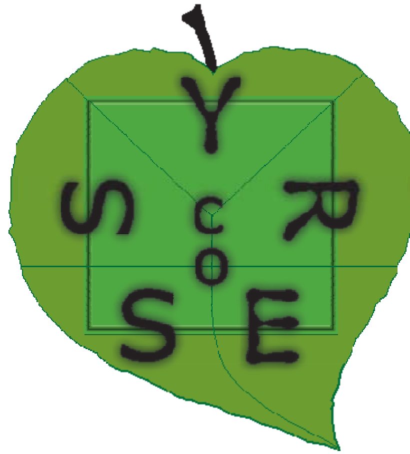


SYRCoSE 2010

Editors:

Alexander Kamkin, Alexander Petrenko,
Andrey Terekhov

Proceedings of the 4th Spring/Summer Young Researchers' Colloquium on Software Engineering

Nizhny Novgorod, June 1-2, 2010

Nizhny Novgorod
2010

Proceedings of the 4th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2010), June 1-2, 2010 – Nizhny Novgorod, Russia:

The issue contains the papers presented at the 4th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2010) held in Nizhny Novgorod, Russia on 1st and 2nd of June, 2010. Paper selection was based on a competitive peer review process being done by the program committee. Both regular and research-in-progress papers were considered acceptable for the colloquium.

The topics of the colloquium include software development methods and tools, functional testing and formal verification, telecommunication software, high-performance computing, software engineering education, and others.

Труды 4-ого Весеннего/летнего коллоквиума молодых исследователей в области программной инженерии (SYRCoSE 2010), 1-2 июня 2010 г. – Нижний Новгород, Россия:

Сборник содержит статьи, представленные на 4-ом весеннем/летнем коллоквиуме молодых исследователей в области программной инженерии, который проводился в Нижнем Новгороде 1-2 июня 2010 г. Отбор статей производился на основе рецензирования материалов программным комитетом. На коллоквиум допускались как полные статьи, так и краткие сообщения, описывающие текущие исследования.

Программа коллоквиума охватывает следующие темы: методы и инструменты разработки ПО, функциональное тестирование и формальная верификация, телекоммуникационное ПО, высокопроизводительные вычисления, образование в области программной инженерии и другие.

ISBN 978-5-91474-015-0

© Авторы, 2010

Contents

Foreword……6

Committees / Referees……7

Software Development Methods, Technologies and Tools

Informational System to Support Development and Usage of Linux Interface Standards
D. Silakov……9

Universal System for Creation and Installation Linux Packages
E. Chernov……17

Single-Window Integrated Development Environment
I. Ruchkin, V. Prus……20

Macromodule Technology
A. Sidnev, V. Gergel……26

On Requirements Completeness Analysis Method
V. Gingina……29

An Approach to Data Validation based on Lifecycle-Bounded Metadata
V. Surpin……33

Functional Testing of Software Systems

A GA-Based Approach for Test Generation for Automata-Based Programs
A. Zakonov, O. Stepanov, A. Shalyto……37

Test Data Generation for Covering Functionality of Database Applications
E. Kostychev, V. Omelchenko, S. Zelenov……43

Testing AJAX functionality with UniTESK
Y. Gerlits……50

Service-Oriented Approach to Integration Testing in Distributed Systems
V. Fedotov……58

High-Performance Computing

GPU-Based Extended Cellular Model Implementation
A. Emelyanov, R. Dmitrienko……60

ParaLab – Visual Way to Parallel Programming
A. Labutina, V. Gergel……63

A DSL for Hardware-Accelerated Grid-Based Scientific Models
A. Gavrilov……69

Event-Driven Simulation

Formalization and Enforcement of Requirements to Modular Discrete-Event Simulation Runtime
E. Chemeritskiy, K. Savenkov..........74

Telecommunication Software Development and Testing

Testing Automation of Projects in Telecommunication Domain
A. Veselov, V. Kotlyarov..........81

Test Suite Development for Conformance Testing of Email Protocols
N. Pakulin, A. Tugaenko..........87

Modeling and Analysis of WAP Protocol Family
M. Alekseeva, E. Dashkova, D. Chaly..........92

Formal Methods for Verification and Test Generation

On the Formal Specification of Automata-based Programs via Specification Patterns
A. Klebanov..........97

On Reasoning about Finite Sets in Software Model Checking
P. Shved..........100

On Context Switch Upper Bound for Checking Linearizability
V. Mutilin..........106

Observable Form of a Timed Finite State Machine
M. Gromov, O. Kondratjeva..........113

On EFSM-based Test Derivation Strategies
N. Kushik, A. Nikitin..........116

Hardware Design and Verification

Comparing GALS Architectures and Communicational Protocols
S. Bykov, S. Mosin..........120

Strategy of Selecting Power Reduction Technique for Energy-Efficient Semiconductor Designs
P. Parnevich, S. Mosin..........123

Contract Specification of Hardware Designs at Different Abstraction Levels: Application to Functional Verification
M. Chupilko, A. Kamkin..........125

An Approach to Test Programs Generation for Microprocessors Based on Pipeline Hazards Templates
A. Kamkin, D. Vorobyev..........130

Analysis and Optimization in Different Fields

Database Index for Approximate String Matching
A. Korotkov..........136

Adaptation of Hierarchical Clustering by Areas for Automatic Construction of Electronic Catalogue
F. Borisyuk, V. Shvetsov..........141

The Method of Programs Compression Based on the Frequency Characteristics of Programs Behaviour
A. Shalimov……………………………………………………………………………...146

Metrized Small World Approach for Nearest Neighbor Search
A. Logvinov, A. Ponomarenko, V. Krylov, Y. Malkov…………………………………………...151

Software Safety and Security

An Approach to on the Fly Activation and Deactivation of Virtualization-Based Security Systems
D. Yefremov, P. Iakovenko…………………………………………………………………157

Software Engineering Education

The Modern Educational Course on Agile Software Development
E. Sorokin, K. Kornyakov………………………………………………………………….162

Programming as a Part of the Software Engineering Education
O. Maksimenkova, V. Podbelskiy…………………………………………………………..165

Foreword

We are glad to welcome you to the 4th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE). This year we have the pleasure of holding SYRCoSE in Nizhny Novgorod, an important economic, transport and cultural center of the Russian Federation. The colloquium is hosted by Lobachevsky State University of Nizhny Novgorod (UNN) and State University – Higher School of Economics (HSE), famous Russian educational and research centers. The event is organized by Institute for System Programming of RAS (ISPRAS) and Saint-Petersburg State University (SPSU) jointly with UNN and HSE.

Program Committee has selected 33 papers that cover different topics of software engineering and computer science. Each submitted paper has been reviewed independently by two or three referees. Participants of SYRCoSE 2010 represent well-known universities, research institutes and IT companies such as HSE, Institute for Information Transmission Problems of RAS, ISPRAS, Lanit-Tercom, Inc., MeraLabs, Moscow State University, National Research Nuclear University "MEPhI", Saint-Petersburg State Polytechnic University, SPSU, Saint-Petersburg State University of Information Technologies, Mechanics and Optics, Special Computing Technologies LLC, Tomsk State University, UNN, Vladimir State University, and Yaroslavl Demidov State University.

We would like to thank all the participants of SYRCoSE 2010 and their advisors for interesting papers. We are also very grateful to the PC members and the external reviewers for their hard work on reviewing the papers and selecting the program. Our thanks go to the invited speakers, Tiziana Margaria (University of Potsdam) and Aleksey Savateyev (Microsoft). We would also like to thank our sponsors, Russian Foundation for Basic Research (grant 10-07-06025-г) and Microsoft Research (in particular, Rostislav Yavorskiy). Finally, our special thanks to Prof. Eduard Babkin (HSE), Prof. Victor Gergel (UNN), and Prof. Oleg Kozyrev (HSE) for their invaluable help in organizing the colloquium in Nizhny Novgorod.

See you next year at SYRCoSE 2011!
Sincerely yours,

Alexander Kamkin, Alexander Petrenko, Andrey Terekhov
May 2010

Committees

Colloquium Chairs

Alexander PETRENKO – Russia
Institute for System Programming, RAS

Andrey TEREKHOV – Russia
Saint-Petersburg State University

Program Committee

Habib ABDULRAB – France
National Institute of Applied Sciences, INSA-Rouen

Sergey AVDOSHIN – Russia
Higher School of Economics

Eduard BABKIN – Russia
Higher School of Economics

Victor GERGEL – Russia
Lobachevsky State University of Nizhny Novgorod

Efim GRINKRUG – Russia
Higher School of Economics

Vladimir HAHANOV – Ukraine
Kharkov National University of Radioelectronics

Vsevolod KOTLYAROV – Russia
Saint-Petersburg State Polytechnic University

Oleg KOZYREV – Russia
Higher School of Economics

Alexander LETICHEVSKY – Ukraine
Glushkov Institute of Cybernetics, NAS

Tiziana MARGARIA – Germany
University of Potsdam

Igor MASHECHKIN – Russia
Moscow State University

Alexander MIKHAYLOV – Russia
National Research Nuclear University "MEPHI"

Valery NEPOMNIASCHY – Russia
Ershov Institute of Informatics Systems

Ruslan SMELYANSKY – Russia
Moscow State University

Valeriy SOKOLOV – Russia
Yaroslavl Demidov State University

Ivan PILETSKI – Belorussia
Belarusian State University of Informatics and Radioelectronics

Vladimir VOEVODIN – Russia
Research Computing Center of Moscow State University

Nina YEVTUSHENKO – Russia
Tomsk State University

Organizing Committee

Eduard BABKIN – Russia
Higher School of Economics

Victor GERGEL – Russia
Lobachevsky State University of Nizhny Novgorod

Alexander KAMKIN – Russia
Institute for System Programming, RAS

Oleg KOZYREV – Russia
Higher School of Economics

Referees

Sergey AVDOSHIN
Eduard BABKIN
Vladimir BASHKIN
Dmitry CHALY
Eugeniy CHERNOV
Mikhail CHUPILKO
Natalia GARANINA
Victor GERGEL
Yevgeny GERLITS
Viktoria GINGINA
Efim GRINKRUG
Maxim GROMOV
Sergey GROSHEV
Vladimir HAHANOV
Pavel IAKOVENKO
Alexander KAMKIN
Eugeni KORNIKHIN
Evgeniy KOSTYCHEV
Vsevolod KOTLYAROV
Kirill KOZLOV
Oleg KOZYREV
Boris KRIVOSHEIN
Victor KULIAMIN
Natalia KUSHIK
Egor KUZMIN
Alexander LETICHEVSKY
Tiziana MARGARIA
Alexander MIKHAYLOV
Vadim MUTILIN
Valery NEPOMNIASCHY
Eugene NOVIKOV
Alexander PETRENKO
Andrey PONOMAREV
Alexey PROMSKY
Yury SCHEKOCHIKHIN
Natalia SHABALDINA
Pavel SHVED
Denis SILAKOV
Kirill SMIRNOV
Sergey SMOLOV
Valery SOKOLOV
Artiom SOLOPOV
Bernhard STEFFEN
Alexander STRAKH
Viktor TATISCHEV
Andrey TEREKHOV
Andrey TRETYAKOV
Andrey TSYVAREV
Anastasia TUGAENKO
Dmitry VOROBYEV
Nina YEVTUSHENKO
Maxim ZHIGULIN

Informational System to Support Development and Usage of Linux Interface Standards

Denis Silakov
Institute for System Programming
at the Russian Academy of Sciences
Moscow, Russian Federation
Email: silakov@ispras.ru

***Abstract*—This paper presents an approach for developing Linux interface standards aimed to improve portability of applications among different Linux distributions. The approach is based on usage of database-driven informational system that simplifies creation and maintenance of interface standards by standardization committees and their usage by application and distribution developers. A logical model of interfaces between Linux applications and distributions is described which is used to design schema of the informational system's database.**

***Keywords*-Software requirements and specifications, Software standards, Data management.**

I. Introduction

The Linux operating system becomes more and more popular. Nowadays it is used not only by enthusiasts, but by many commercial companies, corporations and government organizations. Nevertheless, the market share of Linux in some areas (in particular, on desktops) is still relatively small. One of the main reasons which prevents the growth of Linux popularity in these market segments is lack of applications for this operating system that would satisfy all the needs of target audience.

This lack of applications arises, in particular, from a huge variety of existing operating systems based on the Linux kernel, GNU libraries and utilities and other common components. Such systems are called Linux *distributions*; there are several hundreds of distributions at the moment [1] and the situation is constantly changing – as time goes by, new distributions appear, while the others become obsolete and unsupported, but the total number of distributions is permanently increasing.

Most components that form a distribution are maintained not by distribution vendors themselves, but by different third party developers. This allows to save a lot of resources and efforts, but leads to another kind of problems. The thing is that many developers in the Open Source Software (OSS) world follow the "Release early, release often" policy [2], and it is not uncommon for software updates to appear several times a month. Such often releases lead to situations when a lot of different versions of the same component exist which, in general, provide different functionality. Moreover, distribution vendors often modify software taken from upstream, sometimes slightly, but sometimes significantly – for example, they can add some new unique functionality which will give more advantages to their system with respect to the others. As a result, functionality of the same component in different distributions can vary significantly.

A large variety of distributions provides users with a wide choice of Linux implementations, but such a variety makes it difficult to develop *portable* software that would be able to run in every Linux distribution without any additional actions from the user side. Approaches used by software vendors to increase the number of supported distributions depend on kind of license under which their programs are delivered. From licensing point of view, we should distinguish *open* software, whose source code can be obtained by interested parties for investigation and modification, and *closed*, or *proprietary* software, whose license forbids code modifications.

Developers of open source programs usually leave the task of software adoption for those distribution vendors who want to include their programs. In this case it is distribution engineers who test applications inside particular systems and modify their source code, if necessary. Finally, users themselves can build program from sources (and rely on programs like GNU Autotools that can take care of differences in build environments [3]).

Developers of proprietary software cannot follow this way. Instead, they have to provide binary executable files and shared libraries for their applications that are ready to use "as is", without recompilation or other actions. But it can be very expensive and time consuming to test some application in every existing distribution. That's why many proprietary vendors declare that they only support a few selected systems – usually those that have significant market share, such as SUSE Enterprise Linux or Red Hat Enterprise Linux (for example, IBM XL Fortran supports only these two distributions [4]; Intel Fortran Compiler supports seven systems [5], but this is also not a large number). However, end users normally expect to buy products "for Linux", not "for SUSE" or "for Red Hat".

A promising approach to simplify creation of portable applications distributions is *standardization* – development of requirements that should be satisfied by all standard compatible systems. In our case, *interface* standards are required guaranteeing that every compliant operating system provides certain interfaces (in particular, libraries and functions) that can be used by applications.

Standards are useful not only for proprietary vendors, but

also for developers of open source programs. The thing is that the more modifications are required to adopt a particular application for some distribution, the more likely the modified program will significantly differ from its origin and will be not exactly that thing which the original developer wants it to be. In addition, it's likely that if several programs exist providing the same functionality, then distribution vendors will choose those that require less efforts for maintenance and adoption. Following standards will give developers guarantees that their product will suite perfectly for any standard compliant system and will be unlikely subjected to significant modifications.

Modern Linux distributions are large and provide millions of interfaces of different kinds. For standardization committees, it is important to investigate which interfaces are mostly required and useful; due to a huge number of existing interfaces, some automation of this analysis is desired. But even with careful selection of standardized interfaces, standards can, in turn, become huge, so their size will cause problems for both standardization committees (responsible for standard maintenance and further development) and for developers, who will have to investigate thousands of pages of specification text. Thus, an approach is required to organize development process of an interface standard which will simplify both standard maintenance and development by appropriate committees and standard usage by its target audience – primarily, application and distribution developers.

The remainder of the paper is structured as follows: Section 2 observes the most valuable interface standards in the OSS world and analyzes approaches and techniques used during their development. Section 3 introduces an approach for interface standard development process organization which is based on using of database-driven informational system. Section 4 describes the application of the approach to the Linux Standard Base development process. Finally, Section 5 summarizes the main ideas.

II. Standards in the Open Source World

Portability problem is not a new one for Open Software, and standardization is declared to be one of the key principles of the Open Systems that should solve this problem (at least partially). However, even with such a principle, real life shows that it's not always easy to achieve full compatibility between different products. Problems arise in two areas – standard **development and maintenance** by standardization groups and committees and standard **usage** by its target audience – developers of applications and OS components.

Roots of the first problem lie in a huge number of existing libraries and functions – a modern Linux distribution delivered on a single DVD disk provides several hundreds of libraries which, in turn, export hundreds of thousands of functions. Not all of these functions can be considered as stable, safe, backward compatible, etc. – that is, not all functions can be characterized as a "best practice" and recommended to be used by everyone. One of the main tasks of standardization committee is to select those interfaces that are proved to be useful, and probably try to help to improve those interfaces which are not mature yet. That's why it is important to estimate real needs of applications, capabilities of existing Linux implementations and common practices used to solve particular problems, in order to standardize the mostly requested and important interfaces first. The more so, since besides such interface importance analysis, standardization process involves development and maintenance of specification text, tests and other accompanying products and informational resources – that is, standardization is actually an expensive and time-consuming task, so it is not desirable to waste resources.

Another effect of a large number of existing libraries and interfaces is that standards can become very large, too. This leads to the second problem – large specifications are hard to use for their target audience, since it's not easy to investigate a dozen volumes of specification, several hundreds of pages each. In order to make developers life easier, some standards are accompanied by auxiliary tools, informational resources and other additional products. A common example of such a product is a test suite that can be used to check if application meets all standard requirements. A more sophisticated example is a specialized development environment whose usage during the application compilation and build processes guarantees compliance of resulting program with the standard.

Such auxiliary components form a **standard environment**. All parts of this environment should be kept in sync with each other and with the specification text. For example, if it is decided to remove some interface from the specification, then the test suite for applications should be updated to forbid usage of this interface, the application development tools should be modified to avoid usage of this interface, and so on. Thus, while complicated and feature rich environment of a standard is useful for its target audience, it can significantly complicate development and maintenance of standard and accompanying tools.

One more issue of standardization we'd like to mention is that standards are not always fully suitable for every particular area. It's not uncommon when several standards exist that cover some area or when a small subset of a standard is enough for some class of systems. In such cases, standard **profiles** are developed – unions of existing standards or their subsets aimed to create a specification covering a certain class of systems. As for interface standardization, profiles are asked for when developing highly tailored products – for example, intended to be used only on high-loaded servers or inside mobile devices. Developers of such applications only consider operating systems that can work on their target platforms, and it would be useful for them to have a standard that describes only such particular class of systems. To be sure, existence of specifications that already cover (at least partially) target area can simplify development of a new document, and profile development is usually cheaper then development of a standard from scratch. However, it can introduce its own problems – when selecting subsets of existing standards and then joining these subsets into a single document, it is important to keep internal consistency of resulting specification.

In addition, it can be useful to reuse existing auxiliary tools, and these tools should be also adopted for a new profile – superfluous tests should be dropped, informational resources from different specifications that form the profile should be somehow combined and so on. Thus, profile development is not as cheap as it can seem to be.

All the problems mentioned above are not new and they were faced by different standardization workgroups. Let's consider different approaches used in order to solve them by some famous interface standards that are in use in the Linux world.

A. POSIX and SUS

The most famous and mature open standards for operating system interface are POSIX and Single UNIX Specification (SUS). Initially, these specifications were developed to achieve portability of applications among different UNIX implementations on the source level. This approach supposes standardization of the system Application Programming Interface (API), the core part of which are functions provided by system libraries and declared in appropriate header files. It is guaranteed that any application that meets requirements of some API standard can be compiled from its sources in any operating system compatible with that standard.

Roots of the Single UNIX Specification lie in the Common API Specification, developed in the early 1990th by the COSE alliance formed by all leading UNIX vendors of that time. The main purpose of this alliance was to investigate existing UNIX implementations and create a list of functions that were present in all UNIX systems. The resulting list contained 1170 functions and due to this reason it is also known as Spec 1170. In 1992-1993, during the SUS development, an additional research of 50 leading UNIX applications was performed and additional list of 130 functions was created that were suggested for standardization [7].

Application and distribution analysis during SUS and POSIX development was primarily performed manually and involved deep source code investigation by analysts. In early 1990th, this approach was suitable and allowed to perform a high quality and complete analysis.

A problem with initial versions of POSIX and SUS was that these standards considered only some relatively low level functions and calls, but this was not enough for many applications even in that time – such popular areas as graphical user interface or multimedia were completely out of standardization scope. The need for more areas was understood by standardization committees, and it was decided to develop several SUS profiles – specifications that were based on POSIX but extended it with interfaces specific to particular areas. The SUS version 2 specification presented three profiles – *Base Specification* (predecessor of POSIX 2001), *UNIX98 Workstation* (with GUI requirements based on the Common Desktop Environment – CDE – and the Motif library) and *UNIX98 Server* (specifying additional network services and Java Runtime Environment).

Unlike the base specification, extended profiles were suitable for UNIX-based systems only – for example, there were no free Motif and CDE implementations for Linux. Moreover, there were no concurrent implementations of CDE or Motif at all; concurrent implementations of some other standardized items were allowed, but they had to follow other existing specifications (like Java RE). Thus, during extended UNIX profile development, standardization workgroups didn't have to analyze alternative implementations, they only had to choose some top-level standardization directions – for example, once it was decided that CDE would be a standard desktop environment, there were no need to investigate different (and partially incompatible) implementations of CDE, since there was only one implementation of it in the wild.

On the other side, the POSIX itself was divided on several subsets that also formed a set of profiles – such as POSIX.1b real-time extensions. However, these profiles were even smaller than POSIX and their creation haven't require investigation of some new standardization techniques.

B. LSB

An alternative approach to API standardization is to standardize Application Binary Interface (ABI), giving developers an opportunity to use the same executable files and shared libraries in all compliant systems, without a need for recompilation. The core part of such ABI standards are shared libraries that should be provided by operating system and binary symbols exported by them (*binary symbol* is a binary level entity corresponding to either a function or a global variable exported by library). For application developers, this ABI standardization is more preferable than the one for API, since it doesn't require any actions (neither from developers nor from users) in order to port a program to any standard compliant system. However, ABI standards contain much more limitations for OS – in particular, it is clear that all target systems should use the same format for binary executables and shared libraries. That's why ABI standards often cover less systems than API ones.

Nowadays this approach is used by the Linux Standard Base specification (LSB) which is intended to be applied for Linux based systems only [12]. Roots of LSB lie in POSIX and SUS, and standardization process is also similar in many ways. In particular, LSB developers constantly perform analysis existing distributions and applications in order to select the mostly important and useful interfaces. Initially, the analysis process was also performed manually; but up to now the size of data that should be analyzed increased dramatically, and manual analysis doesn't work fine any more. In particular, during LSB 3.0 development, only interfaces provided by RHEL and SLES distributions were taken into account, while there were several hundreds of different Linux distributions in the world.

LSB has a rich environment, consisting of test suites, development environment for application vendors, online informational resources and other products. All these items are, on the one hand, independent products; on the other hand, they

all represent the LSB in some way and should be kept in consistency with it. The size of all these products makes it hard to perform such synchronization manually; in order to automate this task, a *specification database* was designed to store some information about standardized elements accompanied with a set of tools that were used to synchronize LSB environment components with each other and with LSB itself.

After LSB 3.0 was released and development of the next version was started, it became clear that the current infrastructure implies too many manual work and can't satisfy the all the needs of the LSB workgroup. In December, 2006, Ian Murdock (CIO of Free Standards Group that was responsible for LSB development at that moment) on the LSB Face-to-Face meeting formulated the following problems of the LSB Infrastructure [12]:

- absence of possibilities of Linux ecosystem analysis that would allow to effectively select further development directions;
- complexity of support of several LSB versions at once caused by absence of information about standard evolution in the database;
- high complexity of adding new interfaces to LSB – though the database solved the problem of synchronization of specification text and environment components, the task of populating database with data was not a trivial task;
- lack of auxiliary tools that would help distribution and application vendors to use LSB in the development process.

Summarizing POSIX and LSB experience, we can conclude that as the size of operating systems (measured in a number of interfaces) grows, the amount of work to be performed by standardization committees increases dramatically, and those approaches for standard development that proved to be useful a decade ago nowadays fail to satisfy all the needs of both standardization committees and those developers who use standards. New approaches are required that would help both standardization workgroups and standard users to perform their work effectively.

III. An Approach for Linux Interface Standards Development

In this paper, we present an approach for Linux Interface Standards Development. The approach includes the following stages:

1) Analysis of the Linux ecosystem:
   - selection of popular and mostly important applications, analysis of their requirements for system libraries and functions;
   - collection of information about existing distributions – in particular, about provided libraries and exported functions.

   The set of applications and distributions is constantly evolving, so it is necessary to have data not only with respect to some fixed time point, but collect information about the Linux ecosystem evolution during last several years. It is important to perform constant monitoring of the ecosystem, and results of this monitoring at some certain time points can be used to create next version of a standard, as demonstrated at Fig.1.

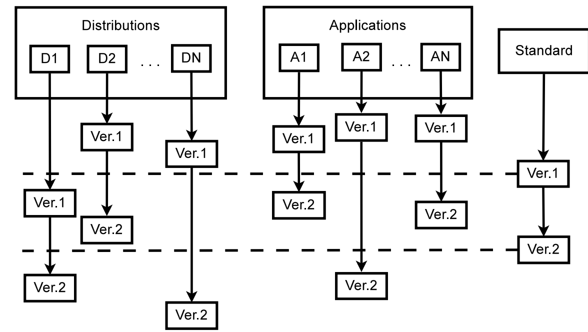


Fig. 1. Analysis of the Linux ecosystem during standard development

2) Preparation of a new standard version. This stage includes selection of interfaces which are mostly needed by applications, proved to be stable and provided by all modern distributions. Then, on the basis of this set of a consistent set of interfaces is constructed which will be included in the specification.
3) Addition of semantic information (in particular, descriptions of functionality that should be provided by interfaces), development of tests, adopting the standard certification system to support certification process for the new version and other tasks that should finalize release of a new standard version.

In order to support this method, we suggest to build an informational system which could be used to automate (at least partially) the mostly time-consuming tasks. The suggested informational system is based on a logical model of interfaces in the Linux ecosystem.

A. Logical Model of Application Interfaces with the Linux OS

In this paper, we concentrate on Application Binary Interface (ABI) – that is, we consider interfaces between binary executables and libraries of applications and shared libraries of distributions. Thus, we consider applications as a set of compiled files (executables and shared objects). In Linux, the main format used for such files is ELF ("Executable and Linking Format"). In our model, we'll include some items related to the ELF format; the general ELF description is provided by the System V ABI Specification [6]; some Linux specific extensions are described in the appropriate LSB sections [8].

All properties of any item which is a part of system ABI or API can be divided in two groups:

- **structural** properties, that can be checked statically – for example, names of functions exported by library or signature of any function from a given header file;

- **semantic** properties, whose analysis usually requires runtime testing – for example, function behavior.

The model described in this paper includes structural interface properties only, abstracting away from semantic aspects.

As elements represented in the model, we use interfaces involved in the process of **dynamic loading** of application files [10]. Compatibility between application and distribution with respect to such interfaces guarantees that application can be successfully *launched* in the distribution – that is, dynamic loader will be able to resolve all external dependencies of application, form the executable image in memory and pass the control to application's main entry.

The following interfaces are considered:

- **libraries** – a special kind of ELF files that can *export* interfaces;
- **binary symbols** exported by libraries – these are binary level entities corresponding to functions and global variables;
- **structure and size of types** used as function parameters and return values;
- **ELF file attributes** – *class* (32bit or 64bit), *target architecture* of a file and *types of sections* that exist in file.

Concentrating on application launching process, the model leaves out of account the following ways of interaction between Linux applications and distributions:

- dynamic loading of shared libraries and dynamic invocation of symbols exported by them at runtime (for example, using the *libdl* library capabilities);
- invocation of external commands and utilities at runtime (for example, using the *system* or *exec* functions).

However, modern recommendations on developing of portable applications forbid usage of such possibilities, unless all files involved in the interaction are part of the application. Indirect dependency on a system library or command cannot be checked by means of the operating system itself (e.g., by dynamic loader), so it is application developer who should check that necessary files exist and provide all required interfaces. However, such checks add complexity to any program, and improperly performed checks can lead to program crash or unexpected behavior [9].

B. Informational System to Support Development and Usage of Linux Interface Standards

In order to support the approach to interface standard development described above, we use an informational system providing the following possibilities:

- planning of further standard evolution;
- creation of new versions of standard and its profiles;
- ensuring consistency of standard environment components;
- checking of how different Linux distributions and applications are compliant with the standard.

The informational system is aimed to automate the most time consuming tasks that arise during the processes described above.

The main components of the system are like the following:

- a **database** with information about both standardized interfaces and interfaces used by existing applications and provided by distributions. The database schema is based on the logical model of interfaces described above;
- automated **data collection tools** used to gather information to populate the database with data;
- automated **generators** that use the database to create components of standard environment.

The database should store information about all interfaces with their characteristics described in the specification which are used by at least one component of the standard environment. If any component during its work requires some information about standardized interfaces which is described in the specification, this information should be either directly queried from the database when such a need occurs, or should be embedded in the component code at compilation time by appropriate automated generators. In particular, if some component needs to know the list of included interfaces, this list should be always taken from the database. This approach guarantees that all components are kept synchronized with each other and with specification text. To be sure, it is required for the specification text itself to be synchronized with the database; one of the ways to achieve it is to generate those parts of the text that are represented in the database – that is, the database should be the only one source of information about standardized items.

Besides the information about standardized items, the database should also contain all the data which is used by several components of standard environment, even though this data doesn't concern the standard itself. This will allow to keep different components synchronized with respect to their common data.

Due to a large number of interfaces that exist in the Linux world and should be subjected for analysis, data collection tools should be as automated as possible. Collection of data about interfaces included in our logical model can be almost fully automated, as demonstrated in authors' work [15]. Moreover, collection of additional information (e.g., header files) which is not used during Linux ecosystem analysis but required for development of different LSB environment components can be also automated significantly [14].

A data work flow diagram in out informational system is shown at the Fig.2.

In order to store information about interfaces that exist in the Linux ecosystem, we suggest to represent each kind of interface as two separate entities in the database schema – the first one will correspond to standardized interfaces of this kind, the other will represent interfaces which are present in distributions and used by applications. The reason is that information about standardized objects and data about ecosystem interfaces are used in different ways – the former is picked up by environment generators, the latter is supposed to be used during the ecosystem analysis, when planning further directions of standard development. In general, these

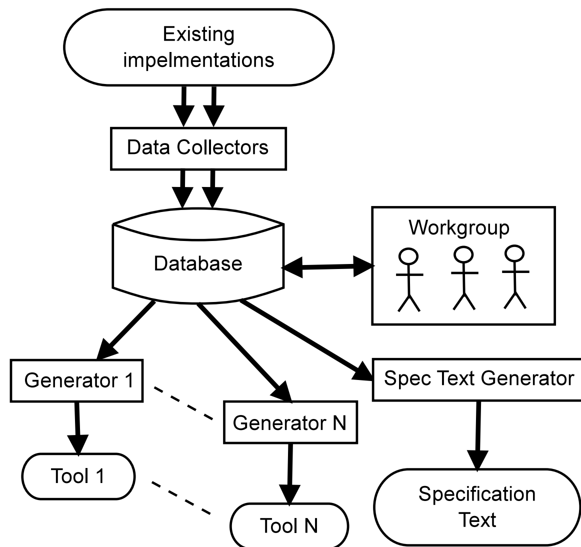


Fig. 2. Data work flow in the informational system

two tasks can require knowledge about different characteristics of the same interfaces. In particular, due to the big amount of existing interfaces that should be subjected to analysis, it can be reasonable to store only those ecosystem data that can be collected automatically; however, standards can be more descriptive and include more characteristics in addition to the collected ones, so the automatically collected data can be insufficient for environment generators.

In order to store information about several versions of a standard (that is, to store standard *history*), the database schema should be extended with attributes containing *temporal* data. Different approaches exist for introducing such extensions; in our work, we use the Temporal Relationship Model (TRM) [11], which is based on the relational model but adds new temporal attributes to every relation. With this model, there is no need to use a specialized temporal DBMS; the database can be served by any relational DBMS – the most popular and widespread kind of DBMS at the moment.

The two obligatory attributes added by the temporal model are the beginning and the end of entity *life period* – a time interval during which the entity preserves its characteristics. In our case, such interval boundaries are standard versions – that is, a time interval for some standardized item indicates a set of standard versions where this item was included with the same characteristics. A special value *NULL* is used to indicate unbounded intervals which correspond to items which exist in the last standard version (that is, that have been never excluded from the specification).

Temporal attributes are added only to those entities that correspond to standardized items; these attributes are not required for entities that represent interfaces existing in the Linux ecosystem. More details about using temporal databases for tracking standard evolution can be found in another author's work – [13].

IV. THE LSB INFRASTRUCTURE PROGRAM

One of the largest standards that specify interfaces of the Linux OS is Linux Standard Base (LSB). The standard is being developed by international consortium named The Linux Foundation which is formed by leaders of the Linux market. The primary content of the standard is formed by lists of libraries that should be present in any compliant Linux distribution, accompanied by lists of binary symbols that should be exported by these libraries. The standard is constantly evolving, and more and more interfaces are added – the latest version, LSB 4.0, describes more than 38.000 functions from 57 libraries. It is noticeable that during the four years passed from LSB 3.0 release, more than 30.000 functions were added.

Such a swift growth of the specification size exposed some significant problems in its development process and surrounding infrastructure. Among the most important issues, the lack of support for Linux ecosystem analysis was mentioned, as well as difficulties with specification text usage by application developers – even LSB 3.0 consisted of several thousands of pages and contained references to several dozens of other specifications [12].

In 2006, the joint Program of The Linux Foundation and Institute for System Programming of RAS was started aimed to improve the LSB Infrastructure. The main purpose of the Program was to resolve existent issues that made difficulties for standard maintenance; it was decided to create an informational system that would allow to both simplify further LSB development and simplify its usage by target audience – Linux application developers and distribution vendors.

By the beginning of the Program, the LSB infrastructure already contained a central database with information about standardized interfaces. That database was used to generate parts of the specification text (lists of libraries, binary symbols, etc.), to create header files and stub libraries for LSB Development Environment and to generate primitive tests checking presence of certain objects (libraries, commands, etc.) in distributions.

During the LSB Infrastructure Program, the following tasks were performed:

- an extension of the LSB database was developed called *Community Database* to store information about interfaces provided by existing Linux distributions and used by Linux applications; automated tools were developed to collect such data and populate the database with it. Nowadays that database contains information about 250 Linux distributions and 1200 applications;
- during the LSB Navigator development, automated tools were created aimed to support analysis of data about existing Linux distributions and applications during the LSB development process. The tools allow to discover potential candidates for standardization and check formal rules that should be met by candidates to be finally included in the specification;
- a temporal extension of the LSB database was developed

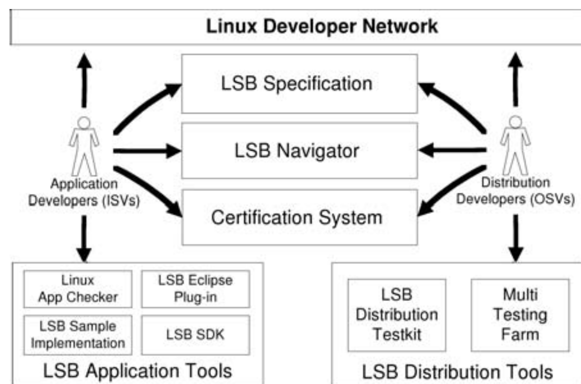


Fig. 3. LSB Environment structure

to store information about all existing LSB versions. All tools that use information from the database were modified to be able to extract data corresponding to any given specification version. Moreover, some products created using the database now support several LSB versions at once – in particular, the LSB Development Environment can be used to build applications compliant with any given LSB version.

Nowadays the work is in progress on improving profile support in the LSB Infrastructure, caused by a need to develop a profile for mobile devices.

The current structure of the LSB Environment is shown at the Fig.3.

The tools developed during the Program allowed to reorganize LSB development process – automation of many time-consuming tasks allowed LSB workgroup members to concentrate on their primary objective – selecting interfaces that should be included in the specification and elaborating descriptions of their behavior. Moreover, the decision making process itself was also significantly improved – the new infrastructure allowed to perform deeper analysis of the Linux ecosystem and to better understand current needs and evolution tendencies of applications and distributions. For example, during the LSB 3.0 development only two distributions were subjected to deep analysis (RHEL and SLES), and information about application needs was limited to direct requests from application developers (expressed in either LSB Bugzilla or mail lists). With the new infrastructure, during the LSB 4.0 development the workgroup analyzed all versions of 12 distributions released during the last three years and more than 1.000 applications.

This, in turn, allowed to significantly increase the number of standardized interfaces from 6.000 in LSB 3.0 to 38.000 in LSB 4.0. Nowadays we can say that the most significant problem with standardization of new interfaces is development of runtime tests; all other tasks (collecting data for the LSB database, keeping components of the LSB Development Environment synchronized, etc.) are highly automated and do not require much engineering efforts.

V. Conclusion

This paper has suggested and approach of developing Linux interface standards aimed to improve portability of applications among different Linux distributions. The approach is based on usage of a database-driven informational system that simplifies creation and maintenance of interface standards and their environment by standardization committees and their usage by application and distribution developers. A logical model of interfaces between Linux applications and distributions is described which is used to design schema of the informational system's database.

Usage of a central database to create different components of the standard environment allows to keep these components synchronized with each other and with the specification text automatically – every change in the database is automatically reflected in all components by means of appropriate generators. Temporal extensions of the database allow to store standard evolution history, which, in turn, allows to support several standard versions by means of the same database and accompanying tools.

Though in this paper we have considered ABI standards, the approach suggested is suitable for developing API standards, too. In order to support API specification, the model of interfaces between Linux applications and distributions should be modified – binary-only elements (e.g., ELF attributes) should be dropped, while entities that are present on source level only (e.g., constants and macros) should be added. Actually, the LSB database, described in this paper, already store some source-level entities and tools exist to automate collection of such information.

The LSB Infrastructure project has demonstrated the practical strength of the method of Linux interface standards development suggested in this paper. The informational system created during the project allowed to automate analysis of the Linux ecosystem and significantly increased the speed of decision making process. The automated data collection tools and database-driven generators eliminated the technical complexity of adding new interfaces to LSB. Finally, the new LSB Infrastructure supports development of profiles based on the LSB specification.

References

[1] The LWN.net Linux Distribution List. http://lwn.net/Distributions/
[2] E. Raymond. *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.* O'Reilly Media, Inc.; Revised &; Expanded edition, 2001.
[3] G.V. Vaughan, B. Elliston, T. Tromey, I. Lance Taylor. *GNU Autoconf, Automake and Libtool*. Sams Publishing; online edition, 2006. [Online] Available: http://sources.redhat.com/autobook/
[4] IBM XL Fortran for Linux. [Online] Available: http://www-01.ibm.com/software/awdtools/fortran/xlfortran/linux/
[5] Intel Fortran Compiler Professional Edition 11.0 for Linux – Installation Guide and Release Notes. [Online] Available: http://cache-www.intel.com/cd/00/00/40/60/406087_406087.pdf
[6] System V Application Binary Interface. 24 April, 2001. [Online] Available: http://refspecs.linuxfoundation.org/elf/gabi4+/contents.html
[7] A. Josey. *API Standards for Open Systems.* The Open Group, 2001. [Online] Available: http://www.opengroup.org/austin/papers/wp-apis.txt

[8] Linux Standard Base Core Specication 4.0. Executable And Linking Format (ELF). [Online] Available: http://refspecs.linuxfoundation.org/LSB_4.0.0/LSB-Core-generic/LSB-Core-generic/elf-generic.html
[9] *Coding practices for compatibility.* Hewlett-Packard Developer & Solution Partner Program. [Online] Available: http://sysdoc.doors.ch/HP/compat.pdf
[10] M. Tim Jones. *Anatomy of Linux dynamic libraries.* IBM developerWorks, 2008. [Online] Available: http://www.ibm.com/developerworks/linux/library/l-dynamic-libraries/
[11] Abdullah Uz Tansel. "Temporal Relational Data Model." *IEEE Transactions on Knowledge and Data Engineering,* vol. 9, N3, pp. 464-479, May-June 1997.
[12] Ian Murdock. *LSB Overview and Progress Report.* LSB Face-to-Face Meeting. December 2006. [Online] Available: http://www.linuxfoundation.org/images/c/c2/Lsb-f2f-200612-overview.pdf
[13] D. Silakov. *Tracking Specification Requirements Evolution: Database Approach.* Proceedings of SYRCoSE 2007, vol. 2, pp. 15-22. Moscow, Russia.
[14] E. Novikov, D. Silakov. *The Automated Analysis of Header Files for Support of the Standardization Process.* Proceedings of SYRCoSE 2009, pp. 27-34. Moscow, Russia.
[15] D. Silakov. *Linux Distributions and Applications Analysis During Linux Standard Base Development.* Proceedings of SYRCoSE 2008, vol. 1, pp. 11-18. St.Petersburg, Russia.

Universal System for Creation and Installation Linux Packages

Chernov Evgeny
Institute for System Programming of RAS
Moscow, Russia
e-mail: ches@ispras.ru

***Abstract*—This paper discuss about the problem with distribution of software for Linux operation system. The problem is that Linux systems are different: different package formats, different names of packages for one program. It leads to creation of several packages for every Linux system for only one program you want to distribute. For resolving the problem mathematical model was developed. Based on the model database with information about dependences and packages of some Linux systems and web service for searching equivalent dependences are developed. Besides, some special tools that can help developers and users to work with different distribution and alien packages are developed.**

Keywords - Software packages; Software portability; Operating systems

I. INTRODUCTION

There are several ways for distribution programs from developer (where it's presented as source code) to users (who use it in compiled form). The first one is a distribution in source code form (Fig. 1).

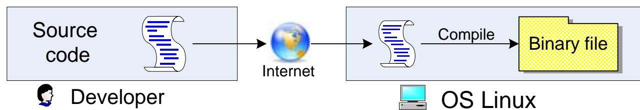
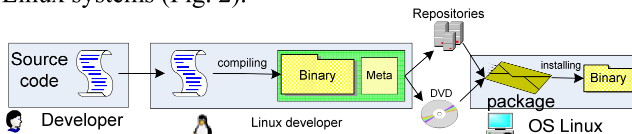


Figure 1. Distribution in source code form.

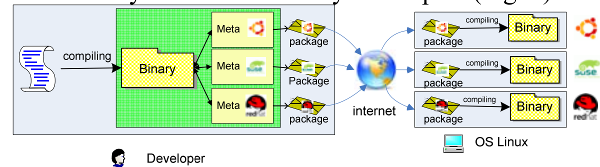
In this case user should compile and install program. But not all users are able to do it. So this way is more convenient for advanced users who know what compilation means and can fix some problems than can appear during compilation.

The second way is a distribution through repositories of Linux systems (Fig. 2).



In this case Linux developers compile the program by yourself in their system and build a package – a special format file that contains archived program and meta-data for it (name, version, architecture, description, checksum and dependency list). Such packages are placed into repositories that are different for every Linux system. Users can install these packages on their systems via special tools from corresponding Linux system. However not all program developers can use this way because Linux developers can't serve all programs.

The third way is a distribution by developers (Fig. 3).



In this case developer build program and create package by himself. However meta-data of package depends on distribution on which it's targeted to install. Names of dependencies are different on different Linux distributions. So developers have to create several packages for every Linux system. This makes it difficult to distribute programs by developers and to find proper packages for users.

So in common developers have to distribute programs by themselves. In this case they are called *Independent Software Vendor (ISV)*. The way is hard because of package incompatibility of Linux system. Different package format can be used in different Linux system and the name of packages of one program can differ. And it leads to the different dependency names.

This paper deals with the system which can determine corresponding names of dependency on different Linux systems automatically and about tools that using the system and help developers to create packages for different distributions and users to install alien packages (created for other distributions). The system is developed as web service. The tools can connect to the service and receive the names of dependencies for particular Linux system.

II. WEB SERVICE AND ALGORYTHM OF DEPENDENCEC SEARCHING

The web service for definition of equivalent dependences on different Linux distributions contains two interfaces:

1) For determination of the status of dependence on the particular Linux. It takes the name of dependency and returns one of the following values:
   - OK – the dependence exists on the distribution (there is a package that provides such entity).
   - NO – the dependence is not present on the distribution (no package provides it).
   - FILE – the dependence is presented in the distribution as a file.
   - MULTIPLE – the dependence is provided by several packages on the distribution.

2) For determination of names of dependences on some distribution. This interface takes list of dependences on any distribution and the information about the distribution (the name, the version, architecture) and returns the list of equivalent dependences on other distributions.

Based on this service the web interface has been developed. It can help user to know the names of some dependences on all supported distributions. Program interface of web service can be used by some systems like system for automatic package building (openSUSE Build Service) and by tools for converting package format – alien.

The web service works on the basis of a database. The main tables in this base are tables of distributions, packages, files and elements of dependences. Besides, there are some tables for links: the first defines, what files are provided by each package, the next two define what elements of dependences packages require and provide. Besides, there is a table for storage of results - equivalent sets of dependency names for different distributions. In case of repeated search of equivalent dependences results takes from this table and new search is not made.

The mathematical model lies in the basis of database structure and algorithm of search of equivalent sets. It operates with the following sets: set of distributions, packages, files and elements of dependences. Links between them (mapping from one set to another) and their properties are defined. Besides, the task of search of equivalent set is formally given.

Informally the task can be formulated by the following ways:

Having set of dependency names from one distribution P and information about the distribution D, determine the set of equivalent dependences on other distributions which satisfy to following conditions:

1) ***The Condition of existence of the decision**: if any distribution does not provide files, which P requires (i.e. the set of files of all packages of this distribution does not include set of files of packages, which package P requires), then the decision for this distribution doesn't exist (i.e. there is no equivalent set for this distribution). Otherwise it should satisfy to following conditions:*
2) ***The Condition of sufficiency of the decision**: the received set for any distribution should include all packages which initial package P on this distribution requires.*
3) ***The Condition of necessity of the decision**: the received set should not contain superfluous elements, i.e. at an exception of any element of it the sufficiency condition should be violated.*

The algorithm of the decision of this task is based on the following assumptions:

1) If any application depends on a certain package on the distribution then this application depends on the files provided by the given package on this distribution.
2) Names of files of the same program on different distributions are identical (thus a path can vary).

Thus, having the information on packages and files which the packages provide, it is possible to find what files the initial package requires on one distribution. Then to find these files on other distribution and to find what packages provide them. Thus we have the list of dependences for other distribution.

The described way of search allows finding the decision, satisfying to a sufficiency condition, but thus it can not satisfy to a necessity condition. Really, many "superfluous" elements can appear in search result. For example, in case of a file on some distribution is provided by several packages all of them will be presented in the result set, however presence only one of them is necessary. Therefore in the course of search it is necessary to delete such "duplicating" packages. Formally, it is possible to leave any of these packages since in any case necessity and sufficiency conditions will be satisfied. However for the best visual representation of the result set some heuristics are used for removing such duplicating packages: to remove the package that provides smaller number of files or not to remove package which name coincides with the name of one of dependences on the initial distribution.

There is one more reason of growth of number of dependences in the result set. The matter is that packages on which depends initial one provides set of files which aren't program and libraries. It can include documentation files, video, the pictures, etc. Such files on other distribution can be provided by set of packages (for example, a file "readme"), however in reality the initial package doesn't require it. So the packages found this way will be superfluous in the result set. To avoid such case it's necessary to specify the first assumption:

If any application depends on a certain package on the distribution then this application depends on <u>some</u> files provided by the given package on this distribution.

Determination of this set of "some" files occurs on the basis of classification of all files on the distribution. There are 7 classes of files: *programs*, *libraries*, modules, *links*, the empty files (*gags*), *not existing* in a package files (they can be created after package installation) and *others*. Each class has some more subclasses (for example, "the program on Perl" or "module Python"). Text files, archives, video are placed in "others" class and in a corresponding subclass ("a text file", "archive", "media").

For each class with a subclass some priority is attributed. For example, "programs" have the highest priority – 1, "text files" – the lowest – 7, "links to modules" – 4 priority. By means of these priorities, in each package it is possible to allocate the *main files* – files with the highest priority. Thus, search of packages on other distributions can be made on the basis of search not everything, but only main files of the package. For example, if the initial package requires on a package that provides 1 program, 5 pictures and 10 text files, then search only one program will be made on other distribution.

The result of work of the web service, in which the described algorithm is implemented, is presented in table 1. The first line contains names of distributions. Further there

are lists of equivalent dependences. The bold type in each line specifies dependence for which search was made.

<table>
<tr><th>OpenSUSE 11.1</th><th>Fedora 11</th><th>Debian 5</th><th>Mandriva 2009</th></tr>
<tr><td rowspan="3">kdebase4-runtime</td><td>kdebase-runtime</td><td>kdebase-runtime</td><td>kdebase4-runtime</td></tr>
<tr><td rowspan="2">kdebase-runtime-libs</td><td>kdebase-bin-kde3</td><td rowspan="2">libkaudiodevicelist4</td></tr>
<tr><td>kdebase-runtime-data</td></tr>
<tr><td>libqt4-x11</td><td>qt-x11</td><td>libqtgui4</td><td>qtguilib</td></tr>
<tr><td>perl</td><td>perl(Test::More)</td><td>perl-modules</td><td>perl</td></tr>
<tr><td>xorg-x11-libXext</td><td>libXext</td><td>libxext6</td><td>libxext</td></tr>
</table>

Table 1. Equivalent dependences search result

As it's apparent from the table, for one dependence on one distribution can exist a little ones on others. Besides, names of dependences can essentially differ.

The described functionality of the web service can be used by some special tools that simplify creation and installing packages for different Linux distributions. There are 4 different tools: for creating and installing universal packages, for installing alien (created for other distribution) packages and for converting packages from one distribution to the package for other distribution.

III. Tools for Management of Universal Packages

The universal package – is a package of a standard format (rpm [1] or deb), storing in itself lists of dependences for all distribution kits. Thus, in the course of installation for any distribution it will be possible to check up, whether dependences of the given package on concrete system are resolved.

Author has developed a tool called '*UPackageBuild*'. The tool works on a basis of the web service and is used for creation of universal packages. This tool takes a package of a standard format (rpm or deb) and Linux distribution name on which it has been created. The list of dependences is taken from the package and is sent to the web service. The web service returns the lists of equivalent dependences for other distributions. The received information is placed in a file "/etc/upackage/package_name.requires" which is added to file-list of an initial package. Initial dependences at a package are cleaned, and to the version the prefix "Universal" is added.

The universal package received by this way is a usual rpm - or deb - package which does not have dependences, but have "/etc/upackage/package_name.requires" file that contains the description of dependences for each distributions. The given package has not own format. That makes it possible to install it by standard utilities (rpm – for rpm-packages and dpkg – for deb-packages). However in this case there is no check of dependences and that can lead to incorrect operation of the program or it can not work at all.

For correct installation of universal packages it is necessary to take advantage of the other special tool – '*Upackage*'. The given tool takes from the installing package the file with dependences, finds in it the list of dependences for the concrete distribution and checks, whether the given dependences on the concrete distribution are resolved or not. In success the package is installed, otherwise the list of packages which are necessary for install before installation of the given package is displayed.

In case of package installation by standard tools, it is checked, if 'Upackage' is installed on the system, then installation interrupts with the requirement to make installation by the given tool. If such tool is not installed, the message is displayed that the given package is intended for installation by Upackage tool, where it is possible to download it and the prevention that in case of package installation by standard tools, the user is responsible that all necessary dependences are installed on the system.

IV. Conclusion

The given paper describes the system for determination of equivalent dependences of packages from different Linux distribution and some special tools that using this system help developers to create packages for different distributions and help users to install alien packages on their system. The offered decision is based on mathematical model where the formal task is set. Then the database of dependences of distributions is developed according to this model. Using the database the web service provides interfaces for the analysis and search of dependences on different Linux distributions. On a basis of web service the author had developed tools for creation and installation of universal packages which can be installed on different Linux distributions.

V. References

[1] Eric Foster-Johnson. RPM Guide. — Indianapolis, Indiana: Wiley Publishing., Inc. 2003 — C. 3-6

Single-window integrated development environment

Ivan Ruchkin
Computer Systems Lab
Moscow State University, CS department
Moscow, Russia
ruchkin.ivan@gmail.com

Vladimir Prus
Computer Systems Lab
Moscow State University, CS department
Moscow, Russia
vladimir.prus@gmail.com

Abstract — **This paper addresses the problem of IDE interface complexity by introducing single-window graphical user interface. This approach lies in removing additional child windows from IDE, thus allowing a user to keep only text editor window open. We describe an abstract model of IDE GUI that is based on most popular modern integrated environments and has generalized user interface parts. Then this abstract model is reorganized into single windowed interface model: access to common IDE functions is provided from the code editing window while utility windows are removed without loss of IDE functionality. After that the implementation of single-window GUI on KDevelop 4 is described. And finally tool views and usability of several well-known IDEs are surveyed.**

Keywords – integrated development environment (IDE); graphical user interface (GUI); usability; widget; single-window interface/design/approach; tool view (utility window); KDevelop; Microsoft Visual Studio; Eclipse; Code::Blocks; NetBeans.

I. INTRODUCTION

There is a wide variety of tools that software engineers use to write the code – from simple text editors, such as Notepad or Kate (which offer only basic text highlighting) to elaborate integrated development environments (IDEs) like Eclipse or Microsoft Visual Studio. IDEs bring together different tools improving convenience, and also providing features not possible with individual tools. For example, reliable code-completion is only possible when the editor, the compiler, and the build system work closely together.

But despite those attractive features many developers find IDE hard to use and stick with simple editors [1]. Reasons for that are different and large usability study [2] is required to completely and accurately determine them. However, anecdotic evidence shows that many users complain about the IDE tool views – which are auxiliary windows typically docked around the editor area, also called utility windows [3][4]. One problem is that a dozen of available tool views just confuse a new user. But a deeper problem is that the tool views are actually required in a number of common workflows. And if a given tool view is repeatedly required, a user is faced with unpleasant choice – either to keep the window always visible, taking space from the source editor, or to constantly open and close it. One way to address this usability issue with tool views is to remove them altogether. Of course, we propose this only as research experiment: remove all tool views, design alternative mechanism to support common workflows and then perform a usability study. These steps allow us to determine a set of tool views that users cannot live without and have to be put back. But we also hope that some utility windows can be eliminated, resulting in an overall usability improvement. One apparent example of removable tool view is build results view which shows errors and warnings. It only makes sense to process error messages from the first to the last one (as later error can be induced by the earlier) and therefore there's no necessity of showing a list of errors – we can immediately display the first error inside the text editor.

So we come to the idea of IDE with the single window hosting a text editor. Researching such approach requires:

- designing conceptual single-window IDE interface,
- implementing it in certain IDE,
- usability testing [2] of implemented GUI and comparing results to existing popular IDEs.

This paper is a work-in-progress report that covers only first two steps. The exact plan of this paper:

- Create a model of IDE tool views by observing them in the most popular development environments [1][5]. This model includes a set of abstract tool views with description of their structure and usage.
- Design a conceptual single-window IDE GUI by removing the utility windows from IDE tool view model.
- Implement the single-window interface in KDevelop integrated development environment.
- Survey existing IDE and find out whether they can be used without tool views.

It is difficult to move functionality of all tool views to text editor window. Trying to do this can cause usability problems [6]. To solve this issue we introduce new widgets and mechanisms to display information inside and near text editor. These widgets are described while designing a conceptual single-window interface.

In the following section the IDE tool view model is proposed.

II. MODEL OF IDE TOOL VIEWS

In this section we build a model of IDE tool views. This model describes a set of abstract tool views and their functions. Each abstract tool view is generalization of similar real tool views from existing IDEs. We build such model to be able to construct a generally useful set of interface improvements, as opposed to fixing problems in a single arbitrary selected IDE.

Development process greatly depends on the programming language. To limit the scope of research we only consider development in a compiled object-oriented language such as C++, Java, or C#. According to [1] [7], the most popular IDEs for these languages are Visual Studio, NetBeans, Eclipse, KDevelop, and Code::Blocks.

First, we need to list the common parts of an IDE interface to show the context in which tool views are used. GUI of a contemporary IDE consists of:

- Main menu, typically with a vast set of commands. It isn't easy for a new developer to discover all of operations in the main menu, that fact can cause usability issues [8].
- Customizable toolbar with command buttons. It facilitates accessing and learning IDE functions, but requires a careful selection of command to expose b default.
- Text editor window. Obviously that's where program code is being edited. This area contains source code and is usually tabbed. Unfortunately, in common IDEs developer is drawn away from the text editor window because a lot of functions and information are spread across other UI elements, mostly utility windows.
- Tool views. These are additional utility child windows inside IDE GUI that contain special instruments: errors view, call graph, project view and many others. Typically views can be docked on any side of text editor window, resized and made auto hiding – popping up when mouse cursor hovers over their header and closing when it leaves. Tool views occupy considerable amount of screen space and compete for it with text editor.
- Status bar. It is a horizontal one text line-high widget in the bottom of IDE GUI that traditionally contains information on current operation progress and rarely some static information about current state of development.
- Context menus in text editor window. That's a powerful tool to provide various operations: a user accesses it quicker than the main menu and learns faster.

Due to work-in-progress paper size limit, we omit the full study of several IDEs and their tool views. The result of this study is a set of abstract tool views, collected from examined IDEs. The key abstraction points were tool view structure and functionality.

We should remark that some tool views are actually documents, displayed outside editor. A good example of document-window is "Output View" in Visual Studio. Such views should be promoted to regular documents and shown in one of text editor tabs. We take only Variables view as an example of such document-like windows, though there are more of them in existing IDEs.

Here we name only the most significant abstract tool views, which compose the tool view model:

- Project view. It is a tree view of categorized project documents. This window is used to navigate through project and interact with documents.
- Files view. It is a tree view of files in file system. This view allows a user to observe the files structure and operate over them.
- Code objects view. It is a tree view of code objects: classes, global functions, global variables and macros. Exact appearance and naming can vary depending on IDE, but all code objects views display class hierarchy as well as class methods and attributes.
- Build results view. Shows a list of build errors and warnings if there were any. Allows a user to navigate to any issue by clicking on it.
- Tasks view. Contains a list of tasks and lets a user navigate from them to text; in most cases, task is a comment from code, marked with a special word, for example "TODO", "FIXME", or "HACK".
- Breakpoint view. Shows a list of all breakpoints, allows a user to switch them on/off, edit their condition and navigate to certain breakpoint.
- Threads view. This window contains a list of debugged application threads. Clicking on a thread results in navigating to its current execution point.
- Call stack view. This view shows the stack of subroutine calls for every program thread while debugging an application. By clicking on any function call a user can open the definition of it in code editor.
- Variables view. This utility window shows a list of variables and expressions monitored while debugging and their current values.

So we have described a model of IDE tool views. This model is sufficient to introduce the single-window design.

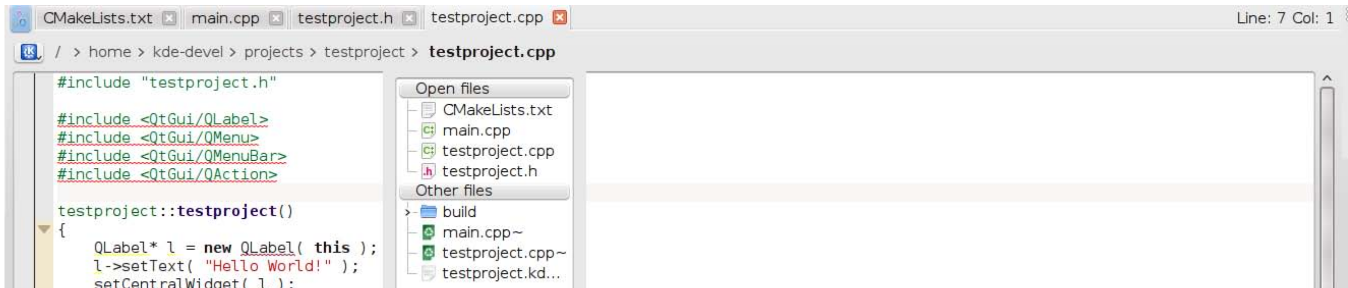
III. Conceptual Single-window Design

This section shows the process and reasoning of creating a single-window IDE design based on tool view model from the previous section.

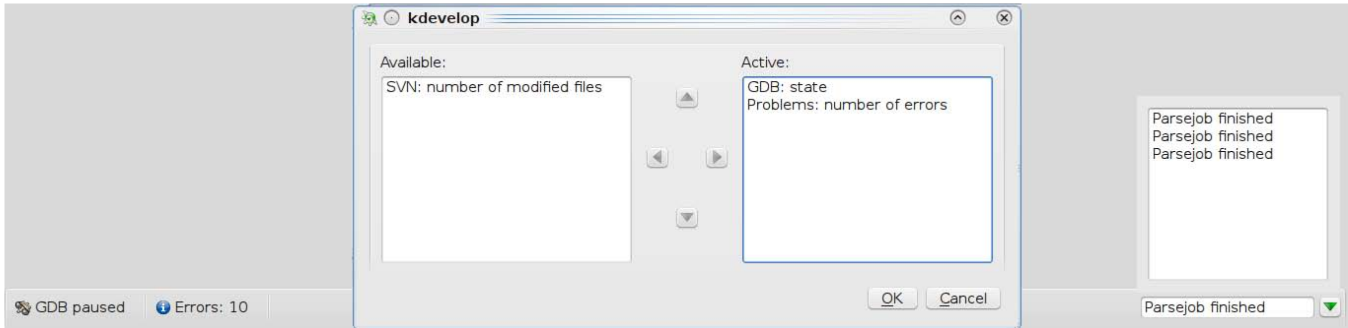
A. *Single-window design: idea*

The idea of this paper is creating a usable graphical interface design for IDE without tool views. Our goal is to reduce the number of open tool views while working with IDE and to carry over the tool views functionality into the text editor.

It is very difficult to meet this goal with the only text editor window. Thus we should introduce several new GUI elements for our single-window design. Those elements address tool view usability problems by (i) taking, when inactive, much less screen space than a tool view (or no space at all) and (ii) appearing when necessary, without explicit user interaction. These visual elements are covered in the next subsection.



Picture 1. Breadcrumbs navigation bar.



Picture 2. Enhanced status bar.

Picture 3. Inline error message.

B. Single-window design: additional widgets

We add widgets into single-window interface or considerably enhance the functionality of existing widgets and use them to achieve the goal from the previous subsection. Each of following paragraphs is dedicated to an additional widget.

Breadcrumbs navigation bar. It is a widget that represents a path in a certain tree to a currently selected object. The most common example is a file system breadcrumbs bar that shows a path to current file or directory. The bar is separated into blocks, each one representing a node in the path. Each block can be decorated with an icon or color and, on mouse click, displays a context menu with the list of elements on the same level of hierarchy. The context menu is used to change the currently selected object or to operate over other objects. The benefit of breadcrumbs lies in the principle of locality [6] [8]: a user tends to work with objects closer to his current work context. For our purposes, we propose that a single breadcrumbs bar supports different modes – that is, different meanings of what a current item mean and what hierarchy is displayed. Mode of breadcrumbs can be conventional (navigation in file system) or advanced (navigation in frame stack or classes). We describe it in details in following subsections. Breadcrumbs navigation for file path is shown on the Picture 1.

Enhanced status bar. Although status bar is a standard part of IDE GUI, it has to be considerably enhanced to meet the needs of single-window design. The enhanced status bar has two parts: static and dynamic. The former part is place for small (not bigger than icon plus a couple of symbols) customizable widgets that are to display information on current IDE state (for instance, number of changed files in VCS working copy). The static area should contain information that user requires occasionally [8]. The dynamic part shows messages about events in the IDE (for example, project build finished). This part shows one message at a time and can display a dropdown list of recent events so a user can choose any of them. Both status bar areas support customizable actions for left click on their parts (static widgets and messages respectively). You can see the enhanced status with several static widgets, dynamic area and setup dialog on the picture 2.

Text editor inserts. We want to show information linked with certain objects (variables, functions, etc.) in program code. Along with traditional ways of presenting information in text editor window (tooltips, context menus) we are going insert widgets between lines. These inline widgets can contain some information and control elements like buttons. Inline widgets are preferable when we want to display something independent on user's action. Good example is showing build errors in text (see picture 3): after build system encounters an error it should be displayed

immediately, not after user hovers mouse over error line or invokes context menu

We come to the discussion of creating single windowed interface from the model created in section II. Now we discuss each of abstract tool views listed in the previous section. We determine their main use cases and show how these tool views can be removed from user workflows.

C. Single-window design: project view

Project view is probably the most common type of utility window. It contains a tree of project files. Exact tree items can vary among IDEs, but the main use case is navigation through existing documents. The project view can be replaced with breadcrumbs bar. Breadcrumbs should display the path to current document, each block is a directory, and the final block is the edited document. In popup display for each block there is a tree view for directory represented by that block. In such way we allow user to navigate through project structure.

D. Single-window design: files view

Files view is very similar to project view in terms of structure and use cases. To replace files view we introduce a new mode for breadcrumbs. In this mode breadcrumbs bar shows a path from the root of file system to the currently edited file. Similarly to previous subsection each block is a directory, except the final one.

E. Single-window design: build results view

Build results window shows a list of errors and warnings. Its main use cases are viewing errors and warnings and showing corresponding source location in the editor. We replace this view functionality by displaying build issues between the lines of text and upgrading breadcrumbs bar.

Concerning errors, it's well known that the first error should be examined first, as other errors can be induced. So, it's reasonable to show the first error as inline widget, and just highlight other errors (for example, with red underlining). The inline widget should contain the text of error message, as well as "Next" and "Previous" to cycle through errors.

Warnings can't be handled like errors: their importance doesn't depend on their order. But displaying all warnings with inline widgets would takes a lot of place and can confuse a user [9]. So, we just underline code lines with warnings and display the warning text (either as inline widget or as a tooltip) only when a user clicks on the line. Also, there should be an action "Ignore this warning forever" because if warning has been shown and wasn't fixed then developer most probably won't ever touch it.

In that way the navigation is carried over to text editor. But what about overview of errors and warnings? We want to handle this use case as well. Thus we mark the blocks of file system breadcrumbs (as well as objects in popup for blocks) with red dots, reflecting that there is an error in corresponding document or that there is a document with an error in the directory.

A user also wants to have information about the number of errors and warnings. We add "Errors number" static widget to status bar. Context menu allows user to enable/disable inline text widgets and breadcrumbs marks. Also there should be a dynamic status bar message about build end with the same functionality. So the build results view has been replaced.

F. Single-window design: code objects view

Code objects view is used to view program structure and go to declaration of a class or any other symbol (variable, function). As shown in the previous subsection breadcrumbs bar can provide navigation through tree structures. So we can add another mode to breadcrumbs or extend the document mode: each document in the tree contains all classes, functions, global variables and macros declared in it. Thus we can remove classes view by replacing it with breadcrumbs navigation.

G. Single-window design: tasks view

Tasks view contains a list of tasks, which are taken from text (from comments like "// TODO" in C/C++). We replace this view similar to build results view. First, we want to carry overview functionality, so we add marks to breadcrumbs indicating that a document has a task inside. Second, we add a several characters-wide inline element after each special word ("FIXME", "TODO", etc.) with buttons for going to next and previous task. And finally we add a static status bar element that displays a number of tasks in project and allows switching on/off breadcrumbs marks and inline widgets.

H. Single-window design: breakpoints view

Breakpoints view contains a list of breakpoints in the project with their attributes: file, line and condition. This utility window is used when a user wants to

- create a new breakpoint,
- view breakpoints and navigate to a random breakpoint,
- edit properties of a certain breakpoint.

We want to cover all three use cases using single-window interface. Creating new breakpoints can easily be transferred to a thin vertical line on the right of text (many IDEs already maintain such operation). Overview of breakpoints can be provided by special marks in the breadcrumbs navigation bar, while it displays path to current file, just like we did it with errors in subsection III-D.

Let's assume that properties editing mostly often occurs after the breakpoint is hit. Then we handle this use case through sending dynamic message to status bar and showing inline widget with breakpoint editable properties. Navigation and global editing of breakpoints are made available through breadcrumbs like build errors (see the previous subsection).

I. Single-window design: thread view and call stack view

Call stack view shows called subroutines for each of program threads. We can replace both these views by organizing following tree structure: the top-level element is a thread and its descendants are function calls in stack. With that said, we introduce a new mode for viewing call stack. Breadcrumbs bar automatically switches to that mode when debugging is paused. User can handle changing modes manually through the first block of breadcrumbs. That's it for call stack view and thread view.

J. Single-window design: variables view

This utility window shows a list of variables and expressions that are watched during debugging process. As

stated before, this view is document-like and should be placed in text editor tabs. A user wants contents of this view during debugging, so we should split the text editor window and display watched variables and expression in parallel with debugged code.

K. Single-window design: resulting interface

In this part of the paper we described a single-window interface design based on IDE tool views model. We have determined only some of single-window interface functional capabilities, used to carry tool views functionality to text editor and additional widgets. Much more functions can be added to that interface to improve usability. For example, contents of context menus for breadcrumbs can be prioritized upon user's selections, but such enhancements are outside of this paper's topic.

Let's proceed to the implementation details of single-window interface.

IV. IMPLEMENTATION DETAILS

The single-window design has been implemented on the base of KDevelop 4 IDE, part of KDE project [10]. The source code of all components of KDevelop is open, which facilitated implementation of desired modifications. Implemented GUI parts can be seen on pictures 1 – 3 above.

A. KDevelop 4 GUI

KDevelop GUI consists of interface parts that have been described in section II. The main window consists of dockable tool views and tabbed text editor view. KDevelop tool views include "File System" view, "Project" view, "Errors" view, "Breakpoints" view. Status bar is almost empty, containing only the progress bar for showing pending operations progress, so this space can be used to implement enhanced status bar. To plug text editor inside the main window KDevelop uses KParts [11], thus allowing a user to edit text with Kate editor [12]. So, single-window interface can be implemented in KDevelop.

B. Implementation

To create single-window GUI in KDevelop the following components have been implemented:

- breadcrumbs navigation bar,
- showing of errors and warnings between lines,
- extended status bar.

Breadcrumbs navigation bar is placed above the code editing window and shows the path to currently edited document. When user clicks any node of the bar, a context menu with file system for that node appears. To speed up access to frequently opened documents the list of currently opened files appears above other files.

Errors and warnings are shown in following way: only the first one appears after the line it refers to. The decision to show only one line has been made because programmer usually tries to fix the first errors as later ones can be caused by the first one. Along with the error information two navigation buttons are shown: "Next" and "Previous". These buttons can take user to neighboring errors.

Enhanced status bar meets the description from section III: it has dynamic and static parts. The static part accepts small widgets for changed files in working copy, number of tasks and number of background parser errors. Presence and order of these widgets can be customized by user. The dynamic part shows messages from removed tool views: that build is started and finished, messages from debugger, background parsing results. Clicking on a message allows user to open the tool view that sent the message.

Here follows a survey of the most popular IDEs.

V. EXISTING IDES

This section examines the sets of tool views and their usability in several popular IDEs [1][5][7]. Along way we note whether any of these IDEs can be used without tool views.

A. Visual Studio

Microsoft Visual Studio [13] is one of the most popular commercial IDEs for developing in the C++ and C# programming languages. Visual Studio has several often used tool views described below.

- "Solution Explorer" – a view displaying the tree of projects and files in current solution. A user operates with this view to create new files, open files (unless they are present in text editor's tabs), and observe the structure of projects.
- "Class View" is a two-part window with a list of classes in the first part and contents of classes (attributes and methods) in the second part. This window is useful for looking through class structure and navigating to their declaration or their methods implementation in code.
- "Output View" – a view containing the results of code build with errors and warnings among them or program execution results. This window is vital for building process because a user views errors and navigates to them with this window.
- "Code Definition Window" – a utility window, browsing the definition for the currently selected object or function. It is used to quickly look at and probably edit the definition of class or function.

Tool views in Visual Studio behave like described in section II and can be put in auto-hide mode, in which they expand only mouse cursor moves to their header and collapse to the screen border when cursor leaves them. That decreases the time user spends with tool views, but user can't remove them permanently: too much information is concentrated in them. For example, user can't open a new file without "Solution Explorer" and can't navigate through errors without the "Output" view. So there's no way to use Visual Studio without tool views.

B. Eclipse

Eclipse [14] is an open-source cross-platform IDE, mainly used for C++ and Java development. Tool views organization in Eclipse is similar to the one in Visual Studio but Eclipse auto-hide mode is less usable than in Visual Studio: it requires an extra click to open a hidden tool view. Also, a user can only toggle auto-hide mode of a certain tool view, not a whole dock area, like in Visual Studio.

Eclipse tool views resemble Visual Studio tool views. "Navigator", "Outline" and "Make targets" windows provide the navigation service through project files, identifiers, and

make targets correspondingly. “Problems” and “Console” views are similar to Visual Studio “Output” view: they allow a user to look at errors and program output and go to errors in code. Thus, tool views in Eclipse play an important role in user’s workflow and cannot be removed without replacing their functions with some other interface elements.

C. *Code::Blocks*

Code::Blocks [15] is an open source cross platform IDE, which is designed for convenient usage with different C++ libraries: GTK+, Qt4, OpenGL, FLTK, wxWidgets, Lightfeather. The user interface of Code::Blocks is less complex (fewer tool views and less their customizability) than one of Visual Studio and Eclipse, but that’s just because of lower number of integrated features. Code::Blocks GUI has several tool views necessary for normal work:

- “Projects” – a window for tree navigation through opened projects. This is counterpart of Visual Studio “Solution Explorer” and Eclipse “Navigator” window.
- “Symbols” – a tree navigation window through functions, classes and global variables. It is similar to Visual Studio “Class View”.
- “Logs” is a tabbed set of several windows: “Search results” (the list of found items), “Build log” (plaintext output of build tool), “Build messages” (a clickable list of errors and warnings), and some others.

Code::Blocks has no auto-hide mode for tool views, and this makes its interface even less suitable for reducing user’s interaction with tool views.

D. *NetBeans*

NetBeans [16] is an open source cross platform IDE, used massively for development on Java platform, but also can be used for some other languages including C++.

Tool views in NetBeans are organized just like in Visual Studio. The most used tool views are:

- “Projects” and “Files” windows – tree views for navigating through projects and file system.
- “Classes” – a window for viewing classes and functions, like “Class View” in Visual Studio and “Symbols” in Code::Blocks.
- “Build” – a window with build results log, allowing a user to move to any error or warning.
- “Navigator” window – a dynamic version of classes window, shows at which place in class and function tree a user’s cursor is situated. This window is useful while browsing through highly nested code.

Tool views in NetBeans can be put in auto-hide mode, called “Minimized” in the NetBeans interface. Unfortunately, a user can change the mode of only one window at a time. Tool views contain information and operations, which are essential for development process (for example, opening a new file through “Files” window), so IDE can’t be used without constant interaction with utility windows. So we can conclude that NetBeans IDE envisages use of tool views in common developer workflows.

VI. CONCLUSION AND FUTURE WORK

This paper has proposed the single-window user interface for IDE to solve the usability problem of tool views. The first result is a model of IDE tool views. The second result of this paper is the conceptual design of single-window IDE interface. Key GUI elements of this interface are:

- breadcrumbs navigation bar,
- enhanced status bar,
- text editor window with inline message display functionality.

The third result of this work is the partial implementation of this single-window design in KDevelop IDE.

Future work involves usability testing [2][6] of single-window interface to find whether usability problems are solved or at least reduced compared to traditional IDE GUI.

REFERENCES

[1] Developpez LLC, “Les meilleurs environnements de developpement” [HTML] (http://general.developpez.com/edi/)

[2] J. Nielsen, “Usability Engineering”, Academic Press, 1993, pp. 23 – 37, 165 – 227.

[3] M. Szymczyk, “Reducing XCode’s Window Clutter”, 2007, [HTML] (http://meandmarkpublishing.blogspot.com/2007/06/reducing-xcodes-window-clutter.html)

[4] Website article: M. Stephens, “10 Things NetBeans Must Do to Survive”, 2003 [HTML] (http://www.softwarereality.com/soapbox/netbeans.jsp)

[5] M. Caron, “Survey on Usability of Integrated Development Environment” [HTML] (http://docs.google.com/present/view?id=addqfjnjc3d6_108gj3w67c3)

[6] Steve Krug, “Don’t Make Me Think A Common Sense Approach to Web Usability”, Indianapolis: New Riders, 2000, pp. 10 – 20, 50 – 96, 138 – 174.

[7] Janel Garvin, “Software Development Platforms - 2009 Rankings”, Evans Data Corporation, 2009, [HTML] (http://www.evansdata.com/reports/viewRelease_download.php?reportID=19)

[8] Morgan Kauffman, “GUI Bloopers 2.0 Common User Interface Design Don’ts and Dos.”, Morgan Kauffman Publishers, 2007, pp. 7 – 51.

[9] Donald A. Norman, “The Design of Everyday Things”, Doubleday, 1989, pp. 187 – 219.

[10] KDE Community, KDevelop website [HTML] (http://www.kdevelop.org/)

[11] Philippe Fremy, “KDE Technology: KParts Components” [HTML] (http://phil.freehackers.org/kde/kpart-techno/kpart-techno.html)

[12] Kate webstite [HTML] (http://kate-editor.org/)

[13] Microsoft, MSDN, Microsoft Visual Studio [HTML] (http://msdn.microsoft.com/ru-ru/vstudio/default.aspx)

[14] Ecliplse Foundation, Eclipse website [HTML] (http://www.eclipse.org/)

[15] Code::Blocks website [HTML] (http://www.codeblocks.org/)

[16] NetBeans website [HTML] (http://netbeans.org/)

Macromodule technology

Victor P. Gergel, Alexey Sidnev
Computational Mathematics and Cybernetics department,
N.I. Lobachevsky State University,
Nizhny Novgorod, Russia
e-mail: alexey.sidnev@itlab.unn.ru

***Abstract*— A development of the parallel, optimal and portable software is a difficult problem. It consists of learning of libraries, programming techniques (such as optimization and paralleling), usage of the libraries and modification of written code. At present there is a set of the optimized libraries for the big number of tasks. But each library is unique, it demands studying and is optimized for specific software and hardware systems. It complicates a choice of the most suitable library (or several libraries) for usage and implementation in the developed software. In this paper, we propose approach for solving the specified problems. The main idea of the approach is the semantic description of programmed code.**

Keywords-macromodule technology; semantic description of code

I. INTRODUCTION

Recently software development has accepted mass character, being hard professional work. Development of parallel programs for high-efficiency computing systems is much more difficult. The main complexity of it consists of learning of libraries, programming techniques (such as optimization and paralleling), and usage of the libraries and modification of written code.

In general, the computer program represents sequence of operations which computer has to execute (Fig. 1).

```
Program Demo
  Action 1
  Action 2
  …
End_of_program
```

Figure 1. The common view of the program

In most cases, separate small operations can be merged in the larger units. In that case the modular representation of program looks like on Fig. 2.

Programs can consist of much number of units. It is possible to select some standard units which describing solutions of some typical tasks among them (an ordering of the data, search of the minimum or maximum value, etc.). These standard units can be developed, placed in libraries of standard units and used. In this case programs look like on Fig. 3.

```
Program Demo
  Unit 1
  Unit 2
  …
End_of_program
```

Figure 2. Modular representation of the program

The similar technology – usage of libraries – is one of the main in a software development. This approach has many advantages. Such as quality implementation of the standard units, essential decrease development coast. At present there is a set of libraries solving tasks of linear algebra (library LAPACK, ATLAS), many-dimensional multicriterion optimization, fast Fourier transform (library FFTW, MKL), etc. There are a number of problems at the library approach. First of all, the programmer has to know a lot of existing libraries. He must know the structure and the rules of usage of the library, because each library is unique. It is difficult to overcome such problems. It is hard to provide portability of programs (work of programs in various hardware and software conditions). In this case, upgrade of programs with replacement of used library units is usually required.

In this paper we propose the technology of macromodular software development which decreases complexity of programming.

```
Program Demo
Unit 1
  Call the unit from library
  Unit 2
  …
End_of_program
```

Figure 3. Modular representation of the program with usage of libraries

II. MACROMODULAR TECHNOLOGY

A. *General Overview*

The main idea of the macromodular approach is the semantic description of programmed code. It is performs description analysis and automatic assembly of optimal

application for target software and a hardware platform (portable, parallel or for the specialized processor). Thus the basis for assembly is debugged and optimized libraries. The software developer does not choice the most suitable library and does not write code to use that library. It is enough to make the macrodescription of a program code and to specify a target platform. Modification of the program under new software and a hardware platform is simplified. It is enough to rebuild the application.

First of all, programmer writes description of code. It consists of action specification, data storage format and so on. Description based on language syntax rules such as preprocessor directives. Preprocessor is a program that processes the code before it passes through the compiler. So it can transform program before actual compilation. After that the programmer selects target software and hardware platform and perform assembly of optimal application.

B. *Usage Example*

As an example we will consider a problem of matrix multiplication [1]. It is popular enough problem. It finds application in problems of a computer graphics, physicists, etc. Matrix multiplication is realized in many libraries (LAPACK, ATLAS, MKL). The example will be considered on the C language.

The implementations of square matrixes multiplication is presented on the Fig. 4. Matrixes A and B are initial matrixes. In matrix C the result of multiplication is stored.

Matrix multiplication is a simple enough task but it is necessary to know:

- Matrixes size.
- Type of matrix elements. It can be simple type (for example, integer type or float type) or complex type (for example, complex number).
- Data storage format. It can be dense representation (in the rows or columns) or sparse (Compressed Row Storage, Compressed Column Storage, Sparse Block Compressed Row Storage, etc) [2].

```
for(i = 0; i < n; i++)
  for(l = 0; l < n; l++)
    for(j = 0; j < n; j++)
      C[i*n+j] += A[i*n+l] * B[l*n+j];
```

Figure 4. Classic matrix multiplication

Macromodular technology uses preprocessor directives for tasks specification, such as matrix multiplication. We will use special directives, named pragmas. Pragmas are the preprocessor instructions, so they are processed before a compilation stage. It often controls actions of the compiler and linker. All unrecognized pragmas are ignored. For example, OpenMP parallel programming model is based on pragmas [3].

Directives of macromodular technology describe the block of code and have the following format: "#pragma mmt <action>(<parameters>)". <Action> is an operation which performs in the specified block, for example matrix multiplication. <Parameters> are input and output arguments of the computing function. Description of matrix multiplication is presented of Fig. 5.

All matrixes in the example are square, have type of elements "float" and have dense representation in memory (so-called style C).

C. *Target Platforms*

We consider CUDA (toolkit version 2.3) [4] and MKL (version 10.0.5.025) [5] libraries. MKL is a library of optimized and threaded math routines for science, engineering, and financial applications that require maximum performance. CUDA is a library for GPU allows developing the programs for nVidia video cards.

Available set of libraries is the basis for select of the target platform. If target system is GPU oriented (has a powerful graphics card) it is more preferable to use CUDA. If target system is CPU oriented (has a powerful processor of Intel) it is more preferable to use MKL.

The operating time of matrix multiplication (type "float") on various platforms is presented on Fig. 6. The first platform has two quad cores Intel Xeon E5320 processors, but very slow integrated video card. The second platform has GPU GeForce 8800 GTS and dual core Intel Core 2 E6650 processor.

On the second platform, performance of matrix multiplication on the GPU is more effectively, than on the processor. First platform with two quad cores processors have shown the best result. On the first platform, GPU does not support general computation. So libraries have various effectiveness on different platforms.

D. *Current Research*

Development of a prototype of the system supporting macromodular technology is at present carried on. A development of the extension for Visual Studio 2008 [6] is now finished. It allows performing preprocessing of a program and a choosing of a target platform.

```
#pragma mmt mmult(A=Matrix<CStyle, float>(n, n), B=Matrix<CStyle,
float >(n, n), C=Matrix<CStyle, float >(n, n))
{
for(i = 0; i < n; i++)
  for(l = 0; l < n; l++)
    for(j = 0; j < n; j++)
      C[i*n+j] += A[i*n+l] * B[l*n+j];
}
```

Figure 5. Macromodule definition of matrix multiplication

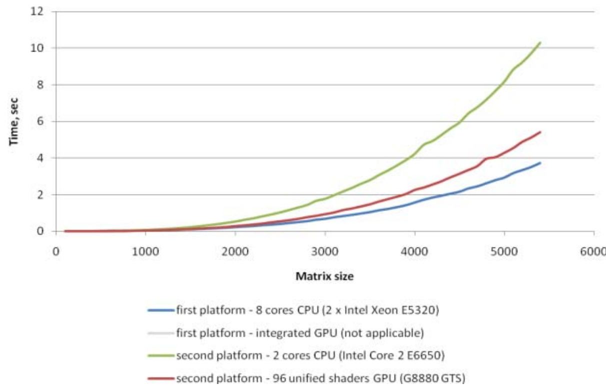


Figure 6. Compare CPU and GPU performance on different platform

III. SUMMARY

The main idea of the macromodular technology is the semantic description of programmed code. It is performs automatic assembly of optimal application for target software and a hardware platform. Thus the basis for assembly is debugged and optimized libraries. The software developer does not choice the most suitable library and does not write code to use that library.

The main ideas of macromodular technology of software development are:

- Standardization of rules of standard unit's usage (the unit name, assignment, the input data and received results). Standard implementation of units is not fixed, so it is defines abstract units. Definition of the abstract unit can be such as on Fig. 7 (on an example of the data ordering). Standardization of abstract units is regulated and it is reported to software developers.
- The extension of standard modules by the available units in libraries (for each unit of library the abstract unit is indicated). Generally, for one abstract unit there can be some implementations in various libraries of standard units (Fig. 8).
- Modular development of programs with usage of abstract units (thereby, the developer should know only definition of abstract units, instead of set of their various implementations).
- Automated choice of implementations of abstract units used in programs. Automation process include the analysis of libraries available in the environment and a choice of the best implementation of abstract units from available set, usage of concrete implementations of abstract units and construction of a final version of programs.

The main advantages of the offered technology are:

- Standardization of the standard units, selected in development process of programs.
- Essential decreasing of practical usage complexity of all set of various implementations of standard units.
- Considerable decreasing of software development complexity – at accumulation of sufficient size of the standardized descriptions of computer data processing.

Appreciable improvement of programs portability between various hardware-software platforms, the localized implementations of programs can have high working speed in the presence of the effective-developed programs.

Unit name: SortData
Purpose: Data ordering
Input data:
- Number of the ordering data
- Initial data

Results:
- Ordered data

Figure 7. Abstract unit definition of a data ordering

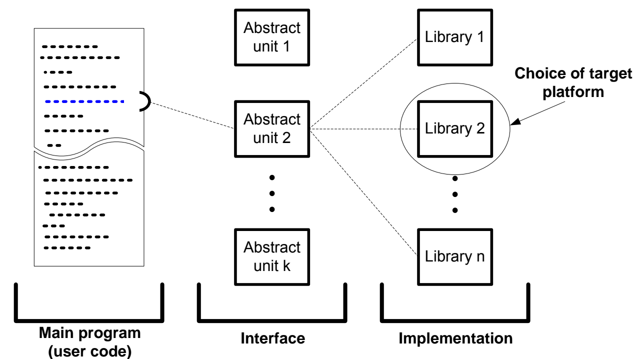


Figure 8. Common view of macromodule technology

REFERENCES

[1] Knuth, D.E. The Art of Computer Programming Volume 2: Seminumerical Algorithms. Addison-Wesley Professional; 3 edition. November 14, 1997. pp. 501.

[2] John R. Gilbert, Cleve Moler and Robert Schreiber. Sparse matrices in MATLAB: Design and Implementation. SIAM Journal on Matrix Analysis and Applications 13 (1), 1992, pp. 333–356.

[3] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, J. McDonald, Parallel Programming in OpenMP. Morgan Kaufmann, 2000.

[4] NVidia CUDA. Reference Manual. Version 2.3. July 2009.

[5] Intel® Math Kernel Library. Reference Manual. September 2007.

[6] Craig S., Marc Y., Brian J. Working with Microsoft® Visual Studio® 2005. Microsoft Press.

On requirements completeness analysis method

Viktoria Gingina
Institute for System Programming
at the Russian Academy of Science
Moscow, Russian Federation
e-mail: vgingina@ispras.ru

Abstract — **Requirements figure prominently in information system development. Both development phase of the system and phase of its verification depends on how qualitative requirements are. That is why it's required to describe requirements as accurately and correctly as it possible. One of the properties that define quality of requirements collecting is completeness. The paper shows that if one obtains sources and symptoms of the requirements incompleteness (completeness absence), classifies, generalizes and clarifies them then one can check requirements for these symptoms while collecting requirements or after that. This will sufficiently decrease the incompleteness of the requirements and thus improve their quality. The paper contains some symptoms of incompleteness have already been revealed and explains the reason of their appearance. These symptoms have been revealed by analyzing the documentation of some important industrial projects.**

Requirements, completeness, incompleteness, incompleteness sources, incompleteness symptoms

I. Introduction

There is the following definition of the term "requirement" in IEEE Standard Glossary of Software Engineering Terminology (1990) [1]:

(1) A condition or capability needed by a user to solve a problem or achieve an objective.

(2) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents.

(3) A documented representation of a condition or capability as in (1) or (2).

The process of developing any information system begins with the phase of requirements collecting. Requirements are used on numerous occasions while developing the system. That is why there should be high quality criteria for requirements.

Requirements are used when modeling the system. They informs about what the system should do, what resources it can rely on, what constraints it should conform to and so on. Any changes and additions to the being developed system architecture is a very hard laborious process especially for the large divaricated one. It causes excessive complication and probably transformation of the architecture in a whole. Thus risks quantity is increasing and developing time is rising. The problem is apparent especially when interconnecting several systems: incorrect requirements can cause their incompatibility.

Requirements are also used to verify if the developed system corresponds to what it's expected to be. Check-on conformance with the requirements is used. If tests used for the verification are not qualitative the testing process will take longer than it was planned. To create good test testers will have to clarify information with the help of analysts and developers and besides the opinion of these experts could be different in points that are incorrectly described in the requirements. Thus the quantity of the failed tests will arise. The results are: low-quality product, broken date of performance, increased cost. And if there is lack of means system development could be paused.

When using low-quality documentation risk rises.

Thus good well-stated, i.e. high-quality requirements are needed. Such requirements should meet following criteria [2]:

Adequacy – requirements meet customer wishes.

Unambiguity – different domain experts understand requirements equally.

Consistency – requirements are exhaustively formulated

Completeness – any situation has its own description (usually general) in the requirements. This paper is devoted to completeness achievement.

II. Requirements completeness

Requirement completeness criterion can be separated into two constituent parts [2]. Complete requirements should describe:

- firstly everything that customer wants to get of the system;

- secondly system behavior in any logically possible situations.

The first aspect is a task for customer analyst. But in the second aspect completeness analysis should be carried out by the requirements completeness expert.

K. Wiegers said that completed requirements give many priorities [3]:

- restriction of rework and redesign when developing,
- project risk decrease,
- system interoperability rise,
- more complete requirements gives fewer mistakes are founded while testing (functional, integration, etc.)

The problem of requirements completeness is not new and is usually considered in respect to rules of good and accurate requirements collecting: ways to interview customer and concerned persons, assessment of user-groups needs, questions that analyst must answer when collecting requirements of different type. But even if all these recommendations are executed requirements often are not complete. Why so?

K. Wiegers notes [3]: "Many software problems arise from shortcomings in the ways that people gather, document, agree on, and modify the product's requirements. <…> the problem areas might include informal information gathering, implied functionality, erroneous or uncommunicated assumptions, inadequately defined requirements, and a casual change process".

There are different reasons of incompleteness. One of them is a human factor. Since requirements are collecting by a human they just can't be fully considered. A commonplace example: there are no constraints for calculation in a computing system. A person who collects requirements (system analyst) believes that some situations are an axiom understandable for everybody so there is no need to describe it in detail – this is another often problem. "System behavior is obvious, usual and do not need any additional explanation" – that is why ambiguous and not evident moments are often not described in requirements. We can also single out moments of "premeditated incompleteness". This is a situation when customers didn't get an agreement or a situation when customer wants to give carte blanche to developers or a situation when system behavior is hard to foretell (impartial non-determinism). K. Wiegers said in his book [4]: "Requirements are never finished or complete. There is no way to know for certain that you haven't overlooked some requirement, and there will always be some requirements that the analyst won't feel it is necessary to record".

In spite of this it is feasible to decrease requirement incompleteness as far as it possible.

Obtaining above-mentioned incompleteness sources gives some understanding of how to find it in requirements. As a result of the research it's planned to get some check-list consists of check-questions that help to define incompleteness. And it's planned to get a set of patterns and anti-patterns that describe situations able to be incomplete. If there is "computes <…>" expression and there is no a precision of computation in requirements this is an example of anti-pattern in requirements description. Element of such lists and sets is not an exact indicator of incompleteness but it's an indicator for a potential place of requirements where description of system property or function is able to be incomplete. Usage of such lists and sets will help to find vulnerable moments in requirements and to define possible but not described situations.

Thus there is a problem to define and to classify possible sources of requirements incompleteness.

Attempts to reveal requirements incompleteness has been already done by the other researches. So it was suggested to describe system behavior in all possible conditions and if there is requirements for situation "A" to define what is happening in all "not A" cases [5]. Also it was suggested to consider requirements in respect to actor (what actor is responsible for what function), to describe all alternative action flows and to justify any requirement [6]. Another method recommends checking if requirements are for all system elements [7]. From the point of view of incompleteness source obtain method such approaches are quite one-sided. Method suggested in this paper includes above-mentioned approaches but considers them as special cases of incompleteness that are not enough. Obtaining incompleteness sources allows to research the problem more profoundly and to discover more symptoms of incompleteness and thus to assess requirements completeness more correctly.

III. Requirements incompleteness sources

Technical documentation for a few industrial systems has been analyzed. This allowed to obtain some symptoms of requirements incompleteness. Considered systems are developed within the framework of large project. This fact vividly demonstrates that requirements incompleteness is critical for implementation and is usual even for quite a good documentation.

In our examples symptoms of incompleteness are expressed as anti-patterns. These are situations that are mostly frequent for the considered projects.

(1) There is a condition-element but not all flow-branches are described.

If there is a description for successful work of some function there should be a description for an erroneous situation. If there is description for "then" condition there should be a description for "else" ("otherwise"). Similarly if there is function description for a set of parameter values there should be a description for any other possible values. E.g. if it's settled that there is some action flow for positive values of real-type parameter then it doesn't mean that nothing happen for zero and negative values of this parameter. Probably author of the requirements has such a behavior in his mind but in that case he should explain and describe it clearly. Otherwise we have incompleteness in the requirements.

So in the LSB specification [8] there was not a description for g_date_clamp function [9] behavior in a case when "date" parameter value was in range between "min_date" and "max_date" parameters. In the same specification there was g_main_context_iteration function [10] description only for the situation when "may_block"

parameter was "TRUE". Situation with "FALSE" value has been omitted.

Special cases of such incompleteness are function requirements that have no description for 0 or NULL values of function parameters. Also there should be clear description for float parameters in the case of Nan and Inf values.

For example in the documentation for a huge industry system S critical defects have been found: a behavior of system interface functions was not described for the case when these functions got faulty incoming data (0 or NULL) instead of file pointer. In SUS 3.0 [11] in the description of ualarm() function [12] there was not requirement for the function behavior when “useconds” parameter is 0.

(2) Changing the data is described in one action branch but there is no any description for the same data in the other branch.

Probably it doesn’t change but the absence of the clear description indicates incompleteness. Some parameter, object pointer, picture on a web-form, content of a file, everything function can affects on - that is what we consider as data in this situation. Values of all these elements form system state. Complete requirements should describe how every function influences to these data elements in all action flows (or there should be a clear instruction that nothing is changed). Furthermore it’s required to note an indirect influence of the sub-functions.

In the above-mentioned system S interface there is a parameter ERROR_ID that gets an identifier of the error took place when using the function. But earlier there were not instructions for correct function processing. An assumption that value was not changed founded to be wrong. In reality the parameter got a “noerror” value.

(3) A new function, type, object, term is used but never described.

For the first view such an omission can look absurd but it quite often occurs. In well-formulated requirements you can suddenly find a link to some function, parameter, data element that is described or explained nowhere in the documentation. For example in the LSB specification [13] in the svcudp_create() function [14] description it was noted that this function was called similarly to svcudp_bufcreate(sock, SZ, SZ) function call. But svcudp_bufcreate() function was never described in the specification. The reason of this incompleteness can be the uncoordinated documentation writing and changing. An ordinary misprint can take place too. And there is a possibility that the function just has been forgotten to describe. Obviously such an error is peculiar to divaricated systems because for a requirements writer it’s more difficult to imagine complex system in a whole. If there are more than one writer the problem will be interconnection between them.

IV. Sources of analyzed documentation

Data analyzed to find incompleteness symptoms is documentation on three industrial systems.

Examples of the defects found in Linux specifications requirements got from the official information of Verification Center of the Operating System Linux [15]. The Center is based at the Institute for System Programming of the Russian Academy of Sciences (ISP RAS) [16]. It integrates a group of projects of developing open source tests and automated verification techniques for Linux-systems. The Center is supported by the Russian Federal Agency [17] for Science and Innovations, by the international consortium The Linux Foundation [18].

Linux Verification Center works on checking that Linux implementations are conform to requirements and specifications. There is information of testing results and found inequalities on Center site. There are many problem reports marked as “incompleteness” among them. As LinuxTesting.org documentation is officially published on their Internet page there is a possibility to show incompleteness symptoms using some real examples of the project.

Other two projects are commercial, closed and do not publish such an information. Internal documents being used for analysis are test-cases and defects reports registered in bug-tracking system in one case and test report documents in the other case.

V. Conclusion

This paper is devoted to a necessity of qualitative description of the system requirements. The paper discloses an importance of the requirements incompleteness and shows its critical influence. Some incompleteness symptoms are obtained and obvious examples of incompleteness demonstration are provided for the huge industrial systems. Obtaining incompleteness symptoms allows improving the project documentation that will give an opportunity to avoid problems when implementing and testing system. It’s planned to continue documentation analysis for different projects to find and classify other incompleteness sources and to get check-lists, patterns and anti-patterns sets. Also it’s planned to examine possibility of incompleteness sources obtain by formalization of requirements collecting process, by requirement modeling, by specification of verification tests and by analyzing results of static and dynamic implementation analysis.

References

[1] http://standards.ieee.org/reading/ieee/std_public/description/se/610.12-1990_desc.html

[2] V.Kuliamin, N.Pakulin, O.Petrenko, A.Sortov, A.Khoroshilov, Requirements formalization on practice, Preprint 13, ISP RAS, Moscow, 2006 (in Russian)

[3] K.Wiegers, Software Requirements: Practical Techniques for Gathering and Managing Requirements Throughout the Product Development Cycle, 2nd edition, Microsoft Press, Redmond, Wash., 2003

[4] K. Wiegers, More About Software Requirements: Thorny Issues and Practical Advice, Microsoft Press, Redmond, Wash., 2006

[5] R. S. Carson, Requirements Completeness: A Deterministic Approach, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.8.735

[6] A. Sarkar, Requirement Management in Testing, Infosys Technologies Limited, Bangalore, unpublished

[7] S. Robertson, J. Robertson, "Mastering the Requirements Process Second Edition", Addison Wesley Professional, 2006

[8] Linux Standard Base Desktop Specification 3.1, Chapter 12. Libraries, 12.2 Interfaces for libglib-2.

[9] http://www.gtk.org/api/2.6/glib/glib-Date-and-Time-Functions.html#g-date-clamp

[10] http://www.gtk.org/api/2.6/glib/glib-The-Main-Event-Loop.html#g-main-context-iteration

[11] The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004 Edition (SUS 3.0), System Interfaces

[12] http://www.opengroup.org/onlinepubs/000095399

[13] Linux Standard Base Core Specification 3.1, Chapter 13. Base Libraries, 13.5. Interface Definitions for libc

[14] http://refspecs.freestandards.org/LSB_3.1.0/LSB-Core-generic/LSB-Core-generic/baselib-svcudp-create-3.html

[15] http://linuxtesting.ru

[16] http://www.ispras.ru

[17] http://www.fasi.gov.ru

[18] http://www.linuxfoundation.org

An approach to data validation based on lifecycle-bounded metadata

Vadim Surpin

The Institute for information transmission problems of the Russian Academy of Sciences (Kharkevich Institute),

Moscow, Russia

vadim@iitp.ru

Abstract – **Data validation is known to be the task performed in almost every business application and occurs through almost all levels of modern multi-tier application. Being a crosscutting concern validation requires some extra effort to make sure that it works right and consistent in all tiers thus increasing time to test the application and possible number of application bugs. The article describes an approach to describe complex lifecycle-bounded validation in a declarative manner making it reusable through the application.**

Data validation is one of the most common tasks in business application. Validation is a process of checking that data conforms to constraints applied to it and producing a list of validation messages that clearly describe failed checks. Problem of data validation attracts more attention since the time when Model-View-Controller (MVC) [1] become a standard for building modern application architecture. Such layered code separation allows to build more secure and scalable software but leads to undesirable code duplication between layers. The validation code is that one that being duplicated. It is being used in all application layers from model to presentation making a problem to coordinate validation rules on different levels.

Last few years there are several attempts to make data validation some more formal description and establish a standard that describes data validation approaches. There are several standards one of which is JSR303 [2] if we look in the area of Java technology [3]. The standard has a working implementation, the one that was a prototype for the standard. It is open-source project Hibernate Validator [4] from the Red Hat company. High page rank of the project official website in Google shows interest of software developers to the data validation problem. Here are some data validation approaches referenced in the JSR303 standard and implemented in Hibernate Validator project.

The standard based on idea of applying constraints to object's fields and offers to look at that constraints as metadata bound to thy fields. The most common constraints such as "not empty", "not large than N symbols", etc. are supplied as out-of-the-box implementation. Software developer may add some more complex field constraints that conforms to the standard. Every constraint may be associated with one or more groups that allows to validate object against several validation sets.

The approach described is suitable to accomplish simple validation tasks when validation constraint set isn't vary very much. Such as in the case of validating domain model objects before they're being send to persistence layer and database. This peculiarity is due to tight integration between Hibernate Validator that was the prototype for the standard and Hibernate [5] object-relation mapping solution. Being good at that field the standard doesn't address validation issues that exist in more complex workflow-based scenarios where typical tasks are:

- Constraints on fields that depend on each other
- Constraints on associated objects or object graph
- Constraints that depend on the lifecycle stage of the business object
- Constraints that depend on context parameters

To deal with first and second problems it is enough to allow object level validation and give a developer possibility to implement validation logic as a program code. This will give also an opportunity to validate complex dependencies between object fields and deep relationship between associated objects.

The lifecycle dependent validation is a common case in application where two or more users work on the same data. In such a case data travel from one user to another in accordance with application workflow, the data contained in the same business objects grows along it's way in workflow so it's consistency depends on the phase of the lifecycle. This is the most common case for every quite complex business application. To address this issue clear principles of business object lifecycle management should be described and implemented in an application architecture.

The MVC architecture states that at least three general classes of objects exist:

- Data access objects or domain-model objects (Model)
- Business logic objects (Controller)
- User interfaces objects (View)

From the perspective of high level system architecture interaction between these classes may be presented as on the UML[6] diagram Fig. 1

The diagram shows that all data changes only when it passes throw the methods of controller object that implements business operations of the system. This means that all object lifecycle-management occurs when data goes across the border between View and Controller layers where the View layer initiates object state change based on the user request and controller performs the requested business operation. That's why passing the boundaries between View and Controller layers is a good place to perform object validation. It solves at least three problems from the validation field:

1. Assure that controller receives correct data that won't corrupt data storage integrity if malicious data will be send by the user.
2. View layer is able to show validation messages informing the user about mistakes in just entered data in context of the requested operation thus giving a developer to supply more specific and clear validation message compared to ones that can be produced without this operation-context dependency.
3. Forces consistency between data check on View and controller layers by using the Don't Repeat Yourself (DRY) [7] principle eliminating code duplication.

The approach suits well to the modern application architecture where system modules have to be terminated by well defined interfaces and the module implementation is a "black box" for the cooperating party. An amount of modern programming languages have a notation of interface in their syntax, e.g. Java language interface syntax may look as follows:

```
@Remote public interface
BusinessOperationsRemote {

        void doSomething(T param);

}
```

The example of a simple business component interface that uses Enterprise JavaBeans 3 (EJB3)[8] technology is shown. Here are it's

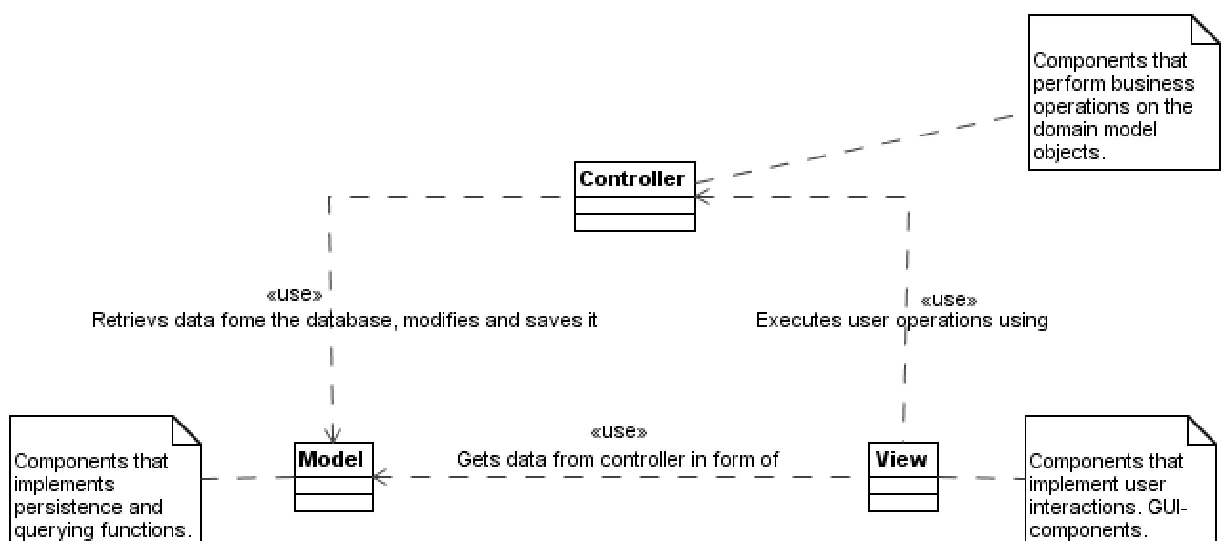


Fig. 1 Interaction between MVC objects

meaningful parts:

1. @Remote – so called “annotation”, an implementation of metadata facility from Java technology. The annotation means that the annotated interface belongs to a business object which lifecycle is managed by an EJB container and the interface methods are accessible remotely by network calls.
2. void doSomething(T param) – business method signature that states the method returns no result and accepts a parameter of type T.

When the component implementation is accessible only to the container which manages that component, it’s interface is visible to both component implementation and client from the view layer which calls business its methods. Keeping that in mind it becomes clear that the interfaces are a proper place to put method parameters validation metadata. The metadata take a form of @Validator annotation on the doSomething() method as follows:

```
@Validator(implementation=DoSomethingValidator.class)

void doSomething(T param);
```

Referenced by the “implementation” attribute class DoSomethingValidator implements the doSomething() method parameters validation logic:

```
public class DoSomethingValidator implements InputValidator {

	@Override

	public List<ValidationMessage> validate(Object… params) {

	…

	}

}
```

This class bytecode should be available both on controller and view layers and can be shipped with business interface description in the same deployment unit. Having access to the BusinessOperationsRemote business interface view layer can simply invoke parameter validation just before the method call:

```
List<ValidationMessage> messages = validator.validate(BusinessOperationsRemote.class, “doSomething”, param);

if(messages == null || messages.isEmpty()) {

	businessOperationsRemote.doSomething(param);

} else {

	//Show validation messages to the user

}
```

The solution described easy integrates with a JSR303 standard-compliant validation using it’s feature to set up validation groups for each object property. These groups may be just fully qualified names of BusinessOperationsRemote interface methods. Such choice of group naming has an advantage of hiding internal object lifecycle from the interfaces client describing transitions between lifecycle phases only in terms of business operation invocations. This gives view layer object only the required knowledge about object lifecycle and removes the need to specify validation groups at view layer. Spreading such information between layers causes numerous errors due to module miscoordintaion during system development process. Miscoordintaion is impossible when using interface level metadata because it can be detected on the compilation stage by compiler error messages.

So the approach described solves the problem of complex object validation in the process of object lifecycle transition process in the way clear to the developer. It coordinates check being performed at differed layers of a multi-tier application and refactoring-friendly since it describes all it’s metadata using language syntax available to the compiler. These simplifies software development that involves data validation facilities (a great part of modern software) and decreases number of hardly testable logical errors in the application design that occur when validation code at different layers gets miscoordinated.

REFERENCES

1. Design Patterns: Model-View-Controller, Java Blueprints, http://java.sun.com/blueprints/patterns/MVC.html
2. Java Specification Request, JSR303, http://jcp.org/en/jsr/detail?id=303
3. Java Technology, http://java.sun.com
4. Hibernate Validator Project, https://www.hibernate.org/412.html

5. Hibernate ORM,
   https://www.hibernate.org/344.html
6. M. Fowler “UML Distilled: A brief guide to the standard object modeling language”, Addison-Wesley Professional, 2003.
7. William Crawford, Jonathan Kaplan “J2EE Design Patterns”, O'Reilly Media, 2003
8. Enterprise JavaBeans Technology,
   http://java.sun.com/products/ejb/

A GA-based approach for test generation for automata-based programs

Andrey Zakonov, Oleg Stepanov (research supervisor), Anatoly Shalyto (research supervisor)
Fac. of Information Technologies and Programming
St. Petersburg State University of Information Technologies, Mechanics and Optics
Saint-Petersburg, Russia
e-mail: andrew.zakonov@gmail.com, oleg.stepanov@gmail.com, shalyto@mail.ifmo.ru

Abstract—**Automata-based approach is often used for developing complex systems. Model Checking is commonly used to check conformance of the system against its specification. However, verification techniques don't allow checking the system in whole, as system consists of not only the model, but also control objects, which are not suitable for model checking. In this paper we propose an approach for testing of automata-based programs. We use EFSM and contracts to extend model with specification requirements and we demonstrate how genetic algorithms could be used to automate generation of tests to find faults in the system in whole.**

Keywords- Automata; EFSM; Testing; Genetic algorithms

I. INTRODUCTION

Automata-based program consists of a finite state machine or any other (often more complicated) formal automata and number control objects, which the model interacts with [1]. The most commonly used technique for verifying automata-based programs is Model Checking [2] because it can be used with very high degree of automation. However Model Checking suits only for verification of the automata, but not the system in whole. Controlled objects behavior and interaction of the automata and their controlled objects are not checked in this approach. Therefore there could be undetected errors left in the automata-based system, even if the automata itself was successfully verified against its specification.

In this paper we propose to use testing to check the automata-based system in whole. Software testing is normally a labor intensive and very expensive task. It accounts for about half of a typical software project life cycle [3]. This means that straightforward approach to testing, such as manual testing, is not the best option. Recently there has been much interest in automated test data generation [4]. Even though testing cannot guarantee the correctness of a program, large number of tests does contribute significantly to the identification and reduction of faults, improving the likelihood that the software implementation will succeed. Therefore this paper includes description of an approach for testing automata-based programs and a way to automate this process using genetic algorithms.

We propose to use testing to check implementation conformance against its specification. Specification given in natural language is suitable only for manual testing. In order to automate testing process, specification must be presented in some formal way. In our approach we include as much specification as it's possible in the automaton, so it would contain the instruments of its own verification. Finite state machines (FSMs) are commonly used for the purpose of automata description. However, a FSM can only model simple reaction of the system to its input events; variables and guard conditions on transitions are needed in order to model a system with complex behavior and data dependencies. Using extended finite state machines (EFSMs), which support variables and guard conditions, is a reasonable choice to describe some of the specification requirements in the automata. As it was proposed in [5] we use contracts [6] to include even more specification requirements in the automata. Having specification requirements included in the program makes it possible to automate checking of these requirements while test is executed. Moreover requirements can be used to aim test generation at detecting situations, when they are not fulfilled.

There is one more reason to use EFSMs. Most programs are designed to interact with some environment: program receives events and input data; automata react to these events and produce some output data. In automata-based programs one uses controlled objects for this purpose: they receive events and provide input data, which can be used in automata in guard conditions on transitions or as other control object functions' arguments. In EFSMs such data are represented as variables. Variable in EFSM could be internal, defined inside the EFSM itself, or external – received from the control object. During testing values of external variables can be provided by the test script.

Considering the proposed description of model and its specification, we defined a test for automata program as a sequence of events and a set of external variables, which lead to specific sequence of transitions (*transition path*) of the automata. As opposed to the traditional approach, where test is a program code, we propose to describe automata test as a transition path, which is much closer to specification level and helps to shorten the gap between the specification and the implementation. Transition path which is interesting for test creation can be easily obtained from natural language specification, but to create test code we need to find sequence of events and set of variables, that would lead to the given path execution. Obtaining sequence of events for the path is straightforward. However set of external variables

is not so easy to guess: one need to find set of values, which would satisfy all transition guards on the given transition path. We propose to apply genetic algorithms to find suitable values for external variables.

Overall, this paper addresses number of problems:

- propose an approach for testing automata-based programs;
- automate test creation by providing a tool, which finds suitable sequence of events and set of external variables for a given transition path and generates test code;
- automate validation of specification requirements, included in the automata, while executing tests;
- attempt to generate tests that lead to violation of specification requirements and so reveal faults in implementation.

The rest of the paper is organized as follows. Section II gives details on proposed approach for testing automata-based programs. Section III describes genetic algorithm applied to find external variables' values. Section IV tells about proof-of-concept tool being developed and preliminary results; Section V concludes.

II. Testing for automata-based programs

The following approach for developing automata-based programs and creating test suites is proposed in this paper:

1) During development include signifigant part of natural language specification in the automata, using EFSM variables, transition guards and contracts.

Controlled objects also have specification and requirements for their inputs/outputs and interaction with the automata. All this specification requirements must be fulfilled during tests execution. Benefit of having controlled object specification included in the automata is that actual implementation of this controlled object becomes less significant for testing. Given the requirements for the object's output, we can check, that automaton reacts well for any data that fulfils given requirements. And vice versa it's acceptable if program fails for the data, which don't fulfill the object's specification.

In our approach we use JML specification language [9] to enrich automata with specification requirements. JML is a design by contract approach and contracts in JML include preconditions, postconditions, and invariants. In our case, such contracts can be defined for automata states and transitions.

2) From natural language specification select interesting scenarious for testing and present them as a sequence of transitions in the automata.

We consider sequence of automata transitions (transition path in the automata) to be a convenient way to describe a test scenario, as this representation of test could be easily derived from a natural language description of a test scenario.

There is number of researches available [7], [8] that addresses the problem of finding transition paths in EFSM to achieve selected coverage criteria (e.g. state or transition coverage in the EFSM). Such techniques can be successfully used together with manual test paths selection and, combined with the approach presented in this paper, could help to automate producing of valuable test suites.

3) Find sequence of events and values of external variables, which would make automata program to execute the disired transition path.

Automaton reacts to the events and perform transitions depending on the values of external variables used in transition guards. Representation of a test as a sequence of events and values of external variables is convinient to programmatically generate test code, but it has very little sense for a developer who works with specification defined in natural language. In our approach developer can describe test scenario in natural language first and then write it down in automata terms as a sequence of transitions, which is straightforward.

We propose an algorithm to automate search for the corresponding sequence of events and set external variables to execute given transition path. There are number of requirements that these variables must meet. First of all the guard conditions on the specified transitions should be carried out. In the second place, all the control object requirements should be fulfilled, because in production use these external variables would be obtained from control objects with given specifications. Optimization algorithms have proven to be efficient for such class of problems [4]. We apply genetic algorithms to solve this search problem. Details on genetic algorithm are described in Section III.

4) Execute generated tests and check filfullment of specification requirements for this tests exectuion.

Test code, which can execute the desired sequence of transitions is useful to perform a runtime check of all the contracts included in the automata. Support of contracts is enough to include most of the specification requirements in the automata and to check them during the tests execution. Specifications written in JML, which we use as contracts in our approach, are annotations for Java code and there are number of tools [10] that are designed to check JML contracts in the runtime or for static check.

Tests that fulfill all the specification requirements doesn't reveal any errors in the program, but still are useful for regression and stress tests. However it is much more important to generate tests, which fail any of the specification requirements for the correct set of external variables and therefore reveal inconsistency between implementation and given specification.

5) Try to find set of external variables which filfulls all guards and control object requirements and fails specification requirements of the program.

To obtain such values we also use genetic algorithm with more sophisticated fitness function, which takes into account not only transition guards and control object requirements, but also all the specification contracts defined for the given path in the automata.

III. AUTOMATIZATION OF TEST DATA GENERATION

A. Optimization problem

Set of external variables can be represented as a vector of values $<x_1, x_2, \ldots, x_n>$, where x_i is an external variable, and n is number of external variables required for this transition path. Fitness function takes this vector as an argument and returns fitness value for an external variables set. The smaller fitness value is the better the proposed vector suits the given transition path. From this point of view task can be considered as a minimization problem, where we look for the set of variables with the minimum fitness value.

B. Candidate encoding

Candidate is a vector of values, as defined above. We use one-point crossover operator, which operates by choosing a random position in the vector, and then new candidate is composed of first candidate's sub-vector before that position and second candidate's sub-vector after that position.

Mutation operator replaces random position of the vector to a new random value.

C. Fitness function

Fitness function aims to provide metric for candidates, which tells how good is this candidate for a specified task. In our case task is to execute given sequence of transitions in the automaton. There is no unambiguous answer for the question of what fitness function to choose.

Approaches for testing of structured programs propose to use such criteria as branch distance [11] for fitness calculation. A branch distance is a measure of how close a particular candidate is to executing the target branch that is missed e.g., $|A-B|$ is the branch distance for the predicate $(A > B)$. The lower $|A-B|$ is the closer is A to B and the closer the candidate is to filfulling the condition. For the filfulled condition branch distance equals zero. There are researches [11], [12] that show effectiveness of described approach for structured programs testing.

In [7] branch distance based approach is used to find input test data that can cause a feasible path in an EFSM model to be traversed. In our research we extend this approach to apply it to automa-based systems. As it was described above, we must take into account not a standalone EFSM, but an automa-based program enriched with system's and control objects' specification. Moreover we aim to find set of variables not only to execute selected path, but to filfull control objects' requirements and ideally to reveal inadequacy of implementation and specification.

To obtain variable values to execute given path there are two types of conditions that should be taken into the account:

- guard conditions on the transitions of the automaton;
- specification requirements of conrolled objects that provide external variables.

These conditions are obligatory to be filfulled. Candidate that fail any of these conditions are not appropriate for test generation, as specification doesn't require system to support such inputs. So in this case fitness function should estimate how close this particular candidate was to filfulling failed conditions.

To give an accurate estimation we examine each state and transition between states on the given path separately. Every transition has the event, which enables it and may have a guard condition and an action section. In the current implementation external variables are introduced in transitions' action sections.

Control objects' specification can be included in transition contracts: preconditions and postconditions. Precoditions verify, that automaton is in correct state to use controlled object; postconditions verify, that external variable value retrieved from the controlled object meets specification requirements.

From this point of view execution of each transition in the path is divided into three small steps:

- receive event, find transition and check guards;
- check preconditions and execute the transition;
- check transition postconditions.

Each of these steps contains coditions that can be failed. Therefore for each of these steps we calculate branch distance. Fitness value for a single transition is calculated as sum of steps' branch distances.

It's important to realize that transitions are executed sequentially. This means that to achieve second transition candidate must successfully complete first one. Therefore transitions in the beginning of the path are somehow more important then transitions in the end. This fact should be taken into the account in the fitness function calculcation. In [7], [12] transition approach level metric is introduced to handle this situation.

For more accurate fitness value we consider step approach level. In such approach each step is assigned a weight value, which depends on the step's position in the path. Last step weight is the smallest, first step weight is the greatest. Overall fitness of the candidate for the given path is calculated as sum of steps' fitness multiplied by their weights.

D. Specification requirements in fitness function

Fitness function described above is aimed to find set of variables that would make possible given path execution. More desirable is to find a candidate, which reveals an inconsistesy between implementation and specification. For this purpose we need take into consideration specification requirements of the system represented as contracts that must be filfulled during the execution. We aim to fail any of these conditions, while guards and controlled objects' requirements are filfulled.

Such task requires iterated approach, as we need to select specific transition, which conditions we want to fail. For example, if we want any of the conditions on the second transition to be failed, we need all the conditions of the first transition to be filfulled, because there may be a dependency between these conditions. For different transitions selected as target fitness function is computed differently. Generally, if k^{th} transition is a target to fail some condition, then all conditions of the transitions with indexes less then k must be fulfilled.

In attempt to fail some conditions we use branch distance turned inside out. If condition is failed then value is zero.

The closer the candidate is to failing the condition the lower the value. This reversed branch distance value is included in path fitness value calculation, similar to common step fitness, described above.

We aim to reveal faults at any transition so we iterate through the given path. At the first step we consider transition path of one transition, the first one. We perform fixed number of attempts to reveal a fault. If any found, test is generated. After fixed number of attempts we move to the next step: consider path of two transitions. We go on like this till we reach the whole given path length.

Finally, after all the iterations are done, for all revealed faults test code is generated, which can be executed separately and used for debugging and bug fixing.

IV. CASE STUDY

In this paper we present a case study that we used in our research. A proof-of-concept tool is being developed during the research. Version of the tool used for the case study contained number of limitations: only integer variable types are supported and current version is capable of providing set of variables to execute given path, but not to reveal faults.

We made up an example of specification for ATM machine and developed an automa-based system for this specification to illustrate our approach.

Sample specification of an ATM machine:

- system must perform withdrawal operations from the specified account on user requests;
- initial amount of money on the account is being retrieved from the bank at the start up. Amount must be a positive number, less or equal to 1000000;
- each time user inputs amount of money on the keyboard a transaction must be initiated. Amount must be greater then 1000 and less then 5000. If wrong input is done user must be notified about an error and operation of the system must be stopped;
- transaction must be successfully completed if after transaction there would be a positive amount of money left on the account. Otherwise transaction must be rollbacked and user must be notified about an error and operation of the system must be stopped;
- while no error occurs user can make withdrawals unlimited number of times.

For the described ATM system it is convenient to introduce number of states: initialization, user input, withdrawal operation, error in entered amount, error during the withdrawal. FSM for this system is presented on Fig. 1:

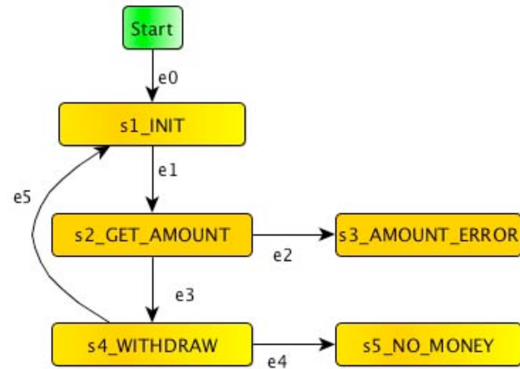


Figure 1. FSM for the ATM system.

Such model contains only basic requirements of the specification. To test such system one would need to examine specification in natural language and write tests manually.

We propose to use EFSM and to include as much specification as possible to the model. Such EFSM is presented on Fig. 2:

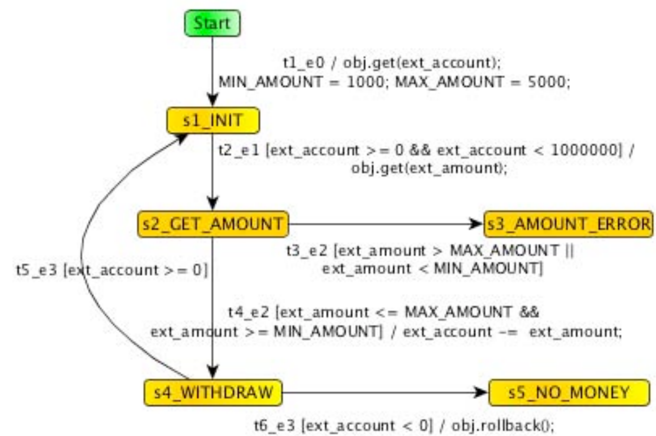


Figure 2. EFSM model of the ATM machine

Model looks more complicated this time, but on the other hand now it contains all the specification requirements, that were described in natural language. Major advantage of such representation is that now requirements are suitable to use in test generation process and for automatic checks during test executions.

Current automata-program interacts with two different control objects:

- control object responsible for bank account management. It provides amount of money on account and performs withdrawal operation;
- control object providing inputs from the user. It can be keyboard or any other device, which is not important for our purpose. Important is that this object provides an amount of money to withdraw.

Control objects' inputs are presented in model as external variables. Transition which retrieves a value from the controlled object contains following code on its label: `<object name>.get(<variable name>)`.

List of external variables with specification requirements for presented on Fig. 2 ATM model:

- `ext_account` – initial amount of money on bank account. This value is retrieved only once on the first transition. Specification requires: `0 <= ext_account < 1000000;`
- `ext_amount` – amount of money to withdraw. This variable can be retrieved unlimited number of times during the execution. Specification requires: `1000 <= ext_amount < 5000`.

Use of external variable with the defined requirements gives us ability not to depend on control object's specific implementation. Controlled objects that would be used in production are not needed for the test generation and for testing of the automata-based program. This can be critical if controlled objects are expensive or complex equipment, which are not available till the deployment of the system. Also it's important if actual controlled object implies manual input (like any keyboard does), because automatic values generation is preferable.

We considered number of different test scenarios to apply our approach. First, scenarios are defined in natural language, for example:

- user withdraws 10 times and on 11th attempt transaction fails, as not enough money on the account;
- user withdraws 5 times and on 6th attempt transaction fails, as not enough money on the account;
- user successfully withdraws 11 times;
- user withdraws 7 times and on 8th attempt incorrect amount of money is inputted.

Detailed description on how to use proposed approach for the first example follows. Test scenario should be described in terms of transactions. Scenario in terms of transition labels for the automaton given on Fig. 2:

```
t1,
t2, t4, t5, t2, t4, t5, t2, t4, t5,
t2, t4, t5, t2, t4, t5, t2, t4, t5,
t2, t4, t5, t2, t4, t5, t2, t4, t5,
t2, t4, t5, t6.
```

Sequence of transitions is given to the proof-of-concept tool as an input. Values of external variables to execute this path is produced automatically:

```
ext_acount = 28688;
ext_account1 = 3198;
ext_account2 = 4612;
ext_account3 = 2280;
ext_account4 = 2310;
ext_account5 = 4311;
ext_account6 = 1786;
ext_account7 = 3867;
ext_account8 = 1217;
ext_account9 = 2739;
ext_account10 = 519;
ext_account11 = 6376;
```

For this set of variables test code can be generated, which provides correct sequence of events and external variables values to the automata-program, so it executes actions, described in test scenario.

V. Conclusion

Simultaneously with Model Checking testing is useful to check conformance of implementation and specification while developing automata-based systems. For effective testing it is important to automate test generation process, as manual test creation is labor intensive and expensive task. In this paper we proposed an approach for testing of automata-based systems and a proof-of-concept tool demonstrating benefits of described approach. Design contracts and EFSM are used to create models containing specification requirements. Genetic algorithm is used to automate the test generation process.

We plan to provide an IDE plug-in for JetBrains MPS (Meta Programming System) [13], which has the StateMachine extension to develop automata-based programs. Seamless integration of test creation into development process would allow detecting possible implementation faults and design flaws at all development stages.

Acknowledgment

The research is conducted in scope of the Federal target program "Scientific and pedagogical personnel of innovative Russia for 2009 - 2013 years".

References

[1] A. A. Shalyto, "Logic Control and "Reactive" Systems: Algorithmization and Programming," Automation and Remote Control, vol. 62, no. 1, pp. 1–29, 2001.

[2] E. M. Clarke, Jr. O. Grumberg and D. A. Peled, "Model Checking", MIT Press, 1999

[3] G. Myers, The Art of Software Testing, 2 ed: John Wiley & Son. Inc, 2004.

[4] McMinn, P., "Search-based software test data generation: a survey: Research Articles," Software Testing, Verification & Reliability, 2004. 14(2): p. 105-156.

[5] O. Stepanov, "Methods of implementation of automata-based object-oriented programs," PhD Thesis (in Russian), SPbSU ITMO, 2009

[6] B. Meyer, "Applying design by contract," Computer, 25(10), pp. 40–51, Oct. 1992.

[7] Kalaji, A.S., R.M. Hierons, and S. Swift. "Generating Feasible Transition Paths for Testing from an Extended Finite State Machine (EFSM)," in Software Testing, Verification, and Validation (ICST), 2009 2nd International IEEE Conference on. 2009. Denver, Colorado - USA: IEEE.

[8] Lai, R., "A survey of communication protocol testing. Journal of Systems and Software", 2002. 62(1): p. 21-46.

[9] G. T. Leavens, A. L. Baker, C. Ruby, "Preliminary design of JML: A behavioral interface specification language for Java," Iowa State Univ., Dept. of Comput. Sci., Tech. Rep. 98-06u, Apr. 2003.

[10] D.R.Cokand, J.R.Kiniry, "ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2," Nijmegen Inst. for Computing and Inform. Sci., Tech. Rep. NIII-R0413, May 2004.

[11] Tracey, N., J. Clark, K. Mander, and J. McDermid. "An automated framework for structural test-data generation," in Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on. 1998.

[12] Wegener, J., A. Baresel, and H. Sthamer, “Evolutionary test environment for automatic structural testing,” Information and Software Technology, 2001. 43(14): p. 841-854.

[13] MPS User's Guide. http://www.jetbrains.net/confluence/display/MPS/MPS+User%27s+Guide.

Test data generation for covering functionality of database applications

Evgeny Kostychev, Vitaly Omelchenko, Sergey Zelenov
Institute for System Programming
at the Russian Academy of Sciences
Moscow, Russian Federation
e-mail: {kostychev, vitaly, zelenov}@ispras.com

***Abstract*—Applications for processing great volumes of data is a very widely used kind of software. In enterprise integration there are tasks of data integration. When solving these tasks, special tools supporting development and execution of applications implementing extract, transformation and load pattern are often used. From the point of view of functional testing, such applications have a specific peculiarity related to a huge number of combinations of input data. Existing approaches and tools solving the problem of test data generation for database application build large arrays of input data based on database scheme or on SQL queries of application under tests. To ensure covering functionality of an application under test using these approaches and tools, a brute force of all available combinations is needed. In the paper, we prpose a method allowing less excessive data generation for covering functionality of database applications. It allows achieving functionality coverage with acceptable amount of test data close to optimal one (one test per one functionality branch) in acceptable generation time.**

Test data generation; database applications; functional testing; data integration; ETL-applications testing

I. INTRODUCTION

Nowadays, availability of more and more increasing volumes of information storages and computing resources for their processing is constantly growing. This yields prevalence of applications working with huge amount of data in various areas. For instance, in the early nineties of the last century, the British national corpus has been created. This is a specially marked and processed big set of printable and audio texts containing 100 million words for more than twenty years' interval [1]. Since then, such national corpora of many languages have appeared and continue to appear and develop. The corpora not only make many traditional problems of linguistics more trivial, but also allow to state and to solve previously impracticable problems primarily related to processing of great volumes of data [2]. Also problems concerned with processing of great volumes of data rise in such areas as statistics, sociology, geophysics, some sections of physics, molecular genetics, problems with climate, meteorology.

Storage and processing great volumes of data is crucial for modern business too. Enterprise systems contain huge volumes of interconnected data concerning consumers and customers, suppliers, partners, equipment and personnel, financial streams, various business transactions. Huge volumes of data are usually stored in structured form in one or several databases under control of one or several DBMS. Often, supporting various business needs require an integration of several subsystems of an enterprise system or even connecting to some parts of external systems. In many such cases, a problem of access to and operating with data stored in different databases in various formats rises. Therefore, data integration applications that perform replication, updating and synchronization of great volumes of data form a widespread kind of integration applications.

Tasks of such applications can be simple themselves but not always easy for implementing (for example, selection of data concerning some person or legal entity). The tasks can also be enough difficult both in conditions of selection/aggregation of input data and in calculation of final result (for example, invoicing clients according to volume of actual service consumption, tariff plans, and provided discounts). Since this kind of tasks frequently occurs when solving the integration problems, there are specialized platforms providing universal (with respect to platforms for data storage) environment for development and running integration applications that effectively solve problems of extraction, transformation and loading of big data [3]. Further we refer to such applications as ETL-applications (Extract, Transform and Loading).

After development or updating of any application, one should check correctness of its behavior with respect to the functional requirements implementing the business needs. The business logic of ETL-applications can be detailed as follows:

- extraction source data that meet corresponding conditions defined in requirements;
- changing values of source data with respect to conditions defined in requirements;
- transformation of input data (changing values and/or representation format) according to requirements;
- loading into target some data that should be loaded according to requirements;
- changing in the target some data that meet corresponding conditions defined in requirements.

Mainly, one checks correctness of implemented behavior by means of functional testing. That is running the system under test (SUT) on specially prepared input data and comparing output data with expected ones. In the paper, we consider the problem of generating input data for functional

testing of ETL-applications. Note that ETL-applications contain all the same steps as any database applications (data extraction, transformation and loading). Therefore, proposed method is applicable not only to ETL-applications but to many other database applications too.

One can estimate quality of functional testing by achieved coverage of functional requirements. Ideally, all functional branches of the algorithm implementing required functionality should be covered. To perform that, one should provide such test data that force SUT to do the following:

- retrieving data from the source and filtering them on all possible combinations of conditions specified in the requirements;
- covering during data transformation all possible functional branches of the algorithm implementing the required transformation;
- possibility to check the absence of unnecessary changes of source and target data;
- processing special data (empty columns, lines of limited length, etc.).

As a rule, a test is defined by some parameters. Obviously, all possible combinations of parameters contain combinations needed for covering the functionality of the application under test. But in real situations, when generating all possible combinations, the number of parameters to be combined causes combinatorial explosion. In this case, besides of unacceptable generation time and total amount of test, resulting test set contains many different test data values not distinguishable for SUT functionality. A problem of analyzing test results rises: the same errors may be revealed on thousands of different values of test data, and thus, time-consuming analysis is required.

Ideally, for covering functional requirements, test data should provide exactly one possible combination of Boolean expression for each functionality branch. The paper describes a method of generating test data with volume close to the optimal set.

The rest of this paper is organized as follows. Section II contains some preliminaries. Section III reviews related work. Section IV describes proposed method. Section V illustrates the proposed method by example. Section VI discusses benefits of the proposed methods. Section VII concludes the paper.

II. PRELIMINARIES

In general, to ensure full coverage of all functional branches, one should use brute force combination of field values depending on which branching should occur when retrieving, when running an application under test. But it leads to the combinatorial explosion causing unacceptable cost on generation time and total volume of test data.

Often, in terms of functionality coverage, values of several fields are related by some semantics, which means that value of one field depends on values of some other fields. As a rule, iterating only the combinations of values meeting this semantics greatly reduces the total number of variants. Let us consider the following example. Let fields `A` and `B` take integer values in the range `[1 .. 30]` and

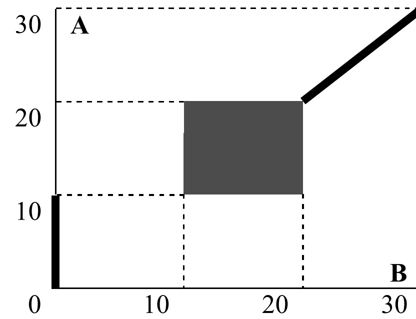


Figure 1. Optimazation of iterating by semantics.

value of the field `B` depends on value of the field `A` as follows (see Figure 1):

- if `1 ≤ A ≤ 10`, then `B = 1`;
- if `11 ≤ A ≤ 20`, then `11 ≤ B ≤ 20`;
- if `21 ≤ A ≤ 30`, then `B = A`.

When producing not all the possible combinations of values of the fields, but only ones meeting the above conditions, the amount of derived combinations is about 8 times reduced (from 900 to 120).

Besides, the functionality of an application under test can have several independent aspects. In this case, it is possible to reduce the number of combinations using so-called diagonal combinator iterating a set of tuples S such that for any i ($1 \le i \le n$) and for any s from S_i there exists a tuple $(s_1,\ldots,s_n)$ from S with $s = s_i$, where S_i is a set of values iterated by i-th subiterator. This method ensures producing a set of test data containing every value of each field. The main advantage of this combination method is that the volume of result test data is significantly less than in the case of using Cartesian product, since the capacity of produced set equals the maximum of capacities of value sets of combining fields instead of product of them.

III. TOOLS REVIEW

Most data generation tools for DB (DTM Data Generator [4], Turbo Data [5], DBMonster [6]) support filling the database tables by a large number of syntactically correct data and provide the following set of features:

- random data generation with ability to specify intervals of numeric types, length of string type, and data format;
- data generation from the list of values with ability to specify the percentage of each list row in generated data set;
- generation by selecting data from specified tables;
- generation by selecting data from files;
- generation of auto incremental data with specified initial value and step;
- data generation from existing libraries;
- random data generation by a mask;
- data generation for dependent tables;
- data generation based on external processes of generation specified by user procedures.

Some tools (AGENDA [7], HTDGen [8]) support the generation of data not only on the basis of constraints defined by schema or user, but also based on SQL queries of database application under test. It allows generating such data that SQL queries of an implementation under test return some meaningful results. However, when generating data is based on an implementation, there is no guarantee of covering all functional branches because the implementation may contain wrong branches or does not implement the required ones. Also this approach does not allow covering all required functional branches of transformation and filtering.

In order to achieve full coverage of all functional branches using any of mentioned above tools a tester has to use brute force combination of field values, depending on which an application under test extracts, transforms and filters data.

The Pinery test generating tool developed at ISP RAS [9, 10] is intended to generate structurally complex test data. Generating data having some syntactic structure is managed by specifying constraints on desired data fragments in the terms of the structure. In particular, Pinery supports so-called conditional constraints that should be used if generation of some part of data depends on values of some other parts of the generated data.

In Pinery, there are many various ways to specify constraints on values of fields. Some of them are:

- enumerating a list of desired values;
- defining a function depending on values of other fields;
- specifying a set data to be belong, for example:
  - a segment of integers;
  - a set of values of another field (e.g., in the case of assignment of values for a foreign key).

Another important kind of constraints is specification of way to combine values of fields. In particular, it is possible to build a hierarchy of different combinators containing Cartesian products, diagonal combinators and custom combinators based on dependences between values of different fields.

These features allow a tester to customize generation more exactly and, as a result, to receive test data with volume close to optimal one.

IV. METHOD DESCRIPTION

The method of directed generation of test data is based on UniTESK [11, 12] approach to model-based testing and consists of the following phases.

The first phase is requirements elicitation: analytics study normative documents for the system under test, identify input data requirements and categorize them. The result of the phase is a requirements catalogue that contains precisely formulated input data requirements, classified into several groups with established links between them. The catalogue is used on the following phases.

The second phase is formalization of requirements. Elicited input data requirements get specified using appropriate formal notation. Such specification is called formal model.

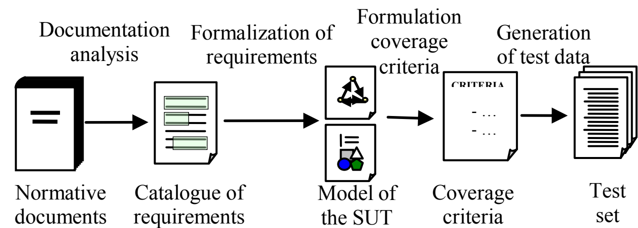


Figure 1. Phases of model-based testing.

The third phase is formulation coverage criteria: the input data domain divides into finite number of subsets (possibly intersecting each other), such that for each subset S, behavior of the SUT on any data from S is uniform. The starting point for such a division is a set of conditions from the requirements catalogue. Besides, the specific semantics of these conditions may force a tester to formulate some additional hypotheses on behavior of the SUT that induce a subdivision of the input data domain into smaller subsets.

The fourth phase is automated tests generation from the formal data model with respect to the coverage criteria. In our approach, automated generation is carried out using the Pinery generator tool. Input data for Pinery are:

- formal description of the test data model, and
- generator configuration aimed at achieving the coverage criteria.

Tests running, reports analysis, defects identification and corrections is beyond the scope of this article. These issues are not discussed here.

V. EXAMPLE

Let us illustrate the proposed method by the following example of test data generation for testing simplified bank credit system.

- The normative document is the following informal description of the bank credit system.
- The client of certain type registers in the system and receives an identifier.
- The client gets a sum of money as a loan that should be repaid in few months.
- The client should monthly repay the sum calculated as current debt divided by quantity of months before the loan termination.
- If actually repaid sum is less than the calculated sum of monthly payment, then the client can be penalized. If actually repaid sum is greater than the calculated sum of monthly payment, then the client can get a bonus. Sizes of penalties and bonuses depend on both client type and credit type.
- The client can declare to get monthly e-mail notifications about the state of the loan, repayments and penalties or bonuses.
- At the end of each month, the application does the following:

A. it reduces the debt by the sum of the client repayment and calculate the penalty or the bonus with respect to the following rules.

I. If the sum of client repayment is less than monthly payment, then:
   1. clients of the type "VIP" should not be penalized;
   2. clients of the type "USUAL" should be penalized with the sum equal to 3% of the overdue payment.

II. If the sum of client repayment is not less than the monthly payment and not more then the current debt, then:
   1. clients of the type "VIP" with the loan of the type "B" and clients of the type "USUAL" with the loan of the type "A" should get bonuses equal to 2% of the repaid sum;
   2. clients of the type "VIP" with the loan of the type "A" should get bonuses equal to 5% of the repaid sum;
   3. clients of the type "USUAL" with the loan of the type "B" should get bonuses equal to 1% of the repaid sum.

III. If the sum of client repayment is more than the current debt, then the bank personnel should be notified about refund.

B. If the client has declared monthly e-mail notification, then the e-mail should be sent.

At the first phase, we analyze the normative documents and build a requirements catalogue. In this example, the requirements catalogue consists of the items of the bank credit system behavior rules (both A with sub-items and B). They can be represented in the form of the diagram (see Figure 3).

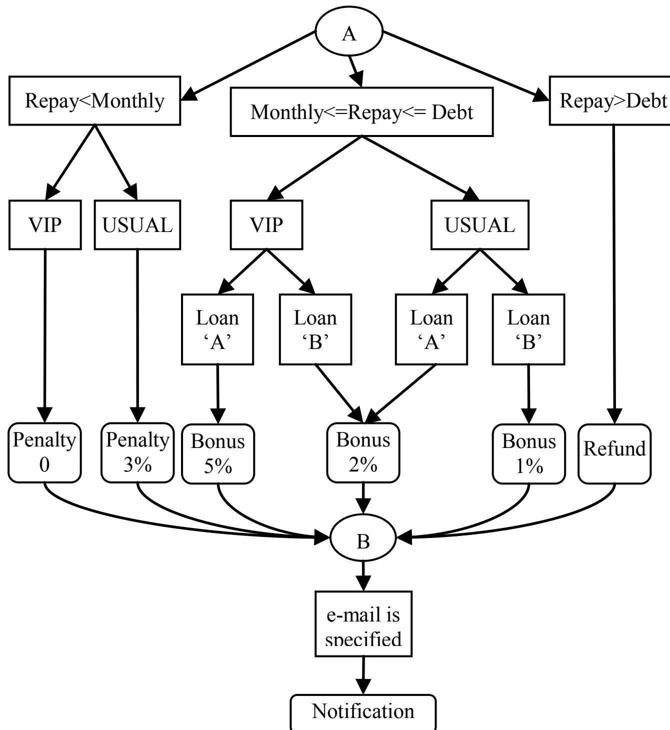


Figure 2. Diagram of bank credit system requirements.

TABLE I. BEHAVIORIAL PARAMERTERS OF BANK CREDIT SYSTEM

Short Name	Full Column Name	Type	Description
MP	MONTH_PAYMENT	NUMBER	Sum to be pay monthly
D	DEBT	NUMBER	Total debt sum
P	REPAYMENT	NUMBER	Actually repaid sum
CL	CLIENT_TYPE	CHAR	Client type: "V" – VIP client "U" – USUAL client
CR	CREDIT_TYPE	CHAR	Credit type: "A" – type "A" "B" – type "B"
E	EMAIL	STRING	E-mail address

At the second phase, we build the formal model of the requirements.

The behavior of the application under test depends on the parameters presented in TABLE I.

The constraint of data consistency is `MP <= D`.

The requirements can be formalized as follows:

```
A.I     P < MP && CL = "V" => penalty 0;
        P < MP && CL = "U" => penalty 3%;
A.II.1     MP <= P <= D
        && (   CL = "V" && CR = "B"
            || CL = "U" && CR = "A"
           )
        => bonus 2%;
A.II.2     MP <= P <= D
        && CL = "V"
        && CR = "A"
        => bonus 5%;
A.II.3     MP <= P <= D
        && CL = "U"
        && CR = "B"
        => bonus 1%;
A.III   P > D => refund.
B.      E ≠ "" => notification.
```

At the third phase, we formulate coverage criteria.

Besides the requirement conditions, we introduce the following additional hypotheses about behavior of the application under test:

1. If `MP <= P <= D`, then behavior of the application can differ in the following cases:
   - `MP = P = D`;
   - `MP = P < D`;
   - `MP < P < D`;
   - `MP < P = D`.
2. If conditions of the requirement A.I hold, then behavior of the application can differ in the following cases:
   - `CL = "V" && CR = "B"`;
   - `CL = "U" && CR = "A"`.

Here we proceed with establishing rules for division the input data domain into subsets with uniform behavior of the

application. First, we formulate such rules separately for each parameter, and then we state how to combine these rules for the whole input data domain.

Let us establish division rules for parameters `MP`, `D`, `P`, `CL`, and `CR`, that relate to requirements from the A group.

The hypothesis 1 yields the following division of input data for parameters `MP` and `D` into two subsets:

- `MP = D`;
- `MP < D`.

Conditions from the requirements of the A group yield the following division rules for parameter `P`.

- If condition `MP = D` holds, then domain for parameter `P` divides into three subsets:
  - `P < MP`;
  - `MP = P = D`;
  - `P > D`.
- If condition `MP < D` holds, then domain for parameter `P` divides into five subsets (taking into account the hypothesis 1):
  - `P < MP`;
  - `MP = P < D`;
  - `MP < P < D`;
  - `MP < P = D`;
  - `P > D`.

Conditions from the requirements of the A group yield the following division rules of input data for parameters `CL` and `CR`.

- If condition `P < MP` holds, then conditions from the requirements of the A.I group yield division into two subsets corresponding to all possible values of the parameter `CL`.
- If condition `MP <= P <= D` holds, then conditions from the requirements of the A.II group and the hypothesis 2 yield division into four subsets corresponding to all possible combinations of values of parameters `CL` and `CR`.
- If condition `P > D` holds, then the condition from the requirement A.III yields no division (or, formally speaking, division into one set).

Next, let us establish division rules for the parameter E that relate to the requirement A.II. The condition from the requirement A.II yields division into two subsets: with empty and non-empty value of `E`.

Next, let us state how to combine these rules for the whole input data domain.

Division of input data domain related to parameters of A group (`MP`, `D`, `P`, `CL`, and `CR`) is induced by Cartesian product (taking into account all dependencies) of divisions for each of the parameters.

Since behavior of the application under test has two independent aspects A and B, then division of the whole input data domain is induced by the diagonal combination of the divisions corresponding to A and B.

At the fourth phase, we generate tests automatically using the Pinery generator tool.

First, we should provide Pinery with formal description of DB scheme (for example, using the DDL-subset of SQL). Next, we should configure Pinery by constraints on data to be generated. These constraints are described in terms of the DB scheme elements.

In our example, there is one table CREDITS with the following fields: MONTH_PAYMENT, DEBT, PAYMENT, CLIENT_TYPE, CREDIT_TYPE, EMAIL. Further we refer these fields by their short names.

In this example, there are two kinds of constraints:

- Constraint on values of one field;
- Constraint on combination method for several fields.

First, we describe constraints of the first kind.

In order to cover subsets with `MP = D` and `MP < D`, we may put, for example, `MP = 6`, `D = 6` and `D = 30`. In order to cover subsets with empty and non-empty E, we may put, for example, `E = ""` and `E = "…@…"` (some address).

In order to configure Pinery with these values, we should describe the following constraints that enumerate lists of values for each field[1]

```
MP = { 6 };
D = { 6, 30 };
E = { "", "…@…" };
```

These are examples of so-called unconditional constraints that specify values valid in all cases.

However, sometimes we must not use unconditional constraints. For example, values of the field P depends on values of fields `MP` and `D`. Thus, we should use two conditional constraints: for the cases `MP = D` and `MP < D`.

In order to make condition `P < MP` hold, we may put `P = MP – 1`. In order to make condition `P > D` hold, we may put `P = D + 1`. If condition MP <D holds, then in order to make condition `MP < P <D` hold, we may put `P = (MP + D)/2`. As a result, we have the following constraints for the field `P`:

```
P[MP<D] =
    {MP-1, MP, (MP+D)/2, D, D+1};
P[MP=D] = {MP - 1, D, D + 1};
```

Values of fields `CL` and `CR` depends on values of fields `P`, `MP` and `D`:

```
CL[ P<MP ] = { "V", "U" };
CR[ P<MP ] = { "A" };
CL[ MP<=P && P<=D ] = { "V", "U" };
CR[ MP<=P && P<=D ] = { "A", "B" };
CL[ P>D ] = { "V" };
CR[ P>D ] = { "A" };
```

Here we proceed with description of a combinator of fields values for generation of CREDITS table tuples.

In all cases, we may combine values of fields `CL` and `CR` using Cartesian product:

```
Product( CL, CR )
```

Similarly, we may combine values of fields `MP` and `D`:

```
Product( MP, D )
```

[1] In Pinery, constraints are described in XML form. In order to increase readability, here we describe constraints in a semi-formal pseudo-code.

Since values of fields CL and CR depend on values of fields P, MP, and D, and values of field P depend on values of fields MP and D, then we should use special "dependent" combinator that describes combinations of the fields that relate to requirements from the A, B and C groups:

```
Depend(  Product( MP, D )
      => P
      => Product( CL, CR )
      )
```

As we mention above, we should use diagonal combinator to combine fields that relate to requirements from groups A and B. So, we have the following combinator for generation of CREDITS table tuples:

```
combinator( CREDITS ) =
   Diagonal( Depend(  Product(MP, D)
                   => P
                   => Product(CL, CR)
                   )
            , E
            );
```

Resulting test data is presented in TABLE II.

TABLE II. GENERATED TEST DATA

D	MP	P	CL	CR	E
6	6	5	V	A	
6	6	5	U	A	@
6	6	6	V	A	
6	6	6	V	B	@
6	6	6	U	A	
6	6	6	U	B	@
6	6	7	V	A	
30	6	5	V	A	@
30	6	5	U	A	
30	6	6	V	A	@
30	6	6	V	B	
30	6	6	U	A	@
30	6	6	U	B	
30	6	18	V	A	@
30	6	18	V	B	
30	6	18	U	A	@
30	6	18	U	B	
30	6	30	V	A	@
30	6	30	V	B	
30	6	30	U	A	@
30	6	30	U	B	
30	6	31	V	A	@

VI. DISCUSSION

Using of a dependent combinator allow us to have a uniform configuration of the generator instead of two (for cases MP = D and MP <D).

Using of a dependent and diagonal combinators allow us to reduce quantity of generated test data by more than 65% in comparison with the general Cartesian product: We have 22 tuples while Cartesian product gives 64 tuples (3*2*2*2 = 24 for case MP = D, plus 5*2*2*2 = 40 for case MP < D). Nevertheless, our test set has the same quality as the Cartesian product test set (with respect to the formulated coverage criteria).

There are two aspects that make relative reduction of tests quantity to increase.

First, the more possible values of fields are, the more economy we have. For instance, if we have in our example one addition client type and one addition credit type , then economy in our approach is 70% (3*3*3*2 + 5*3*3*2 = 144 for Cartesian product against (3 + 1*3*3 + 1) + (3 + 3*3*3 + 1) = 44 in our approach).

Second, the more independent aspects of behavior of the application under test, the more economy we have. Suppose in our example, that actual payment has a type with two possible values ("O" – payment under the clearing settlement and "E" – payment by cash), and the application under test uses this type in some additional aspect of behavior (for example, calculating some statistics). Then the quantity of tuples in our approach does not increase (since we use diagonal combinator), while the quantity of tuples for Cartesian product doubles.

VII. CONCLUSION

In the paper, we propose the method of automated generation of test data for functional testing of applications that process huge volumes of data. The method is aimed to cover functional branches of an application under test. The main benefit of the method is that on the one hand it allow a tester to achieve coverage of functionality of an application under test, but on the other hand generated test data are more optimal then in existing tools:

- generation process is less time-consuming, and
- test report analysis is less labor-consuming and less time-consuming.

Generation of the data by means of Cartesian combination of all fields provides full coverage, but resulting test data are practically always superfluous. Redundancy extremely grows under increasing number of combined fields and cardinality of sets of their values, on which the behavior of the application under test depends.

The proposed approach is based on requirements analysis and formalization. Usage both Cartesian product combinators, and dependent and diagonal combinators allows a tester to reduce a test set without loss of test data quality and to obtain test data with volume close to the optimum.

The approach is supported by the Pinery generator of structurally complex data.

REFERENCES

[1] http://www.natcorp.ox.ac.uk/corpus/index.xml

[2] N. Oostdijk and P. Haan, Corpus-Based Research into Language. In honour of Jan Aarts, Amsterdam/Atlanta, GA, 1994, VII.

[3] M. L. Songini, QuickStudy: Extract, Transform and Load (ETL), 2004, Computerworld,

http://www.computerworld.com/s/article/89534/QuickStudy_ETL

[4] DTM Data Generator. http://www.sqledit.com/dg/

[5] Turbo Data. http://www.turbodata.ca/

[6] DBMonster. http://dbmonster.kernelpanic.pl/

[7] D. Chays, Y. Deng, P.G. Frankl, E.J. Weyuker, An AGENDA for testing relational database applications, Software testing, verification and reliability, 2004, VOL 14; PART 1, pages 17–44.

[8] C. Binnig, D. Kossmann, E. Lo, Testing database applications, 2006, Proceedings of the 25th ACM SIGMOD international conference on management of data / Principles of database systems, Chicago.

[9] A.V.Demakov, S.V.Zelenov, S.A.Zelenova. Pinery generator of structurally complex data: implementation of new capabilities of UniTESK // Proceedings of ISP RAS, Moscow, 2008, vol.14, part 1, 119–136.

[10] A.V.Demakov, S.V.Zelenov, S.A.Zelenova. Using abstract models for the generation of test data with a complex structure. Programming and Computer Software, 2008, vol.34, N 6, 341–350.

[11] I.B.Bourdonov, A.S.Kossatchev, V.V.Kuliamin, A.K.Petrenko. UniTesK Test Suite Architecture. Proc. FME'2002 conference, LNCS, 2391. Copenhagen, Denmark, 2002, 77—88.

[12] UniTESK Technology Web-site. http://www.unitesk.com/

Testing AJAX functionality with UniTESK

Yevgeny Gerlits
Lomonosov Moscow State University
Moscow, Russian Federation
Email: gerlits@ispras.ru

***Abstract* —AJAX (Asynchronous JavaScript and XML) is a very promising technology for building interactive web applications. At the same time, AJAX significantly complicates the development of the client side of web applications. The paper demonstrates the possibility of utilizing the UniTESK test development technology for testing the client side functionality of AJAX web applications. Using UniTESK, test systems are developed for 8 AJAX web applications. Then the fault revealing capability of the test systems is evaluated in experiments.**

Keywords-AJAX; model based testing; UniTESK, asynchronous interface

I. INTRODUCTION

A classic web application is built around the notion of web pages and generally consists of a set of static web pages or server side programs that generate web pages. Such a web application is sufficiently inferior in interactivity to a web application developed with AJAX. The main reason is that the user communicates with the classic web application synchronously, that is he supplies input to the browser, e.g. clicks on a submit button or a link, and then waits until the browser refreshes the page. As opposed to this, web applications developed with AJAX can retrieve data from the server asynchronously in the background without interfering with the display and behavior of the existing page.

At the same time, improving interactivity with AJAX sufficiently increases the complexity of the client side development. Using the JavaScript programming language, an AJAX application developer should implement an intermediate level between the browser and web-server which is responsible for handling user actions, managing browser-server dialog, and changing the interface according to web server responses. This task is hard enough to make a couple of faults.

In this paper, we consider the problem of testing the client side functionality of AJAX web applications. We show that qualitative tests can be elaborated using UniTESK [1, 2], an industrial model based test development technology designed in Institute for System Programming of Russian Academy of Sciences.

UniTESK was initially applicable to only systems with synchronous interfaces. After a period of time, an approach [3, 4, 5, and 6] was designed and implemented that extends this technology to asynchronous interfaces. Since then UniTESK has given a good account of oneself in testing several classes of complex applications with asynchronous interfaces such as internet protocols, components of a distributed operating system, and functions of the standard binary interface of Linux. Actually, these successful applications of UniTESK suggested that we apply this technology to AJAX web applications.

UniTESK offers a test suite architecture consisting of a set of components that are used as building blocks to organize test systems. In the paper, we present a technique for developing these components so that the test system they form aims at revealing faults in the client side of the AJAX web application under test. We do not consider the problem of testing the server side of AJAX applications in this paper.

After presenting the approach to testing systems with asynchronous interfaces proposed by UniTESK and our technique of its use, we conduct several experiments in which we practically apply them. The obtained results show the applicability of UniTESK and the technique for testing the client side functionality of AJAX web applications. At the end of the paper, we present a comparison between our approach and the existing approaches to highlight the key advantages of UniTESK and our technique. We also discuss the main limitations and drawbacks of our approach.

The paper is structured as follows. Section II is devoted to the AJAX technology. We consider the architecture, the behavioral model, and the main features of a typical AJAX application. Section III outlines the UniTESK approach to testing systems with asynchronous interfaces. In section IV, we present our technique for testing the client side of AJAX web applications with UniTESK. We empirically evaluate the applicability of UniTESK and the technique in section V. Section VI compares our approach with the existing techniques. We conclude with a summary of our key contributions, and suggestions for future work in section VII.

II. AJAX

AJAX is an approach to web interaction that combines a set of well known technologies to achieve high interactivity of web applications. In this section, we consider the architecture, the behavior and the main features of a typical AJAX application.

Let us discuss AJAX applications comparing them with web applications that we call “classic”. The architectures of both the classic and AJAX applications are shown in Fig. 1.

A classic web application consists of a set of web pages. Some web pages may be described in static HTML

This work was supported by the RFBR (grant 09-01-00576-a)

(Hypertext Markup Language) files; the others may be generated by the server side programs. A web page is displayed to the user, containing lists of links and form elements that allow the user to drill down to further web pages.

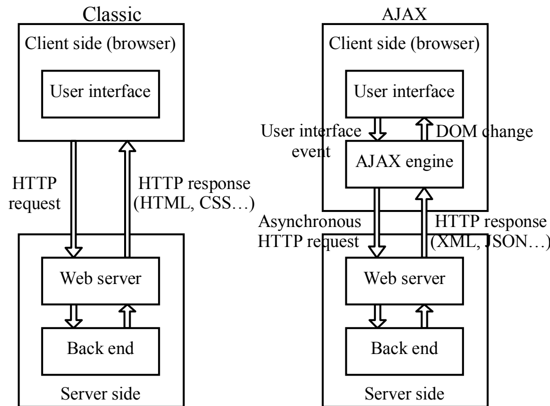


Figure 1. The architectures of classic and AJAX web applications

The main functionality of a classic web application is implemented at the server side. Some animation and additional functionality can be provided using client side programming languages and technologies, but it doesn't change the main behavioral model of the application. This model works as follows: the user supplies input to the browser, e.g. types a URL (Uniform Resource Locator), clicks on a hyperlink, or submits a form; the browser sends the HTTP (Hypertext Transfer Protocol) request for the URL to the web server; the web server responses with a new web page; the browser renders the page and waits for the user's next input.

The key features of classic web applications are as follows:

1. The user interacts with the web application synchronously, i.e. he requests for the next web page only after the response to the previous request has been handled by the browser and the appropriate web page has been displayed.
2. HTTP requests are issued for entire web pages and the entire page gets refreshed as a result of this action.
3. HTTP requests are issued by the browser, and HTTP responses are handled by the browser.
4. HTTP requests occur as a direct consequence of user actions.

As contrasted with a classic web application, the user communicates with an AJAX application asynchronously. The behavioral model proposed by AJAX works as follows:

1. The user performs an action on the web interface, e.g. clicks on a hyperlink, or a button.
2. An appropriate user interface event is fired.
3. The handler of this event, a JavaScript function, is called. It builds an asynchronous HTTP request, sets a callback function that will handle the response, and issues the request to the web server.
4. The web server replies with the data.
5. The callback function is called, it reads the data and changes the client side state that includes the DOM (Document Object Model) state, cookies, and global JavaScript variables.

According to this model, the user is able to go on working with the AJAX web application right after the user interface event handler has been executed, i.e. the user does not has to wait until the client-server dialog has been completed as it is happens in case of a classic web application. Because of the small size of the transferred data, the browser responds very quickly and the user does not feel any delay.

The key features of AJAX web applications are as follows:

1. The user interacts with the AJAX application asynchronously, i.e. he goes on working with the application while asynchronous HTTP requests are issued and responses are handled in the background.
2. The web server does not response with the entire web page, it responses with data that the client side JavaScript uses to dynamically refresh a small part of the currently displayed page.
3. HTTP requests are formed, issued and handled by JavaScript functions.
4. User actions can trigger the execution of JavaScript functions that may change the client side state and perform communication between the client and server, but JavaScript functionality is also able to work independently from user actions. It is usually achieved with special JavaScript functions that use timers to call other JavaScript functions.
5. The JavaScript programming language doesn't support multithreading. The browser uses one thread to handle user actions and execute JavaScript functions, including user interface event handlers and callback functions.
6. Concurrent HTTP requests are possible in some AJAX web applications, i.e. the next HTTP request may be issued before the response to the previous one has been handled.

In the paper, we consider AJAX applications, the client-server dialog of which complies with the behavioral model presented in this section. It doesn't matter which mechanism an AJAX application uses to perform asynchronous client-server communication. Let us note that the use of the XMLHttpRequest [7] object implies a sequence of HTTP responses to a single HTTP request. We take this fact into account.

We also suppose that an AJAX application itself is able to perform client-server communication independently from user actions.

III. Testing Asynchronous Interfaces with UniTESK

UniTESK is a model based test automation technology. It can be used for testing systems with synchronous and asynchronous interfaces. A synchronous interface implies that the subsequent action on the interface may be performed only after the interface has already responded to the previous

action. The interface of a software system is considered to be asynchronous if this system can simultaneously interact with several other systems or interactions may be initiated by the system itself. The approaches and test suite architectures for testing systems with synchronous and asynchronous interfaces differ. We will discuss UniTESK implying only the asynchronous case in the remainder of the paper.

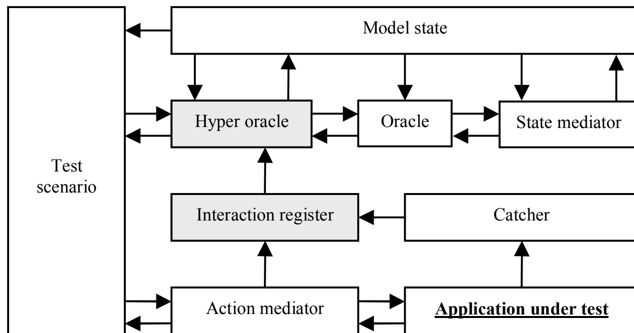


Figure 2. The UniTESK test suite architecture for testing systems with asynchronous interfaces

Each test system developed with UniTESK consists of a set of components. UniTESK defines the number of the components, their responsibilities and relationships. Fig. 2 contains the test suite architecture proposed by UniTESK for testing systems with asynchronous interfaces. Some of the components are already implemented and are used as is independently from the type of the application under test. Their representations have the gray background in the figure 2. The other components should be implemented by the tester and their implementations vary depending from the application under test. UniTESK provides formal descriptions to describe these components, extensions of some of the industrial programming languages to develop them, and software instruments to translate formalisms into the code in the target industrial programming language. The following formal descriptions are provided: *specifications, mediators,* and *scenarios*.

The test system developed with UniTESK supposes that the interface of the application under test consists of atomic operations of two types: *stimuli and reactions*. The test system supplies input to the application by means of applying stimuli to it. The application outputs through reactions that the test system evaluates. Reactions can be of two types: immediate or deferred. *Immediate reaction* is a reaction that is visible from outside immediately after affecting target system. When testing an application with an asynchronous interface, reactions to some stimuli may not be observed immediately because of the internal processes in the application. *Deferred reaction* is a reaction that is visible from outside later some time after a set of affecting target system.

Stimuli and reactions are the notions of UniTESK. The tester has to represent the real application interface through stimuli and reactions, and provide UniTESK with this interface. The following question could be set. Is it always possible to represent an arbitrary asynchronous interface through atomic stimuli and reactions? We haven't heard of a formal proof of it, but we also haven't heard of a contrary instance refuting it.

A formal interface of the application under test consisting of stimuli and reactions is fixed in *specifications*. Requirements to the application behavior are also fixed in *specifications* in the form of pre-conditions and post-conditions of stimuli and reactions, and *invariants of data types*. Specification also contains data structures that model the state of the application under test, i.e. describe the *model state*. The model state reflects the state of the application under test during testing. The requirements in specifications are imposed on the model state changes. *The pre-condition for the stimulus* describes constraints on the state, in which the test system is able to apply the stimulus. Violation of the precondition for the stimulus represents that the test is made incorrectly. The immediate reaction does not have its precondition. The post-condition for the stimulus and post-condition for the immediate reaction are the same things. *The post-condition for the stimulus* defines the requirements to the result of its application, i.e. to the state change and possibly the return value of the application operation the stimulus refers to, e.g. when applying the stimulus leads to the call of a public application operation that returns a value. *The pre-condition for the deferred reaction* describes if appearance of the reaction in the given state is possible. When precondition for the deferred reaction is violated, incompliance between the behavior of the application and its specification is registered. *The post-condition for the deferred reaction* checks compliance of the result obtained when the reaction emerges, to the expected one.

UniTESK defines the structure of specifications. The main goal of this structure is to provide the test completeness metric.

Specifications are translated into the test suite architecture components that take part in the verification of stimuli and reactions: *model state, action oracles* and *state mediators*.

To be able to verify requirements to stimuli and reactions, the test system should somehow link specifications to the application under test. *Action Mediator* component is generated from the formal description called mediator. It performs actions on the application under test, i.e. really applies stimuli. It also registers immediate reactions. The other component, implemented in the target programming language, registers the appearance of the deferred reactions. It is called *catcher*. The component that keeps information about the order of stimuli and reactions is called *interaction register*. The exact order of stimuli and reactions can not always be observed when testing a system with an asynchronous interface; therefore the UniTESK approach to testing systems with asynchronous interfaces was designed to be able to take advantage of the observable partial order of stimuli and reactions. So, interaction register usually keeps information about the detected partial order of stimuli and reactions.

The component of the UniTESK test suite architecture, which is called *test scenario*, is generated from the formal description of the same name and is used to combine

operations that test logically related aspects of the application functionality. These operations are called *scenario functions*. Each scenario function applies a set of logically related stimuli to the application under test, supplying values for their parameters. To apply a single stimulus, the scenario function passes its call to test oracle, test oracle passes the stimulus to action mediator, and action mediator finally applies the stimulus.

Stimuli are applied and reactions appear during the execution of the scenario function. The completion of the scenario function indicates that all the stimuli have already been applied and all the reactions have been cached. After the scenario function has been executed, *hyper oracle* begins evaluating the observable behavior of the application under test. Information about the detected order of stimuli and reactions is utilized during the evaluation process as follows. The test system goes over all the possible orders of stimuli and reactions that conform to the partial order detected. For each particular order, each stimulus, and reaction test oracle checks the pre-condition, state mediator synchronizes the model state with the state of the application under test, and again test oracle checks the post-condition. If this procedure discovers at least one order, for which all the constraints on stimuli and reactions are met, the test system claims that the behavior of the application under test is acceptable.

To completely automate the execution of UniTESK tests and automatically generate sequences of test inputs, the developer has to define test scenario automata. A special component of the UniTESK test suite architecture goes over all the states of test scenario automata and calls each scenario function in each accessible state. To define test scenario automata, a function should be implemented that returns the state of test scenario automata after each scenario function call. In theory, the state of test scenario automata is constructed on the base of the model state. In practice, it may be an arbitrary function. This function allows the test system to construct test scenario automata incrementally during testing.

UniTESK imposes the following restriction on the behavior of the application under test: after applying a set of stimuli to the application, it demonstrates a set of reactions during a finite period of time and goes to a state in which no reactions appear spontaneously. Such states are called *stationary*. Stationary states allow the test system to perform the evaluation process and call the next scenario function at the state in which the previous scenario function finished.

In this section, we have only outlined the main characteristics of the approach we use for testing the client side of AJAX web applications. The details can be found in [3, 4, 5, and 6].

IV. Testing AJAX applications with UniTESK

In this section, we present a technique for developing the UniTESK test suite architecture components so that the test system they form aims at revealing faults in the client side functionality of AJAX web applications.

A. *The technique*

In practice, functional testing of web applications aims at discovering faults of two types: general faults such as dead links and incorrect markup, and business logic faults concerning the behavior of the web application under test. Business logic faults are discovered when the web application under test incorrectly reacts to a logically related set of stimuli. The technique we propose in this section aims at discovering faults concerning the behavior of the client side functionality of AJAX applications.

At the first step, the requirements to the behavior of the client side of the AJAX application under test are extracted. When testing a web application, it is natural that there aren't any well-structured documents describing functional requirements. The probability of getting the requirements to the client side of the AJAX application is even lesser. We do not propose a method for the extraction of the requirements in the paper, because elaboration of such a method requires additional investigations and a separate paper is better to be written on the matter. We only assume here that the result of the requirements extraction procedure is a set of well-structured documents describing the requirements to the client side of the AJAX application under test.

At the second step, the extracted requirements are to be formally fixed in specifications in the form of pre-conditions and post-conditions of stimuli and reactions, and invariants of data types. To be able to formalize the requirements using the software contracts proposed by UniTESK, the tester must represent possible interactions of the client side functionality with its environment as a set of stimuli and reactions.

We believe that an adequate model is shown in Fig. 3. This model conforms to the behavioral model of a typical AJAX web application presented in section II, but it only concerns the client side of the application. An individual action on the application interface represents a stimulus if this action leads to the modification of the client side state or if an asynchronous HTTP request is issued. A user interface event occurs as a result of such an action. The handler of this event is called. It may change the client side state or issue an asynchronous HTTP request. The result of its execution is modeled as a reaction. The new proxy server component of the test system intercepts the request issued by the user interface event handler. It in turn issues the HTTP response. It is modeled as a stimulus. The callback function is called that handles this response. The client side state can be modified as a result of its execution or something else can happen. It is modeled as a reaction.

The client side functionality of the AJAX web application under test may change the client side state or issue an asynchronous HTTP request independently. Such an activity is modeled as a rection.

Having this model, the requirements to the stimuli and reactions can be formalised. Stimuli are specified trivially. A reaction results to the client side state change and possibly an asynchronous HTTP request. So, the postcondition for the reaction should asses the client side state change and the HTTP request in case the request is issued as a result of the reaction.

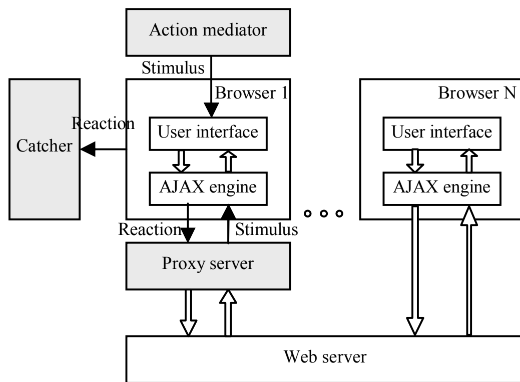


Figure 3. Interactions of the client side of an AJAX web application with its environment

In order that the test system may really verify the behavior of the AJAX application, action mediator, catcher and proxy server test suite architecture components are implemented at the third step.

Action mediator contains functions that programmatically perform actions on the application interface.

Catcher must detect the reactions, and extract and save the client side state changes after them. The single threaded nature of JavaScript helps a lot for the extraction of the client side state changes. If the extraction of the client side state change is accomplished by a JavaScript function, it is guarantied that there aren't another activity that modifies the client side state at the same time.

Proxy server is not a part of the UniTESK test suite architecture. It is a new component specifically desigent to support testing of AJAX applications. Proxy server has two responsibilities:

- intercept asynchronous HTTP requests;
- apply stimuli that model the responses of the target web server.

The use of proxy server allows modeling the real situation of multiple users working with a single web server. The server side state can be changed by the users. Proxy server is able to respond taking the possibility of the server side state changes into account.

The client side state changes and the intercepted HTTP requests are used by the state mediator to synchronize the state of the requirements model with the state of the AJAX application under test during the verification procedure.

At the fourth step, specifications are used to determine the test coverage criteria. The higher is the criteria, the more complecate are test scenarios.

At the fifth step, test scenatios are developed so that the choosen test coverage criteria could be achieved during testing.

Testers often do not take faults concerning multiple asynchronous HTTP requests into account, because of their low probability. A typical example of such a fault can be the following: the second asynchronous HTTP request is issued before the response to the previous one has come; due to network delay, the response to the second request comes before the response to the first one; the callback function that handles the second response removes a DOM element; the response to the first request comes; its callback function crashes trying to access the deleted DOM element. It is obvious, that the proposed technique for modeling stimuli and reactions alows developing scenatio functions aiming at testing multiple asynchronous HTTP requests.

B. *Application domain*

The approach to testing systems with asynchronous interfaces proposed by UniTESK has two main application conditions:

1. A formal interface consisting of atomic stimuli and reactions may be provided for the real interface of the application under test. This formal interface should adequately model the real application interface.
2. After responding to a set of stimuli, the application under test must go to a stationary state in which no reactions can appear spontaneously.

The technique we have just presented explains how to get a formal interface complying with the first condition.

As concerns to the second condition, we have mentioned in section II of the paper that AJAX web applications may have client side functionality that changes the client side state and communicates with the server independently from user actions and at an unpredictable time. Formally, there are no stationary states in such applications. If such functionality is out of the scope of testing, it usually may be ignored or deactivated by hand. If the test system must take such functionality into account, it has to model stationary states. For instance, the test system may artificially execute a piece of JavaScript during the evaluation process in order that the application under test does not change the client side state or issue an HTTP request.

At the moment, we can not imagine a client side functionality of an AJAX web application that can not be modeled and tested using UniTESK and our technique.

V. Empirical Evaluation

In order to evaluate the applicability of the UniTESK test development technology and the technique of its use presented in the paper for testing functionality of the client side of AJAX web applications, we perform a set of experiments.

We collect 8 AJAX design patterns. Each pattern describes how the objects, components, and levels constituting the AJAX web application should interact in order that the application could respond to user actions in a certain way or a certain interactivity effect could be achieved. The patterns primarily describe client sides of AJAX web applications. Implementing them allows us to get AJAX applications that both implemented differently and behave differently.

We implement each pattern in an AJAX web application. So, we have 8 AJAX applications. After that, using the UniTESK technology and our technique, we create a test system for each AJAX web application developed. In order

to assess the fault-revealing capability of the test systems we intentionally introduce faults into the source code of the AJAX web applications, perform testing and count the percent of the faults revealed. This section presents the results of our experiments.

A. *AJAX design patterns*

Here we briefly introduce 8 AJAX design patterns and their implementations for which we develop test systems. Detailed description of the patterns can be found in [8, 9, and 10].

Pattern: Explicit Submission. *Problem:* How can information be submitted to the server? *Solution:* Instead of automatically submitting upon each browser event, require the user to explicitly request it, e.g. submit upon a button click. *AJAX application:* A simple authorization form.

Pattern: Periodic Refresh. *Problem:* How can the application keep users informed of changes occurring on the server? *Solution:* The application periodically issues asynchronous requests to gain new information, e.g. one request every five seconds. *AJAX application:* An application alerts the user as a new comment has been added.

Pattern: Submission Throttling. *Problem:* How can information be submitted to the server? *Solution:* Instead of submitting upon each JavaScript event, retain data in a browser-based buffer and automatically upload it at fixed intervals. *AJAX application:* An application that submits a single field periodically as changes are made.

Pattern: Predictive Fetch. *Problem:* How can you make the AJAX application respond quickly to user activity? *Solution:* Have the application anticipate likely user actions and call the server in preparation. *AJAX application:* An application that preloads the next page of the article.

Pattern: Browser-side Cache. *Problem:* How can you make the AJAX application respond quickly to user activity? *Solution:* Retain server results in a browser-side cache. Whenever the application performs an asynchronous request, it first checks the cache. If the query is held as a key in the cache, the corresponding value is used as the result, and there is no need to access the server. *AJAX application:* A simple calculator that performs calculations on the server and retains the results in a client-side cache.

Pattern: Guesstimate. *Problem:* How can you cut down on calls to the server? *Solution:* Instead of requesting information from the server, use a historical data and make a reasonable guess on the client. *AJAX application:* An approximate calculation of the number of registered users.

Pattern: Pseudo-threading. *Problem:* AJAX web applications are single-threaded. Some of them require complex processing on the client. If the thread of execution is busy performing such processing, users won't be able to perform input. *Solution:* Instead of solving the entire problem at once and returning, a processing function is called once in a while, incrementally processes a bit more of the problem, before yielding. *AJAX application:* Sorting of a big table on the client.

Pattern: Multi-stage Download. *Problem:* How can you optimize downloading performance? *Solution:* Break content download into multiple stages, so that faster and more important content will arrive first. *AJAX application:* An application that downloads additional links after the main content of the article has been downloaded.

B. *Experiments*

To implement test systems for the AJAX applications introduced in the previous subsection, we exploit both the Java and JavaScript programming languages. The JavaTESK [11] toolkit is used to implement the UniTESK test suite architecture components and run the test suites developed. The Selenium Remote Control [12] testing tool is used to drive the browser, programmatically perform actions on the web interface, and access the resulting DOM states. We exploit Mozilla Firefox as a browser in our experiments. Our technique of the use of UniTESK introduces the proxy server component in the test suite architecture. We implement this component using the Java programming language. It is universal, i.e. implemented once it is included in all the test systems.

We perform five experiments for each AJAX web application and corresponding test system. Thus forty experiments are conducted in the total. Each experiment consists in introducing a single fault into the source code of the application, running the corresponding test system on the application, and analyzing the test results. Table 1 summarizes the results of the experiments performed.

TABLE I. THE RESULTS OF THE EXPERIMENTS

AXAX application for	Introduced	Revealed	%
Explicit Submission	5	5	100%
Periodic Refresh	5	4	80%
Submission Throttling	5	4	80%
Predictive Fetch	5	5	100%
Browser-side cache	5	5	100%
Guesstimate	5	3	60%
Pseudo-threading	5	4	80%
Multi-stage download	5	4	80%
TOTAL	40	34	85%

Here are some examples of the faults introduced: building incorrect HTTP requests in JavaScript functions, removing user interface event handlers, wrong modifications of the DOM, removing an XMLHttpRequest object from the pool of XMLHttpRequest objects, setting timers with wrong time intervals, removing identifiers of HTML elements and etc. All the faults appear at the client side of the AJAX applications.

The test systems reveal 85% (in the mean) of all the errors introduced. We believe it is a good result that confirms the applicability of UniTESK and the technique of its use for testing functionality of AJAX web applications. It is worth noting that the percentage of the faults revealed depends on the quality of the test systems developed.

VI. COMPARISON WITH THE EXISTING APPROACHES

We didn't manage to discover another approach specifically designed for testing the client side of AJAX applications. In this section, we present an overview of the existing AJAX functional testing approaches. The approaches test an AJAX application as a whole; therefore

they are able to reveal faults in both the client side and server side of AJAX applications. We compare them with the approach we propose in the paper, i.e. the UniTESK technology complemented with the technique of its use.

A. Approaches proposed by the scientific community

We succeed in discovering three approaches specifically designed for functional testing of AJAX web applications:

- Invariant Based Testing [13];
- State Based Testing [14] ;
- Search Based Testing [15].

All the approaches use a FSM (Finite State Machine) model of the AJAX web application under test to produce tests; therefore we label them as *FSM based test generation approaches*.

The Invariant Based Testing approach is rather directed to revealing faults in dynamical DOM states such as dead links, incorrect markup, and the absence of widgets, DOM elements, and error messages; than organizing complex test situations in which the test system applies a set of logically related stimuli to the application and verifies the reactions to these stimuli. Accomplishing the latter is the primary purpose of the approach we propose in the paper, i.e. the UniTESK technology complemented with the technique of its use. So, the Invariant Based Testing approach and our approach aim at revealing faults of different types; therefore there is no point in their further comparison.

The State Based Testing approach divides test creation into two stages. At the first stage, the FSM model of the AJAX application under test is constructed on the base of a set of preliminarily recorded real execution traces of the application. The states of the FSM are abstracted from the real DOM states. The transitions are the JavaScript method invocations triggered by user events or server responses and modifying the DOM. At the second stage, tests are generated on the base of the traversal of the FSM extracted at the first stage. The test generation is accomplished so that the generated tests are able to automatically reveal faults leading to the modification of a correct sequence of states in the FSM model of the application.

Because the FSM model is constructed on the base of the real behavior of the application, the approach is expected to show its best in regression testing. The authors strengthen the approach by providing the ability to express general requirements to the behavior of the application in the form of pre-conditions and post-conditions. This feature of the approach makes it possible to apply it for functional testing. An advantage of the software contracts proposed by UniTESK is that they additionally provide test coverage criteria. The State Based Testing approach deals with concurrent asynchronous HTTP requests, but it only warns whether there may be a problem. As opposed to this, our approach reveals faults concerning multiple asynchronous HTTP requests. The authors of State Based Testing claim that their approach is a good complement to the classic functional testing.

The Search Based Testing approach is based on the State Based Testing approach. The authors propose a technique that enhances the fault revealing capability of the tests generated. The main features of the approach remain the same.

A common advantage of the State Based Testing and Search Based Testing approaches over our approach is that they are better automated. The approaches are designed for testing only AJAX applications, the authors of the approaches tried to automate them as much as possible. In contrast to the approaches, the UniTESK technology doesn't take the AJAX specific features into account, because it was developed to be applicable for general purpose software. That is why developing some of the UniTESK test suite architecture components is a fairly labor-intensive task. For instance, special functions should be implemented in order that action mediator could programmatically perform actions on web interface elements. Each particular AJAX application requires its own functions because there aren't two AJAX applications that have the same interface. Other functions should be implemented in order that catcher could get DOM states after the reactions.

B. Approaches used in industrial practice

We examined existing test automation tools that support functional testing of web applications. The tools that are positioned as AJAX test automation tools implement the Capture and Playback [16] approach. According to the approach the tester records the user actions; saves them in a script; enhances the recorded script with verification points, where some property or data is verified against an existing baseline; plays back the script and observes the results. The Capture and Playback approach is very useful for regression testing. It is also widely used for functional testing of classic web applications.

In order to support testing of AJAX applications, Capture and Playback testing tools implement either a method for automatically detecting responses to asynchronous HTTP requests or a method for detecting DOM state changes. Such a method allows a Capture and Playback testing tool to determine whether the application has already responded to the user action during the playback stage. The Capture and Playback approach supporting AJAX is implemented in IBM Rational Functional Tester [17], SWEA [18], and many other test automation tools. The Capture and Playback approach doesn't aim at creating complex test sequences like the approach we propose in this paper. Using it leads to the generation of a big amount of test scripts. A script usually verifies a sequence of possible user actions. Week modularity is a common disadvantage of such scripts. As opposed to this, the test suite architecture is one of the most competitive advantages of UniTESK.

The most flexible of the existing AJAX functional testing techniques is to use a combination of a unit testing framework and a software library which makes it possible to programmatically perform actions on the application interface and then access the resulting DOM state. An example of such a technique is the JUnit [19] unit testing framework complemented with the Selenium Remote Control testing tool. By analogy with the Capture and Playback testing tools, AJAX support is limited to designing and implementing either a method for detecting responses to

asynchronous requests or a method for detecting DOM state changes. Let us note that this technique is flexible because it provides minimal support for test automation. In fact tests are handmade, but can be executed automatically.

VII. Conclusion

In this paper, we demonstrate the applicability of the UniTESK test development technology for testing the client side functionality of AJAX web applications. We outline the approach to testing systems with asynchronous interfaces proposed by UniTESK, present the technique for modeling and testing AJAX applications with UniTESK, practically evaluate UniTESK and our technique, and compare our approach with the existing approaches.

Though UniTESK can be used to develop test systems for AJAX web applications, UniTESK is not an AJAX-specific testing technique. Developing tests for AJAX with UniTESK is a very labor-intensive task. The future work may consist in enhancing the automation level of the approach we propose in the paper.

In this paper, we ourselves develop AJAX applications. Then we apply UniTESK to them. In our future work, we should apply UniTESK to a couple of applications really working in Internet.

Our approach can only be used for testing the client side functionality of AJAX web applications. On the one hand, the approach is directed to the client side faults that are typical and specific for AJAX web applications. On the other hand, we do not test the server side at all. Future investigations may consist in designing an AJAX testing technique that will take both the client side and the server side faults into account.

References

[1] I. Bourdonov, A. Kossatchev, V. Kuliamin, and A. Petrenko, “UniTesK test suite architecture,” Proc. FME 2002, LNCS 2391, Springer-Verlag, 2002, pp. 77-88.

[2] I. Bourdonov, A. Kossatchev, V. Kuliamin, and A. Petrenko, “UniTesK: Model Based Testing in Industrial Practice,” Proc. the 1st European Conference on Model-Driven Software Engineering (ECMDSE), Nuremberg, Germany, Dec. 11-12, 2003, pp. 55-63.

[3] V. Kuliamin, A. Petrenko, N. Pakoulin, I. Bourdonov, and A. Kossatchev, “Integration of Functional and Timed Testing of Real-time and Concurrent Systems,”. Proc. of PSI 2003, LNCS 2890, Springer-Verlag, 2003, pp. 450–461.

[4] A. Khoroshilov., “Specification and Testing Systems with Asynchronous Interfaces,” Preprint of the Inst. for System Programming, Russ. Acad. Sci., Moscow, 2006.

[5] V. Kuliamin , A. Petrenko, and N. Pakoulin, “Practical Approach to Specification and Conformance Testing of Distributed Network Applications,” Proc. ISAS'2005, Berlin, Germany , April 25-26, 2005, pp. 60-73.

[6] N. Pakulin and A. Khoroshilov, “Development of formal models and conformance testing for systems with asynchronous interfaces and telecommunications protocols,” Programming and Computer Software, vol. 33, number 6, Nov. 2007, pp. 316-335, doi: 10.1134/S0361768807060035.

[7] XMLHttpRequest object specification: http://www.w3.org/TR/XMLHttpRequest/

[8] M. Mahemoff, Ajax design patterns. Sebastopol, CA: O’Reilly Media, Inc, 2006

[9] Nicholas C. Zakas, Jeremy McPeak, Joe Fawcett, Proffessional Ajax 2nd edition. Indianapolis, Indiana: Wiley Publishing, Inc., 2007

[10] Wiki on AJAX containing a comprehensive collection of AJAX design patterns:
http://ajaxpatterns.org/

[11] JavaTESK toolkit for testing Java applications with UniTESK:
http://www.unitesk.com/

[12] Selenium Remote Control web application functional testing tool: http://seleniumhq.org/projects/remote-control/

[13] Ali Mesbah and Arie van Deursen, “Invariant-based automatic testing of Ajax user interfaces,” Proc. the 31st International Conference on Software Engineering (ICSE'09), IEEE Computer Society, 2009, pp. 210-220.

[14] A. Marchetto, P. Tonella, and F. Ricca, “State-based testing of Ajax web applications,” Proc. 1st IEEE Int. Conference on Sw. Testing Verification and Validation (ICST’08), IEEE Computer Society, 2008, pp. 121-130.

[15] A. Marchetto, P. Tonella, “Search-based testing of Ajax web applications,” Proc. the 2009 1st International Symposium on Search Based Software Engineering, May 13-15, 2009, pp. 3 – 12.

[16] G. Meszaros, “Agile regression testing using record and playback,” Proc. the conference on Object Oriented Programming Systems Languages and Applications, 2003, pp. 353-360.

[17] Rational Functional Tester web application functional testing tool:
http://www-01.ibm.com/software/awdtools/tester/functional/

[18] SWEA web application functional testing tool:
http://webiussoft.com/

[19] JUnit unit testing framework:
http://www.junit.org/

Service-oriented approach to integration testing in distributed systems

V.N.Fedotov

Software Engineering Department
Institute for System Programming
vnfedotov@acm.org

Abstract

Development of the service-oriented technologies has turned the market of integration platforms, becoming new challenge for IT experts worldwide. Key principles of SOA paradigm do not allow to apply the solutions tested on the client-server architecture. On the other hand, SOA offers its own set of tools to cope with majority of problems. In this paper it is shown how to effectively troubleshoot the distributed system by means of service-oriented approach.

1 Introduction

With development of communication technologies, business process automation becomes increasingly urgent task both for business, and for the state. Seen not so long ago just as a tool to reduce costs, now BPA is a key to survival in a changed world where internet and mobile services rule.

Majority of companies already has ERPs, CRMs, HRMs and many other applications managing their daily business. But do these applications automate company business processes? They are certainly providing some automation but it has nothing to do with business process, as they cannot interact with each other. So, BPA is only possible when there is a way to 'glue' applications together. In literature such way often called 'enterprise application integration' (EAI[1]).

There isn't a 'right' solution for application integration. As different integration concepts were developed, they didn't replace each other, instead they're competing until now. You still can find message-oriented middleware from early 90-s, integration brokers, and even CORBA, which is almost dead, because it didn't survive the competition with the newest trend - service-oriented architecture (SOA[2]).

While using many ideas from CORBA, SOA has taken completely different technological approach by using XML and J2EE to simplify development of integration components dramatically. In SOA these components are called 'services', as they mostly resemble web-services, but are arranged in a way defined by SOA guiding principles. These principles actually form SOA paradigm, as they distinct service-oriented architecture from bunch of web-services, just like OOP principles distinct object-oriented code from just some C++ code.

2 Enterprise as a Black Box

When thinking about how to make two applications interact with each other, you probably will come out with something like client-server approach, which is typical for Internet and various network applications.

Unfortunately, client-server architecture doesn't work in EAI as there is a tens of applications interconnected with each other in different ways by different protocols. In that case client-server approach requires each application to contain an adapter for every other application, causing serious flexibility and scalability issues.

These issues are the reason of using middleware solutions. Middleware platform are in the middle, obviously, of every interaction, mediating and routing messages between applications. For testing it means that instead of one interaction, you need to test two interactions, which isn't too bad. But SOA make matters worse, as it compose middleware platform from dozens of interconnected services. Now you need to test twenty or thirty interactions. As SOA solution becoming mature, its becoming more complex, with composition services and orchestrations, so testing efforts are only increasing.

Common solution to lower the efforts putted in integration testing is simple - don't do integration testing. Without advanced testing approach, this is the only way to cope with tight release schedule typical for SOA projects.

3 Test stubs as a solution

Of course there are better solutions. How to test integration in a distributed system - isn't a most recent question. It is also having a simple answer: instrument SUT in a such way so that interactions between its components became transparent. But there is actually quite a broad choice of technical approaches.

Logging is a most simple approach. Analysis of transactions journal provide all the necessary information about way that components interact. However logs can be pretty hard to reach, too big to read and it's impossible to analyze them automatically.

Testing tools traditionally offer a different solution - test stubs. Stubs form a virtual environment, surrounding each component and giving a possibility to inspect all external links of a component under test. However, stubs are way too invasive, as they are literally muffling external links, thus it is impossible to test entire business process. Also, test stubs are inaccessible from the test scripts, so all assertions must be contained within the test stubs themselves, thus test logic became sprayed over test environment. Because of that it is nearly impossible to support regression test sets in a virtual environment.

Various academical papers[3][4] propose instrumenting SUT by way of instrumenting its components to provide more information about interactions with other components and make that information accessible from test scripts. Unfortunately, such instrumentation requires a complete revamp of release process adopted by enterprise, thus making that approach inappropriate for most companies.

4 Alternative

As an alternative to the solutions described above we propose a new way to test application integration in a distributed system. Our solution offer similar approach, but entirely different technical implementation. In short it can be summarized as 'connect everything to ESB[5]'.

We propose to use ESB as a universal proxy for every interaction in a SUT, thus providing us with capability to monitor and control these interactions. Using of ESB also grants access for testing tools through JMX and event queues, which provide us a way to automate analysis of interactions inside the SUT and even modify messages on the fly to reach broader test coverage.

These are the key advantages of proposed solution:

- ESB provides transparency for all interactions between SUT components;
- test assertions are located in one place;
- tests can be easily automated;
- the minimum quantity of tests covers a maximum quantity of SUT components;
- thanks to a uniform configuration provided by ESB, the test environment becomes more flexible and controlled;
- testing process is focused on business process, so there is a minimum risks of occurrence of late integration defects.

5 Conclusion

In this paper we've shown a way to use service-oriented technologies for creation of flexible test environment, which allows to simplify integration testing of distributed systems by adding necessary transparency to interactions between components of SUT.

On the basis of the proposed approach it is planned to create the completed methodology of testing distributed systems based on service-oriented architecture, having presented original techniques of construction of test scenarios and carrying out a performance testing.

References

[1] Gregor Hohpe, Bobby Woolf. *Enterprise Integration Patterns.*

[2] OASIS Reference Model for Service Oriented Architecture 1.0 http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf

[3] Cesare Bartolini, Antonia Bertolino, Sebastian Elbaum, Eda Marchetti. *Whitening SOA testing.*

[4] Youngkon Lee. *Double layered SOA test architecture based on BPA - simulation event.*

[5] David Chappell. *Enterprise Service Bus.*

GPU-BASED EXTENDED CELLULAR MODEL IMPLEMENTATION

A. A. Emelyanov, R. M. Dmitrienko
Special Computing Technologies LLC
N.Novgorod, Russian Federation
office@hopcomp.net

A Cellular Automaton (Cellular Model) is a convenient and efficient way to solve a broad class of problems that belong to different areas of science, including (but not limited to) physics, math, chemistry, biology. A lot of natural phenomena and common tasks that are described via differential equations are easily modeled this way. But in spite of obvious advantages, CA-based algorithm implementations for common CPUs are highly parallel and hence extremely resource-intensive. Thus, the idea of SIMD GPU-based CA implementation looks feasible. In this paper, we will discuss the problems we faced while developing GPU-based cellular model implementation, and the ways we solved them.

CA – Cellular Automaton

SIMD – Single Instruction Multiple Data

GPU – Graphics Processing Unit

I. INTRODUCTION

A Cellular Model (a Cellular Automaton) is a set of cells and their corresponding states. These cells form a grid; a certain set of rules determines each cell's new state depending on the nearby cells' current states at any given moment of time. The most common are the automata whose cells' new states are determined only by their own current state and those of corresponding adjacent cells. Cubic grids are the most popular, but the irregular grids are possible as well. Different kinds of cellular automata are studied: synchronous and asynchronous, deterministic and probabilistic, with regular and irregular cell disposition. An Extended Cellular Model is a generic mathematical model of a Cellular Automaton that encapsulates all of the aforementioned kinds of Cellular Automata.

One of the most important properties of an Automaton is the “Locality of the rules”. It means that the cell's new state is only determined by the adjacent cells and probably by itself. This fact allows us to suppose that it is possible to implement a Cellular Automaton using a SIMD processing unit, such as a GPU. First we'll discuss the limitations of the SIMD architecture (as implemented by a GPU) that are related to the problem of efficient Cellular Automata implementation.

II. PROBLEM DEFINITION AND ANALYSIS

When it comes to implementation of highly parallel GPU-based algorithms, the most important limitation that arises is caused by the memory access collisions. A memory access collision is a situation when more than one of the active threads attempt to access memory either for reading, or for writing. At the hardware level, this problem is usually partially solved by the means of memory organization: either a collision-free memory paragraph access method, or a collision-free separated memory bank access, where each bank is spread across the whole address space. Thus, the problem of efficient algorithm implementation is solved by choosing an optimal layout of input and output data in the memory and an optimal sequence of memory access attempts by the threads.

From the software implementation point of view, a Cellular Automaton is a closely coupled net of finite state machines (the cells), where the connections represent the input and output data of these FSMs. From the programming point of view, the only difference between a Synchronous and an Asynchronous Cellular Automata is the global cell states' buffer that is required by a Synchronous CA. Considering the fact that the GPU threads access memory at different moments of time, it is possible to implement both Synchronous and Asynchronous Cellular Automata equally efficiently (when all of the other conditions are the same). A Movable Cellular Automaton may be reduced to an ordinary Cellular Automaton by adding the cell's coordinate to the cell's state variables list, while establishing connections between each and every cell. Thus, instead of recalculating cell adjacency before new cell states' calculation, it is safe to assume that every other cell is a neighbor of the current cell, and hence it becomes possible to use these “neighbor”'s coordinates for state recalculation. This kind of algorithm transformation does not increase the processing load, as the quantity of operations remains the same, only their order is changed. Implementing determined and probabilistic Cellular Automata is even simpler. Probabilistic state recalculation algorithms are slightly more complex, but since the random numbers (which are essential in this case) are calculated independently for each cell, and it is possible to use GPU registers to store such small amounts of data, memory access collisions are avoided. Furthermore, it is possible to precalculate the random numbers' sequence and

store it optimally in the memory before launching the computing algorithm.

So, the only problem of the GPU-based Cellular Model implementation that remains is the irregularity of the grid. Let's suppose that the cells' connections are predetermined and bear no regular structure. This is highly likely to cause collisions when threads will attempt to access memory areas that store the parameters of the cells, since the order of access attempts is not known beforehand (it depends only on the Cellular Automaton's cells' interconnections). It is also important to note that this problem is irrelevant for movable Cellular Automata, as the cells' interconnection structure is unknown beforehand at the beginning of every calculation step (and thus must be recalculated); so, it is possible to connect each and every cell, while maintaining the same efficiency. However, this is not the case for Automata with irregular grids: there can be much less actual connections than the squared number of cells, so that the SIMD architecture-based implementation's efficiency will drop significantly.

Based on the aforementioned suggestions, the problem that we need to solve is actually a problem of designing of an efficient implementation of a Cellular Automaton with irregular connections' structure. Implementing a Cellular Automaton with regular connections' structure is just a special case of this problem.

III. The proposed solution

Each cell contains its state variables and a list of neighboring cells. In-memory data layout is done this way:

a_i - cell i

U^i - a set of neighbors of cell i

B^i - a set of cells, for which the cell I is a neighbor

$a_j \in B^i \Leftrightarrow a_i \in U^j$

$M(A)$ - cardinality of the set A

Step 1. The cells are ordered by the number of cells that a cell is adjacent to, beginning from the greatest number.

$$M(B^i) \geq M(B^{i+1})$$

Step 2. Cells' state variables are laid out this way: the first variables of cells' states are stored beginning from a memory paragraph's boundary, then the second variables are stored beginning from the next paragraph's boundary, and so on.

p_j^i - state variable j for then cell i

$S(p_j)$ - size of the state variable j

PH - size of a memory paragraph

N - quantity of cells

In-memory layout of parameters:

$$p_1^1\ p_1^2\ p_1^3\ p_1^4 \ldots p_1^N\ SPACE_1$$
$$p_2^1\ p_2^2\ p_2^3\ p_2^4 \ldots p_2^N\ SPACE_2$$
$$\ldots$$
$$p_K^1\ p_K^2\ p_K^3\ p_K^4 \ldots p_K^N\ SPACE_K$$

where K is the number of state variables.

SPACE – memory boundary alignment space

Size of

$$SPACE_j = \left(PH - \left(\left(S\left(p_j\right) * N \right) modPH \right) \right) modPH$$

where a mod b is a reminder of a/b.

It does not matter whether all of the state variables fit into the same memory paragraph or not. Nevertheless, it is important to keep the number of used paragraphs low while avoiding memory access collisions.

Step 3. The list of cell's neighbors is a sequence of cells' numbers (according to the one we've got at the Step 1); the fields that correspond to the non-neighboring cells are filled with 0 (-1 may be used if 0 is taken by the first cell).

A separate thread is launched per each cell. A single thread may correspond to a number of cells in case the total amount of cells exceeds the thread count limit. Each thread reads its corresponding cell's neighbors' states and recalculates the cell's state variables accordingly. If the neighboring cell is marked as 0 (i.e. there's no such neighbor), the thread enters waiting state until a valid cell is found.

Memory access collision avoidance and processing efficiency are achieved by the fact that the waiting threads attempt no memory access, and the number of such threads may grow significantly while the neighbor list is being processed. It is worth mentioning that SIMD threads are always synchronous and hence are put into waiting state only when they miss a conditional branch of the code which is entered by some other threads (they remain in the waiting state until others exit the branch). Thus, this algorithm may become especially efficient when applied to Cellular Models with irregular cells' disposition (where the efficiency depends significantly on the structure of connections). A major performance gain is also possible for Asymmetric Cellular Automata, where the density of cellular interconnections is distributed unevenly and has certain distinct peaks and troughs.

IV. Conclusion

We have described our approach to the GPU-based Cellular Models implementation. This approach is frequently employed for our development which is done in conjunction with subject matter experts from different areas of science, such as physics, chemistry, meteorology and so on. We have implemented it using NVidia GPUs (as a part of the CUBLIC(TM) project), and the estimated performance gain (as compared to the transformation of the source Automaton into an Automaton where each and every cell is connected to every other cell) ranges from 2 to 5 times.

References

[1] Kalgin Konstantin Victorovich KalginKV@gmail.com Supercomputer Software Department (SSD, ssd.sscc.ru), “Cellular Aitomata on GPU” *http://ssdonline.sscc.ru/kalgin/cuda/ca.gpu.pdf*

[2] Psakhie, S.G.; Horie, Y.; Korostelev, S.Yu.; Smolin, A.Yu.; Dmitriev, A.I.; Shilko, E.V.; Alekseev, S.V. (1995). «Method of movable cellular automata as a tool for simulation within the framework of mesomechanics». Russian Physics Journal 38 (11)

[3] Psakhie, S.G.; Horie, Y.; Ostermeyer, G.P.; Korostelev, S.Yu.; Smolin, A.Yu.; Shilko, E.V.; Dmitriev, A.I.; Blatnik, S.; Spegel, M.; Zavsek, S. (December 2001). «Movable cellular automata method for simulating materials with mesostructure». Theoretical and Applied Fracture Mechanics 37 (1-3)

ParaLab – Visual Way to Parallel Programming

The software system for investigating the parallel algorithms

Anna Labutina, Victor Gergel
Nizhny Novgorod State University
Nizhny Novgorod, Russia
e-mail: anna.labutina@cs.vmk.unn.ru

In this paper we introduce a software system which allows to carry out and visualize computational experiments for studying and researching the parallel algorithms of solving complicated computational problems in imitation mode on one single sequential computer. User can "assemble" a parallel computational system of cluster type that consists of multiprocessor and multicore nodes connected with the network, set up the problem to be solved, carry out the parallel solving algorithm, collect and analyze the results of computational experiments. To estimate the execution time of parallel method on current hardware system we use the sophisticated models. For every implemented parallel method we proved the theoretical estimations of the execution time by comparing the real time of the execution on the NNSU high performance cluster with the time, that can be calculated using the model.

High performance computing, parallel computing, parallel computations modeling, cluster, multiprocessor architecture, multicore architecture.

I. Introduction

The development of the computer architecture and network technologies, together with investigations of new time-consuming scientific and applied problems that demand massive computations showed high necessity of parallel computations, made high performance computing the cornerstone of programming and computational technology.

But despite of science needs and the actuality of parallel computations, so far they are not as widely used as it was predicted. One of the possible reasons is the necessity of developing new parallel algorithms to solve the new computationally intensive problems. It is well-known that the speedup of solving the task on parallel computational system can only be achieved when the algorithm is divided into set of independent processes that can be run simultaneously. The other reason is that the debugging of parallel code is a high complexity problem, which makes it necessary to fully understand the behavior of the system of computational processes run in parallel. That is why competence in modern high perforamnce computational system design trends, in new tools developed to achieve parallelism, the ability to create models, methods for solving the problems in parallel are the major qualities for specialists in applied mathematics, computer science and information technologies.

One more reason why "real" parallel programs are so hard to understand is that there is no simple tool to visualize their behavior. It is getting even worse now when unified interfaces are used to organize an access to multiprocessor computational systems. Such interfaces allow user to put the task into the queue remotely, and then after a while get the result of program execution stored in the file.

An access to real parallel computational system is not necessary for learning the basic principles of parallel algorithm's execution. Textbooks will help to gain the sufficient level of theoretical knowledge. ParaLab system introduced in this paper will help to get the practical expertise.

II. Working with ParaLab

While working with ParaLab user has an access to a wide range of tools to set the computational experiment parameters. She can model the computational system, chose the problem, carry out the parallel algorithm, collect and analyze the results of computational experiments.

A. *Modeling the Parallel Computational System*

ParaLab allows to simulate the parallel computational experiments execution on multiprocessor (*SMP*) and *multicore* architectures. The computational system appears to consist of the *computational nodes* (*computers*). Each node has one or more *processors*, and each processor has one or more *cores*. The ParaLab system architecture doesn't limit the maximum amount of cores in processor and processors in one node, but for the sake of visualization we limit the number of cores to be equal to 1, 2 or 4 and number of processors to be 1 or 2.

In order to simulate the computer system, it is necessary to determine the network topology, the number of computational nodes, the number of processors and cores on one node, the performance of each core, and the characteristics of the communication network (latency, bandwidth and data communication method). It should be noted that the computer system is assumed to be homogeneous in the ParaLab system, i.e. all the computational nodes have the equal amount of processors, every processor consists of the same number of cores, cores possess equal performance, and all the communication lines have the same characteristics.

The data communication network *topology* is defined by the structure of communication lines among the computer

system nodes. The system ParaLab supports the following network topologies: farm, ring, star, mesh, hypercube, full graph (clique).

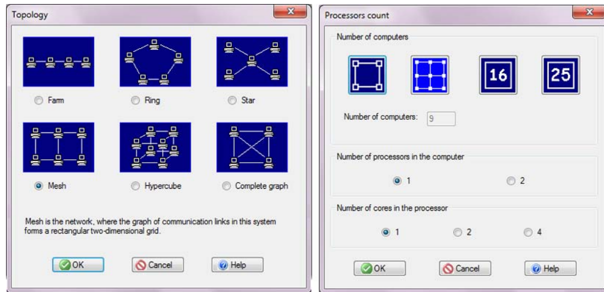


Figure 1. Dialog windows to set up the computational system parameters

The system ParaLab allows user to set the desirable number of nodes for the selected topology. The choice of the system configuration is performed in accordance with the type of the topology used. Thus, for instance, the number of processors in a two-dimensional grid must be a perfect square (the sizes of the grid both vertically and horizontally are the same), while the number of the processors in a hypercube must be a power of 2.

The *performance* of the core in the ParaLab system is measured by the number of floating point operations per second (flops). It should be noted that to estimate the execution time of the experiment, it is assumed that all the computer instructions correspond to the same floating point operation.

The time of data transmission among the processors determines the *communication overhead* of the parallel algorithm execution in a multiprocessor system. The main set of parameters, which makes possible to estimate the data communication time, contains the following values:

- *latency* (α). It is the time, which characterizes the duration of preparing a message for transmission, as well as the duration of the searching for the route in the network, etc.;
- *network bandwidth* (β). It is defined as the maximum amount of data, which can be transmitted in a certain unit of time through a data communication channel. This characteristic is measured, for instance, in Mbits per second.

Among the data communication methods, implemented in ParaLab, there are the following two well-known communication methods (see, for instance, Kumar, et al. (1994)). The first method is aimed at passing *messages* as indivisible information blocks (*store-and-forward routing* or *SFR*). The second communication method is based on representing the transmitted messages as a set of information blocks of smaller sizes (*packets*). As a result, the data transmission may be reduced to passing packets. In case of this communication method (*cut-through routing* or *CTR*) the transit processor may perform transmitting the data along the chosen route directly after the reception of the next packet without waiting for the termination of receiving all the message data.

B. *Selecting the Problem and the Parallel Method*

The following widely used parallel algorithms applied to solving complicated computational problems in various scientific and technical applications are implemented in the system ParaLab: the algorithms for data sorting, the algorithms for matrix operations, the algorithms for solving the systems of linear equations, graph processing, the algorithms for solving differential equations in partial derivatives and the algorithms for global multiextremal optimization.

As a rule, for every task there are several solving parallel methods implemented. For the matrix-vector multiplication task we implemented algorithms based on block, row-wise and column-wise matrix decomposition. For the matrix multiplication problem there are parallel Fox's and Cannon's algorithms and the algorithm based on striped matrix decomposition. For the problem of solving the system of linear equations we present the parallel variants of Gauss method and conjugate gradient method. For the sorting problem we implemented parallel variants of bubble sort, Shell sort and quick sort. For the graph processing task there are parallel algorithm for building minimal spanning tree, Dijkstra's and Floyd's algorithms for shortest paths problem. For the problem of solving the differential equation in partial derivatives we have parallel Gauss-Seidel algorithm. Parallel index method is implemented for the problem of multiextremal optimization.

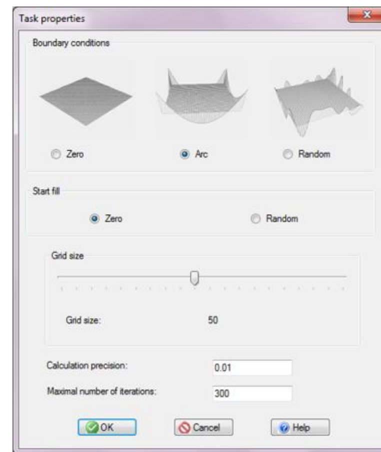


Figure 2. Dialog window to set the parameters for the problem of solving differential equation in partial derivatives

The main problem parameter in the system ParaLab is the initial data amount. For the problem of sorting this is the size of the array. For the matrix operations and the problem of solving a system of linear equations this is the order of the matrices. For the problem of processing graphs this is the number of graph vertices.

User can set additional parameters for some types of problems. For example there is a possibility to chose the boundary conditions for the problem of solving the differential equation in partial derivatives, to chose the type of function for the problem of multiextremal optimization, to

create a graph with the help of built-in graph editor for the graph processing problem.

C. *Carrying out the computational experiment*

ParaLab provides various forms of graphical demonstration of parallel computation results in order to observe the process of carrying out a computational experiment of solving complicated time consuming computational problems. Before the parallel algorithm execution user can set the visualization parameters for demonstration speed, the mode of communication operation visualization, the required level of details to be shown.

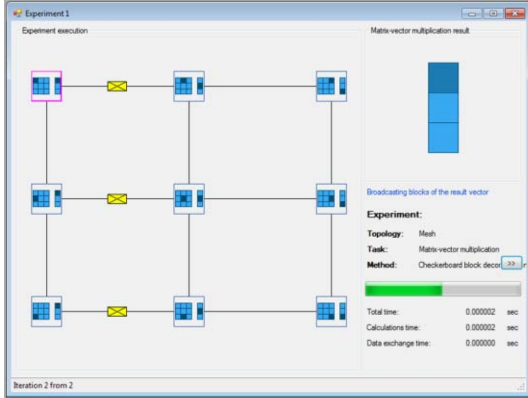


Figure 3. The window of the computational experiment while solving the problem of matrix-vector multiplication

The system ParaLab provides different schemes of carrying out experiments to give convenient possibilities for studying and using parallel algorithms of solving complicated computational problems. Problems may be solved in the sequential execution mode, in the time sharing mode with the possibility to simultaneously observe the algorithm iterations in all the computational experiment windows. Carrying out experiment series that require long-continued computations, may take place in the automatic mode with the possibility of saving the results of solving problems in the experiment log. Experiments may be also carried out in the step-by-step mode.

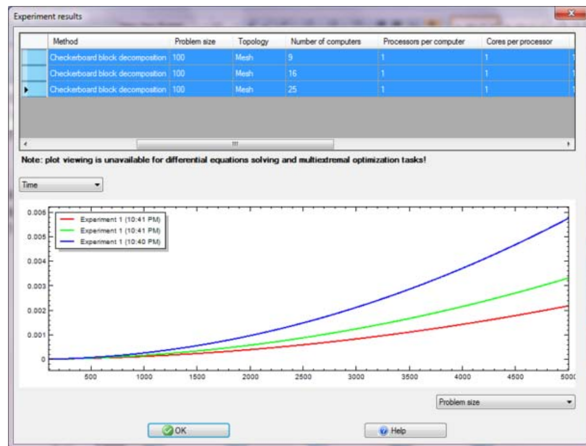


Figure 4. The experiment log window

D. *Accumulating and Analyzing the Experiment Results*

To accumulate the results of the executed experiments, ParaLab provides a special memory, which is hereinafter referred to as the experiment log. The results are stored in the experiment log by the system automatically. Accumulated results can be used for observing and analyzing. The stored data can be also taken to restore the previous state of the experiment – that allows to rerun the experiment. Also it allows to continue computations from the suspended state.

Saving the current experiment in a file, all the accumulated results will be also saved.

For the experiments saved in the experiment log, we build the graph that shows how the execution time and the speedup depend on problem and computational system parameters. These graphs are built in accordance with the theoretical models we use to estimate the execution time of the parallel algorithm.

III. MODELING PARALLEL COMPUTATIONS

A. *Model for the local computations*

While creating a model to estimate the time of local computations we assume that this time is the sum of the calculation time and the memory access time:

$$T_1 = T_{calc} + T_{mem} \quad (1)$$

Here the calculation time is the result of multiplication of the executed operations number N by the time of one operation execution τ. The memory access time is the result of division of the maximum amount of data M by the memory bandwidth β_{RAM}. To make the estimation more precise we should consider that the data comes from memory not in byte-by-byte mode but in full cache lines, the length of one cache line is equal to L bytes. The worst case is when every data element should be downloaded from the memory and it falls in the separate cache line. Thus, the model for the local computations execution time can be the following:

$$T_1 = N \cdot \tau + L \cdot M/\beta_{RAM} \quad (2)$$

We should also consider the RAM latency α_{RAM} that can significantly influence the time of computations:

$$T_1 = N \cdot \tau + M \cdot (\alpha_{RAM} + L/\beta_{RAM}) \quad (3)$$

This model doesn't reflect the modern processor architecture, where the processor has small but fast local memory, which is called cache memory. In order to get the fast access to the necessary data this data is downloaded from RAM to cache before the computations with the use of different prediction algorithms. This download can be performed simultaneously with computations and doesn't affect the time of computation execution. The situation when the necessary data is not in the cache and the processor should wait for them to be downloaded from RAM is called *cache miss*. To make the model of computational time more precise we need to know the number of cache misses

appeared during computations. With this new information we can correct the time that the processor spends on waiting for the data to be downloaded from the RAM:

$$T_1 = N \cdot \tau + \gamma \cdot M \cdot (\alpha_{RAM} + L/\beta_{RAM}) \quad (4)$$

where γ is the cache miss ratio (number of cache misses divided by the number of cache access operations), which can be theoretically estimated.

Thus, to estimate the time of local computations execution we need to know:

- α_{RAM} – RAM latency,
- β_{RAM} – RAM bandwidth,
- γ – cache miss ratio,
- τ – the time of one operation execution.

To make a decision about the model accuracy the computational experiments were carried out on the computer with the Intel core 2 quad Q6600 processor. The architecture of this processor includes first-level caches with the bandwidth of 153 Gb/sec and latency of 1,22 nsec. The RAM of the target system has a bandwidth of 12,4 Gb/sec and latency of 8,31-80 nsec. The algorithm of matrix-vector multiplication was executed. The code for this algorithm is:

```
for (i=0; i<Size; i++) {
  pResult[i] = 0;
  for (j=0; j<Size; j++)
    pResult[i] +=
      pMatrix[i*Size+j]*pVector[j];
}
```

To calculate the time of one operation execution τ we measured the time spent on performing the algorithm for small object size, when matrix and vectors can fit in cache L1. We divide this time by the number of performed operations and get the time of one operation execution τ = 3,78 nsec.

TABLE I. COMPARISON OF THE EXPERIMENTAL AND THEORETICAL EXECUTION TIME OF THE MATRIX-VECTOR MULTIPLICATION ALGORITHM

Matrix Size	Experimental Time	Theoretical Time	Relative Error
100	0,0001	0,0001	0,0062
1000	0,0076	0,0076	0,0011
2000	0,0303	0,0304	0,0021
3000	0,0688	0,0685	0,0043
4000	0,1222	0,1217	0,0036
5000	0,1909	0,1903	0,0035
6000	0,2748	0,2740	0,0029
7000	0,3741	0,3729	0,0033
8000	0,4894	0,4872	0,0044
9000	0,6186	0,6164	0,0036
10000	0,7637	0,7611	0,0034

In current version of ParaLab the simplier model for estimating the time of local computation is realized. This model only uses the number of operations and the time of one operation execution τ. We plan to implement the described approach to local computations time estimation in the next version of ParaLab.

B. *Model for data passing operations execution*

The time necessary for transmitting data between the processors defines *the communication overhead* of the duration of parallel algorithm execution in a multiprocessor computer system. The basic set of parameters, which can help to evaluate the data transmission time, consists of the following values:

- *initializing time* (α) characterizes the duration of preparing the message for transmission, the search of the route in the network etc.;
- *control data transmission time* (t_c) between two neighboring processors (i.e. the processors, connected by a physical data transmission channel); to control data we may refer the message header, the error detection data block etc.;
- *transmission time of one data byte along a data transmission channel* ($1/\beta$); the duration of this transmission is defined by the communication channel bandwidth.

Let's consider *store-and- forward routing* (*SFR*). In case of this approach the processor, which contains a message for transmission, gets all the amount of data ready for transmission, defines the processor, which should receive the data, and initializes the operation of data transmission. The processor, to which the message has been sent, first receives all the transmitted data and only then begins to send the received message further along the route. The time of data transmission t_{comm} for the method of transmitting the message of m bytes along the route of length l is defined by the expression:

$$t_{comm} = \alpha + \left(t_c + \frac{m}{\beta}\right) \cdot l \quad (5)$$

If the messages are long enough, the control data transmission time may be neglected, and the expression for data transmission time may be written in a simplified way:

$$t_{comm} = \alpha + \frac{m}{\beta} l \quad (6)$$

Let's consider *cut-through routing* (*CTR*), when the receiving processor may send the data further along the route immediately after receiving the current packet without waiting for the termination of the whole message data transmission. The data transmission time in case of packet communication method will be defined by the following expression:

$$t_{comm} = \alpha + \frac{m}{\beta} + t_c \cdot l \quad (7)$$

If we compare the obtained expressions, it is possible to notice that in the majority of cases the packet communication leads to faster data transmission. Besides, this approach decreases the need for memory for storing the transmitted data. Different communication channels may be used for packet communication simultaneously. On the other hand, the implementation of the packet communication requires the

development of more complex hardware and software. It may also increase the overhead expenses (initialization time and control data transmission time). Deadlocks may also occur in case of packet communication.

C. The data passing operations in multiprocessor and multicore architectures

As it was previously mentioned, in ParaLab the computational system consists of computational nodes, the network links between them are determined by the topology (farm, ring, etc.). Every node has one or more processors, every processor consists of one or more cores. We assume that the internal links between cores in frame of one computer (busses) form the full graph topology.

To make the time estimation model easier we assume that the computations and data passing operations cannot overlap, which means that the computations stop when the cores are performing the data transmission, and vice versa.

Every collective data passing operation between cores can be divided into 3 stages:

1) Data transmission between cores in frames of one computational node and sending the data into the external network (via network adapters),

2) Data transmission between different computational nodes through the local network (in ParaLab if all the data was sent to the network from one core then it is visualized like one envelope, if the data was sent by different cores it is shown like passing the pile of envelopes),

3) Receiving the data from the network adapter by the different cores in frames of one computational node.

To estimate the time spent on data passing operation we need to know:

- the size of one data unit in bytes,
- the number of units being transmitted for every pair of cores that perform the data passing operation on the current iteration of the algorithm.

To calculate the final time of the communication operation we only take into account the time of the second stage (passing the data through local network). The time spent on transmitting the data through the bus is 3 to 4 degrees less than that.

D. An Example of Computational Experiment Time Estimation

Let's consider the complexity of the parallel algorithm for matrix-vector multiplication based on rowwise matrix decomposition. Every core performs the multiplication of the matrix stripe by the vector, each stripe has *n/p* rows, where *n* is the size of the matrix and *p* is number of cores. One scalar product of the matrix row and a vector involves *n* multiplications and (*n-1*) additions. Let's assume that the multiplication and addition have the same duration τ. Besides, let us assume that the computer system is homogeneous, i.e. all the processors of the system have the same performance. With regard to the introduced assumptions, the computation time of the parallel algorithm is:

$$T_p(\mathrm{calc}) = \lceil n/p \rceil \cdot (2n-1) \cdot \tau \qquad (8)$$

The 'all gather' operation is used to put the result vector on all the processes of the parallel program. This operation can be performed in $\lceil log_2 p \rceil$ iterations. At the first iteration the interacting pairs of processors exchange messages of size $w\lceil n/p \rceil$ bytes (w is the size of one element of the vector in bytes). At the second iteration the size becomes doubled and is equal to $2w\lceil n/p \rceil$ etc. As a result, the all gather operation execution time when the Hockney [2] model is used can be represented as:

$$T_p(\mathrm{comm}) = \sum_{i=1}^{\lceil \log_2 p \rceil} \left(\alpha + \frac{2^{i-1} w \lceil \frac{n}{p} \rceil}{\beta} \right) = \\ = \alpha \lceil \log_2 p \rceil + w \lceil n/p \rceil \left(2^{\lceil \log_2 p \rceil} - 1 \right) / \beta \qquad (9)$$

where α is the latency of data communication network, β is the network bandwidth. Thus, the total time of parallel algorithm execution is

$$T_p = \frac{n}{p}(2n-1)\tau + \alpha \log_2 p + \frac{w(n/p)(p-1)}{\beta} \qquad (10)$$

(to simplify the expression (9) it was assumed that the values *n/p* and $log_2 p$ are whole numbers).

Let us analyze the results of the computational experiments carried out in order to estimate the efficiency of the discussed parallel algorithm of matrix-vector multiplication. Besides, the obtained results will be used for the comparison of the theoretical estimations and experimental values of the computation time. Thus, the accuracy of the obtained analytical relations will be checked.

The experiments were carried out on the computational cluster on the basis of the processors Intel XEON 4 EM64T, 3000 Mhz and the network Gigabit Ethernet under OS Microsoft Windows Server 2003 Standard x64 Edition.

The comparison of the experiment execution time T_p^* and the theoretical time T_p calculated in accordance with the expression (10), is shown in Table 2.

TABLE II. THE COMPARISON OF THE EXPERIMENTAL AND THEORETICAL EXECUTION TIME FOR PARALLEL ALGORITHM OF MATRIX-VECTOR MULTIPLICATION BASED ON ROWWISE MATRIX DECOMPOSITION

Matrix Size	**2 processors**		**4 processors**		**8 processors**	
	T_p	T_p^*	T_p	T_p^*	T_p	T_p^*
1000	0,0069	0,0021	0,0108	0,0017	0,0152	0,0175
2000	0,0132	0,0084	0,014	0,0047	0,0169	0,0032
3000	0,0235	0,0185	0,0193	0,0097	0,0196	0,0059
4000	0,0379	0,0381	0,0265	0,0188	0,0233	0,0244
5000	0,0565	0,0574	0,0359	0,0314	0,028	0,015

Now let us describe the way the parameters of the theoretical dependencies (values τ, w, α, β) were evaluated. To estimate the duration τ of the basic scalar computational operation, we solved the problem of matrix-vector multiplication using the sequential algorithm. The computation time obtained by this method was divided into the total number of the operations performed. As a result of

the experiments the value of τ was equal to 1.93 nsec. The experiments carried out in order to determine the data communication network parameters demonstrated the value of latency α and bandwidth β correspondingly 47 msec and 53.29 Mbyte/sec. All the computations were performed over the numerical values of the double type, i.e. the value w is equal to 8 bytes.

IV. CONCLUSION

The Parallel Laboratory software system (ParaLab) provides the possibility of carrying out computational experiments for studying and investigating the parallel algorithms of solving complicated computational problems. The system may be used for organizing a set of laboratory works on various courses in the area of parallel programming. This laboratory works will allow the learners to do the following:

- *to model multiprocessor systems* with various data communication network topologies,
- *to obtain the visual presentations of the computational processes and data communication operations* which take place in case of parallel solving various problems,
- *to construct the efficiency estimations* of the parallel methods to be studied.

In general, ParaLab is the integrated environment for studying and investigating the parallel algorithms of solving complicated computational problems. A wide set of available means to visualize the process of carrying out an experiment and to analyze the obtained results allows to study the parallel method efficiency on various computer systems, to make conclusions concerning the scalability of the algorithms and to determine the possible parallel computation speedup.

The processes of study and research realized by ParaLab are aimed at mastering the fundamentals of parallel computation theory. They allow the leaners to form the basic concepts of the models and methods of parallel computations through observation, comparison and analysis of various visual graphic forms demonstrated in the course of the experiment execution.

ParaLab is mainly used for training purposes. It may be used by University professors and students for teaching/studying and investigating parallel algorithms of solving complicated computational problems using the set of the laboratory works, applied to various courses in the area of parallel programming. ParaLab may be also used to conduct research into estimating the efficiency of parallel computations.

For those who only start to study the problem of parallel computations, ParaLab is very useful, as it allows them to master the parallel programming methods. Experienced users may use the system in order to estimate the efficiency of new parallel algorithms, which are being developed.

REFERENCES

[1] Foster, I. Designing and Building Parallel Programs: Concepts and Tools for Software Engineering. Reading, MA: Addison-Wesley (1995).

[2] Hockney, R. W., Jesshope, C.R. Parallel Computers 2. Architecture, Programming and Algorithms. – Adam Hilger, Bristol and Philadelphia (1988).

[3] Kumar V., Grama A., Gupta A., Karypis G. Introduction to Parallel Computing. – The Benjamin/Cummings Publishing Company, Inc., (1994).

[4] Quinn, M. J. Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill (2004).

[5] Rajkumar Buyya. High Performance Cluster Computing. Volume 1: Architectures and Systems. Volume 2: Programming and Applications. Prentice Hall PTR, Prentice-Hall Inc. (1999).

[6] Xu, Z., Hwang, K. Scalable Parallel Computing Technology, Architecture, Programming. – Boston: McGraw-Hill (1998).

[7] Voevodin V.V., Voevodin Vl.V. Parallel Computations. Saint-Petersburg, BHV (2002).

[8] Gergel V.P. Theory and Practice of parallel computations. BINOM (2007).

[9] Korneev V.V. Parallel Computational Systems. – Moscow, Knowledge (1999).

[10] Tanenbaum E. Computer Architecture. – Saint-Petersburg, Piter (2002).

A DSL for Hardware-accelerated Grid-based Scientific Models

Alexander N. Gavrilov

R&D Department
Lanit-Tercom, Inc.
Saint-Petersburg, Russia
E-mail: Alexander.Gavrilov@lanit-tercom.com

***Abstract*—This paper presents an ongoing effort to develop a domain specific language that would simplify exploiting hardware acceleration in grid-based scientific models. The project is being implemented in Common Lisp, which has the best built-in metaprogramming capabilities among industrial-level compiled programming languages. The DSL is applied to a hydrodynamic model of the atmosphere.**

***Keywords*-GPU computing; SSE; lisp; metaprogramming; DSL; simulation**

I. Introduction

Grid-based physics simulations are among the most easily parallelizable programming tasks, belonging to the category of so called embarrassingly parallel applications. They also are very close to the kind of calculations that modern GPU chips are designed to perform, and therefore should benefit greatly from using them.

Unfortunately, currently available free tools for programming NVidia GPUs are basically limited to the official C compiler [1] and accompanying libraries distributed by NVidia Corporation. Moreover, before PGI released a GPU-capable version of their commercial Fortran compiler [2] near the end of 2009, that was the only available tool at all.

The C-based tools are designed to provide very close control of the GPU, and thus require the programmer to manually manage GPU memory allocation, data alignment, textures and other low-level features of the platform. This makes GPU programming quite difficult to approach for people who are not professional system programmers.

Moreover, with existing tools it is still impossible to use the same code for both CPU and GPU, and fully exploiting the high performance computing features of modern CPUs in complex cases requires manual use of even more low-level SSE intrinsic functions. Which cases are to be considered complex depends on the capabilities of the compiler, but these usually include any loops that contain conditionals – even though the actual SSE instruction set is flexible enough to be able to express them[1].

All these issues were encountered during an attempt to speed up execution of an atmospheric model described in [3], which resulted in the creation of the DSL.

[1]This however requires dropping the lazy evaluation property of conditionals, i.e. works by evaluating both branches and then choosing which result to use.

II. Related Work

In addition to the official C toolkit, there exist a number of GPU programming libraries for other programming languages like Java[4] or Python[5]. However, they mostly only wrap the run-time library that is used to manage GPU memory, load code to the GPU and execute it. The actual GPU code still has to be written in C.

These wrappers already remove a large part of the burden and make GPU programming a lot more accessible, as user reports indicate. However, as noted above, due to the lack of native meta-programming support in the mentioned programming languages it is impossible to achieve seamless integration. The Python version goes further than the Java wrapper and allows C code to be expressed as a data structure tree instead of strings, but that is the limit of what can be done. This also means that these languages are not a very good choice for implementing a new DSL.

There are also some existing differential equation solver libraries, e.g. the OpenCurrent[6] C++ library created by a member of NVidia Research, but they are quite irrelevant to the task of implementing a specific unique calculation.

III. Methodology

A. Initial problem

The project started as an attempt to optimize a hydrodynamic model [3] of the atmosphere. The initial program was written in Fortran; Waterloo Maple was used to derive the mathematical expressions and generate most of the code.

As follows from their name, grid-based simulations are based on a representation of the state of the world produced by sampling the relevant physical quantities at regular space intervals. These samples are then grouped into 2D or 3D arrays depending on the model, and recalculated iteratively using a set of mathematical formulae as in-model time progresses.

Since most of the time the value of a cell being calculated depends only on array cells within a fixed nearby area, these calculations can be easily parallelized along array axes. Some models however improve computational stability by employing additional smoothing schemes that are serial along one of the axes; this is the case for the model in question. These schemes are an obstacle

to achieving the best performance and may result in a bottleneck on highly parallel architectures like the GPU.

Another major source of complications seen in this model is the use of a skewed grid, i.e. a scheme where physical coordinate grids used to sample different quantities are displaced in relation to each other by a certain fraction of the grid step (normally 1/2). This can be used to noticeably improve the precision of the model, but results in a certain technical difficulty:

The most straightforward and natural way for a human to handle a skewed grid scheme is to actually allocate a common grid with twice as many indexes, and use a sparse sub-grid for every specific quantity; any other way is very error-prone. However this naive approach wastes half of the memory bandwidth and prevents the use of some optimizations with strict memory positioning requirements, e.g. Intel SSE instructions.

The skewed grid issue is most naturally solved via automated transformation, i.e. meta-programming.

B. Implementation language

ANSI Common Lisp[7] is a dynamically typed multi-paradigm compilable programming language that is defined by an ANSI standard [8] finalized in 1994, and has multiple independent implementations. It allows writing code in functional, imperative or object-oriented style.

Like all languages of the lisp family, Common Lisp provides excellent meta-programming support via macros, which, unlike the identically called feature of the C preprocessor, are actually full-featured Lisp functions that are configured to be automatically called by the compiler, and operate on syntax trees. Macros can do anything that ordinary functions can do, including operations like creating temporary files and calling external programs.

The existence of macros transforms arbitrary meta-programming from something that usually requires modifying the compiler via patching or plug-ins, or implementing external preprocessors that operate on raw program text, into a relatively mundane task that is done to some extent by every proficient lisp programmer.

On the other end of the abstraction spectrum, almost every Common Lisp implementation provides facilities that can be used to access raw memory from lisp code and directly call external C libraries. One notable implementation in this regard is Embeddable Common Lisp([9], [10]), which compiles all lisp code by transforming it into C and calling the system's C compiler. This allows it to provide support for using C directly from lisp code, much like C compilers and inline assembler.

A couple of widely accepted utility libraries (e.g. [11]) can be used to call C functions from lisp code in an implementation-independent way. When the developers of the C library take care to maintain binary compatibility (as is the case with NVidia drivers), this makes the development and build process a lot more convenient by removing the need for any kind of wrappers written in C.

C. Target platform

At the hardware level modern GPU systems produced by NVidia are multi-core SIMD processors capable of executing divergent code[12], with native support for multi-threading and a hardware scheduler. Processor cores also include special units for texture fetching and interpolation, limited caches for textures & constants, and on-chip shared memory buffers for inter-thread communication. Ordinary memory was not cached until the recently announced Fermi architecture[2].

This hardware is exploited through a programming model that involves launching a large grid of threads, every one of which executes the same function to process one small chunk of input data. The threads are not required to follow the same execution path, but unless the pattern of branch divergence is matched to the underlying SIMD hardware it results in heavily reduced performance.

More specifically, the thread grid is partitioned into identical blocks of up to 1024 threads. Threads within one block can communicate via shared memory and a barrier primitive; different blocks are independent and may be scheduled by hardware in any order.

When a block starts execution, it is assigned to a processor core and gets a slice of its shared memory buffer and register pool[3]. These memory and register requirements determine how many blocks can run on one core at once. The threads of all running blocks are executed by the core's pre-emptive scheduler in static groups of 32 threads, which are officially called warps.

If all threads of a warp execute the same instruction, it is executed simultaneously in SIMD fashion; divergence forces the scheduler to serialize processing by temporarily disabling SIMD units for threads that don't need the instruction that it is going to execute.

In a similar fashion, if all threads of a warp access adjacent addresses in global memory, the scheduler can coalesce these operations into one large memory transaction, thus reducing the overhead caused by the lack of a traditional memory cache. Memory access transaction coalescing is heavily affected by data alignment.

GPU code cannot allocate or free memory, call functions by pointer or use recursive functions. GPU memory cannot be directly accessed from host code, but modern video cards allow mapping specially allocated regions of host memory into GPU address space.

D. Current state of the DSL syntax

This work on rewriting the original program has led to the creation of the following DSL:

1) Infix formula syntax: Common Lisp normally uses identical prefix syntax for function invocation and arithmetic operations. This is necessary for achieving the existing degree of meta-programming support, and tolerable

[2]Since Fermi-based cards are new technology that was not available for purchase at the moment of writing, this paper does not take that architecture into account.

[3]Each core has 16KB of shared memory, and, depending on the GPU version, 8192 or 16384 32-bit registers

in ordinary code, but not at all convenient for expressing huge mathematical expressions that are often used in the problem domain.

Fortunately, this can be fixed using the extensibility of the lisp reader (i.e. low-level code parser). A straightforward add-on with a simple expression parser allows writing infix expressions delimited by curly brackets:

```
{ NEW_DT[i,MW+1] := (TMP_ANU[MW-1]*TMP_EPS[MW+1]
                     +TMP_ANU[MW+1])
                  / (1.0-TMP_EPS[MW+1]*TMP_EPS[MW-1]) }
```

The parser recognizes arithmetic operators, array references, function calls and assignments. The syntax reflects the one used by Maple and many other computer algebra systems.

2) Virtual arrays: As a way to solve the skewed grid issue outlined above, the system is based on virtual arrays that have separate logical and physical dimension structure.

Dimensions may be arbitrarily reordered, and any particular dimension can be configured to either contain regular gaps, or distribute consecutive logical indexes along a hidden fixed dimension. Either of these special modes achieves contiguous arrangement in memory for logical indexes that are separated by a specific fixed stride.

```
(def-multivalue DR ((i 1 N1) :by 2) (k 1 MW :by 2)))
(def-multivalue PL ((i 1 N1) (k 1 {MW+1} :bands 2)))
```

Virtual arrays are accessed via an *iref* macro, which is similar to the standard *aref* function, but applies appropriate static transformations to its index arguments. The infix expression parser can be configured to generate either of those as a representation of its array reference syntax.

3) Virtual array index loops: A useful macro allows iterating over logical dimensions of a virtual array in a way that reflects the underlying physical structure.

```
(do-indexes OUT_U (i k)
  (format vel (formatter "~12,3E ~12,3E ~14,5E ~14,5E~%")
          (iref xcoord i) (iref zcoord k)
          (iref OUT_U i k) (iref OUT_V i k)))
```

The actual loop variables in the macro expansion correspond to physical dimension indexes, while the public logical indexes are expanded to expressions based on them. When said logical index variables are used as arguments to the iref macro within the loop body, the expressions are automatically simplified, and the resulting code appears as if it was written directly for the physical structure.

The order of index names in the loop header (which must match the names used in the virtual array definition) determines the loop nesting order. The iteration ranges are derived from the array definition. Extended syntax can be used to specify iteration direction and stepping, or reduce its range.

4) The compute statement: Finally, main code translation features of the library are encapsulated in the compute statement, which allows one to define a complete computation, composed of a target array, index range limits, iteration directions, and a mathematical expression to compute.

```
(compute PL ((i :skip (1 1) :step (* 2))
             (k :step (* 2)))
  { t3*t10/2.+2.*t1*_grp(t2/t3)*t12 }
  :parallel i
  :with { t1 := PL[i+1,k];
          ... })
```

Every such statement is converted to a separate GPU invocation. Additional clauses can be used to specify which index to focus parallelization efforts on, define inter-value dependencies, or pass additional optimization hints to the GPU compiler.

If the code translator fails to handle some parts of the expression, the statement is expanded into ordinary lisp loops. This ensures graceful integration of the functionality in the language, and, very importantly, allowed the optimized code generator to be implemented gradually *after* the program was fully converted from Fortran to Lisp.

E. Code optimizations

In order to ensure good performance of the generated code, the compute statement applies a number of optimization techniques. They can be classified into general expression restructuring and hardware-specific modifications.

1) Expression restructuring: The following generic simplifications are applied to the computation:

- Nested trees of associative-commutative operators are flattened and reordered to reveal the internal structure. This can be inhibited where needed[4] using a special _grp(...) syntax, which works like unbreakable parentheses.
- Very basic arithmetic expression simplification is performed.
- Branch-on-sign[5] expressions are simplified using hints about value signs supplied by the developer.
- Common sub-expression elimination with loop-invariant value extraction is performed. CSE exploits associativity of arithmetic operations.

2) Efficient GPU memory usage: The GPU has several ways to store and access data with varying characteristics. In order to achieve good performance it is necessary to use the best method for the task. For instance:

- The code generator supports using texture fetches to access arrays:
  Before Fermi, textures were the only way to achieve cached access to memory. In C using them requires rewriting all statements that access the data, but the generation-based approach reduces the complexity to a simple hint clause with a list of arrays that should be cached.
- Values that are the same for all threads of a block are stored in shared memory:

[4]Since floating-point calculations are not associative, automated reordering sometimes leads to severe precision degradation and situations like 0/0.

[5]The program that was used as a base for the experiment uses many expressions like (ifsign x 0.0 0.5 1.0). The branch value argument often contains constants and other sub-expressions that don't affect the sign.

Due to rather simple data access patterns combined with complex computations being common in the problem domain, shared memory is generally underused and thus cheaper than registers. Using it when appropriate increases the number of blocks that can run in parallel.

3) Serial inner loop representation: Due to memory coalescing requirements, the innermost loop must always be mapped to an in-block thread grid. If it has a serial data dependency, this results in a performance problem.

First of all, this means that the whole of the inner loop range must be handled within one block in order to allow data exchange through shared memory. This reduces the number of independent blocks.

The second issue is that the data-dependent part must still be executed serially using inter-thread synchronization barriers. The only way to improve performance here is to move as much code as possible out of this part. This especially applies to global memory access expressions.

In order to handle this case, the code generator includes special dependency analysis and code extraction functionality.

4) Operation reordering: Reducing the number of registers needed by every thread is instrumental in increasing the degree of parallelism and hiding memory latency. Unfortunately, optimal expression DAG reordering under register pressure is proven to be a NP-complete problem ([13], [14]), so no compiler can be expected to produce the best possible code in all cases.

The code generator applies a simple ordering heuristic that seems to help the official NVidia compiler produce better register mappings.

IV. Results

The above DSL was used to produce both SSE-ified CPU code (integrated via ECL's inline C feature), and GPU kernels. The CPU code, including the original program, was tested on an Intel Core 2 Quad 2.66GHz CPU[6], while the GPU version was executed using an NVidia GeForce GTX 275 graphics card.

The SSE version showed a 14x speedup against the original Fortran+Open-MP version. This can be explained in the following way:

- Expected 4x speedup due to the use of single-precision floating-point SIMD instructions.
- 40% speedup was observed after ifsign expression simplification was implemented.
- 2.5x speedup probably can be explained by better cache usage behavior, removal of function calls from inner loops, etc.

The GPU version was observed to be about 5 times faster than SSE code run on 3 of 4 cores. This number slowly grows as the array dimensions are increased; this is expected due to effects like diminishing utility of the CPU's caches.

[6]Only 3 cores were used for measurement to reduce the impact of other processes running on the same computer.

The raw peak FLOPS characteristics of the GPU allow one to expect a 26x maximum speedup. However, these numbers are computed under assumption that the MADD/-MUL double-issue feature is active; when it is not (and no code consists only of those two instructions), the expected speedup decreases to somewhere around 8-10x.

GPU code profiling and some calculations show that serialized inner loops are a noticeable local bottleneck in the current version.

V. Future work

These results show that the approach works for this particular model. The next major goal is to make the system more easily usable for implementation of similar programs, while also improving its performance. This requires fixing some shortcomings in the current implementation of the library.

A. New code generation back-end

The low-level C code generation back-end has been implemented in a hurry, and is too inflexible. This has become an obstacle in exploring better representations of serialized inner loops and other features.

This is being addressed by means of implementing a new, more generic, and separately usable GPU interface library for Lisp. The end result should be more or less similar to the existing bindings for Java and Python, but allow writing GPU code using normal Common Lisp syntax and a subset of the standard library with appropriate restrictions and extensions. Of course, the set of supported data types and operations will (at least initially) reflect the needs of this project.

The core part of the new library already works, but many specific features like textures are missing.

B. New DSL syntax

The syntax outlined above is both too general for the problem domain, and too specific in some technical matters like inter-iteration dependencies.

This prevents easily implementing features like non-SMP parallelism (e.g. multiple GPUs or clusters), and makes the code difficult to write for people who are not acquainted with the implementation of the library.

For example, it doesn't make much sense to declare different physical index orderings for different arrays. The array dimensions correspond to physical spatial dimensions, and it is more natural to select a global arrangement for them. Neither is it necessary to allow transposing indexes like this:

```
(compute foo (i j) { bar[j,i] })
```

Explicitly disallowing this would enable adding more code generation features.

One additional possibility is integrating with the Maxima[15] computer algebra system, which is also implemented in Common Lisp. Things like ifsign simplification more properly belong in the context of a CAS.

REFERENCES

[1] "CUDA Downloads," NVIDIA Corporation, 2010. [Online]. Available: http://developer.nvidia.com/object/cuda_download.html

[2] "PGI CUDA Fortran Compiler," The Portland Group, 2009. [Online]. Available: http://www.pgroup.com/resources/cudafortran.htm

[3] S. Kshevetskii and N. Gavrilov, "Vertical propagation, breaking and effects of nonlinear gravity waves in the atmosphere," *Journal of atmospheric and solar-terrestrial physics*, vol. 67, no. 11, pp. 1014–1030, 2005.

[4] "Java bindings for CUDA." [Online]. Available: http://www.jcuda.org/

[5] A. Klockner, "PyCUDA," Brown University. [Online]. Available: http://mathema.tician.de/software/pycuda

[6] J. Cohen, "OpenCurrent," NVidia Research. [Online]. Available: http://code.google.com/p/opencurrent/

[7] K. Pitman, *Common Lisp HyperSpec*, LispWorks, 1996. [Online]. Available: http://www.lispworks.com/documentation/common-lisp.html

[8] American National Standards Institute and Information Technology Industry Council, *American National Standard for information technology: programming language — Common LISP: ANSI X3.226-1994*. pub-ANSI:adr: pub-ANSI, 1996.

[9] G. Attardi, "The embeddable common Lisp," in *Papers of the fourth international conference on LISP users and vendors*. ACM, 1994, p. 41.

[10] J. J. Garcia-Ripoll, "Embeddable Common Lisp." [Online]. Available: http://ecls.sourceforge.net/

[11] J. Bielman and L. Oliveira, *CFFI User Manual*, 2005-2009. [Online]. Available: http://common-lisp.net/project/cffi/manual/index.html

[12] *CUDA Programming Guide 2.3*, NVIDIA Corporation, 2009. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf

[13] J. Bruno and R. Sethi, "Code generation for a one-register machine," *Journal of the ACM (JACM)*, vol. 23, no. 3, p. 510, 1976.

[14] C. Kessler, "Scheduling expression DAGs for minimal register need," *Computer Languages*, vol. 24, no. 1, pp. 33–53, 1998.

[15] *Maxima 5.20.1 Manual*, 2009. [Online]. Available: http://maxima.sourceforge.net/docs/manual/en/maxima.html

Formalization and enforcement of requirements to modular discrete-event simulation runtime

E. V. Chemeritskiy, K. O. Savenkov

This paper presents an extendable architecture for a discrete-event simulation runtime (DESR). The architecture is based on a set of logic blocks. Each block encapsulates a part of the DESR functionality and provides an interface to that functionality for other blocks. Differences in requirements that are imposed by different simulation problems are encapsulated in distinct logic blocks. Interface of each block is formally specified and there is a possibility of its automated check. Instances of the logic blocks are combined to get DESR for a particular simulation problem. Therefore, there is no need either in performance trade-offs or in a custom development of the DESR.

General Terms: **Discrete-Event Simulation, Simulation Runtime, Reuse, Modular Design**

I. Introduction

THIS paper is devoted to the development of the architecture for a DESR consisting of a set of logical blocks. The particular set of blocks, their interfaces and functionality give ability to build the runtime taking into account the requirements of a custom simulation task.

The Computer Systems Laboratory[1] (CSL) conducts multiple diverse research projects related to the simulation of distributed systems. The terms of such problems include the modeling of on-board systems (aviation, naval, automotive) computer networks and instruction set of processors. All these problems are focused on modeling the functionality of a computer system (data processing) and its performance as well as used apparatus – discrete-event simulation.

In CSL, such problems have been solving since 1982 and with increasing number of projects and directions it was decided to create a unified DESR [1]. It is based on a single approach to the computer systems simulation and uses a specialized language to describe simulation models.

However it became clear that this approach is not entirely suitable for the development of high-tech research projects. New challenges bring with them the need for simulation in a variety of detail levels. New requirements for scalability, performance and response time appear. General purpose solution is a compromise between expressive power and efficiency. Another problem is that it requires a tremendous effort to upgrade and maintain it in the grease condition in the future development.

As a result incompatible changes have been made in the unified DESR for each major project and currently there are several well-used variants of the same product.

This paper proposes the other way - to develop the architecture of the DESR as a collection of logical blocks. Each of these blocks encapsulates the functionality of the DESR and provides an interface to other units to use this functionality. A set of blocks has been designed so that the differences in the requirements for the DESR made by different tasks were encapsulated in separate blocks, retaining the overall structure of the DESR. The option to compose a DESR from needed copies of the different blocks gives ability to create a DESR configured to solve a custom simulation task with no need to compromise in terms of system performance and without wasting the developers' effort to create and support a new DESR.

This paper is based on a comparative analysis of different editions of the runtime DYANA used in several projects: 1) functional simulation of on-board marine systems [3], 2) hardware-in-the-loop simulation of on-board aircraft systems [4], 3) simulation of performance of neuroprocessor instruction set [5] and 4) on-board automotive information system simulation. Several DESR of well-known simulation systems have been also reviewed: AutoMod, SLX, Extend, SIMAN V, ProModel, GPSS/H.

Typically, the runtime consists of a set of modules [6],[7]. However, decomposition of a runtime into blocks based on the reduction of overhead costs for retargeting to a certain task hasn't been addressed yet. This problem has two important features:

1. The blocks must encapsulate functionality which is likely to change when adapting the runtime for a new simulation task;
2. Modification mechanisms for the runtime structure and the particular implementation of the runtime blocks should be researched;
3. The requirements for block implementations and monitoring mechanisms to ensure its compliance should be specified. The requirements may concern the block interface (has methods with a specific signature, is written in the same language, is connected as an object file, etc.), and its behavior.

Manuscript received March 22, 2010.

E. V. Chemeritskiy is with the Faculty of Computational Mathematics and Cybernetics, Moscow State University, Moscow, (e-mail: tyz@lvk.cs.msu.su).

K. O. Savenkov is with the Faculty of Computational Mathematics and Cybernetics, Moscow State University, Moscow ,(Phone: +7(495)939-46-71; fax: +7(495)939-25-96; e-mail: savenkov@cs.msu.su).

[1]The Computer Systems Laboratory of the Faculty of Computational Mathematics and Cybernetics of Moscow State University

The research results into a set of blocks encapsulating the differences of the examined DESR. On the basis of the proposed blocks a mathematical model of the DESR has been constructed. The paper describes in detail the functionality of each of the proposed block and the formal specifications of the interfaces of these blocks. Some mechanisms for its automated check have been proposed according to the analysis of designed specifications.

II. COMPONENTS OF THE RUNTIME

A generalized DESR scheme is proposed on the basis of a comparative analysis of different variants of DYANA runtime [2]-[5] and several well-known simulation systems: AutoMod, SLX, Extend, SIMAN V, ProModel, GPSS/H [8],[9].

The terms of this paper are borrowed from [10] with some generalization. The basic concepts of generalized DESR are *event* (a signal notifies DESR on the changing of model state), *logical object* (LO) (entity that is able to schedule events) and *resource* (provide LO with some services). Resources are presented by model time, cells of a memory (variables), clipboard information, semaphores, and so on. LO may delay event arrival until some condition depending on a state of the model resources set is satisfied. This condition is called a *delay condition*. As a result of the event arrival DESR can produce a number of actions called the *event handling*.

All event transactions take place in the *handling blocks*. Each block is a container for events generated by LO. Any handling block can be provided with an individual scheduler to properly rank the elements of the block. Runtime environment may contain several types of handling blocks:

1. *Current event block* (CEB) contains ready to be handled events.
2. *Future event block* (FEB) stores events with a delay condition depending only on the model time.
3. *Delayed event block* (DEB) contains events with complex delay condition depending on set of resources. There are two general approaches to condition check: polled waiting and related waiting. Respectively, there are two types of DEB according to these approaches: *related event block* (REB) and *polled event block* (PEB).

The work of DESR blocks is coordinated by the *dispatcher* throughout the model execution. The result message sequence is written to output *trace*.

More information about logical blocks composing the runtime environment and their functionality within the DESR is available in [11].

III. STATIC SEMANTICS

A. Static structure of the runtime

Runtime environment w connects a set of resources $\mathbb{R}$ and logical objects $\mathbb{L}$ of the model with the dispatcher d.

$$w = < R, L, d > \quad (1)$$

It is important to note that the resources and the logical objects are binded to the only dispatcher throughout the model execution.

B. Resource

The set of all resources is denoted by $\mathbb{R}$. For each resource $r \in \mathbb{R}$ a set of variables (memory cells) $V \subseteq \mathbb{V}$ and event container E are attached. The variables from the set $\mathbb{V}$ can take values from the set $\mathbb{D}$.

There are two types of resources: direct access resources $\mathbb{R}_{direct}$ and indirect access resources $\mathbb{R}_{indirect}$. Direct access resource keeps track of the values of related variables using the map $\mathcal{E}: \mathbb{R}_{direct} \to \mathbb{D}$. In this case the resource stores only a “local copy” of the original model data. Over a set of indirect access resources a map $\mathcal{D}_R: \mathbb{R}_{indirect} \to \{d, \emptyset\}$ to the set of dispatchers is defined. This mapping allows one to distinguish between attached and free model resources.

$$r = < V, E > \quad (2)$$
$$\mathcal{E}: \mathbb{R}_{direct} \to \mathbb{D} \quad (3)$$
$$\mathcal{D}_R: \mathbb{R}_{indirect} \to \{d, \emptyset\} \quad (4)$$

C. Message and trace

The output trace t is presented by the message sequence. The message alphabet is denoted by $\mathbb{M}$.

$$t = \{m_k\}_{k=1}^{\infty} \quad (5)$$

D. Event

Suppose there exists a set of all possible events $\mathbb{E}$. Each event has a trace message m and the type of event p. There are several event types in accordance with type of handling blocks intended to store this event $p \in \{CEB, FEB, PEB, REB\}$.

There are a number of maps defined over event set $\mathbb{E}$. The logical object arisen the particular event could be found by the mapping $\mathcal{L}: \mathbb{E} \to \mathbb{L}$. Suppose that there is a set of event attributes of all kinds $\mathbb{A}$. Then the mapping $\mathcal{A}: \mathbb{E} \to 2^{\mathbb{A}}$ defines an attribute set for each particular event.

Suppose there is a set of predicate symbols Pred defined over a set of resources $\mathbb{R}$ and depending on the values D of attached variables V. So the delay condition of event $e \in E$ could be presented as a formula of propositional logic (quantifier-free first-order logic) $\mathcal{C} \in \mathrm{Cond}$ over the set of predicates Pred.

The map changing the states set of resources $\mathfrak{R}$ and the states set of logical objects $\mathfrak{L}$ associated with the event $e \in \mathbb{E}$ is referenced as the modification $\mathcal{M}: \mathfrak{R} \times \mathfrak{L} \to \mathfrak{R} \times \mathfrak{L}$. There are only several ways to change model state:

1. To attach a new logical object $l \in \mathbb{L}$ to the dispatcher d,
2. To attach a new resource $r \in \mathbb{R}$ to the dispatcher d,
3. To change resource variables value r.V,
4. To change logical object activity limit value $\bar{a}$.

$$e = < m, p, \mathcal{C}, \mathcal{M}, \mathcal{A} > \quad (6)$$
$$\mathcal{L}: \mathbb{E} \to \mathbb{L} \quad (7)$$

Event type imposes restrictions on the delay condition. The delay condition of events of type "CEB" is always true. Events of type "FEB" essentially depends only on the model time and

types "PEB" and "REB" by contrast are independent of model time.

The notation $\mathrm{Var}(\phi)$ denotes a set of significant variables of $\phi \in \mathrm{Cond}$ and T denotes a resource containing the model time. Then, the following expressions are correct [2]:

$$\forall e\ (e.p = \mathrm{CEB} \rightarrow \mathrm{Var}(e.\mathcal{C}) = \emptyset) \tag{8}$$

$$\forall e\ (e.p = \mathrm{FEB} \rightarrow \mathrm{Var}(e.\mathcal{C}) = \mathrm{T}) \tag{9}$$

$$\forall e\ \big((e.p = \mathrm{PEB} \vee e.p = \mathrm{REB}) \rightarrow \mathrm{T} \notin \mathrm{Var}(e.\mathcal{C})\big) \tag{10}$$

The delay condition of each "FEB" event is true at a single point on the axis of time.

$$\forall e\ \Big(e.p = \mathrm{FEB} \rightarrow \big(\exists!\, \mathrm{D} \in \mathbb{D}\ (e.\mathcal{C}(\mathrm{D}) = \mathrm{true})\big)\Big) \tag{11}$$

Introduce a special operator $\mathcal{T}: \mathbb{E} \rightarrow \mathbb{D}$ to determine its value.

$$\mathcal{T}(e) = \mathrm{D} \in \mathbb{D}: \mathcal{C}(\mathrm{D}) = \mathrm{true} \tag{12}$$

There is also a dependency between the type of event and its attribute set. The events of one type have the same attribute set. Attribute set of the event with a type different from "CEB" includes all of its attributes.

$$\forall e_1 \forall e_2 \big(e_1.p = e_2.p \rightarrow \mathcal{A}(e_1) = \mathcal{A}(e_2)\big) \tag{13}$$

$$\forall e_1 \forall e_2 \big(e_1.p = \mathrm{CEB} \rightarrow \mathcal{A}(e_1) \subseteq \mathcal{A}(e_2)\big) \tag{14}$$

E. Logical object

The logical object $l \in \mathbb{L}$ uses the event generator $g \in \mathbb{G}$ to schedule events. For each generator $g \in \mathbb{G}$ its current state $s \in \mathbb{S}$ and the step function $\mathcal{N}: \mathbb{S} \rightarrow \mathbb{S} \times \mathbb{E}$ are defined.

Newly scheduled events are stored in a local event queue $E = \{e_i \in \mathbb{E}\}$ [3] with size limited by the capacity $c \in \mathbb{N} \cup 0$. In addition to capacity the behavior of the logical object is controlled by the activity level $a \in \mathbb{N} \cup 0$ and the activity limit $\bar{a} \in \mathbb{N} \cup 0$.

Over the set of logical objects the mapping $\mathcal{D}_L: \mathbb{L} \rightarrow \{d, \emptyset\}$ to the dispatcher set is defined. This mapping allows one to distinguish between attached and free logical objects.

$$g = < s, \mathcal{N} > \tag{15}$$

$$l = < g, E, a, \bar{a}, c > \tag{16}$$

$$\mathcal{D}_l: \mathbb{L} \rightarrow \{d, \emptyset\} \tag{17}$$

Generator $g \in \mathbb{G}$ constructs an event sequence $\{e_n\}_1^\infty$ using the recurrence relation $s_{n+1} \times e_{n+1} = g.\mathcal{N}(s_n)$. Generator schedules events in order of nondecreasing model time.

$$\forall n, k \in \mathbb{N}\ \big(n > k \rightarrow \mathcal{T}(e_n) \geq \mathcal{T}(e_k)\big) \tag{18}$$

The generator has ability to synchronize its own time (presented explicitly or implicitly, through the scheduled events) with the model time. In this case the step function returns an empty symbol as the event $s_{n+1} \times \theta_e = g.\mathcal{N}(s_n)$. If the generator has planned all the events then step function returns an empty symbol as the state $\theta_s \times e_{n+1} = g.\mathcal{N}(s)$.

F. Handling block

Handling block consists of one or several event containers and a scheduler $\mathcal{S}: 2^{\mathbb{E}} \rightarrow \{e_i\}$, $\mathcal{S}(\mathbb{E}) \subseteq \mathbb{E}$. The scheduler ranks elements of attached containers and choose a certain event range.

There are several types of handling blocks b: "CEB", "FEB", "PEB" and "REB". Each block type has its own distinctive features.

Blocks typed as "CEB", "FEB" and "REB" have a more complex structure than a block of type "PEB". Two containers are attached to these blocks. Block "FEB" also contains a model clock (defining the event horizon) and the simulation threshold l. "REB" block has a resource container R_{REB} intended to store resources changed on the current iteration of event handle loop.

$$b_{CEB} = < E_{CEB}, \hat{E}_{CEB}, \mathcal{S}_{CEB} > \tag{19}$$

$$b_{FEB} = < E_{FEB}, \hat{E}_{FEB}, \mathcal{S}_{FEB}, l > \tag{20}$$

$$b_{PEB} = < E_{PEB}, \mathcal{S}_{PEB} > \tag{21}$$

$$b_{CEB} = < E_{REB}, \hat{E}_{REB}, \mathcal{S}_{REB}, R_{REB} > \tag{22}$$

The event scheduler gives as the result an ordered set of events with the delay condition met. The results of the scheduler of "CEB" and "FEB" blocks includes all such events whereas the schedulers of the remaining block types allowed not giving all such events.

$$\forall X \in \{\mathrm{CEB}, \mathrm{FEB}, \mathrm{PEB}, \mathrm{REB}\}\ \forall d \in \mathbb{D}$$

$$\forall e \in \mathcal{S}_X(E)\ e.\mathcal{C}(d) = \mathrm{true} \tag{23}$$

$$\forall X \in \{\mathrm{CEB}, \mathrm{FEB}\}\ \forall d \in \mathbb{D}\ \mathcal{S}_X(E) = \{e: e.\mathcal{C}(d) = \mathrm{true}\} \tag{24}$$

$$\forall X \in \{\mathrm{PEB}, \mathrm{REB}\}\ \forall d \in \mathbb{D}\ \mathcal{S}_X(E) \subseteq \{e: e.\mathcal{C}(d) = \mathrm{true}\} \tag{25}$$

The delay condition of event moved to "FEB" block essentially depends on the model time, and is satisfied at only point on the time axis. Scheduler of this block gives a set of events with a minimum time:

$$\forall e \in \mathcal{S}_{FEB}(\mathbb{E})\ (\mathcal{T}(e) = \min_{e \in \mathbb{E}} \mathcal{T}(e)) \tag{26}$$

In addition time of each scheduled event does not exceed the simulation threshold l:

$$\forall e \in \mathcal{S}(\mathbb{E})\ \mathcal{T}(e) \leq l \tag{27}$$

G. Dispatcher

Output trace t, handling blocks set B and direct access resource container R_{direct} are attached to the dispatcher d.

$$d = < t, B, R_{direct} > \tag{28}$$

Handling block set includes a single block of "FEB", and can also include no more than one block of every other type. Thus the assertion

$$\{b_{FEB}\} \subseteq B \subseteq \{b_{CEB}, b_{FEB}, b_{PEB}, b_{REB}\} \tag{29}$$

[2] Here and henceforth the operator "." will be used to denote the tuple element.

[3] We assume that the event queue E has several predefined operations:

1. $|E|$ – returns number of events in the queue,
2. $pushBack(E, e)$ – adds event to the back of the queue,

$popFront(E)$ – takes event from the head of queue.

IV. OPERATIONAL SEMANTICS

In this chapter the concept of runtime environment and rules of its changing are introduced. Algorithms for the initialization of the runtime and model running are described.

A. Resource state

For each resource $r =< V, E >$ the state $\mathfrak{r}$ is determined as a tuple consisting of variable value $r.V$ [4] and a set of events from attached container $r.E$ [5].

$$\mathfrak{r} =< eval(V), eval(E) > \quad (30)$$

B. Logical object state

The state $\mathfrak{l}$ of the logical object $l =< g, E, a, \bar{a}, c >$ is defined as a pair of corresponding event generator $l.g$ and the event set of events contained in the attached to the logical object container $l.E$.

$$\mathfrak{l} =< s, eval(E) > \quad (31)$$

C. Handling block state

State $\mathfrak{B}$ of handling block set (29) is defined as a collection of the contents of containers attached to them.

$$\mathfrak{b}_{CEB} =< eval(E_{CEB}), eval(\widehat{E}_{CEB}) > \quad (32)$$
$$\mathfrak{b}_{FEB} =< eval(E_{FEB}), eval(\widehat{E}_{FEB}) > \quad (33)$$
$$\mathfrak{b}_{PEB} =< eval(E_{PEB}) > \quad (34)$$
$$\mathfrak{b}_{REB} =< eval(E_{REB}), eval(\widehat{E}_{REB}), eval(R_{REB}) > \quad (35)$$

D. Dispatcher state

Dispatcher state $\mathfrak{d}$ is defined as a collection of the contents of trace t, attached handling blocks state $\mathfrak{B}$ and the content of attached resource container.

$$\mathfrak{d} =< eval(t), \mathfrak{B}, eval(R_{direct}) > \quad (36)$$

E. Runtime environment state

Runtime environment condition $\mathfrak{w}$ is characterized by the state of resources $w.\mathbb{R}$ and logical object $w.\mathbb{L}$ of the model and the state of the dispatcher $w.d$.

$$\mathfrak{w} =< R, L, d > \quad (37)$$

F. The rules of the runtime state changing

Rules (38)-(40) are intended to add a direct or indirect access resource or logical object to the dispatcher.

Rule (41) describes the changes in the state of the runtime after the value of resource variables changed.

Rule (42) defines the run of the model.

$$\frac{r \in \mathbb{R}_{direct} \;\&\; r \notin d.R_{direct}}{d.R_{direct} = r \cup d.R_{direct}} \quad (38)$$

$$\frac{r \in \mathbb{R}_{indirect} \;\&\; r.d \neq d}{r.d = d} \quad (39)$$

$$\frac{l \in \mathbb{L} \;\&\; l.d \neq d}{l.d = d, lookForEvents(l)} \quad (40)$$

$$\frac{r \in \mathbb{R}_{indirect} \;\&\; set(r, D)}{r.D = D, R_{REB} = r \cup R_{REB}} \quad (41)$$

$$\frac{E_{FEB} \neq \emptyset}{handleCycle(), advanceTime()} \quad (42)$$

G. Searching events to transit into the handling blocks

$lookForEvents(\, l \in \mathbb{L}\,);$

The algorithm checks the readiness of the logical object to schedule events and moves events created by them into the handling blocks attached to the dispatcher. If the number of such events in the handling blocks has reached the limit, some of them are buffered into the local queue.

```
#Used while event handling – decrement activity level
a = a − 1;
IF ( a = 0 )
  #There are no more events produced by l
  IF ( |E| = 0 )
    #Local event queue is empty
    WHILE ( s ≠ θs & (a < ā ∨ |E| < c) ) DO
      #Event generator can schedule events and local queue is not full
      #Invoke event generator
      s × e = G(s);
      IF ( e = θe )
        #Generator cannot schedule events yet
        BREAK;
      FI
      IF ( a < ā )
        #Activity level is less than activity limit
        #Add event to handling block
        addEvent(e);
        #Increment activity level
        a = a + 1;
      ELSE
        #Handling block contain a limit event number
        #Add event to local event queue
        E.pushBack(e);
      FI
    OD
  ELSE
    #Local event queue is not empty
    #Add a number of events less or equal to activity limit
    limit = min(|E|, ā);
    WHILE ( a < limit ) DO
      #Transmit event from local queue to handling blocks
      addEvent(E.popFront());
      a = a + 1;
    OD
  FI
FI
```

H. Addition of event to the handling blocks

$addEvent(\, e \in \mathbb{E}\,);$

The event is placed in a handling block in accordance with its type.

[4] To indicate the values of the object X the operator $eval(X)$ will be used.

[5] Under the value of the container the set of elements contained therein is infered.

```
#Load an event to handling block of the same type
IF ( p = CEB )
   Ê_CEB = Ê_CEB ∪ e;
ELSEIF ( p = PEB )
   E_PEB = E_PEB ∪ e;
ELSEIF ( p = REB )
   Ê_REB = Ê_REB ∪ e;
#Event type is "FEB" – other choices are sort out
ELSEIF ( t = c )
   #Event time is equal to current model time
   Ê_FEB = Ê_FEB ∪ e;
ELSEIF ( t > l )
   #Event time is greater than simulation threshold
   l.s = θ_s;
ELSE
   E_FEB = E_FEB ∪ e;
FI
```

I. Event handle cycle

handleCycle();

Algorithm iteratively retrieves ready events from handling blocks, sorts them and calls for event handling. Model time does not change during this process.

```
WHILE ( TRUE ) DO
   #Search for direct access resources with changed variable value
   ∀r ∈ R_direct
      IF ( eval(r.V) ≠ ℰ(r) )
         #Add resource to container of changed resource
         R_REB = r ∪ R_REB;
         #Refresh value
         r.V = r.𝒢;
      FI
   #Compose the contents of REB major event container
   #Add events with satisfied delay conditions
   E_REB = {e: e ∈ Ê_REB & e.𝒞 = true}
   #Add the rest to resource containers
   ∀e ∈ Ê_REB \ E_REB
      ∀r ∈ Var(e.𝒞)
         r.E = e ∪ r.E;
   #Leave in auxiliary REB container only events with satisfied delay condition
   Ê_REB = E_REB;
   #Add events depending on changed resources into the REB major event container
   ∀r ∈ R_REB
      E_REB = r.E ∪ E_REB;
   #Add events independent on resource state into CEB auxiliary container
   Ê_CEB = Ê_CEB + Ê_FEB;
   #Compose the contents of CEB major event container
   E_CEB = Ê_CEB + 𝒮_PEB(E_PEB) + 𝒮_REB(E_REB);
   E = 𝒮_CEB(E_CEB);
   IF ( |E| = 0 )
      #No event was chosen
      BREAK:
   FI
   #Handle chosen events
   ∀e ∈ E
      handleEvent(e);
OD
R_REB = ∅;
```

J. Event handling

handleEvent(e);

Handled event is excluded from the handling block. Then the model state changed in appropriate to this event way and the information recorded in the trace. The dispatcher also searches for the events created by the same logical object to transfer into attached handling blocks.

```
IF ( p = CEB ∨ p = FEB )
   #Event does not depend on resource state
   Ê_CEB = Ê_CEB \ e;
ELSEIF ( p = PEB )
   E_PEB = E_PEB \ e;
ELSEIF ( e ∈ Ê_REB )
   #Event delay condition was always satisfied
   Ê_REB = Ê_REB \ e;
ELSE
   #Event was added into resource container
   ∀ r ∈ Var(e.𝒞)
      r.E = r.E \ e;
FI
#Change model state according to event
ℜ × 𝔏 = e.ℳ(ℜ × 𝔏);
Add message to the output trace e.m;
getEvents(L(e));
```

K. Model time advancing

advanceTime();

Selected by the scheduler of "FEB" block, events are transferred from the major container to the additional one. During this model time is changed to the arrival time of any of the selected events (each of these events has the same arrival time).

```
#Move event scheduled to current model time to auxiliary container FEB
Ê_FEB = 𝒮_FEB(E_FEB);
E_FEB = E_FEB \ Ê_FEB;
#Set a new model time
c = e.t, e ∈ Ê_FEB;
```

V. Requirements to the Runtime Blocks

Mathematical model allows identifying interfaces of blocks that make up the runtime. Thus a sufficient condition for the possibility of substituting a new runtime is the compliance of its interfaces with the requirements.

There are several classes of requirements:

1. PRED – specifies pre-condition,

2. POST – specifies post-condition,
3. RET – determines the method return value,
4. EQ – specifies equivalence to the described algorithm.

Requirements for the interfaces of the blocks depend on its type, and the configuration of the runtime as a whole. All specifications are listed in table 1.

VI. Observations on the implementation

Configurable runtime environment is developed as a compile time library written in C++. The flexibility of the runtime components is achieved through the use of template classes. Thus each of these blocks is represented as a single class. The new runtime environment with the necessary properties can be created on the basis of the class layout.

The signature of class interfaces are variable and depend on the configuration of the runtime. Nevertheless, they can be checked at compile time. As appropriate tool the library Boost Concept Check Library (BCCL) can be used [12]. This library allows formal describing of the requirements for abstract data types (concepts) used in templates and verify their compliance with these requirements.

The most difficult requirements to verify are the ones to interface of the handling blocks. Depending on the configuration of the runtime their functionality can have significant differences. But the number of fundamentally different configurations of the handling block is low. Thus partially specifying a template of dispatcher class and using BCCL can impose restrictions on the interfaces of the handling blocks for any possible configuration.

Semantic requirements for class interfaces can be checked with unit testing. Tests for the blocks can be incorporated directly into the library being developed so that using the predefined flag tests new plug-in logical block [13]. Then successfully tested blocks can be incorporated into the library itself.

VII. Conclusion

Implementation of the developing library results into ability to quickly build high-performance runtime environment with the necessary properties. This only requires new instances of some blocks. Thus a sufficient condition for the correctness of the constructed DESR is the interface compliance to formulated specifications which can be verified automated.

Tested blocks can in turn be included into the library. With the increasing number of block instances the share of reusable code will increase whereas the cost of developing new runtime will be reduced to the layout of the ready-made blocks.

References

[1] V. G. Moloney, R. L. Smelyansky, "An integrated approach to modeling distributed computing systems", Programming N.1, 1988 pp. 57-67 (In Russian)

[2] A. Bakhmurov, A. Kapitonova, R. Smeliansky, "DYANA: An Environment for Embedded System Design and Analysis", 5-th International Conference TACAS'99, Amsterdam, The Netherlands, March 22-28, 1999. Springer (LNCS Vol.1579), pp.390-404

[3] V. V. Balashov, A. G. Bahmurov, D. Yu. Volkanov, R. L. Smelyanskiy, M. V. Chistolinov, N. V. Yushchenko, G. T. Mamontov, P. Yuhta "Experience of the program DYANA implementation for simulation and integration of on-board computing systems", Abstracts of reports XXVI conference in memory of an outstanding designer gyroscopic devices N. N. Ostryakov - St. Petersburg: Central Research Institute Elektropribor, 2008. pp. 60-61 (In Russian).

[4] V. V. Balashov, A. G. Bakhmurov, M. V. Chistolinov, R. L. Smeliansky, D. Yu. Volkanov, N. V. Youshchenko, “A Hardware-in-the-Loop Simulation Environment for Real-Time Systems Development and Architecture Evaluation”, In Proc. of the Third International Conference on Dependability of Computer Systems DepCoS-RELCOMEX 2008, Szklarska Poreba, Poland, June 26-28 2008.

[5] A. G. Bahmurov, E. G. Egisapetov, O. V. Novikov, V. V. Prus, K. O. Savenkov, R. L. Smelyansky, "Tool support for software development process for a special processor-based CPU L1879VM1 ", Methods and means of processing information. Proceedings of the Second All-Russian Scientific Conference. - M.: Publishing Department, Faculty of Computational Mathematics and Cybernetics. Moscow State University, 2005, pp.450-456

[6] C. D. Pegden, “Introduction to Simio”, Proceedings of the 40th Conference on Winter Simulation (Miami, Florida, December 07 - 10, 2008), pp. 229-235.

[7] R. C. Crain, J. O. Henriksen, “Simulation using GPSS/H”, In Proceedings of the 31st Conference on Winter Simulation: Simulation---A Bridge To the Future - Volume 1 (Phoenix, Arizona, United States, December 05 - 08, 1999), pp. 182-187.

[8] T. J. Schriber, D. T. Brunner, "Inside discrete-event simulation software: how it works and why it matters", Proceedings of the Winter Simulation Conference, 2005, pp. 11 pp.+

[9] T. J. Schriber, D. T. Brunner, "Inside discrete-event simulation software: how it works and why it matters", Proceedings of the Winter Simulation Conference, 1996, pp. 11 pp.+

[10] P.J. Sanchez, "Fundamentals of simulation modeling", Proceedings of the Winter Simulation Conference, 2007, 9-12 Dec. 2007 Page(s):54 – 62

[11] K. O. Savenkov, E. V. Chemeritskiy, “Discrete-event simulation runtime: from genericity to extendability and reuse”, Simulation-2010, submitted for publication.

[12] Boost library [Online]. Available: http://www.boost.org.

[13] A. H. Bagge, V. David, M. Haveraaen. “The axioms strike back: testing with concepts and axioms in C++”, In Proceedings of the Eighth international Conference on Generative Programming and Component Engineering (Denver, Colorado, USA, October 04 - 05, 2009).

Table 1. Requirements to components of the runtime.

Block	Method	Block type	Configuration	Requirement: Type	Requirement: Specification
Resource	$\mathrm{get(void)}: \mathrm{D} \in \mathbb{D}$			RET	$\mathrm{eval(r.V)}$
	$\mathrm{check(void)}: \{\mathrm{true, false}\}$	$\mathrm{r_{direct}}$		RET	$\mathcal{E}(\mathrm{r}) \neq \mathrm{eval(r.V)}$
	$\mathrm{set}(\mathrm{D} \in \mathbb{D}): \mathrm{void}$	$\mathrm{r_{indirect}}$		POST	$\mathrm{r.V = D}$
			Related waiting	PRED	r.bind has been invoked.
				POST	d.changed has been invoked.
	$\mathrm{bind(d): void}$			POST	$\mathrm{r.d = d}$
	$\mathrm{addEvent}(\mathrm{e} \in \mathbb{E}): \mathrm{void}$			POST	$\mathrm{r.E = e \cup r.E}$
	$\mathrm{deleteEvent}(\mathrm{e} \in \mathbb{E}): \mathrm{void}$			POST	$\mathrm{r.E = r.E \setminus e}$
	$\mathrm{getEventsRange(void)}: < first, last >$			RET	$< r.E.first, r.E.last >$
Trace	$\mathrm{addEvent}(\mathrm{m} \in \mathbb{M}, \mathrm{t} \in \mathbb{D}): \mathrm{void}$			POST	The message has been added to the trace.
Event	$\mathrm{getType(void)}: \{\mathrm{CEB, FEB, PEB, REB}\}$			RET	$\mathrm{e.p}$
	$\mathrm{getMessage(void)}: \mathbb{M}$			RET	$\mathrm{e.m}$
	$\mathrm{getLogicalObject(void)}: \mathbb{L}$			RET	$\mathcal{L}(\mathrm{e})$
	$\mathrm{change(void): void}$			POST	$\mathfrak{R} \times \mathfrak{L} = \mathrm{e}.\mathcal{M}(\mathfrak{R} \times \mathfrak{L})$
	$\mathrm{getTime(void)}: \mathbb{D}$	$\mathrm{e_{FEB}}$		RET	$\mathcal{T}(\mathrm{e})$
	$\mathrm{condition(void)}: \{\mathrm{true, false}\}$	$\mathrm{e_{PEB}}$ $\mathrm{e_{REB}}$		RET	$\mathrm{e}.\mathcal{C}$
	$\mathrm{addToResources(void): void}$		Related waiting	POST	$\forall \mathrm{r} \in \mathrm{Var}(\mathrm{e}.\mathcal{C})\ \mathrm{r.E = e \cup r.E}$
	$\mathrm{removeFromResources(void): void}$			POST	$\forall \mathrm{r} \in \mathrm{Var}(\mathrm{e}.\mathcal{C})\ \mathrm{r.E = r.E \setminus e}$
Logical object	$\mathrm{bind(d): void}$			POST	$\mathrm{l.d = d}$
	$\mathrm{lookForEvents(void): void}$			EQ	To algorithm lookForEvents(l)
	$\mathrm{setCapacity}(\mathrm{c} \in \{\mathbb{N}, 0\}): \mathrm{void}$		Event buffer	POST	$\mathrm{l.c = c}$
	$\mathrm{setLimit}(\bar{\mathrm{a}} \in \{\mathbb{N}, 0\}): \mathrm{void}$			POST	$\mathrm{l}.\bar{\mathrm{a}} = \bar{\mathrm{a}}$
Handling block	$\mathrm{addEvent}(\mathrm{e} \in \mathbb{E}): \mathrm{void}$			POST	$\mathrm{b = e \cup b}$
	$\mathrm{removeEvent}(\mathrm{e} \in \mathbb{E}): \mathrm{void}$			POST	$\mathrm{b = b \setminus e}$
	$\mathrm{getEventRange(void)}: < first, \mathrm{last} >$			RET	$\mathrm{eval}\big(\mathrm{b}.\mathcal{S}(\{\mathrm{e: e \in b}\})\big).$
	$\mathrm{addIndependentEvents}(< first, last >): \mathrm{void}$	$\mathrm{b_{CEB}}$	$\mathrm{b_{CEB} \in d.B}$	POST	$\mathrm{b_{CEB}} = \{< first, last >\} \cup \mathrm{b}$
	$\mathrm{addEvents}(< first, last >): \mathrm{void}$	$\mathrm{b_{CEB}}$	$\mathrm{b_{CEB} \in d.B}$	POST	$\mathrm{b_{CEB}} = \{< first, last >\} \cup \mathrm{b}$
		$\mathrm{b_{FEB}}$	$\mathrm{b_{CEB} \notin d.B}$	POST	$\mathrm{b_{FEB}} = \{< first, last >\} \cup \mathrm{b}$
	$\mathrm{timeAdvance(void): void}$	$\mathrm{b_{FEB}}$		POST	$\mathrm{b_{FEB}.c} = \min_{\mathrm{e \in b_{FEB}}} \mathcal{T}(\mathrm{e})$
	$\mathrm{addResource}(\mathrm{r} \in \mathbb{R}): \mathrm{void}$	$\mathrm{b_{REB}}$	$\mathrm{b_{REB} \in d.B}$	POST	$\mathrm{b_{REB} = r \cup b_{REB}}$
		$\mathrm{b_{FEB}}$	$\mathrm{b_{REB} \notin d.B}$ Related waiting	POST	$\mathrm{b_{FEB} = r \cup b_{FEB}}$
	$\mathrm{reset(void): void}$	$\mathrm{b_{PEB}}$		POST	Resource container has been reset.
		$\mathrm{b_{FEB}}$		POST	Ready to handle events have been reset.
		$\mathrm{b_{CEB}}$		POST	Unhandled events have been reset.
Dispatcher	$\mathrm{addResourceDirect}(\mathrm{r} \in \mathbb{R}_{\mathrm{direct}}): \mathrm{void}$			EQ	To rule (1)
	$\mathrm{addResourceIndirect}(\mathrm{r} \in \mathbb{R}_{\mathrm{indirect}}): \mathrm{void}$		Related waiting	EQ	To rule (2)
	$\mathrm{resourceChanged}(\mathrm{r} \in \mathbb{R}): \mathrm{void}$		$\mathrm{b_{REB} \in d.B}$	POST	$\mathrm{b_{PEB}.addResource(r)}$ has been invoked.
			$\mathrm{b_{REB} \notin d.B}$ Related waiting	POST	$\mathrm{b_{FEB}.addResource(r)}$ has been invoked.
	$\mathrm{addLogicalObject}(\mathrm{l} \in \mathbb{L}): \mathrm{void}$			EQ	To rule (3)
	$\mathrm{addEvent}(\mathrm{e} \in \mathbb{E}): \mathrm{void}$			EQ	To algorithm addEvent()
	$\mathrm{handleEvent}(\mathrm{e} \in \mathbb{E}): \mathrm{void}$			EQ	To algorithm handleEvent()
	$\mathrm{handleCycle(void): void}$			EQ	To algorithm handleCycle()
	$\mathrm{run(void): void}$			EQ	To rule (5)

Testing automation of projects in telecommunication domain

Alexey Veselov, Vsevolod Kotlyarov
Saint-Petersburg State Polytechnic University, Saint-Petersburg, Russia
a.veselov@ics2.ecd.spbstu.ru, vpk@ics2.ecd.spbstu.ru

Abstract – **This paper presents an integrated approach to testing automation of telecommunication projects along with proposals to automation of conformance testing. The underlying idea is to benefit from combining formal verification and testing automation techniques in order to improve product quality.**

I. INTRODUCTION

Testing is known to be an essential step in any modern industrial software development process. The importance of software testing and its impact on software quality can't be underestimated. To date, spending up to 40% of efforts on testing is not uncommon for a software company. Therefore, testing automation and testing efficiency – the issues addressed in this paper - are very important.

This paper describes a part of technology which combines formal verification, testing and code generation techniques in order to improve software quality, reduce efforts and costs. This technology covers the whole software development process which starts from the initial requirements and goes through formalization, verification, simulation, code generation and testing to a software product. The technology's part, which is described below, relates to the techniques which on the one hand can be used for automation of functional tests generation, and on the other hand can be applied for conformance testing.

The underlying idea of the proposal is to benefit from presence of verification step in the technology. Verification is a powerful technique which can significantly improve software quality and formally prove absence of mistakes in algorithm. But it also has another important advantage: a model is created on the verification step, and each path in the model's behavior tree can be considered as a potential test case. The number of such paths in industrial-scale projects is enormous, so a special filtering technique is required to obtain suitable (optimal) test suite. This approach is applicable not only to automation of functional tests generation, but also for automation of conformance testing (in case if a model of the corresponding standard is provided). An example of such standard is CDMA2000 [1]. CDMA2000 is a family of 3G mobile technology standards which use CDMA (Code Division Multiple Access) channel access to send data, voice, and signaling data between mobile phones and cell sites. Importance of compliance of any CDMA2000 device and cell site with this standard is obvious. In practice, compliance is checked by rather significant amount of conformance tests which are prepared manually. Proposed approach to conformance testing allows avoiding manual tests development by means of reusing artifacts of verification step.

This paper outlines the main principles of automated test suite generation on the base of formalized specifications in the language of basic protocols using the verification tool VRS [2, 3] and tests generation tool TAT [4].

II. TECHNOLOGY OVERVIEW

Despite the paper is focused on testing automation part of the technology, overview of the whole technology chain is required for proper understanding of application domain of the proposing ideas.

The technology is intended for automation of manual efforts in software development process as much as possible (by applying code generation techniques) and increasing products' quality (because of combining formal verification and testing).

According to the technology chain, illustrated in Fig. 2, software development starts from understanding and formalizing of the requirements. On the formalization step, engineers create a model of the system in terms of basic protocols. Basic protocol is a formal description of some action which shall be performed if the system goes into a certain state (pre-condition is satisfied), and a new state of the system after performing this action (post-condition). In other words, basic protocol is a "small piece" of system's behavior. Basic protocols are represented in the following notation:

$$\forall X : \alpha(X) \xrightarrow{\mu(X)} \beta(X)$$

where X is protocol parameters' list, α is a pre-condition, β is post-condition and μ is action; α, μ and β may depend on X. This is a Hoare's triplet [5].

The formalism of basic protocols is based on the theory of agents and environments with the insertion function [6]. It was proposed by A. Letichevsky et al. for creating system behavior models suitable for automated verification [7, 8]. Basic protocol is just a small MSC chart. Fig. 1 illustrates a basic protocol example.

Two basic protocols can be concatenated when post-condition of the first one is equal to pre-condition of the second one. All possible concatenations of basic protocols construct the model's behavior tree. Each path in this tree is a possible scenario of system's behavior.

Formalization is mostly manual work, but it can also be automated or simplified in some sense. For example, the user can perform formalization in terms of use cases (Use Case Maps (UCM [9]) or Message Sequence Charts (MSC [10])) or an UML [11] model. Special tools are used for basic protocols generation from UCM, MSC and UML.

The second step of the technology chain is verification. The system's model, represented in terms of basic protocols, is

verified against some properties using model checker of the VRS tool.

A set of basic protocols can be interpreted as a set of states and transitions, i.e. as a state machine. Having a state machine which describes system's behavior, it is possible to generate system's logic code in a target language. But there is no tool which can generate executable code from basic protocols, so intermediate representation is required. Specification Description Language (SDL [12]) was chosen to be the language of this intermediate representation. SDL is rather simple and well-known language for describing state machines. Presence of intermediate language makes it much more convenient for the engineers to debug models.

So, the third step is conversion from basic protocols representation to SDL representation. This conversion is performed by special tools.

As soon as SDL code is obtained, it is possible to generate code in target language. Generation of target code is the fourth step. A user has opportunity to choose any SDL to target code translator, but in our practice we use Sdl2cpp because it is seamlessly integrated into the technology chain. Sdl2cpp tool is efficient and reliable translator from SDL to C++. Sdl2cpp can generate configuration file for TAT (Test Automation Toolset) with test environment description. So, in case of using Sdl2cpp together with TAT, test environment configuring becomes transparent for a user (no manual efforts required).

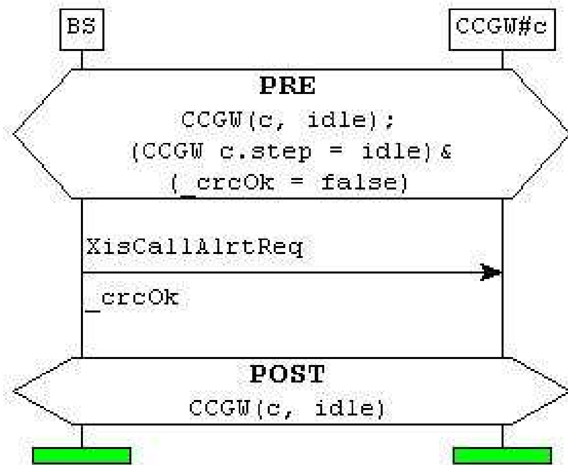


Fig. 1. Basic protocol

As soon as model's executable code is obtained and test environment is configured, it is possible to run tests and perform real-time simulation. It is also rather convenient to do step-by-step simulation in IBM Rational SDL Suite [13] with tracking of system's behavior in graphical mode.

The fifth step is creation of test coverage criterion. Model in terms of basic protocols is a behavior tree with enormous number (more precisely – infinite number) of potential scenarios. It is impossible to cover this entire tree with tests, so a filtering criterion is required. The user is free to choose any criterion, but usage of requirements coverage criterion makes it is possible to generate relatively small test suite which covers functional requirements. This criterion requires creation of "chains" for each requirement. "Chain" is a sequence of key impacts on the system under test (SUT) and systems' responses which cover a particular requirement (requirement may be covered by more than one "chain").

Creation of chains is a process of requirements interpretation, so it is mostly manual work, especially if requirements are presented in natural language. Chains creation is significantly simplified if customer provides requirements in formal notation, for example in terms of UCM or MSC.

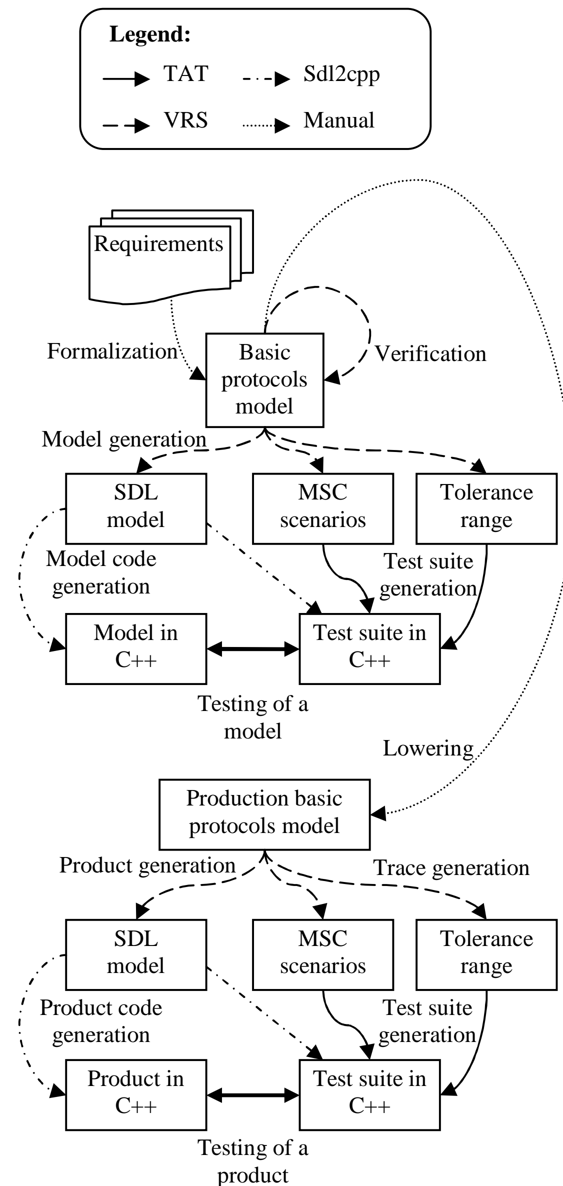


Fig. 2. Overall technology scheme

The sixth step is test scenarios generation. VRS tool uses chains for guided search in model's behavior tree and generates a trace if it can find a path with chain's key events. As a result, a set of scenarios which cover functional requirements is generated. These scenarios are represented in terms of MSC charts. For the requirements which for some reasons can't be expressed in terms of chains (for example,

non-functional requirements), test cases shall be created manually.

TAT tools are used for generation of executable test environment code from MSC charts. So, the next (seventh) step is generation of a test suite in a target language using TAT tools. This step also includes substitution of symbolic parameters of MSC scenarios with actual values taken from tolerance range.

The eight's step is running generated test suite against the model.

The technology chain consisting of the mentioned eight steps works fine when scale of a project is rather small. But when the project starts to grow, verification becomes a bottle neck because of the state explosion problem. Verification of industrial-scale projects requires abstracting from some details, therefore basic protocols model must be detailed after verification and before production code generation. It is usually required to maintain two models within the scope of one project: high-level model which is used for verification, modeling and simulation, and production model which is used for product code generation. Each model requires its own test suite.

So, the ninth step is detailing. This step is also known as lowering because the model is filled with low-level details. Usually it implies adding signals' parameters, replacing functional stubs (when a complex behavior is encapsulated in one signal; e.g.: usage of "InitSession" signal instead of describing real sequence of messages for session initialization) with actual sequence of signals and adding omitted signals. Lowering is another one manual part of the technology.

As soon as the model is filled with low-level details, it can be used for product code generation. To generate product code, the user has to repeat steps 3 and 4 for the detailed model. To generate a test suite for the product, the user has to repeat steps 6 and 7 (step 5 is not required because chains for the high-level model are applicable to the detailed model).

III. TESTING AUTOMATION

The whole technology chain consists of several steps. Steps 5, 6, 7 and 8 relate to tests preparation, generation and execution activities.

Tests preparation starts from requirements interpretation and formulating coverage criteria in terms of "chains" – sequences of key events. These chains are used as oracle for guided search in model's behavior tree. It is natural that from infinite number of possible system's behaviors, test engineers want to choose reasonable quantity of scenarios which cover SUT requirements. This result can be achieved in case of using chains: VRS tool will automatically generate corresponding traces (test cases). In the worst case, the number of test cases will be equal to the number of chains, but in practice some traces cover more than one requirement, so quantity of tests is usually less than quantity of chains. Special tool is responsible for selecting minimal test suite from generated traces. This tool also marks the points (in MSC traces), where coverage of each particular requirement starts and ends, along with the places of key events.

As soon as a set of scenarios is selected according to the coverage criteria, it is required to replace scenarios' parameters with concrete values (if they contain symbolic parameters).

VRS tool can work in two modes. The first one is regular model checking where all the parameters have concrete values. The second mode is called "symbolic"; in this mode tolerance range for each parameter is evaluated after applying each basic protocol, so generated MSC charts (scenarios) contain symbolic parameters, and VRS provides their values' constraints (tolerance ranges). So, in case of using "symbolic" mode, there is one additional step: replacement of MSCs' symbolic parameters with concrete values taken from tolerance range.

One MSC with symbolic parameters is actually a set of equivalent behaviors (the same sequence of events for any parameters' values from tolerance range). So, taking into account parameters' constraints and dependencies between parameters, the user can obtain a set of test scenarios with same sequence of events, but different parameters' values.

A set of concrete values for each trace we will call a profile. The user is free to choose any profile, but usually the following profiles are chosen:

- "left edge" profile - all independent parameters have minimum possible values (from the tolerance range);
- "right edge" profile - all independent parameters have maximum possible values (from the tolerance range);
- "middle" profile – all independent parameters have arbitrary values (from the tolerance range);
- "error" profile – at least one parameter has value out of tolerance range. Test case with any "error" profile shall always fail.

Scenarios with concrete parameters can be used for test suite generation.

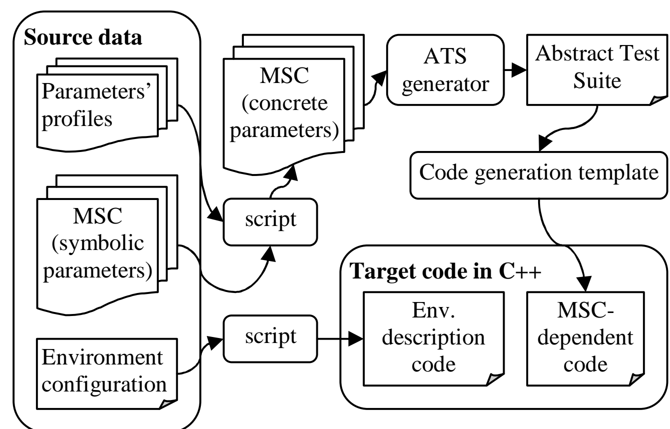


Fig. 3. Test suite generation

Test Automation Toolset (TAT) is used for generation test suites in target language on the base of scenarios represented in terms of MSC charts and configuration XML files with environment description.

Test suite generation consists of several major steps:

- Analysis of XML configuration files and generation of environment-specific code in target language;

- Generation of Abstract Test Suite (ATS) – representation of a test suite as a set of states and transitions between them (state machine) in TCL. ATS is generated on the base of MSC charts;
- Generation of a test suite in target language on the base of ATS.

These steps are illustrated in Fig. 3

TAT has very flexible mechanism of code generation templates. Using templates, TAT can be configured for generation of test suite code in any language. Currently, the most powerful template is for C++, but several other languages are also supported, including Java SE, Java ME and TTCN-3 (only C++ code generation is considered in the scope of this paper).

Generated test suite interacts with a system under test via IP sockets (both TCP and UDP are supported) or via Unix domain sockets (both stream and datagram are supported). A user can specify which sockets to use in TAT configuration files along with the following settings:

- Initialization code which will be executed before the beginning of each test case. Sometimes some initialization of a system under test is required before running a test case. The code which is responsible for moving SUT to its initial state shall be placed in corresponding section of TAT XML configuration file (initialization section).
- Finalization code which will be executed after a test case is finished regardless of the verdict (pass/fail). It is a good practice to free resources when they are not in use any more, this can be done in corresponding section of TAT XML configuration file (finalization section).
- Timeouts of waiting for incoming signals. The user can specify how much time to wait for signal(s) in milliseconds. If test environment does not receive the signal(s) which it expects (according to the MSC scenario) within this time interval, corresponding test will be marked as failed. It is possible to specify default timeout for all incoming signals and, if necessary, redefine this parameter for any particular incoming signal.
- Signals' distinguisher code (if not provided by Sdl2cpp). This code is responsible for distinguishing incoming messages. In case of using TAT together with Sdl2cpp, this code is generated automatically. If TAT is used as stand-alone tool, the user has to provide this code manually.
- Signals' sending/receiving code (if not provided by Sdl2cpp). The user may provide own code for serialization and deserialization of messages and their parameters. In case of using TAT together with Sdl2cpp, this code is generated automatically. If TAT is used as stand-alone tool, the user has to provide this code manually.
- Instances and interfaces definition (if not provided by Sdl2cpp). All the instances (used in MSCs) must be described in TAT XML configuration file along with their interfaces (port numbers for IP sockets or file names for Unix domain sockets). Instances may be of two types: "env" (abstraction of test environment) and "model" (abstraction of system under test). In case of using TAT together with Sdl2cpp, this description is generated automatically. If TAT is used as stand-alone tool, the user has to provide it manually.
- Message delimiter (if not provided by Sdl2cpp).
- User-defined log format. TAT has two built-in formats of logs: logs in terms of MSC charts and plain text logs. But, if required, the user can define own log format.

TAT has seamless integration with Sdl2cpp: Sdl2cpp can not only generate target code from SDL, but also completely configure TAT for testing the generated code. But even in case of using TAT together with Sdl2cpp, manual adjusting of test environment remains possible because TAT configuration can be split into two parts (stored in separate files): generated by Sdl2cpp and manually provided. Manually provided part has higher priority, so default environment settings (generated by Sdl2cpp) can be redefined by the user (if required).

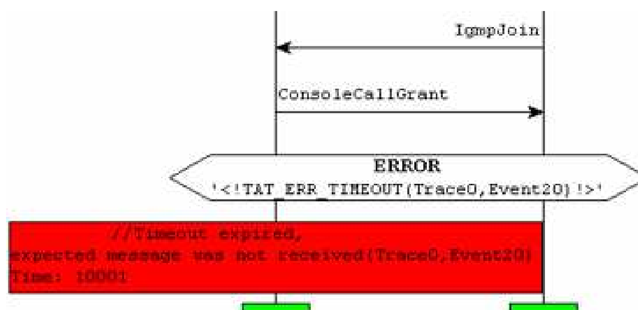


Fig. 4. Fragment of MSC log file with an error indication

As soon as a test suite in target language is generated by TAT, it can be used for testing SUT. The results of testing can be observed in log files (an example of a log file fragment in MSC format is represented in Fig. 4).

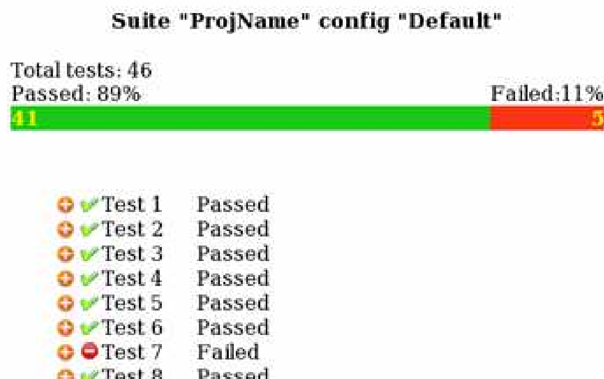


Fig. 5. Test report

TAT tool called Offline Test Results Analyzer (OTRA) is responsible for "offline" test results analysis. It compares source MSC charts (the charts which were used for test suite

generation) with logs in MSC format and generates a test report in HTML format. Fig.5 illustrates a test report example.

Each line in a test report is a link to another HTML page with comparison details and failure description (if corresponding test failed).

Described approach of automated test suites generation on the base of formal specifications can considerably reduce efforts and time of test cases preparation. Of course, this approach is not applicable to covering any requirement because sometimes requirements can not be expressed in terms of MSC charts or a sequence of key events. So, test engineer must understand the scope of its applicability and create tests for the not covered requirements manually. But, in practice (especially in telecommunication domain), majority of the requirements (especially functional requirements) can be automatically covered by tests in case of applying this technology.

Automation of functional tests generation is a powerful feature of the described testing automation approach, but it can also be applied for checking that interaction between SUT and environment takes place in compliance with some standard. It is well-known that in telecommunication domain compliance with a standard is at least not less important than checking functional requirements.

The underlying idea of the proposals about automation of conformance testing is similar to the idea of automation of functional testing, but has some peculiarities.

IV. AUTOMATION OF CONFORMANCE TESTING

Conformance testing is concerned with the assessment of the extent to which an implementation of system conforms to a specification. [14]

The process of interoperability testing and conformance testing is usually time consuming and expensive. However it is essential because the cost of releasing a product (especially in telecommunication domain) that does not operate correctly is very much higher.

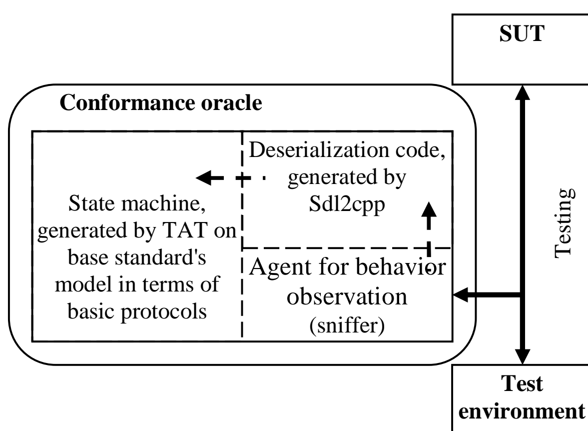


Fig. 6. Conformance oracle

And the problem here can be formulated as follows: if I have a SUT which interacts with test environment or with other system(s), how can I check whether this interaction takes place in accordance with some standard or not?

It is not easy, but very important question.

Another important question is: “Are there any inconsistencies in the standard’s specification?”

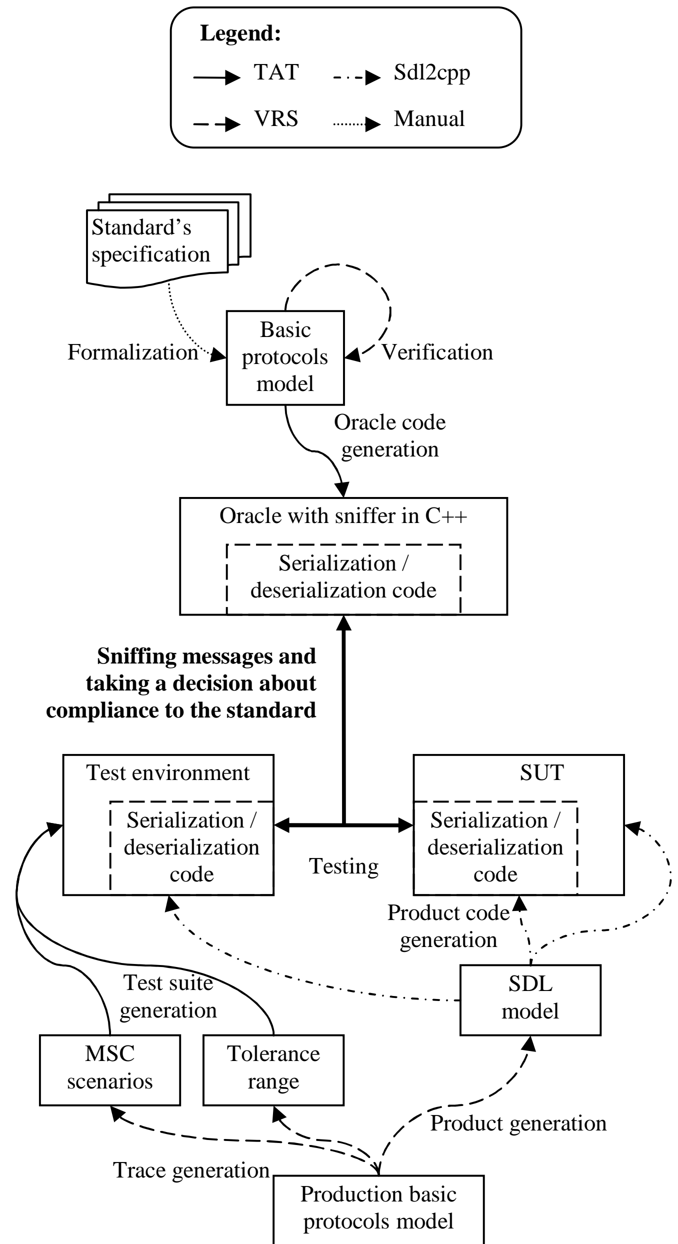


Fig. 7. Conformance checking approach

The proposing approach, illustrated in Fig. 7, tries to answer both of them. The first its step, verification, can answer the second question. Verification of a standard is not an easy task. Formalization of the standard’s specification is the most effort-consuming part of this step, and it can take much time. Unfortunately, formalization is a process which in common case can hardly ever been automated (because it is a process of requirements understanding and interpretation). However, verification can not only formally prove absence of errors, but also help us answer the first question.

In case of using notation of basic protocols, formalized model is represented as a set of states and transitions between them, i.e. as a state machine.

This state machine can be used as an oracle if place it between SUT and test environment or between SUT and other system(s) (which conform the standard).

So, the second step is generation of target code of this state machine and using it together with an agent for behavior observation (for example, a network sniffer for telecommunication domain). TAT code generation engine is going to be used for generation of the state machine code.

Normally, when Sdl2cpp generates SUT's target code from SDL, it also generates serialization and deserialization functions for each message which can be sent to the environment or received from the environment. These functions are both used on the SUT side and on the test environment side for encoding and decoding messages. So, it is not required for the user to define messages' structure (and corresponding serialization/deserialization functions) twice. In other words, Sdl2cpp can automatically configure TAT and share encoding/decoding code with it. This makes message exchange transparent for the user who works on the "signals" level of abstraction and is not concerned about what is actually sent via sockets. However, observation of socket-level messages remains possible. The same idea is also applicable to the oracle which, of course, requires a means of mapping sockets-level messages to high-level signals.

So, conformance oracle will consist of the following parts (as shown in Fig. 6):

- State machine generated by TAT on base of standard's model in terms of basic protocols;
- Deserialization code generated by Sdl2cpp;
- Agent for behavior observation.

The third step is testing. On the testing phase SUT and test environment communicate as usual (through signals exchange via network). And the only difference is presence of the conformance oracle which intercepts all the signals using network sniffer, decodes them using deserialization functions (provided by Sdl2cpp) and takes a decision about compliance with the standard.

Decision about compliance with the standard is taken in accordance with the following algorithm:

1. The oracle intercepts a message and tries to decode it. If the message can't be decoded, it does not conform to the standard.
2. If the message is successfully decoded, the oracle checks existence of any transition from its current state where condition of the transition is receiving this signal. If there is no such transition, the system does not conform to the standard.
3. Then the oracle checks signal's parameters. If parameters' values are within admissible range, the oracle performs transition to the next state and waits for the next signal; otherwise, the system does not conform to the standard.

The more test are executed with the conformance oracle, the more the user is confident that SUT conforms to the standard.

V. CONCLUSION

The developed approach of testing automation was successfully applied in several large-scale projects in telecommunication domain.

The maximum benefit (cost and time savings; increase of quality) from this approach can be achieved in case of using it as a seamless part of the overall technology, represented in section 2. But usage of TAT as a stand-alone test suite target code generator from formal specifications remains possible, so described testing automation proposals are also applicable to other software development processes.

Conformance testing automation proposals seem to be very promising because in spite of high complexity of standards formalization, automation of conformance testing together with standards verification can significantly increase product quality.

REFERENCES

[1] CDMA Development Group - http://www.cdg.org/
[2] Drobintsev P.D. Integrirovannaia tehnologia obespechenia kachestva programmnih produktov s pomoshiu verifikacii i testirovania. Kand. dis., SPbGPU. 2006. 238 p.
[3] Letichevsky A., Kapitonova J., Letichevsky Jr., A., Volkov V., Baranov S., Weigert T. Basic protocols, message sequence charts, and the verification of requirements specifications, Computer Networks: The International Journal of Computer and Telecommunications Networking, v.49 n.5, p.661-675, 5 December 2005.
[4] TAT User's Manual © 2001-2005 MOTOROLA.
[5] Hoare C.A.R. Communicating sequential processes, Prentice Hall, London, 1985.
[6] Letichevsky A.A., Kapitonova J.V., Volkov V.A., Vyshemirskii V.V., Letichevsky Jr. A.A. Insertion Programming // Cybernetics and Systems Analysis, Volume 39, Issue 1 (January 2003), p.16-26.
[7] Letichevsky A.A., Kapitonova J.V., Volkov V.A., Letichevsky Jr A.A., Baranov S.N., Kotlyarov V.P., Weigert T. System Specification with Basic Protocols // Cybernetics and Systems Analysis, Volume 41, Issue 4 (July 2005), p.479-493.
[8] Baranov S., Jervis C., Kotlyarov V., Letichevsky A., and Weigert T. Leveraging UML to deliver correct telecom applications in UML for Real: Design of Embedded Real-Time Systems by L.Lavagno, G. Martin, and B. Selic (editors), pp. 323–342, Kluwer Academic Publishers, 2003.
[9] Recommendation ITU-T Z.151. User requirements notation (URN), 11/2008.
[10] ITU Recommendation Z.120. Message Sequence Charts (MSC), 11/99.
[11] OMG Unified Modeling Language - http://www.omg.org/spec/UML/2.2/
[12] ITU-T Recommendation Z.100, CCITT Specification and Description Language (SDL), 03/93.
[13] IBM Rational SDL Suite - http://www-01.ibm.com/software/awdtools/sdlsuite/
[14] ETSI Conformance Test Specification - http://portal.etsi.org/mbs/testing/conformance/conformance.htm

Test suite development for conformance testing of email protocols

Nikolay Pakulin
ISP RAS
npak@ispras.ru

Anastasia Tugaenko
ISP RAS
tugaenko@ispras.ru

***Abstract*—The method for testing electronic mail protocols in the Internet to conform to the standards based on formal specifications is presented. The method is based on automated testing technology UniTESK in which functional requirements are formalized as pre- and postconditions and test sequence is generated on-the-fly from finite state machine (test state machine) traversal. The method is illustrated by the test suite development for SMTP and POP3 protocols using JavaTESK – a specification extension of Java language.**

***Keywords*-formal specifications; model based testing; protocols testing; conformance testing.**

I. INTRODUCTION

Emails are fundamental to modern communications between people. Hundreds of millions of emails float every day around the Internet. Reliability and correctness of the emailing infrastructure is vital to the modern information society. In this article we concern two aspects of these questions – reliability of (1) mail transfer in the Internet and (2) delivery of the email to recipient, typically a human being.

Most of emails in the Internet are transferred by means of SMTP – Simple Mail Transfer Protocol [1]. It is a text-based protocol with two parties: a client and a server. Client issues commands and server executes them, returning status code and other details if needed. SMTP has its own overlay network over Internet comprised by numerous mail servers and relay agents used to forward emails between various domains. A feature of SMTP is that each physical server could operate as both SMTP client and SMTP server: being a server it accepts an incoming email and becomes a client to forward it to a next hop.

SMTP is used to send messages, but when an SMTP implementation identifies that this is the final destination of an email it stops forwarding the mail and places it in an internal (implementation-specific) storage. To retrieve emails from the storage end-users utilize other protocols: POP3 (Post-Office Protocol, version 3 [2]) or IMAP4 (Internet Mail Access Protocol version 4 [3]). Both protocols are text-based with distinct roles of a client and a server. Clients access storage issuing protocol commands and servers provide required information in their replies. Typical POP3 and IMAP4 implementations support only one role at a time.

On its way from the originator to the recipient an email is processed by a number of intermediate servers. A typical case is that those servers come from different vendors thus having different implementations of the email protocols. The total reliability of emailing infrastructure substantially depends on the reliability of each server in the way and the compatibility between implementations.

Nowadays protocol conformance testing is the basic method of attesting implementations compatibility. That rationale for this statement bases on the suggestion of good protocol quality: if two implementations confirm to a protocol specification then they are compatible, they can correctly communicate with each other.

Despite of more than twenty-year history of mail protocol service and existence of dozens of SMTP/POP3/IMAP4 implementations there are no open and implementation-agnostic conformance test suites for those protocols. We believe that there are several reasons of this.

First, the simplicity of mail protocols is seeming. Let's explore the mail protocols features in the context of testing:

1) Mail protocols are underspecified: a large part of functionality is left to implementation developers; specifications prescribe several variants of possible system behavior;
2) mail protocols are nondeterministic, the standard allows various system behavior alternatives including refusal in mail message delivering or connection tear;
3) mail protocols requirements differ in the level of obligations (MUST, SHOULD, MAY);
4) protocol architecture is extensible, protocols implementation may use different extensions for supplemental functionality or even overlapping functionality.

Listed features demonstrate complexity of the task of conformance test for email protocols.

Second, developers has to focus testing on another big issue of mail server development: processing of a large number of settings necessary for practical use – parameters of routing, authentication, security, mail messages depository etc. We studied test suites developed by several open source implementations. The test suites turned out to be tightly coupled with the implementations under test (IUT) and non-portable to servers from other vendors, they use testing implementations features setting parameters, access to internal state, execution in the same process as the implementation. Tests designed for one implementation could not be applied to other implementations. Moreover, as shown by the analysis, these tests are inappropriate for

conformance testing, they are oriented to checking implementation for numerous settings correctness that don't directly connected with SMTP and POP3 standards. The problem is that such approach to testing doesn't guarantee servers' compatibility to the standard. The situation is even worse – our conformance tests detected a serious functional defect in James server – cycling at certain conditions – that was overlooked by the James functional tests.

Also it is necessary to note that there is a special tool developed in Apache Project – Mail Protocol Tester (MPT) – for verifying correctness of server replies. The input data for this tool is a script that specified server stimuli and expected server replies. The program uses regular expressions for comparison of IUTs replies and expected replies from defined scripts. If reply mismatches program stops and throws the mistake message. Apache James MPT does not support branching, cycles and parameters usage in tests.

Exact relationship between tests and standard's requirements allows hard confidence estimating in terms of external user of mail service. As the example with server James shows, the thoroughly testing of internal functions of implementation doesn't guarantee the functional quality of implementation in real environment.

Proceeding from the above arguments, we believe that the problem of conformance testing to the standards of mail protocols has significant practical importance. The ultimate goal of our research is to develop an opensource general-purpose conformance test suite for SMTP, POP3 and IMAP4. In this paper we present a model-based approach to conformance testing of email protocols. The presented approach focuses solely on conformance and does not consider other aspects of email infrastructure validation, such as interoperability testing, performance testing, reliability testing etc.

The paper is structured as follows: section II gives an overview of the existing approaches to protocol conformance testing and discusses why model-based testing is used in our approach. Section III provides a quick introduction to UniTESK technology that lays in the basis of our approach and Section IV introduces the proposed test development process. In section V we present the test suite for SMTP and POP3 developed so far and Section VI discusses pros and cons of the proposed approach. Section VII summarizes results achieved by now and highlights directions of future research.

II. Mail protocol testing

In the contemporary industry conformance testing of protocol implementations is mostly based on manual development of test suites consisting of independent test programs written in specialized or general-purpose programming language. Such programs are referred to as test cases; they implement stimulus test sequence generation, passing generated test inputs to the IUT, reading and analysis of observed outputs [4].

Let's consider requirements for test suite. Test suite for conformance testing must possess the following properties:

1) Requirements traceability. Tests must correlate with standard requirements. It must be clear for each requirement which test it is covered by.
2) Variety of settings for implementations features (MUST, SHOULD, MAY and others). There must be an option to define the set of requirements supported by an IUT and avoid requirements that IUT does not implement.
3) Completeness of test suite in terms of requirements coverage. Resulting test suite must cover at least all obligation requirements.

Test cases approach doesn't provide evident traceability of requirements. Requirements completeness in terms of coverage in such approach is also complicated. We rejected TTCN3 [5] and JUnit [6] because they don't provide formal connection between tests and requirements.

Besides that large number of tests presented as separate programs results in code redundancy or complex and complicated connections. It is necessary to use methods permitting decomposition of test suites and providing requirements traceability.

Required possibilities are given by tools based on formal method approach. Utilizing of formal specifications allows to:

1) define formal connections between requirements and tests; automatically backtrace quality of testing in terms of specification coverage;
2) using model repeatedly for checking correctness of implementations behavior;
3) generate test stimuli in terms of model and automatically filter redundant stimuli.

Also when choosing a method for generating test sequences it is necessary to take into account features of mail protocols. Particularly because of protocol behavior is nondeterministic and underspecified, one should choose approaches providing test sequences generation with a glance of IUT replies. As well when testing mail protocols it is complicated to make prediction for result or define equivalence of traces. Automatic verdict generation from specifications postconditions may solve the problem of verifying correctness of IUT behavior.

There are many instruments and approaches for testing. NModel [7] represents model system in C# language and provides basic facilities for on-the-fly testing and coverage maximization. But for on-the-fly testing test developer must write separate program describing complex traversal strategy. The current stable version of SpecExplorer [8] doesn't support on-the-fly testing.

Toolkits UniTESK [9] and Conformiq Qtronic [10] support formal specifications notation, automated on-the-fly test stimuli generator (code-level, there is no need to write separate program) and automated test results analysis. For this project the UniTESK was selected.

In the UniTESK technology formal specifications which formalize requirements as pre- and post- conditions are

used for generation of test sequences. Also for test sequences generation must be given a certain finite state machine (test state machine). The test process in UniTESK is automatic traversal of test state machine in which IUT behavior is automatically verified by test oracles; test oracles are generated from formal specification. The utilizing of formal specifications allows automating verification of behavior correctness and estimation of hard confidence; presenting test as state machine makes possible to automatically generate long and various sequences of test events.

Authors used presented method for developing test suites for protocols SMTP and POP3. From tools implementing the UniTESK approach JavaTESK [11] was chosen. JavaTESK uses the programming language Java with a number of extensions for record formal specifications and specify tests.

III. UniTESK technology overview

The standard format for Internet protocol standardized documentation is defined by documents RFC (Request for Comment). Requirements in these documents are stated in English and correspond informal text that describes desirable system behavior. In the UniTESK technology (Fig. 1) specialized specification languages – extensions of Java and C – are used for record requirements. In this work was used Java extension JavaTESK.

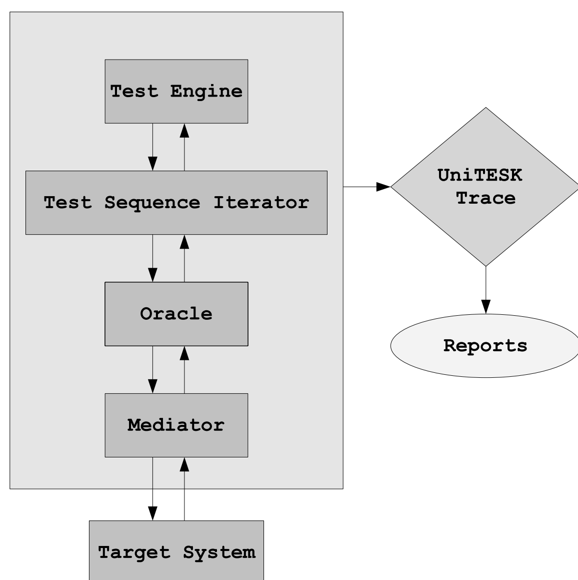


Fig.1. UniTESK Test Architecture

Recording the informal requirements of standardized documentation in formal language represents the protocol model. In UniTESK approach the formal model is constructed in terms of finite state machine. Transitions between states may be given in explicit or in implicit way. In case of explicit definition of transition the model contains algorithm for calculating next state and protocol reaction. Presentation of implicit transition is a predicate which defines restrictions on acceptable states and protocol reactions.

Specification in JavaTESK usually consists of one or few specification classes which describe states and transitions of modeling protocol. Protocol transitions are presented as special methods (specification methods). In addition there is a possibility to define restrictions on acceptable set of states by means of type invariants (type restrictions) and state variable invariants.

Definition of implicit transitions realized as pre- and postconditions. In preconditions there are restrictions on acceptable stimulus parameters values and on states from which stimuli may be given. The IUT may react on stimuli by changing state, giving a reaction or both. Postconditions define acceptability of demonstrated behavior.

For modeling IUT behavior one uses the set of data structures which referred to as abstract states. For verdict pronouncement about the correctness of IUT behavior UniTESK uses data from model abstract state.

In UniTESK both stimuli for the IUT and its reactions are described in terms of model; model is defined by the formal specifications. Correlation between the model and the IUT is established by an intermediary – mediator – which translates stimulus parameters from model form to protocol messages, IUT reactions to model representation and if necessarily transports changes from the IUT state to the abstract state.

Test scenario defines stimulus sequence applied to IUT. The metamodel of finite state machine is used as the theoretical basis for constructing scenarios. In JavaTESK test state machine is defined in scenario class which contains the procedure for calculating current state and iterator of test stimuli. The JavaTESK tool contains test engine for constructing test stimuli sequences from test state machine description.

IV. The proposed method for mail protocols conformance testing

Mail protocols may be in several states. When receiving certain stimulus they generate and send reactions and jump to another state or leave in current. With a glance of this fact on the basis of instrument UniTESK the method for testing mail protocols was developed. The developed method contains the following main steps:

1) Analysis of knowledge domain. Developing of examples and elementary tests. This step doesn't give any visible results but it is important for detailed protocol understanding and helps in implementation of next steps.
2) Creation of requirements catalogue. Requirements catalogue is a database or a table with description of requirements. Catalogue's record contains not only requirements description but also requirement identifier, type (syntax or functional), severity, link to the place in the RFC and maybe other attributes.
3) Designing of lite protocol model. Creation of experimental tests – test state machines with only one state. Lite protocol model includes information about commands and about possible reactions to these commands. Experimental test consists of

specification, mediator and scenario classes. Specification class on this step includes only signatures of methods; all verifications are making in scenario class. Such test referred to as linear test; applying stimuli and reading reactions are process in certain order defined in scenario class.

4) Designing of conceptual protocol model. Extracting states of basic protocol. Creation of test state machines with dedicated states. Expanding experimental test – addition the block which makes transitions between states. Conceptual model defines behavior of observing system as operations on some set of abstract components and composing objects. These components are used only for behavior modeling and may not conform to the model extraction. Addition of block for making system transitions between states turns test from linear to automatic test. In this test construction of stimulus sequence is made from test state machine traversal; applying stimuli and reading reactions are made only from acceptable states.
5) Requirements formalization. Relocation of verification of IUT responses into specification class. Checking completeness and consistency of requirements is made while formalizing requirements. The result of this step is the formal protocol specification written on one of the special program languages. In our case the Java extension JavaTESK was used.
6) Enhancement of scenario and specification for covering all requirements. In this step scenario classes contain only stimuli. The order of stimuli is formed from state machine traversal and depends on applying stimuli in certain states conditions. Usually one scenario class is responsible for the certain requirements section. For covering all formal requirements few scenario classes may be needed.
7) Execution of test suites and analyzing the results. Analysis may show that not all requirements are covered by generated test suite. If not all requirements are covered then step 6 must be repeated until covering all requirements from catalogue.

V. Method application for protocols SMTP and POP3 testing

The first step in writing tests for SMTP and POP3 implementations was analysis of knowledge domain, sending emails directly from server console. Then the requirements from RFCs were marked and categorized for types: commands and replies, routing, notifications, server settings, mail headers, mail body, etc. On this basis the lite protocol models were designed; test state machines consisted of only one state from which sent commands (for SMTP: EHLO, HELO, MAIL FROM, RCPT TO, DATA and others, for POP3: USER, PASS, LIST, STAT, RETR, DELE, TOP and others), received replies (for SMTP: three digit numeric code – reply code, for POP3: "+OK" or "-ERR" replies) and left at the same state. On this step implementations of test suites generation had a formal interface, specification classes was consisted of only methods' signatures; all IUTs behavior correctness verifications were made in scenario classes. Scenario classes consisted of methods applying (by means of mediator classes) stimuli to the IUTs, reading servers responses and returning verdicts about correctness of severs behavior. Mediator classes transformed the IUT stimuli format to model systems format and vice versa.

Then the basic protocol states were marked. On this step the new blocks responsible for making model system's transitions between states were added to the scenario classes. Specification classes were not changed.

In the next step blocks responsible for verification of IUT behavior correctness and blocks that make transitions between systems states were relocated from scenario classes to specification ones. Scenario classes solely applied stimuli to the IUT. From now certain commands could be passed only from acceptable states of state machines. The possibility of such verification is achieved by recording acceptable states in preconditions of specifications. Iterator every time checks the current state and whether the applying of the next command is allowed in current state. Also it gives the opportunity to check the fact that servers don't send commands from forbidden for such commands states.

VI. Discussion

While testing systems it always necessarily to know when testing may be considered as completed, what requirements have already been tested and what requirements are to be tested. Test cases testing cannot answer this question because correlation between tests and requirements is given informally in form of traceability matrix.

The utilizing of formal specifications allows formulation the exact unambiguous hard confidence criteria – testing may be completed when all elements of appropriate formal specifications are covered. UniTESK uses procedure of counting covered requirements and allows to define some selecting criterion for scenarios – if applying of certain scenario doesn't increase test coverage then system misses it and moves to the next scenario.

One of important advantages of this method is separating the class in which makes the pronouncement of verdict. The oracle which is generated from specification postconditions is responsible for verdict pronouncement. Due to this one hasn't to invent special functions for checking the correctness of IUT behavior.

To the lows of the method based on formal specifications one may attribute the absence of the quick test suites updating ability. When testing in terms of test cases new test results immediately from a simple test program. When testing in terms of formal specifications for developing new test it is necessary to thoroughly study requirements, formalize and classify them. Only after these preparations one may set out to write specification, mediator and scenario classes. Through this the period for new test development is increasing. But after specification,

mediator and scenario classes have written one got not a single test but a set of tests responsible for corresponding requirements class.

VII. Results and further research

For protocol SMTP were marked 51 basic requirements, 43 of them are related to server commands and replies (11 of them are mandatory and 4 are optional), 8 related to routing (all are mandatory). For protocol POP3 were marked 58 requirements for all functionality, 5 of them are mandatory and 6 are optional. All marked requirements are covered. Developed test suites were applied for testing open source mail protocols implementations – Apache James, hMailServer, Postfix and Dovecot. In the course of testing the following disagreements between protocol implementations and standards [1, 2] were detected:

- absence of required commands supporting;
- protocol rules violation (passing commands from forbidden for such commands states);
- wrong reply codes to the protocol commands;
- cycling while redirecting mail.

The feature of mail protocols is that mail protocols are extensible. Certain extensions supplement existing functionality, i.e. add new requirements which are not in a contrast with requirements from main standard. But there are also such extensions which radically alter the protocol structure thereby discarding some requirements from the basic standard. For checking such extensions one should modify test suites in such a way that requirements that are amended by extensions have not been verified. Otherwise there will be no opportunity to reach 100% coverage of all obligatory requirements. In connection with many protocols be extensible there is a need for tools providing ability for generating test suites for testing different extensible protocols' implementations both supporting extensions and supporting only basic standard functionality.

VIII. Conclusion

The paper presents a new approach to mail protocol testing. The approach belongs to model-based testing domain, it uses contract specifications to formalize protocol specification and on-the-fly test sequence generation. The implementation of the approach is based on UniTESK technology. Distinctive features of this method are automated test sequences generation on basis of formal specifications, test coverage calculating which allows constructing stimuli in optimal way and also the presence of separate component – oracle – responsible for verdict about IUT behavior correctness returning.

Developed method was applied for testing of long used mail protocols implementations. In one of the tested implementations was found a critical defect – under specific circumstances while redirecting message the server is resending the mail to itself and the message never reaches the recipient.

References

[1] IETF RFC 5321. J. Klensin. Simple Mail Transfer Protocol. 2008.

[2] IETF RFC 1939. J. Myers, M. Rosem, Post Office Protocol – Version 3. 1996.

[3] IETF RFC 3501. M. Crispin. Internet Message Access Protocol – version 4rev1. 2003.

[4] ISO/IEC 9646. Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 1: General concepts. Geneva: ISO, 1994.

[5] ETSI ES 201 873-1 V3.1.1. Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. Sophia-Antipolis, France: ETSI, 2009.

[6] Unit testing framework [URL] http://www.junit.org/.

[7] Jonathan Jacky, Margus Veanes, Colin Campbell, Wolfram Schulte. Model-based Software Testing and Analysis with C#. Cambridge University Press, 2008.

[8] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, Lev Nachmanson. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer Microsoft Research, Redmond, 2007.

[9] A. Barantsev, I. Burdonov, A. Demakov, S. Zelenov, A. Kossatchev, V. Kuliamin, V. Omeltchenko, N. Pakoulin, A. Petrenko. UniTesK Approach to Test Development: achievements and Prospects. Proceedings of ISP RAS, No. 5, 2004.

[10] End-to-End Testing Automation in TTCN-3 environment using Conformiq Qtronic and Elvior MessageMagic. 2009

[11] JavaTESK: getting started (in Russian). Moscow, 2008.

Modeling and Analysis of WAP protocol Familyl

Marina Alekseeva, Ekaterina Dashkova, Dmitry Chaly
P.G. Demidov Yaroslavl State University
150000 Yaroslavl, Sovetskaya 14, Russia
marya_87@mail.ru, dea.yar@mail.ru,
chaly@uniyar.ac.ru

Abstract

The importance of data networks and multimedia cannot be overestimated in contemporary world, at the dawn of Information Era. Information collection, processing and distribution systems are the key points for applying the scientific knowledge. Specification and verification of communication protocols and boosting their performance became the topics of the day. Due to the development of mobile systems the wireless networks are in good demand.

Thanks to technological advances in wireless data transfer the hardware and software developers offer a wide range of new services like mobile internet. Innovations must be useful, convenient, flexible, fast and secure enough with the least possible amount of errors during operation.

One of the ways to provide mobile internet is a WAP stack of protocols. WAP abbreviation translates into Wireless Application Protocol or Wireless Access Protocol. The second reading is more precise in reflecting the meaning of WAP technology, purpose of which is to grant access to information stored directly in the Internet.

We propose a modification for Wireless Transaction Protocol (WTP is the transport layer of WAP) which improves the original flow control algorithm. The work includes new ideas of developing and improving WAP as one of the important technologies.We use NS2 (Network Simulator) which provides an opportunity to specify such network protocols and simulate their behaviors.

Index terms: *Wireless Application Protocol, Network Simulator, flow control protocol*

I INTRODUCTION

In the beginning of the 21st century computer networks became the most important way of communication. Wireless networks spread worldwide and became one of the contemporary technologies. There are several protocols which provide access to the wireless internet. One of the most popular of them at the beginning of the 21st century was Wireless Access Protocol (WAP) [6]. The most exact definition of WAP technology: to provide access to the information from the internet for the mobile devices which have some specific characteristics:

- amount of memory devices;
- small size of the phone's screen, as well as the limitations of his keyboard;
- low processor speed;
- low bandwidth of the communication channel;
- possible long timeouts.

WAP was developed to overcome these problems and that is its major difference from HTTP and TCP/IP. Note that the user does not resort to the assistance of additional devices, such as modem. WAP is a protocol that describes the way in which information from the Internet displays on the small screen of mobile phone.

WAP is a stack of protocols which consists of several layers. According the OSI (Open Systems Interconnection) model [8] WAP contains six layers (Application layer, Session layer, Transaction layer, Secure layer, Wireless Datagram Protocol and Bearers). Each layer is intended to perform a well-defined function. We are focusing on the transport layer. In our paper we are focusing on the transport layer, modifying flow control algorithm using ideas, which are the fundamental principles of ARTCP algorithm [1].This protocol offers a new method of flow control based on rate control of the transfer of segments in the network.

Protocol ARTCP (Adaptive Rate Transmission Control Protocol) borrows some of the mechanisms of the TCP protocol which is the main transport protocol of the internet. Protocol ARTCP differs from TCP standard. Segments are sent to the network not in the form of the burst, but separated by time intervals. Sending rate depends on receiving rate as both parameters are included into the ratio which is changing the principle of system's work. Algorithm

has a mechanism of adaptating according to the evolving conditions of a network.

We will check properties of our algorithm with the help of a network simulator. There are two modeling approaches: analytical approach and simulation approach. At the moment with the help of NS2, which provides an opportunity to specify such network protocols and simulate their behaviors, our model of the modified protocol WTP is in its final step of development.

II MAIN PART

A. System modeling. NS2.

Simulation is widely-used in system modeling for applications ranging from engineering research, business analysis, manufacturing planning, and biological science experimentation [5].

In the development of such kind of model we can use a static models as well as dynamic ones. And under the static model we understand the models which can be used to analyze the network conditions in the particular curtain moments of time, for example, analytical methods of calculations from the queuing theory. Under the dynamic model we understand – discrete stochastic models, for example, processes of generation of requests or processes of their services. A great amount of programming features for imitating modeling exists nowadays: libraries of functions for the standard compilers as well as specific programming languages [7].

There are some examples which show that different performance indicators can be took off from the model with the help of colored Petri Nets. CPN/Tools – is an instrument that gives an opportunity of visualized corrections, performance and analysis of colored Petri Nets (CPN - Colored Petri Nets). In CPN/Tools non classical variant of colored Petri Nets are used: time is added in the structure as well as embedded programming language is used CPN ML (it is based on Standard ML). That is why Network Simulator is the most perspective specialized packet for different performance characteristic analysis of various of protocols. NS2 allows to build communication protocol models of almost all kinds of complexity. There exist some protocol models which were built by the protocols developers with the help of NS2. This fact can tells much about the high quality of the models developed with this instrument [2].

Network Simulator (version 2), widely known as NS2, is an event driven simulation tool which is very useful in studying the dynamic nature of communication networks. NS2 provides users with a way of specifying such network protocols and simulating their behaviors.

NS2 allows to create different types of traffic from the simplest ones which obeys the Poisson law to self-similar. Undoubted advantage of this simulator for the wireless network situation is its ability to define the nodes movements with the help of *Scenario Generator* mechanism. It is important to underline embedded opportunity of animation of different scenario results. Full as well as mere version of NS2 contains an instrument for animation of results of the model performance – nam (Network Animator). It is implement on Tcl/Tk. Only full version of NS2 contains a tool Xgraph. It helps graphically display the modeling results. This fact is very important as its significantly simplify efforts for results visualizing (there is no need to install specific software especially for providing visibility of model performance).

NS2 suggest two steps of work. The first step is constructing a model with the help of programming on C++, and finally the use of the Object-oriented Tool Command Language (OTcl) for analysis of the model and simulating the network conditions. It allows us to include our C++ programming code to the NS2 environment (fig.1) [3].

We decided that NS2 is the most convenient tool for modeling the network behavior.

B. WTP.

The Wireless Transaction Protocol is responsible for reliable message delivery. Maximum Transfer Unit (MTU) is a maximum size of a packet in networks. WTP fragment and reassembly the messages that exceed the size of the network MTU (Maximum Transfer Unit). There are three classes of operation for this protocol. We are focusing only on class-2 operation for Wireless Transaction Protocol. Flow control in cases of fragmented messages, is performed by sending fragments in groups. Every group of

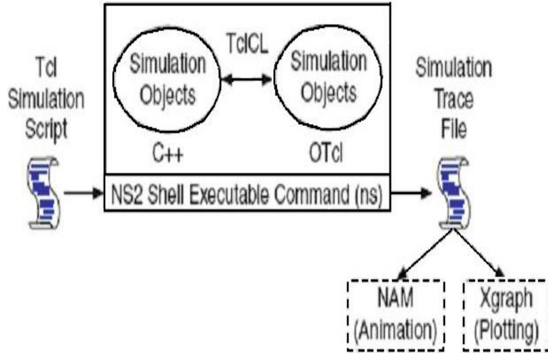


Рис. 1: Basic architecture of NS2

packets requires only one acknowledgement. The last packet of each group contains a special flag. This flag indicates the end of the group and receiver knows when to send an acknowledgment. The size of each group depends on the link characteristics and the device memory. It is necessary to avoid extra packet retransmission and data loss. Receiver sends a negative acknowledgement (NAK) if the end-of-group packet is received whilst intermediate packets are missing. This operation is repeated until the entire group is received and a positive acknowledgment is sent. If timeout expiries before any acknowledgement is received (ACK/NAK), then only the last packet of the group is retransmitted, and sender knows what packets have been lost. Wireless Transaction Protocol tries to minimize the number of unnecessary retransmissions [4].

C. Proposed model.

Extended Segmentation and Re-assembly of WTP protocol use either a sliding window based transmission or the traditional stop and wait mechanism [6]. The fundamental ideas of sliding window and stop and wait protocols are classical. They are used in various of transport protocols, but we presumes that this algorithms can be improved or more effective analogue can be developed. ARTCP flow control algorithm is based on the sliding window algorithm as well as on its own mechanism. The major advantage of ARTCP algorithm is that fact that it doesn't interpret the packet loss just as network overflow, this fact helps to avoid undesirable decrease of efficiency of network performance.

In our model we have three parameters, which we will change depending on network performance. Let t_s is the time interval between consecutive packets of the group which are sent from the sender SENDER to the receiver RECEIVER. And t_r is the interval between consecutive packets of the group which are received by the RECEIVER.

$$t_r = t_{r-1} * alfa + (1 - alfa) * t',$$

where

$$alfa = \frac{1}{(Am_p - 1)},$$

$t' = (now - t_{rl})$, now - current time, t_{rl} - the time when the previous packet of the group was received. Let Am_p - as the number of packets in the group. In our model there are two types of acknowledgments (ACK is a positive acknowledgment, NAK is a negative acknowledgment) (fig.2). When receiver sends an acknowledgment it transfers t_r with the help of it. Sender calculates the ratio $\frac{t_s}{t_r}$. Depending on the result of this ratio sender has several situations for analysis and further actions. The first one $\frac{t_s}{t_r} = 1$ reflects the perfect network conditions. The second one is $0.85 < \frac{t_s}{t_r} < 1$. All parameters can be changed by increasing Am_p, decreasing t_s and timeout:

$$Am_p = 2 * Am_p;$$

(exponentially increase Am_p);

$$t_s = \frac{t_s}{2};$$

(exponentially decrease t_s);

$$timeout = \frac{timeout}{2}$$

(exponentially decrease timeout);

We conclude that if $0.70 < \frac{t_s}{t_r} < 0.85$ there is no enough data for our algorithm to make a decision how to modify parameters (conditions of a network correspond to the established parameters). The fourth one is $\frac{t_s}{t_r} < 0.70$, so the network is congested, all parameters can be changed by decreasing Am_p, increasing t_s and timeout:

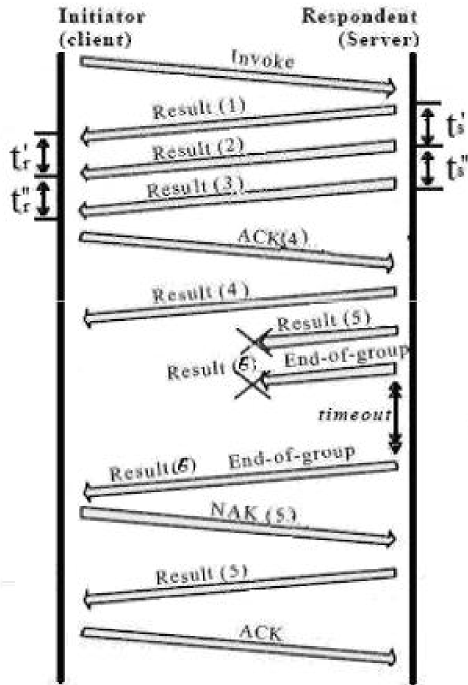


Рис. 2: Scheme of work

$$Am_p = \frac{Am_p}{2}$$

(exponentially decrease Am_p);

$$t_s = 2 * t_s$$

(exponentially increase t_s);

$$timeout = 2 * timeout$$

(exponentially increase timeout);

III CONCLUSION

With the growth and development of communication systems the economic benefit from increased efficiency of communication protocols (such as WAP) can be very substantial. WAP requires serious improvement. We have developed a new method of flow control, which is based on the management of transmission speed, the number of outgoing information and the waiting time for a response from the recipient.

We concluded that the control algorithms of data streams using the insufficient number of network features. So they faced with the same error: overloading the network, and the subsequent loss or corruption of data. We propose the idea of the analysis of additional characteristics of the network and its subsequent improvement, depending on the incoming data.

We analyzed the transport layer (Wireless Transaction Protocol - WTP) of WAP protocol and concluded that the mechanism of flow control can be flexible and adapt to the conditions of the wireless environment. Wireless channels are characterized by high levels of errors and narrow bandwidth, therefore the network congestion occurs frequently (duration of waiting for a response from the server increases or data is lost). We have studied the weaknesses of the flow control mechanism of several protocols. We have made changes in the structure of the transmitted packets, entered additional information in the packet header, in a number of key functions of the protocol, which are responsible for sending, receiving packets and settings of timeouts. In the future, we can create a protocol based on verified and analyzed model.

ACKNOWLEDGMENTS

The article was prepared within the scope of Finnish - Russian University Cooperation in Telecommunications Program. The authors would also like to thank the dean of Yaroslavl Demidov State University Computer Science Department P.G. Parfenov for interest and support of this project and the head of scientific-educational center "Center of Innovation Programming"Professor V.A. Sokolov for helpful advices.

REFERENCES

[1] I. V. Alekseev, V. A. Sokolov, D.U. Chaly "Modeling and analysis of Transport protocols for computer networks", *Yaroslavl State University*, 2004. (in Russian)

[2] V. A. Sokolov, D.U. Chaly, "Methods for investigating the behavior of transport protocols in congested network Distributed Inform.-computational resources, and math. modeling, MKVM-2004, p.129.

[3] T. Issariyakul, E. Hossain, "Introduction to Network Simulator NS2", DOI: 10.1007/978-0-387-71760-9 1, Springer Science + Business Media, LLC 2009, p.20-23.

[4] C. Ladas, R. M. Edwards AMIEE, G. Manson "WAP performance on an end-to-end scheme", The Centre for Mobile Communications Research (C4MCR), The University of Sheffield.

[5] http://www.isi.edu/nsnam/ns/

[6] http://www.openmobilealliance.org/Technical/wapindex.aspx - WAP specification.

[7] Y. Koucheryavy "NS-2 as a universal means of simulation networks", Tampere University of Technology, Telecommunications Laboratory, Tampere, Finland.

[8] A.S. Tanenbaum "Computer networks", - St. Petersburg, Peter, 2003. - 992 p.

[9] Aust Stefan, Nikolaus A. Fikouras, Görg Carmelita "Enabling Mobile WAP Gateways using Mobile IP", Department of Communication Networks (ComNets), Center for Information and Communication Technology (IKOM) University of Bremen, Germany.

[10] Moon Il-Young "Performance Analysis of WAP Packet Transmission Time and Optimal Packet Size in Wireless Network", School of Internet Media Engineering, Korea University of Technology and Education, Republic of Korea. Springer-Verlag, Berlin Heidelberg, 2006.

[11] http://nile.wpi.edu/NS/

On the Formal Specification of Automata-based Programs via Specification Patterns

Andrey A. Klebanov

Abstract—Model checking is a well developed verification technique still it is not widely adopted. One of the reasons is that defining formal specification is an error-prone and time-consuming task. This paper gives an overview of the ongoing research which focuses on expressing verifiable requirements in controlled natural language in the framework of automata-based programming.

Index Terms—Model checking, Specification, Temporal logic, Language parsing and understanding.

1 Introduction

AUTOMATA-BASED programming [1] – is a software development paradigm based on the extended finite-state machine model. In this approach programs are represented as a system of automated controlled entities which behavior is described by the system of interacting state machines.

Model checking [2] could be successfully applied to automata-based programs [3], [4]. The idea of model checking is to verify the consistency between finite-state model (*Kripke structure*) and formal specification expressed as a set of temporal logic formulae. In verification the main advantage of automata-based approach over traditional ones is a high extent of automaticity as in automata-based programs behavior model is defined a priori. Several methods [5], [6], [7] have been developed to automatically transform both a control system into a verifiable model and a counter-example produced by a verification tool back into automata model. Still for all the approaches a significant obstacle exists – considerable mathematical background is required for expressing specification as a temporal logic formula.

Design by Contract approach [8] could partly solve this problem [9] as contracts are much simpler formalism. However they are second to temporal logics in expressive power – they can do no better than specifying invariance, precondition or postcondition properties.

This paper presents an approach which battles temporal logics' complexity. The requirements are expressed in a controlled natural language defined by a formal grammar introduced further. The grammar is based on a set of specification patterns [10], [11] – a generalized description (both formal and in natural language) of a commonly occurring requirement on a permissible state sequences in a finite-state model of a system. Thus for each requirement equivalent verifiable formal mapping could be defined.

In [3] specification patterns are mentioned in the framework of automata-based programming: "… it is important to consider temporal properties patterns (structures) which are most suitable and appropriate for automata-based programs verification. Existence of such patterns would allow focusing on classes of temporal properties of automata models which definitely would facilitate flow chart development for automata-based programs verification." Still only one requirement (which is an instance of existing pattern) is outlined and no further development is provided.

The rest of the paper is organized as follows. Section 2 covers some background material on the nature of specification patterns and how they can be adapted for automata-based programming. Section 3 discusses specification patterns applicability analysis results. Formal grammar to derive verifiable requirements is introduced in Section 4. Finally, Section 5 makes a conclusion.

2 Specification Patterns

Specification patterns system has been introduced in [10], [11]. They are based on a specifications analysis for the programs developed in a traditional (i.e. non automata-based) way.

Patterns could be classified according to the hierarchy based on their semantics. Eight patterns which belong to one of the groups ("Occurrence" and "Order") are outlined. Patterns which belong to the group "Occurrence" specify occurrence or absence of the states in which a given state formula holds. "Order" group contains patterns which describe order of the states during system execution.

Pattern is described by its name (or set of names), intent, mappings to some formalisms (LTL, CTL and etc.), example of use and relationships with other patterns.

Each requirement has a scope – an extent of the system execution over which it should hold. Five kinds of scopes are defined:

- Global – entire execution path.

The research is conducted in scope of the Federal target program "Scientific and pedagogical personnel of innovative Russia for 2009 – 2013 years".

- *A. A. Klebanov is with the Computer Technologies Department, Saint-Petersburg State University of Information Technologies, Mechanics and Optics (SPbSU IFMO), Saint-Petersburg, Russia. E-mail: klebanov.andrey@gmail.com.*
- *The research is supervised by O. G. Stepanov, PhD, who is with the Computer Technologies Department, SPbSU IFMO, Saint-Petersburg, Russia.*

TABLE 1
"UNIVERSALITY" PATTERN

Intent	The pattern is used to describe a portion of a system's execution which contains *only* states that have a desired property. Also known as "Henceforth" and "Always".		
Mapping	LTL	Scope	Mapping
		Globally	□(P)
		Before R	◊R → (P U R)
		After Q	□(Q → □(P))
		Between Q and R	□((Q & !R & ◊R) → (P U R))
		After Q until R	□(Q & !R → (P W R))
	CTL	Scope	Mapping
		Globally	AG(P)
		Before R	A[(P | AG(!R)) W R]
		After Q	AG(Q → AG(P))
		Between Q and R	AG(Q & !R → A[(P | AG(!R)) W R])
		After Q until R	AG(Q & !R → A[P W R])
Example and known uses	The pattern could be used to specify either entire model or some group of states properties. For example when it's desired to express requirement like: "If an automaton is in state s, then P holds."		
Relationships with other patterns	The pattern is closely related to the "Absence" and "Existence" patterns. Universality of a state can be viewed as absence of its negation.		

- Before - execution path up to a given state.
- After – execution path after a given state.
- Between – execution between two given states.
- After-until – the same as "Between", but the right end of the interval where property holds is optional.

For the state-oriented formalism intervals are left-closed and right-open.

As an example, "Universality" pattern is presented in Table 1. Original "Example and known uses" section's contents is substituted with an automata-oriented example, this is a key idea behind patterns adaption for automata-based paradigm.

3 SPECIFICATION PATTERNS APPLICABILITY ANALYSIS

Creating a new pattern system for the formal specification of automata-based programs wouldn't make much sense without preliminary applicability analysis of the specification patterns described before. To carry it out it's necessary to analyze how requirements for automata-based programs (developed in SPbSU IFMO, Yaroslavl State University, Concern AVRORA and available at [12]) could be expressed via specification patterns. An example of intermediate results organization is presented in Table 2. Columns "Requirement" and "Original formal mapping" represent original requirements from the source (column "Source") expressed in natural language and one of the formalisms correspondingly. Pattern which instantiation with a real requirement leads to a formal equivalent is provided in column "Pattern, Scope". Equivalence proof is provided where required.

TABLE 2
INTERMEDIATE ANALYSIS EXAMPLE

Requirement	Original formal mapping	Pattern, Scope	Source
If either heater of one of the valves failure has happened, then coffee machine (automaton A_0) will mandatory change its state to the state 5.	AG((y_{31} = 4 | y_{32} = 4 | y_2 = 4) & y_0 = 2 → A(y_0 = 2 U y_0 = 5)))	Response (constrained), Globally AG(P → A(S)), P: (y_{31} = 4 | y_{32} = 4 | y_2 = 4) & y_0 = 2, S: y_0 = 2 U y_0 = 5	[4]

Altogether 77 requirements for 13 programs from 15 sources have been analyzed. 87% of the requirements could be expressed via five patterns. Remaining 13% couldn't be expressed due to the limitations of the pattern system or issues of the concrete automata-based model of the system. The percentage between used patterns is presented on the Fig. 1 and between used scopes – on the Fig. 2.

4 CONTROLLED NATURAL LANGUAGE GRAMMAR

Several approaches to extract verifiable requirements form the natural language specifications have been developed. Among the most popular [13] are natural language processing and formal grammars-based derivations.

In this paper grammar-oriented approach is used. The grammar is based on the specification patterns. Also wide spread variants of some patterns (i.e. "Response" and chain patterns) are added explicitly. So the requirement could be expressed both in natural language and in any formalism supported by the pattern system. An extract of the grammar is presented in Table 3; placeholders for real requirements are in `monospace font`.

To exemplify this, requirement from [4] is considered: "Coffee machine control system never gets to the state where it doesn't respond to either system timer events, or buttons "OK" or "Cancel". In the automata-based model of the coffee machine control system requirement "Doesn't respond to either system timer events, or buttons "OK" or "Cancel" corresponds to the predicate `act = end`. The adverb "never" implies to use "Absence" pat-

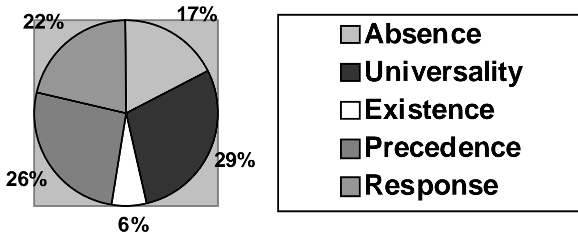


Fig. 1. The percentage between used patterns.

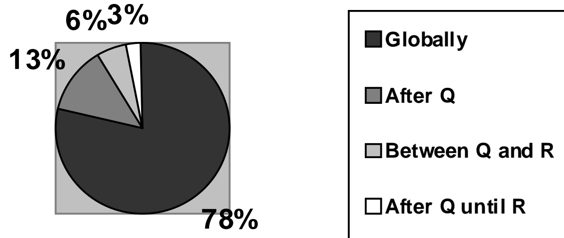


Fig. 2. The percentage between used scopes.

tern with a "Global" scope. The derivation follows:

<requirement> → <scope> <pattern> → For all the states holds <pattern> → For all the states holds <absence> → For all the states holds that never `P`.

Instantiating `P` with a real requirement leads to a desired requirement in natural language: "For all the states holds that never `act = end`." Formal expressions in CTL and LTL used for verification purposes are `AG(!act = end)` and `□(!act = end)` correspondingly.

5 Conclusion

Requirements expression as temporal logic formulae is error-prone and time-consuming task. An approach which facilitates this process has been introduced in this paper.

There are a few open issues left to work on in future. First of all it's tool support. It has been shown in [9] how JetBrains Meta Programming System [14] could be used both to develop and verify automata-based programs. Currently formal specifications are integrated with code but only as temporal logic formulae. The major improvement would be to replace them with the natural language specifications as described above. Besides similar to [13], [15], [16] some wizard to guide a user during property construction could be implemented. Finally, further use cases of the pattern system could be investigated.

TABLE 3
Controlled Language Grammar Extract

<requirement>	::= <scope> <pattern>
<scope>	::= «For all the states holds that» | «Before the state where `Q`, holds that» | «After the state where `Q`, holds that» | «Between the states where `Q` and `R`, holds that» | «After the state where `Q`, before the state where `R`, holds that»
<pattern>	::= <absence> | <universality> | <existence> | <constrained existence> | <precedence> | <response> | <constrained response> | <chain precedence> …
<absence>	::= «never `P`.»
…	…
<response>	::= «always if `P`, then eventually `S`.»
…	…

References

[1] N.I. Polikarpova, A.A. Shalyto, *Automata-based programming*, Piter, 2009. (in Russian)

[2] E.M. Clarke, O. Grumberg, D.A. Peled, *Model Checking*, MIT Press, 2000.

[3] K.A. Vasileva, E.V. Kuzmin, "LTL Verification of Automaton Programs," *Modeling and Analysis of Information Systems*, vol. 14, no. 1, pp. 3–14, 2007. (in Russian)

[4] E.V. Kuzmin, V.A. Sokolov, "Modeling, Specification, and Verification of Automaton Programs," *Programming and Computer Software*, vol. 34, no. 1, pp. 38–60, 2008. (in Russian)

[5] V.S. Gurov, B.R. Yaminov, "*Automata-based Programs Verification without Translation into Verification Tool's Input Language*," Proc. Conf. Scientific Software in Education and Research. 2008. (in Russian)

[6] M.A. Lukin, A.A. Shalyto, "*Automation of Visual Automata-based Programs Verification*," Proc. 15th Int'l. Conf. Advanced Intellectual Technologies and Innovation in Education and Science. 2008. (in Russian)

[7] E. Kurbatsky, "*Verification of Automata-Based Programs*," Proc. Sec. Spring Young Researchers Colloquium Software Engineering. 2008.

[8] B. Meyer, *Object-Oriented Software Construction, 2nd Edition*, Prentice Hall PTR, 2000.

[9] A. Borisenko, P. Fedotov, O. Stepanov, A. Shalyto, "*Reliable Software with Complex Behavior Development*," Proc. 5th Central and Eastern European Software Engineering Conf. in Russia. 2009.

[10] M.B. Dwyer, G.S. Avrunin, J.C. Corbett, "*Property Specification Patterns for Finite-state Verification,*" Proc. 2nd Workshop Formal Methods in Software Practice. 1998.

[11] M.B. Dwyer, G.S. Avrunin, J.C. Corbett, "*Patterns in Property Specifications for Finite-state Verification,*" Proc. 21st Int'l. Conf. Software Engineering. 1999.

[12] *Programming Technologies Department, Saint Petersburg State University of Information Technologies, Mechanics and Optics*, http://is.ifmo.ru/

[13] S. Konrad, B.H.C. Cheng, "*Facilitating the Construction of Specification Pattern-based Properties,*" Proc. IEEE Int'l. Requirements Engineering Conf. 2005.

[14] *JetBrains Meta Programming System*, http://www.jetbrains.com/mps/index.html

[15] R.L. Smith, G.S. Avrunin, L.A. Clarke, L.J. Osterweil, "*PROPEL: An Approach Supporting Property Elucidation,*" Proc. 24th Int'l. Conf. Software Engineering. 2002.

[16] O. Mondragon, A.Q. Gates, S. Roach, "*Prospec: Support for Elicitation and Formal Specification of Software Properties,*" Proc. Runtime Verification Workshop. 2004.

On Reasoning About Finite Sets in Software Model Checking

Pavel Shved
Institute for System Programming, RAS
email: shved@ispras.ru

Abstract

A number of static checking techniques is based on constructing and refining an abstract reachability tree (ART) and reasoning about Linear Arithmetics. For example, in BLAST, each program statement is represented as a series of assignments of a linear functions to variables, and the procedure of predicate discovery relies on Craig interpolation of linear arithmetics and equality with uninterpreted function symbols.

In this paper we propose an approach to extend the domain of mathematical operations a checker described can reason about with the certain operations with finite sets: adding and removing elements, testing whether set contains a particular element, or is empty. It being implemented, the ART doesn't split at each operation. The tradeoff of it is more complex formulas for a solver to handle and incapability of using set-related operations in loops.

We implemented the algorithm, proceeding from the resriction of making no modification in LA+EUF interpolation algorithm. We also provide results of the performance evaluation of the algorithm proposed and of the other known way to reason about finite sets.

Index Terms

Software verification and validation

1. Introduction

The approach to static verification through utilizing CEGAR [6] with reasoning about linear arithmetics has been proven succesful by such tools as SLAM [2] and BLAST [5]. The advantage of such approach to analysis is its precision (interprocedural analysis, lower false positive rate) compared to the other static analysis techniques. However, the tradeoff to precision has always been the speed and scalability of the process.

The tools described above work by constructing the locations reachable from the entry point by all syntactically plausible program paths. The locations reached are tagged with a set of linear constraints, which are discovered by an interpolation procedure (in BLAST it's Craig interpolation). These constraints limit the possible state of program variables just roughly, but precise enough to prove unreachability of specially marked "error locations".

The analysis stops when the program proves reachability of the error location or marks it with an empty region (i. e. with an unsatisfiable conjunction of constraints), having examined all possible paths. To avoid exploring an infinite amount of paths (for example, if there's a loop), the checker doesn't explore the locations that are tagged with a contraints implied by the constraints of some other location. The reason is that such exploration would produce a superset of paths and states as the children of the other node would.

All that being said, the analysis algorithm still has to produce different paths if the program branches execution, with an **`if`** statement, for example. Even if the control-flows are then combined together the ART will remain split, and each branch will be tracked separately until one of the branches is proven infeasible. This doubles the time required to verify the program.

It is a natural disadvantage of the approach, but such branching is an inherent property of the program being verified. However, the code verified consists not only of the program code itself, which comes as an input. Another part of code represents the instrumented safety property being verfied, and is a native language implementation of the abstract concepts behind the logic being verified. When an integrated verification system is built, the developers should pay attention that such verification code doesn't add extraneous complexity to the code of the original program.

Some safety properties can be defined in terms of finite sets. For example, a heap model may be represented as a set of currently available memory locations. Each `malloc()` operation adds a unique pointer to that set, and each `free(p)` call deletes a pointer p from that set. At each `free(p)` invocation a check is performed, whether the pointer being freed belongs to that set, and if it doesn't, the control flows to error location. Also at the end of the program a check that that set is empty is performed—i. e. that there's no memory leak.

A classical approach to this problem, universal quantification trick, attributed to [7], is that the set is represented as a single varible. This variable represents one of possible elements currently in the set. This is achieved by the following means. Each opertaion of adding element to set is implemented as an **`if`** statement, one branch of which overwrites the characteristical variable, the other keeping it intact. However, each branch doubles the amount of ART nodes consecutively build, thus expanding it exponentially. Ditto for deletion form the set.

One of the approaches to reducing the numbers of nodes checked is Large-Block Encoding [4]. While this approach is generic and will be of help in the case under consideration,

it suffers from a yet unsolved disadvantage. At the current state of art, it is not known how to produce a useful trace from the entry point to the error location.

The solution we propose is to define set operations that will be treated differently by the checking engine. We present a way to build a path formula that utilizes existing LA+EUF interpolation procedure to devise constraints caused by performing operations on sets. We also describe the model of regions, by which the ART nodes will be tagged.

2. Reasoning about finite sets

To make model checker support reasoning about finite sets, we need to modify certain parts of the algorithm. Second, we need to describe what will be the description of the regions. Third, we should modify predicate discovery algorithm, so that it yields information about finite sets. But at first, we define the concepts the approach works with.

2.1. Concept of a finite set

In this model, each set is recursively defined as either

- **Empty set** that contains no elements, or
- **Union** of an element (or, more precise, of a set that contains one element), which is an arbitrary expression, and of another set; or
- **Subtraction** of an element from a set.

Each set is a finite chain of these operations. That means that at the beginning of each program, all sets are considered empty[1]. This is necessary for all sets to be finite, since an unspecified set (even if it's finite) may have an arbitrarily big lenght.

The relevant operations, as shown in the table 1, operate with sets as with first-class values. These operations are called *set construction operators*, and they shall not branch execution.

Along with some operators to construct sets, a number of *set testing operations* (shown in table 2) are introduced:

- **Check if set contains a particular element**, the element being specified as an expression;
- **Check if set is empty**.

These operations don't branch execution; only examining the result of these checks is what branches execution. However, the common usage pattern is that one of the alternatives of such a check leads to an error state immediately, thus the amount of ART nodes explored doesn't expand exponentially to the number of checks. The problem of the classical quantification trick is in branching that happend at the points of set construction operators.

2.1.1. Definitions used in the article.

To shorten the descriptions, we present notations for some concepts used.

Set *constitution* and a set itself are different concepts. Constitution is how the set is constructed, according to set construction operators described in 2.1. A set that has a constitution C is denoted as $[C]$. The set constitution may be denoted like this:

1. contrary to integer variables that are considered unspecified untill they're assigned a value

Table 1. Set construction

`S = SetEmpty();`	place an empty set to S
`S = SetAdd(P,expr);`	place $P \cup \{expr\}$ to S
`S = SetDel(P,expr);`	place $P \setminus \{expr\}$ to S

Table 2. Set examining

`rslt=SetInTest(S,expr);`	check if $expr \in S$
`rslt=SetNotEmptyTest(S);`	check if S is empty

$$C = \emptyset + \{x_1 + z_1\} - \{2 * y_1 + 5\} + \{x_2\} + \{x_3\} - \{y_2\} \quad (1)$$

A number of sets can have a given constitution, depending on other external constraints. For example, given that $x > 5$, any set that contains exactly one number that is greater than ten, has a constitution of $C = \emptyset + \{x + 5\}$.

A set constitution may be presented in a *normalized form*, with consequent add and subtraction operations grouped together. If we unroll the constitution, up to a specific depth, and group together the consequent additions and subtractions, the constitution S may be presented in the following form:

$$S = S_n + \alpha_n - \beta_n + ... + \alpha_1 - \beta_1 \quad (2)$$

where α_i and β_j are sets, and

$$\forall i > 1 \rightarrow (\alpha_i \neq \emptyset) \wedge (\beta_i \neq \emptyset) \quad (3)$$

The normalized form is used to present different set constitutions in a universal way.

As usual, $foo.bar$ means *bar* "field" in the foo object. The notation is inherited from object-oriented programming and, we suppose, is easier for programmers to follow.

Also the following definitions will be used: given a function f, $D(f)$ is the domain of f, and $E(f)$ is the codomain of f. We will use the notation mostly for maps; thus a codomain of a map is implied to be the smallest set possible.

2.2. Predicate discovery algorithm for finite sets

In order to discover predicates specific to the finite sets, we need to build a path formula that describes a concrete state of the program variables in the error location. When we try to add the logic related to finite sets to this formula, it should retain the following properties. First, it should remain **precise**, i.e. only program states that happen during real execution of the program should satisfy the formula. Second, it should be written in **static single assignment form** (or, "SSA form").

Since the formula built without finite sets concept satisfies these properties, we can utilize it for building the formula *with* sets concept. The idea is that set construction operators do not have direct effect to path formula. However, these operators are followed to calculate the exact constitution

of each set encountered in the trace. The expressions that represent elements of sets are stored in SSA form. When set testing operator is encountered in the trace, the consitution of the set queried is already known; the constraints for the values of set elements are precise, and are described by the rest of the formula.

If the Craig interpolation procedure succeedes the predicates discovered form constraints for program variables. Under certain conditions, discussed in section 2.2.3, these constraints contain enough information to prove that the error path is infeasible. Note that the constraints should not include any symbols of the variables that represent sets. Only "usual" variables (that contain integers) are the domain of these contraints.

2.2.1. Formula for inclusion check. Each check separates the execution into two branches. Path formula is always built among one branch (i.e. the path from root to current node is exactly known). So, to implement inclusion check, for each of these two branches a separate predicate is needed:

1) SAT if the element belongs to the set, UNSAT otherwise
2) UNSAT if the element belongs to the set, SAT otherwise

Let's build the predicate that satisfies the requirements outlined under the first point. It can be built recursively. Given a constitution of a set, C, and the element (expression) x we want to check for inclusion into $[C]$, the predicate is:

- **Empty set**: $C = \emptyset$, then $x \in [C] \Leftrightarrow false$;
- **Union**: $[C] = [C_1] \cup \{y\}$ then $x \in [C] \Leftrightarrow (x = y) \vee (x \in [C_1])$;
- **Subtraction**: $[C] = [C_1] \setminus \{y\}$ then $x \in [C] \Leftrightarrow (x \neq y) \wedge (x \in [C_1])$;

Since error trace is always finite, the recursive unrolling always terminates, and the formula is always built. Since the path formula is in terms of LA+EUF, Craig interpolant always exists [9], and only (SSA-terms of) variables that appear in source code are used in them.

The predicate built that way satisfies the requirement under the first point. Its negation also satisfies the second point. So, we've built proper predicates for both branches for set inclusion check.

2.2.2. Path formula for set emptiness. As described in section 2.2.1, we need to present algorithm for two predicates:

1) SAT if the set is empty, UNSAT otherwise
2) UNSAT if the set is empty, SAT otherwise

We can notice that **a set is empty iff each element that was added to it was removed from it afterwards**.

Given the set $[S]$ such that, in a normalized form (see section 2.1.1),

$$S = S_n + \alpha_n - \beta_n ... \alpha_1 - \beta_1 \tag{4}$$

let's define S^- function as follows:

$$S^-(S, n) = \bigcup_{i=1..n} \beta_i \tag{5}$$

Figure 1. Demonstration of limitation of predicate-discovery algorithm — source code

```
x = 1; y = 2;
S = SetAdd(SetEmpty(), x);
x = 3; y = 1;
S = SetDel(S, y);
if (SetNotEmptyTest(S)) error();
```

Then the predicate

$$(\beta_1 \neq \emptyset) \wedge \left(\bigwedge_{i=1..depth(S)} \bigwedge_{a \in \alpha_i} \bigvee_{b \in S^-(S,i)} (a = b) \right) \tag{6}$$

is true if and only if a set with constitution S is empty, which is the predicate that satisfies the conditions under the first point.

We should note that the predicate (6) is finite, since all the sets α_i are finite and the result of S^- function is a finite set.

Though finite, (6) is nevertheless large, and it also contains disjunctions. This allows us to suggest that check for emptiness will be more complex and less scalable than check for inclusion. We will see if it's correct in section 4.

We could use negation of (6) as a second predicate, but due to its complexity we should search for another way. We note the following. Given a set with constitution S, the second predicate is equivalent to

$$x \in [S] \tag{7}$$

where x is a variable that doesn't appear anywhere in the trace. Indeed, if set is empty then for every x (7) is false, which means that it's unsatisfiable. If set is not empty, then there exists at least one element that belongs to it, and this makes (7) satisfiable.

The negation of (7) doesn't yield the first predicate, since the set $[S]$ is finite, hence there always exists an element that doesn't belong to it; this makes the negation of (7) satisfiable even if set being checked is not empty.

2.2.3. Correctness of predicate discovery. The predicate discovery procedure described above sometimes yields predicates that are false for each program location. Let's consider a sample program shown on figure 1.

If we convert this program to SSA form (and expand set checking predicate), we will get the program shown at figure 2. Interpolation procedure that may yield the predicate $x0 == y1$, it proves that `error()` is inreachable.

However, when we convert it back from SSA form, the predicate would look like $x = y$. If we check figure 1 again, we note that **in no program location this predicate was true**! That means that this predicate discovery procedure based on LA+EUF interpolation can't yield a predicate good enough if the value added to (or removed from) the set was later changed.

Figure 2. Demonstration of limitation of predicate-discovery algorithm — SSA form

```
x0 = 1;
y0 = 2;
S0 = SetAdd(SetEmpty(), x0);
x1 = 3;
y1 = 1;
S1 = SetDel(S0, y1);
if (x0 <> y1) error();
```

Figure 3. Instrumented program for which predicate discovery is correct

```
x = 1; y = 2;
Add1 = x;
S = SetAdd(SetEmpty(), Add1);
x = 3; y = 1;
Rm1 = y;
S = SetDel(S, Rm1);
if (SetNotEmptyTest(S)) error();
```

We can mitigate this obstacle by instrumenting before each set construction function call an assignment to a special variable that doesn't change later (see figure 3). It is possible unless a set construction operator is in a loop.

So, we later assume that **no set construction operation should be in a loop**. We beleive that it's an inherent limitation of our approach. However, even with this restriction the approach amy be useful if it performs well in experiments.

2.3. Regions for finite sets

Some variables encountered during the ART construction are considered to contain sets. The basis, based on which they're considered as such, is the application of relevant operations to them, and storing the result of those into them. The list of supported operations is in table 1. If a result is stored into a variable, or a variable is a set operand of any of those operations, the constitution of such a set is tracked in this ART node and its subsequent children.

The regions are constructed from the very beginning of the processing; this behaviour resembles lattice-shape-analysis [3]. During that initial step a set constitution (in the shape depicted at section 2.1) is tracked. When a value of an expression is added to or removed from a set, the expression is converted by assigning new names to its underlying variables. The converted expression then becomes a part of this set constitution. The rationale is similar to what was discussed in seciton 2.2.3: a value may change after it was added to a set, and set checking should take the old value into account. Instead of checking this requirement, we design region processing in such a way that it tries to work for programs that don't satisfy this condition, but does not guarantee results in this case.

Sometimes constitution tracking alone may be of help. For example, if it's known that a value was recently added to a particular set, the test whether it's empty should return negative response (note the first minterm in (6)). However, in other cases it's insufficient, and the relationship between expressions that represent set elements should be taken into account.

This is where the predicates discovered by the modified trace analysis procedure come into play. Each time a variable is added into a set, all predicates at the current region (we assuime Cartesian abstraction [1]) are also added to the set region, the variables in these predicates being converted in the same way as set elements have been. When determining the post-region of a set testing operation, both these accumulated predicates and set constitutions are taken into account. If they are sufficient to prove the program state infeasible the post operation adds to the cartesian predicate region the predicate that keeps the information about the variable the result is stored in.

Formally speaking, each region $reg = post(reg', e)$ consists of the following components:

- $reg.binding : variable \rightarrow variable$ (such that $E(reg.binding) \cap D(reg.binding) = \emptyset$) is the current mapping between actual variables, in terms of which the CFA blocks are expressed, and those used within sets. This binding is used to convert elements and predicates at current point. It is updated at each location, when e is a basic block of assignments.
- $reg.sets : variable \rightarrow constitution$ is a mapping from program variables to set constitutions (described in section 2.1). Elements of the sets are expressions over the $E(reg.binding)$ set of variables. This mapping is updated at each construction operation (see fig. 1); the mapping between how constitution is updated and what operation is along the edge is obviously inferred from the description in seciton 2.1;
- $reg.predicate_i$ is the predicate over the $E(reg.binding)$ set of variables. In a post-region of reg' after an edge e, a predicate belongs to this set if and only if:
  - it belongs to $reg'.predicate$ set; or
  - it is equal to one of the predicates in cartesian region in $post(region', e)$, the variables being updated according to $reg.binding$.

When at the location a set testing operator is encountered, the region machinery should decide whether the post-region is feasible. To notify the model checker that an infeasible location is encountered, the predicate over the variable the result is stored into is added to the Cartesian abstraction region.

In a current region reg such a predicate should be added when testing operation in e should fail on each set constitution that belongs to reg' region. Having rewritten the element expression involved in set testing operation with use of $reg'.binding$, and having applied the relevant formula of those in section 2.2, we get a formula $F_{[S]}(E(reg.binding))$. That is the exact formula

Figure 4. Header file with set-related functions

```
typedef int Set;
Set SetEmpty();
Set SetAdd(Set, ...);
Set SetDel(Set, ...);
int SetInTest(Set, ...);
int SetNotEmptyTest(Set);
```

that would have appeared in this location if we started error trace analysis now.

However, such analysis, if it had already been completed for a path this location belongs to, had yielded several interpolants. We know that the error trace—even before it's converted to SSA—already has some variables not changing their values (these are the instrumented variables introduced in section 2.2.3), so these values will be reasoned about in interpolants. These interpolants provide the necessary predicates to prove set-related path infeasible if it really is (since the formulas are equivalent). It means that the formula (with unbound variables $E(reg.binding)$)

$$F_{[S]} \wedge \bigwedge_{p \in reg.predicates} p \tag{8}$$

is UNSAT if the path to current location is on a subpath of an infeasible error path checked for an error before. However, if Cartesian abstraction succeeds in propagating set-related predicated to other paths, this formula may work too. That is one of the main benefits of lazy analysis [8].

2.3.1. Region coverage. Region machinery should address another problem: region coverage. For finite sets it is possible, but unnecessary to devise the relevant formula.

Since finite sets can not be operated with in cycles, there will never be a situation when one region would cover another one due to reasons devised when analyzing set-related operations. So the usual coverage checking procedure would suffice.

3. How it is embedded into BLAST

A special header with C funciton headers is created (see figure 4). It utilizes variadic arguments, since it's not known, what the types of the expressions added to sets will be.

When BLAST interprets the source code of a program that uses these functions, it replaces each function call with a separate edge in the CFA. Set functions are undefined, but a separate edge will be created anyway.

Set construction functions will be inserted as is. Set testing funcitons are intended to use in bodies of **if** operators. However, they will anyways be represented with a separate edge, the program being transformed to something like this:

```
tmp = SetInTest(S, expression);
if (tmp != 0) { ... } else {...}
```

Figure 5. Memory operations implemented in terms of finite sets

```
int counter = 1;
/* Set of allocated regions */
Set memory;
ptr malloc()
{ counter += 1;
  memory = SetAdd(memory, counter);
  return counter; }
void free(void* p)
{ if (!SetInTest(memory,p)) error();
  memory = SetDel(memory,p); }
void check_leaks()
{ if (SetNotEmptyTest(memory)) error(); }
```

Figure 6. Memory operations implemented with path splitting

```
int maybe(); //returns an arbitrary bool
int counter = 1;
/* One of the pointers in set */
void* M = 0;
/* One of the pointers deleted from set */
void* F = 0;
ptr malloc()
{ counter += 1;
  if (maybe()) M = counter;
  return counter; }
void free(void* p)
{ if (M == p) M = 0;
  if (F == p) error();
  if (maybe()) F = p; }
void check_leaks()
{ if (M != 0) error(); }
```

To avoid branching at the point of function call assignment, the predicates for such a check devised in section 2.2 $pred$ are utilized in the following formula:

$$(tmp = 1 \wedge pred_1) \vee (tmp = 0 \wedge pred_2) \tag{9}$$

where $pred_1$ and $pred_2$ are the predicates devised in each of secitons 2.2.2 and 2.2.1: one is satisfiable when condition being checked is true, and the other is satisfiable when it's false.

This doesn't affect correctness of any claims made above, although requires some extra prover work as formulas produced this way contain more disjunctions.

4. Performance evaluation

We evaluated our algorithms for simple programs. Each test program consists of consequent allocations of several

Table 3. Evaluation results (seconds elapsed; "X" means CSIsat failure)

# of regions allocated	1	2	3	4	5	6	7	8	9	10	15
"Trick" with checking leaks	1	5	52	540	1553	> 2000	> 2000	> 2000	> 2000	> 2000	> 2000
Sets with checking leaks	1	4	10	X	80	X	X	X	X	X	X
"Trick" without checking leaks	1	4	41	443	1289	> 2000	> 2000	> 2000	> 2000	> 2000	> 2000
Sets without checking leaks	1	3	6	17	36	70	200	333	X	X	X

memory regions and consequent deallocations of them, followed by an optional check for unfreed memory. We also introduced double-free errors and memory leaks to verify correctness of our approach. All such tests didn't demonstrate any errors in the algorithms used.

The checking for memory operation safety was similar to that presented in the introduction. For finite set it's presented on figure 5, and for universal quantification trick—on figure 6[2].

The complexity of the test programs range from one to fifteen allocated regions. We also thought that it would be fruitful to check how algorithms behave in absense of leak checking, because formula to check emptines. (6), is more complex than other set-related formulas. The results are in table 3.

We found out that after amount of allocations exceeds a certain limit, algorithms start failing due to the failures of the underlying Craig interpolant generator, CSIsat. We believe that the reason is that formulas fed to it appear to be more complex than it is capable to handle. We also see that leak checking (i. e. emptiness checks) is way more complex than reasoning if an element belongs to a set.

5. Conclusion

In this paper we showed that certain common properties checked by static analysis frameworks can be represented in terms of finite sets. Sections 2.1 and 2.2 contain sound algorithms to discover predicates and refine abstraction with utilization of set-related functions in C code.

The algorithms proposed have a serious limitation: each variable in expressions being added to/removed from sets may not be used later in the code. Further research could discard this requirement if it extended the logic used beyond LA+EUF, by adding some set-related concepts into interpolation procedure. In this article we failed to demonstrate that it is unnecessary.

The concepts described here were prototyped as a patch for BLAST of version 2.5, the prototype being of low quality and was just to check if the concepts presented here are correct and viable.

Several simple artificial programs were generated for the experiments. The tests held compare the approach proposed with already known solutions. It's clear that our novel algorithm performs better than the known one. However, the complexity of formulas generated limits the scalability of the approach.

Thus, the approach proposed in the paper doesn't perform well in the experiments with the currently used interpolating tools. Given also the severe limitations on its applicability, we think that further improvement of the prototype developed to make it useful in industrial application is ineffectual.

2. note that in the original paper [7] the elaborated algorithm was not presented. The algorithm on figure 6 was devised by us, but we beleive that Bandera tool has something similar.

References

[1] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstractions for model checking c programs. *Proc. TACAS*, page 268–283, 2001.

[2] T. Ball and S.K. Rajamani. The slam project: Debugging system software via static analysis. *Proc. POPL*, page 1–3, 2002.

[3] D. Beyer, T.A. Henzinger, and G. Théoduloz. Lazy shape analysis. *Proc. CAV, LNCS*, 4144:532–546, 20006.

[4] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu, and Roberto Sebastiani. Software model checking via large-block encoding. In *Proceedings of the 9th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2009, Austin (TX), November 15-18)*, pages 25–32. IEEE Computer Society Press, Los Alamitos (CA), 2009.

[5] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *Int J Softw Tools Technol Transfer*, 9:505–525, 2007.

[6] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. *Proc. CAV, LNCS*, 1855:154–169, 2000.

[7] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, Psreanu, Robby C.S., and H Zheng. Zheng, h.: Bandera: extracting finite-state models from java source code. *ICSE'00: Proc. 22nd Intl. Conf. on Software Engineering*, page 439–448, 2000.

[8] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. *Proc. POPL*, page 58–70, 2002.

[9] K.L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, pages 101–121, 2005.

On Context Switch Upper Bound for Checking Linearizability

Vadim S. Mutilin

Institute for System Programming, RAS
Moscow, Russia

***Abstract*—Approaches that tackle multithreaded programs suffer from state explosion problem. Promising idea is bounding the number of context switches of running threads. Recent work [10] shows that most bugs can be detected even with two context switches. Despite of the fact that it was successful in practice we still can not be sure that no bug has escaped. In this paper we use context-bounding for checking linearizability property, which proved to be useful both for simplifying specifications and usage of programs and as a common property for finding potential bugs in the same way as race conditions. For linearization we provide an algorithm, which returns an upper bound of context switches. Having the upper bound we can be sure that if the program is not linearizable then context-bounding algorithm will show it.**

I. Introduction

Lately multithreaded programming has become widely spread and gained popularity. Aiming at increasing effectiveness, programs consist of several threads that do the work in parallel. But their development is much harder than development of sequential ones. This is caused by the fact that the scheduling order of instructions from different threads cannot be predicted before actual execution and the developer should foresee correct work of the program for all possible interleavings of instructions.

Here we consider programs providing interface (API), which other programs use for interacting with them. Interface consists of *operations* (procedures) which can be *executed* (invoked) with different values of *parameters*. Operation execution ends with returning a value called a result. Moreover, operations can be executed in different threads simultaneously. Execution of operations in several threads we will call *parallel execution*, and execution in a single thread – *sequential execution*.

Parallel execution is *linearizable*, if it is equivalent to some sequential execution conforming to specification. Formally, the notion of linearizability will be defined in section II. As we can see, the task of checking linearizability is a special case of functional testing where we check whether the program meet functional requirements given as specification. But in contrast to general case, linearizable programs require specification only for sequential executions.

As shown in [7], linearizability is local and nonblocking property. Locality means that if we have shown that operations working with the same object are linearizable then they are linearizable together with any other linearizable operations working with different objects. Hence hereafter we will consider operations working with single shared object. In linearizable program running operation does not require invocation of new operations for its completion. This is nonblocking property. Moreover, proof of properties for programs which use linearizable operations is simplified because behaviour of linearizable program is reduced to sequential executions.

Linearizability has much in common with such properties as serializability [8], [11], atomicity [3], [15], sequential consistency [9]. In contrast to it linearizability requires a specification while these properties impose restrictions on the program only. In some works the term atomicity is used as synonymous to linearizability, in the others it means what we call self-linearizability.

The problem of checking linearizability in general case is not decidable. The key issue is lack of restrictions on the number of threads and operations which occur in parallel executions. There is only manual proof approaches for general case [7], [14]. The automatized approaches work only in the special case, which we consider in our work. In these settings the number of threads is finite and the number of operations in each thread is also finite. For example, in [1] it is shown that with limited number of threads, states of implementation and specification the problem can be solved by model checking.

The principle source of the complexity is a great amount of executions resulting from interleavings of instructions. Partial order reduction, heuristic search, as well as context-bounding methods are used to reduce the amount of them. The last one is used in our paper. The main idea of it is to bound the number of context switches in parallel executions. As shown in [10] in practice tricky errors can be detected even with small number of context switches. But the question of choosing an upper bound of switches which is sufficient for ensuring linearizability is still open. That is the question we give answer here that forms novelty of our work.

Next section gives a formal definition of linearizability. Section III describes a formal model of program. In section IV we prove a theorem on which the algorithm given in section V is based. This algorithm gives an upper bound of context switches sufficient for checking linearizability. Section VI discusses significancy of upper bound produced by the algorithm. Section VII contains comparison with related works.

```
1   volatile int x;
2   volatile boolean b = false;
3
4   boolean insert(int i) {
5     if(b==false) {
6       x=i;
7       b=true;
8       return true;
9     } else return false;
10  }
11  void delete() {
12    b = false;
13    return;
14  }
15
16  int lookUp() {
17    if(b==true)
18      return x;
19    else return -1;
20  }
```

Fig. 1. Cell Example

insert:
$\{true\}$
$op\langle insert\rangle_begin\langle i\rangle$ $op\langle insert\rangle_end\langle r\rangle$
$\{r = \neg b \wedge (b \wedge (b' = b) \wedge (x' = x) \vee \neg b \wedge (b' = true) \wedge (x' = i))\}$
delete:
$\{true\}$
$op\langle delete\rangle_begin\langle\rangle$ $op\langle delete\rangle_end\langle\rangle$
$\{b' = false\}$
lookUp:
$\{true\}$
$op\langle lookUp\rangle_begin\langle\rangle$ $op\langle lookUp\rangle_end\langle i\rangle$
$\{(b' = b) \wedge (b \wedge (i = x' = x) \vee \neg b \wedge (i = -1))\}$

Fig. 2. Cell Specification

II. The Notion of Linearizability

A. History

Suppose we have a set of operations $op\langle name\rangle$, each of which has *begin* (invocation) $op\langle name\rangle_begin\langle parameters\rangle$ and *end* (response) $op\langle name\rangle_end\langle result\rangle$.

History is a finite sequence of events $\alpha{:}op\langle name\rangle_begin\langle args\rangle$ and $\alpha{:}op\langle name\rangle_end\langle res\rangle$, where α is a thread. End *matches* begin, if thread names and operation names agree.

Consider an example of Cell program shown in Fig.1 written in Java language, which will be used hereafter. Examples of histories:

$h_1 =$
$\alpha : op\langle insert\rangle_begin\langle 0\rangle$,
$\beta : op\langle delete\rangle_begin\langle\rangle$,
$\alpha : op\langle insert\rangle_end\langle true\rangle$,
$\gamma : op\langle lookUp\rangle_begin\langle\rangle$,
$\beta : op\langle delete\rangle_end\langle\rangle$,
$\gamma : op\langle lookUp\rangle_end\langle 0\rangle$

$h_2 =$
$\alpha : op\langle insert\rangle_begin\langle 0\rangle$,
$\alpha : op\langle insert\rangle_end\langle true\rangle$,
$\gamma : op\langle lookUp\rangle_begin\langle\rangle$,
$\gamma : op\langle lookUp\rangle_end\langle 0\rangle$,
$\beta : op\langle delete\rangle_begin\langle\rangle$,
$\beta : op\langle delete\rangle_end\langle\rangle$

$h_3 =$
$\alpha : op\langle insert\rangle_begin\langle 0\rangle$,
$\alpha : op\langle insert\rangle_begin\langle 1\rangle$,
$\beta : op\langle insert\rangle_end\langle true\rangle$,
$\beta : op\langle insert\rangle_end\langle true\rangle$

Definition 1: History is sequential if

1) The first event is a begin of operation.
2) Each event except the last one is immediately followed by matching end.

In the example, history h_2 is sequential and h_1, h_3 are not sequential.

Thread history(projection, subhistory) in a history H ($H \mid \alpha$) is a subsequence of all events in H, which have thread name α. For instance, $h_1 \mid \alpha = \alpha$ $op\langle insert\rangle_begin\langle 0\rangle$, α $op\langle insert\rangle_end\langle true\rangle$. Two histories H, H' are *equivalent*, denoted as $H \sim H'$, if for any thread α subhistory $H \mid \alpha$ equals to $H' \mid \alpha$. In the example $h_1 \sim h_2$. History is a *well-formed* if any subhistory $H \mid \alpha$ is sequential. All histories considered in the paper are well-formed.

Operation is *pending* in a history if some begin is not followed by matching end. $complete(H)$ is maximal subsequence of H consisting only from begins and matching ends (pending operations are removed).

A set S is *prefix-closed* if for any history H in S holds that any prefix of H is also in S. *Sequential specification of a program* is a prefix-closed set if sequential histories. History H *conforms to specification* if $H \in S$. Specification can be presented in different forms. Fig. 2 shows specification of Cell in the form of pre and postconditions.

B. Definition of Linearizability

History H induces irreflexive partial order on operations $<_H$, such that $e_0 <_H e_1$ if $end(e_0)$ precedes $begin(e_1)$ in H

Definition 2: History H is linearizable if it can be extended (appending zero or more responses) to some history H' for which

1) $complete(H')$ is equivalent to some sequential history S which conforms to specification.
2) $<_H \subseteq <_S$.

History h_1 is linearizable, because it is equivalent to h_2 while preserving partial order of operations. History h_3 is not linearizable, because any sequential specification which is equivalent to it contains two sequential successful $insert$ operations that contradicts to specification.

By *reachable* history of a program we shall mean a history which can actually occur in the program. Later on we will define the notion of reachable history on the base of execution trace. Program is *linearizable* if any reachable history is linearizable.

C. Self-linearizability of Program

For checking linearizability it is helpful to define a notion of self-linearizability independent from specification. In work [6] self-linearizability is called atomicity, but there is no formal definition. They introduce atomicity using sufficient conditions. Here we give a formal definition.

Definition 3: Program is self-linearizable if for any reachable history H there exists a reachable sequential history H' such that $H' \sim H$.
If a program is self-linearizable then by checking that all reachable sequential histories conform to specification we show that program is linearizable.

III. Program Model

Program (system, implementation) is a triple

$$Sys = \langle s_0, S, P \rangle$$

where $s_0 \in S$ is an initial state, S is a set of shared states, P is a finite set of opeartion subprograms.

Each subprogram P is a quadruple

$$P = \langle l_0, L, \nu, T \rangle$$

where $l_0 \in L$ is an initial local state, L is a set of local states (control states), $\nu : T \to \Sigma$ is a labeling function, $T \subseteq L \times G \times C \times L'$ is a set of transitions.

$\Sigma = \{\tau, op_begin\langle parameters\rangle, op_end\langle result\rangle\}$. All transitions from initial local state are labeled by $op\langle name\rangle_begin\langle arguments\rangle$, intermediate are transitions labeled by τ. Transitions which are labeled by $op\langle name\rangle_end\langle result\rangle$ finish the subprogram.

In a transition $c \in C$ is a command $S \to S$ changing the state (instruction, sequence of instructions), $g \in Gd$ is guard condition $S \to \{true, false\}$. We assume that sets of local states of different subprograms do not intersect. Start and end transitions of operations do not change the shared state.

In Cell example, the program model consists of a set S including states of variables x,b and stacks of each subprogram, an initial state $s_0 = \{x = 0, b = false,$ empty stacks $\}$ and a set $P = \{p_{insert}, p_{delete}, p_{lookUp}\}$. Evidently, that subprogram stacks can be separated from S thus showing that this part of the state is accessible to one subprogram only, but this feature is not essential in the paper.

p_{insert}:
$L^i = \{4 \dots 10\}$, $l_0^i = 4$
$T^i = \{\ t_1^i = (4, true, invoke\langle i\rangle, 5),$
$t_2^i = (5, true, \hat{b} = get\langle b\rangle, 5'),$
$t_3^i = (5', \hat{b} = false, nop, 6),$
$t_4^i = (5', \hat{b} \neq false, nop, 9),$
$t_5^i = (6, true, put\langle x, i\rangle, 7),$
$t_6^i = (7, true, put\langle b, true\rangle, 8),$
$t_7^i = (8, true, ret\langle true\rangle, 10),$
$t_8^i = (9, true, ret\langle false\rangle, 10)\ \}$
$\nu(t_1^i) = op\langle insert\rangle_begin\langle i\rangle$
$\nu(t_7^i) = op\langle insert\rangle_end\langle true\rangle$
$\nu(t_8^i) = op\langle insert\rangle_end\langle false\rangle$
For the others $\nu(t^i) = \tau$

p_{delete}:
$L^d = \{11 \dots 14\}$, $l_0^d = 11$
$T^d = \{\ t_1^d = (11, true, invoke, 12),$
$t_2^d = (12, true, put\langle b, false\rangle, 13),$
$t_3^d = (13, true, ret, 14)\ \}$
$\nu(t_1^d) = op\langle delete\rangle_begin\langle\rangle$
$\nu(t_3^d) = op\langle delete\rangle_end\langle\rangle$
$\nu(t_2^d) = \tau$

p_{lookUp}:
$L^l = \{16 \dots 20\}$, $l_0^l = 16$
$T^l = \{\ t_1^l = (16, true, invoke, 17),$
$t_2^l = (17, true, \hat{b} = get\langle b\rangle, 17'),$
$t_3^l = (17', \hat{b} = true, nop, 18),$
$t_4^l = (17', \hat{b} \neq true, nop, 19),$
$t_5^l = (18, true, \hat{x} = get\langle x\rangle, 18'),$
$t_6^l = (18', true, ret\langle \hat{x}\rangle, 20),$
$t_7^l = (19, true, ret\langle -1\rangle, 20)\ \}$
$\nu(t_1^l) = op\langle lookUp\rangle_begin\langle\rangle$
$\nu(t_6^l) = op\langle lookUp\rangle_end\langle \hat{x}\rangle$
$\nu(t_7^l) = op\langle lookUp\rangle_end\langle -1\rangle$
For the others $\nu(t^l) = \tau$

In order to *execute a program* it is necessary to provide user threads $\Psi = \psi_u^1, \dots, \psi_u^n$, which will be executed. User thread ψ_i is defined as a sequence of operation subprograms $p_0, \dots, p_{n_i}$ with values of input parameters. Thread starts execution in an initial state of subprogram p_0. After the end of each subprogram it moves from the end state to the initial state of the next subprogram. Thread finishes execution after finishing the last subprogram p_{n_i}.

For the given user threads Ψ we define *execution state* as $g = (s, l^1, \dots, l^n) \in G$, where s is a shared state, l^i is a local state of the thread ψ_i. $g_0 = (s_0, l_0^1, l_0^2, \dots, l_0^n)$ is an initial state. A set of all execution states we denote as G.

We will use the following definitions:

1) $enabled(t, s) \equiv t.guard(s)$.
2) $pre(t)$ is a start state of t, $post(t)$ is an end state.
3) $local(\alpha, g)$ returns a local state of the thread p in g.
4) $shared(g)$ returns a shared state s.
5) $t(\alpha)$ means that t is executed in thread $\alpha \in \Psi$.
6) $active(t(\alpha), g) \equiv pre(t) = local(\alpha, g)$.
7) $enabled(t(\alpha), g) \equiv active(t(\alpha), g) \wedge enabled(t, shared(g))$.

Define a transition relation $\longrightarrow$. There is a transition $g \xrightarrow{t(\alpha)} g'$ from $g = (s, l^1, \dots, l^\alpha, \dots, l^n)$ if $enabled(t(\alpha), g) = true$ and $g' = (\hat{s}, l^1, \dots, \hat{l}^\alpha, \dots, l^n)$, where $\hat{s} = t.command(s)$ and $\hat{l}^\alpha = post(t)$.

Execution trace of a program is a sequence $t_1(\alpha_1), \dots, t_m(\alpha_m)$ such that $g_0 \xrightarrow{t_1(\alpha_1)} g_1 \xrightarrow{t_2(\alpha_2)} \dots \xrightarrow{t_m(\alpha_m)} g_m$. *Thread trace* is a projection of execution trace on a thread. *Operation trace* in a thread is a projection of thread trace on an operation. Execution history for a trace $\sigma = t_1(\alpha_1), \dots, t_m(\alpha_m)$, denoted as $H(\sigma)$, is a sequence of labels $\nu(t_i(\alpha_i))$ with all τ labels removed. History H is reachable if there exists a trace σ such that $H = H(\sigma)$.

Let a thread α executes $insert\langle 0\rangle$, β executes $delete\langle\rangle$, γ executes $lookUp\langle\rangle$. Consider the examples of traces.

$$\sigma_1 = t_1^i(\alpha), t_2^i(\alpha), t_1^d(\beta), t_3^i(\alpha), t_5^i(\alpha), t_6^i(\alpha), t_7^i(\alpha), t_1^l(\gamma), t_2^l(\gamma), t_3^l(\gamma), t_5^l(\gamma), t_2^d(\beta), t_3^d(\beta), t_6^l(\gamma) \quad (1)$$

$H(\sigma_1) = h_1$.

$$\sigma_2 = t^i_1(\alpha), t^d_1(\beta), t^i_2(\alpha), t^i_3(\alpha), t^i_5(\alpha), t^d_2(\beta), t^i_6(\alpha), t^d_3(\beta), t^i_7(\alpha) \quad (2)$$

$$\sigma_3 = t^l_1(\gamma), t^i_1(\alpha), t^l_2(\gamma), t^i_2(\alpha), t^l_3(\gamma), t^i_5(\alpha), t^l_5(\gamma), t^i_6(\alpha), t^l_6(\gamma), t^i_7(\alpha) \quad (3)$$

IV. An Upper Bound of Context Switches

A. The Notion of Independence

We will use classical definition of independence [4], [5], [12] (Definition 4) and extend it for an arbitrary set of user threads (Definition 5).

Definition 4: $D(\Psi)$ is a symmetric dependence relation for an execution Ψ, iff for all $(t_1(\alpha), t_2(\beta)) \notin D(\Psi)$ (independent) implies that the two following conditions hold for all reachable states g:

1) From $enabled(t_1(\alpha), g)$ and $g \overset{t_1(\alpha)}{\longrightarrow} g'$ follows that $enabled(t_2(\beta), g') = enabled(t_2(\beta), g)$,
2) If $enabled(t_1(\alpha), g)$ and $enabled(t_2(\beta), g)$ then there exists unique state $\hat{g}$ such that $g \overset{t_1(\alpha),t_2(\beta)}{\longrightarrow} \hat{g}$ and $g \overset{t_2(\beta),t_1(\alpha)}{\longrightarrow} \hat{g}$.

Definition 5: D is a symmetric dependence relation for a program iff $(t_1, t_2) \notin D$ (independent) implies that $\forall\Psi, \forall\alpha, \beta$ $\exists D(\Psi) : (t_1(\alpha), t_2(\beta)) \notin D(\Psi)$.

Note 1: Independent transitions can be interchanged in a trace, but the history of the trace and the final state will stay unchanged. [5]

Consider an example of dependence relation for Cell program. We can see that some transitions do not access shared data. Such transitions are obviously independent. Consider transitions accessing shared data $t^i_2, t^i_5, t^i_6, t^d_2, t^l_2, t^l_5$. Among them we can distinguish transitions which only read or write variables. $r_b(w_b)$, $r_x(w_b)$ denote transitions performing read (write) of variables b and x correspondingly. Then r_b: t^i_2, t^l_2; w_b: t^i_6, t^d_2; r_x: t^l_5; w_x: t^i_5. Transition pairs performing read/read of arbitrary variables, write/write or read/write of different variables are independent with each other. The others are considered as dependent. Hence dependence relation D includes $\{(t, t') \mid t \in \{t^i_6, t^d_2\}, t' \in \{t^i_6, t^d_2, t^i_2, t^l_2\}\} \bigcup \{(t, t') \mid t \in \{t^i_5\}, t' \in \{t^i_5, t^l_5\}\}$ and symmetric pairs.

B. The Notion of Dependence Cycle

Let $\sigma = t_1(\alpha_1), \ldots, t_m(\alpha_m)$ be a trace of a program. Define successor relation.

Definition 6: Successor relation (without transitive closure)

1) $t_i(\alpha_i) < t_j(\alpha_j)$, if $t_i(\alpha_i)$ precedes $t_j(\alpha_j)$ in the trace and one of the following conditions holds
   a) $t_i, t_j \neq \{op_begin, op_end\}$, $(t_i, t_j) \in D$ and $\alpha_i \neq \alpha_j$,
   b) $t_i = op_end$, $t_j = op_begin$ and t_i, t_j do not belong to the same operation.
2) $t_i(\alpha_i) = t_j(\alpha_j)$, if t_i, t_j belong to the same operation.

Statement 1: Let a trace σ contains neighbouring pair $t_i(\alpha_i)$, $t_j(\alpha_j)$ and $t_i(\alpha_i) \not\leq t_j(\alpha_j)$. Let σ' be the trace obtained from σ by interchanging $t_i(\alpha_i)$, $t_j(\alpha_j)$ in the reverse order. Then the history of σ' is equivalent to the history of σ ($H(\sigma') = H(\sigma)$).
Follows from independence of $t_i(\alpha_i)$, $t_j(\alpha_j)$, because $t_i(\alpha_i) \not\leq t_j(\alpha_j)$.

Sequence by successor relation is a sequence of transitions related by definition 6: $t_{i_1}(\alpha_{i_1}) \leq t_{i_2}(\alpha_{i_2}) \leq \ldots \leq t_{i_p}(\alpha_{i_p})$, where $t_{i_j}(\alpha_{i_j})$ is a trace element.

Cycle is a sequence by successor relation $t_{i_1}(\alpha_{i_1}) < t_{i_2}(\alpha_{i_2}) \leq \ldots < t_{i_{p-1}}(\alpha_{i_{p-1}}) = t_{i_p}(\alpha_{i_p})$, in which $t_{i_1}(\alpha_{i_1}) = t_{i_p}(\alpha_{i_p})$ and $t_{i_1}(\alpha_{i_1})$ precedes $t_{i_{p-1}}(\alpha_{i_{p-1}})$. Element $t_{i_1}(\alpha_{i_1})$ is called the *start* of a cycle and $t_{i_{p-1}}(\alpha_{i_{p-1}})$ is the *end*.

Consider examples of cycles. Trace σ_2(2) has a cycle

$$t^i_2(\alpha) < t^d_2(\beta) < t^i_6(\alpha) = t^i_2(\alpha) \quad (4)$$

Trace σ_3(3) has two cycles

$$t^l_2(\gamma) < t^i_6(\alpha) = t^i_5(\alpha) < t^l_5(\gamma) = t^l_2(\gamma) \quad (5)$$

$$t^i_5(\alpha) < t^l_5(\gamma) = t^l_2(\gamma) < t^i_6(\alpha) = t^i_5(\alpha) \quad (6)$$

Schematically these cycles are shown in Fig. 3(a,b). Arrows mean that transitions are related by $<$. Trace $t^d_1(\beta), t^l_1(\gamma), t^l_2(\gamma), t^d_2(\beta), t^l_3(\gamma), t^d_3(\beta), t^l_5(\gamma), t^l_6(\gamma)$ has no cycles (Fig. 3(c)).

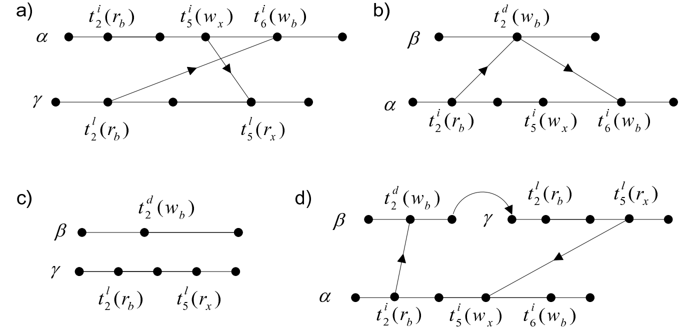


Fig. 3. Examples of Cycles

Two cycles with ends t_{k_1}, t_{k_2}, belonging to the same operation, where t_{k_1} precedes t_{k_2} are *equivalent* if there is no cycle with a start t' preceding t_{k_2} and not preceding t_{k_1} ($t' \in [t_{k_1}, t_{k_2})$).

The equivalence of cycles defines an equivalence relation and corresponding division of cycles into equivalence classes.

Holding cycle is an equivalence class of cycles.

C. The Notion of Context Switch

Let $\sigma = t_1(\alpha_1), \ldots, t_k(\alpha_k), t_{k+1}(\alpha_{k+1}), \ldots, t_n(\alpha_n)$ be a trace.

We say that there is a *context switch* between $t_k(\alpha_k)$, $t_{k+1}(\alpha_{k+1})$ if $\alpha_k \neq \alpha_{k+1}$. Context switch is *nonpreemptive* if $t_k(\alpha_k)$ is the last transition in the operation trace. Otherwise context switch is called *preemptive* which means that a scheduler suspends the executions of the running thread at an arbitrary point. Note that context switches between different operations in the same thread are defined as nonpreemptive.

The number of preemptive context switches in a trace σ we denote as $csw(\sigma)$. For instance, $csw(\sigma_2) = 5$, $csw(\sigma_3) = 8$.

Theorem 1: If the number of holding cycles in a trace σ is k then there exists a trace σ' such that $H(\sigma) = H(\sigma')$ and $csw(\sigma') \leq k$.
Proof of the theorem can be found in the Appendix on page 6.

Corollary 1: If there is no reachable traces with cycles then program is self-linearizable.
If there is no cycles in a trace σ then it is equivalent to a trace σ' without preemptive switches. Hence σ' is a sequential trace and the program is self-linearizable.

V. CycleCount Algorithm

Traditional model checking algorithms suffer from state explosion problem. It is not difficult to show that the number of executions grows exponentially both in the length of thread traces and in the number of threads. More precisely, $\frac{(nm)!}{(m!)^n}$, where n is the number of threads, m is the maximal length of thread trace. Promising approach for solving this problem is bounding the number of context switches. This idea appeared in [13] and in [10] it was evaluated in practice. In the approach the number of executions with k switches is estimated as $(n^2 m)^k n!$, i.e. polynomial in the trace length. Empirical results show that 90% of state coverage can be achieved with eight context switches and the majority of errors are detected even with two switches. The novel result of our paper is that we can guarantee linearizability if we apply context-bounding search algorithm [10] with a bound calculated by CycleCount algorithm (Fig. 4). The linearizability of each trace found during the search we check using one of existing methods. For example, if we have a recognizing automaton as in [1] we can check it in $O(m)$.

CycleCount algorithm takes user threads Ψ, i.e. $\psi_1, \ldots, \psi_n$, where $\psi_i = p_0, \ldots, p_{n_i}$ and for each subprogram p it takes a complete set of operation traces $traces(p)$. The completeness means, that any operation trace that can occur in executions of $Sys(\Psi)$ should be in $traces(p)$. Execution traces of threads in the algorithm are overapproximated as $traces(\psi_i) = traces(p_0) \times \cdots \times traces(p_{n_i})$. Execution traces of program with Ψ are overapproximated as all possible interleavings of thread traces $traces(\psi_1), \ldots, traces(\psi_n)$.

Complexity of the algorithm is $O(n(mk)^3 l^2)$, where n is the max number of threads, m is the max number operations in a thread, k is the max number of traces for a single operation, l is the max operation trace length.

For Cell example the complete set of operation traces is as follows:
insert:
$\sigma_1^i = t_1^i, t_2^i, t_3^i, t_5^i, t_6^i, t_7^i$
$\sigma_2^i = t_1^i, t_2^i, t_4^i, t_8^i$
delete:
$\sigma_1^d = t_1^d, t_2^d, t_3^d$
lookUp:
$\sigma_1^l = t_1^l, t_2^l, t_3^l, t_5^l, t_6^l$
$\sigma_2^l = t_1^l, t_2^l, t_4^l, t_7^l$

$K := 0$
For each thread ψ_i
  For each operation $p_j \in \psi_i$
    For each operation trace $\sigma_k \in traces(p_j)$
      For each $\sigma' \in op' : \psi(op') \neq \psi(op_j)$
        Find $D_k(\sigma') = \{t \in \sigma \mid \exists t' \in \sigma' : (t', t) \in D(\Psi)\}$
        If $D_k(\sigma') \neq \emptyset$, then
          Let $E(\sigma') = \{\sigma'\}$
            $\cup \{traces(\hat{op}) \mid \psi(\hat{op}) \neq \psi(op'), \psi(op_j)\}$.
        Else let $E(\sigma') = \emptyset$.
      For each $\sigma' \in op' : \psi(op') \neq \psi(op_j)$
        Mark transitions $t \in \sigma_k$: $L(t) = \{B, E, \tau\}$.
          Let $L_E = \bigcup_{\hat{\sigma} \in E(\sigma')} D_k(\hat{\sigma})$.
          If $L_E \neq \emptyset$
            Let $L_B = \{t' \in D_k(\sigma') \mid t'$ precedes $t_r\}$.
            where t_r is the most right end in L_E.
          Else let $L_B = \emptyset$.
          Transitions in L_B we mark as B (starts),
            L_E – E(ends), the others – τ.
        $K(\sigma_k)$ is the number of continuous intervals
          of $\{E, \tau\}$, i.e. ends not separated by starts.
        $K := K + K(\sigma_k)$

Fig. 4. CycleCount Algorithm

Suppose that a thread α executes operation p_{insert}, β executes p_{delete}, γ executes p_{lookUp}. Then $CycleCount(\beta, \gamma) = 0$, because there is no cycles. $CycleCount(\alpha, \gamma) = 2$, because we have cycles with ends t_6^i, t_5^l. $CycleCount(\alpha, \beta, \gamma) = 3$, because we have cycles with ends t_5^i, t_6^i, t_5^l. Cycle with end t_5^l (5) is shown in Fig. 3(a). Cycles (4,6) with end t_6^i are shown in Fig. 3(a), 3(b). Cycle

$$t_2^i(\alpha) < t_2^d(\beta) = t_3^d(\beta) < t_1^l(\gamma) = t_5^l(\gamma) < t_5^i(\alpha) = t_2^i(\alpha) \tag{7}$$

is shown in Fig. 3(d). Here $t_3^d(\beta) < t_1^l(\gamma)$, because $\nu(t_3^d(\beta)) = op_end$ and $t_1^l(\gamma) = op_begin$.

VI. Discussion

While formulating CycleCount algorithm in section V we assumed that we have complete set of operation traces. One way to get the traces in practice is to run a test suite. Moreover, our experience in analysing multithreaded programs suggests that in most cases complete set of traces can be generated even by sequential executions. In the worst case it requires minimal number of switches. For ensuring completeness the existing tools measuring path coverage [2] can be used.

Note, that the number of context switches in execution is bounded by maximal number. There are at most

$$\sum_{\psi_i \in \Psi} \sum_{p_1^i, \ldots, p_{n_i}^i} Max_{\sigma \in traces(p_j^i)}(|\sigma| - 1),$$

context switches, i.e. not more than the sum of context switches in the longest operation traces for each thread. Besides, context switches can occur only between transitions inside operation trace, i.e. $|\sigma| - 1$. Table 5 shows maximal

Threads	CycleCount	POR	Maximum
β, γ	0	1	6
α, γ	2	3	9
α, β, γ	3	3	11

Fig. 5. Context Switch Bounds

bounds for Cell example. We can see, that in comparison with maximal bounds CycleCount algorithm has an advantage.

Results of the algorithm CycleCount can be improved. We notice three reasons why the algorithm gives crude estimates. First, theorem 1 does not take into account the fact that several cycles can be broken by one switch. For instance, two cycles in Fig.3(a) can be broken by one context switch instead of two.

Second, CycleCount algorithm does not detect contradictory cycles, i.e. cycles which can not appear together in one trace, but can appear in different ones. For example, while running $CycleCount(\alpha, \beta, \gamma)$ cycle (7) does not appear together with (4), because the same transition t_5^i is both start of the cycle (4), and end of (7) related with the same transition t_5^l.

Third, algorithm does not consider nonappearance of transitions between the other ones. Such situation occur in acquiring locks. Transitions from mutual exclusion intervals cannot interfere with each other, i.e. at first the instructions from the one interval will be executed then the rest ones.

VII. Related Work & Conclusions

Novelty of the results of the paper is in estimation of upper bound of context switches which provided to context-bounding algorithm will guarantee linearizability of the program. Along with context bounding algorithm there are heuristic search and partial order reduction (POR) which reduce the number of executions. In comparison with heuristic search our method can guarantee linearizability. The same as in our estimations the key notion in POR is independence. In this sense our derivation of upper bound is a reduction of non linearizable traces to non linearizable traces with at most k switches.

Table 5 shows the number of context switches in traces found with POR search in the lucky case of a search order. The benefit of CycleCount can be explained by the fact that it knows operation traces in advance, hence reductions can be calculated before search of actual program executions.

Our future work is to improve CycleCount algorithm and implement it with one of model checking engines.

References

[1] Rajeev Alur, Ken McMillan, and Doron Peled. Model-checking of correctness conditions for concurrent objects. In *LICS '96: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, pages 219–228, Washington, DC, USA, 1996. IEEE Computer Society.

[2] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[3] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 256–267, New York, NY, USA, 2004. ACM.

[4] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. *SIGPLAN Not.*, 40(1):110–121, 2005.

[5] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. Foreword By-Pierre Wolper.

[6] John Hatcliff, Robby, and Matthew B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *In Proceedings of the International Conference on Verification, Model Checking and Abstract Interpretation*, pages 175–190. Springer, 2004.

[7] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[8] A. Khoroshilov. *Specification and testing components with asynchronous interfaces. PhD thesis*. ISP RAS, Moscow, 2006.

[9] Leslie Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, 1983.

[10] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *SIGPLAN Not.*, 42(6):446–455, 2007.

[11] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[12] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 377–390, London, UK, 1994. Springer-Verlag.

[13] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *In TACAS*, pages 93–107. Springer, 2005.

[14] Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 129–136, New York, NY, USA, 2006. ACM.

[15] Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Transactions on Software Engineering*, 32(2):93–110, 2006.

Appendix

Theorem 2: If the number of holding cycles in a trace σ is k then there exists a trace σ' such that $H(\sigma) = H(\sigma')$ and $csw(\sigma') \leq k$.

Proof.

Let $\beta_1, \ldots, \beta_n$ are operations in a trace σ.

$\beta_1 = t_1^1 \ldots t_{q_1}^1$

$\ldots$

$\beta_n = t_1^n \ldots t_{q_n}^n$

A *division* Ξ is

$\beta_1 = \beta_1^1 \ldots \beta_1^{p_1}$

$\ldots$

$\beta_n = \beta_n^1 \ldots \beta_n^{p_n}$

where β_i^l is a union of neighbouring transitions in the same operation trace.

Given Ξ we can define partial order HB ($\beta_i^l \prec \beta_j^m$) on it.

Definition 7: Partial order HB on Ξ is *consistent* if

1) $\forall i \; \forall \; l < m$ holds $\beta_i^l \prec \beta_i^m$
2) $\forall \; t_1 < t_2$ in σ, t_1 in β_i^l, t_2 in β_j^m holds $\beta_i^l \prec \beta_j^m$.

Lemma 1: If partial order HB is consistent on the division Ξ then $\exists \sigma'$ such that $H(\sigma) = H(\sigma')$ and $csw(\sigma') \leq (p_1 - 1) + \cdots + (p_n - 1)$.

As σ' it is sufficient to take any trace composed from β_i^l, with respect to partial order HB. The number of context switches for β_i is $p_i - 1$. σ' can be derived from σ by interchanging transitions $t_i(\alpha_i) \not\leq t_j(\alpha_j)$, because it preserves equivalence of histories (Statement 1). From consistency of HB it follows

that transitions for which $t_i(\alpha_i) \le t_j(\alpha_j)$ need not to be interchanged.

Now for proving theorem we need to choose division Ξ such that $\sum_{i=1}^{n}(p_i - 1) \le k$ and to choose consistent partial order HB on it.

Choosing division. For each β_i. If there is no cycles then $\beta_i = \beta_i^1$. Otherwise, suppose there is s holding cycles. $t_{k_1}, t_{k_2}, \ldots, t_{k_s}$ are the most left ends of cycles in each holding cycle (i.e. t_{k_j} is the most left end of the cycles in a holding cycle j).

Suppose that $t_{k_1}, t_{k_2}, \ldots, t_{k_s}$ appear in β_i in the same order. Chose division β_i:

$\beta_i^1 = [op_begin, t_{k_1})$
$\beta_i^2 = [t_{k_1}, t_{k_2})$
$\beta_i^3 = [t_{k_2}, t_{k_3})$
$\ldots$
$\beta_i^{s+1} = [t_{k_s}, op_end]$

We define relation $<$ on Ξ as follows. $\beta_i^l < \beta_j^m$ if one of the following conditions holds

1) $i = j$, $l < m$
2) $\exists\ t_1$ in β_i^l, t_2 in β_j^m: $t_1 < t_2$.

We define HB ($\prec$) on Ξ as a transitive closure of $<$. Lets show that $\prec$ is consistent partial order. Consistency immediately follows from definition of $<$. Lets show that $\prec$ is a partial order.

Proof by contradiction. Let exists β_i^l, β_j^m such that $\beta_i^l \prec \beta_j^m$ and $\beta_j^m \prec \beta_i^l$.

Hence there exists chains

$\beta_i^l < \beta_{i_1}^{l_1} < \ldots < \beta_j^m$
$\beta_j^m < \beta_{j_1}^{m_1} < \ldots < \beta_i^l$
or
$\beta_i^l < \beta_{i_1}^{l_1} < \ldots < \beta_j^m < \beta_{j_1}^{m_1} < \ldots < \beta_i^l$

Hence there exists a sequence $t_1(\beta_i) \le t_2(\beta_{i_1}) \le \ldots \le t_w(\beta_j^m) \le t_{w+1}(\beta_{j_1}) \le \ldots \le t'(\beta_i)$. Two signes are strictly less, because there should appear $\hat{\beta} \ne \beta_i$.

Either t_1 precedes t' then let $t_a = t_1$, $t_b = t'$ or t_1 does not precede t'. Then in one of the equivalences $\ldots < t_b(\hat{\beta}) = \ldots = t_a(\hat{\beta}) < \ldots$, t_a precedes t_b. Because otherwise, if in all equivalencies t_b precedes t_a, then t_1 precedes t'.

t_a precedes t_b then exists a cycle: $t_a(\hat{\beta}) < \ldots \le t'(\beta_i) = t_1(\beta_i) \le \ldots < t_b(\hat{\beta})$. This cycle can not be equals to any cycle with the end which is not later than t_a. Hence one of the most left ends of cycles in holding cycle lies after t_a and not after t_b. That contradicts to the chosen division. •

Observable Form of a Timed Finite State Machine

Maxim Gromov, Olga Kondratjeva
Faculty of Radiophysics
Tomsk State University
Tomsk, Russia
E-mail: {gromov, kondratjeva_olga}@sibmail.com

***Abstract*—This paper is devoted to the problem of an observable form for a given Timed Finite State Machine. This problem, together with the problem of the number of states in the observable form, has theoretical value, since it shows the way to build an observable form, and answers the question, does an observable form exist for any TFSM. Also it has a practical use in testing, since Finite State Machine methods of test generation rely on the fact that specification of a system is an observable FSM and those methods are intended to be applied for TFSMs.**

***Keywords*-observable form; non-deterministic Timed Finite State Machine;**

I. Introduction

Problem of testing communication protocols has drawn a lot of attention of different researches. The most popular paradigm for such testing is model based testing, when a system and its specification are supposed to be described as models of some kind, and the problem of testing is re-expressed as a problem of checking relations on those models. One of the most popular model for communication protocols description is a Finite State Machine (FSM) [1] and there are lots of tests generation methods for FSMs [6]–[8]. These methods suppose, that an FSM is given in observable form [2]. But FSM as a model has some limitations, one of which is that FSM does not consider time explicitly, but a communication protocol usualy has different behaviour depending on the time elapsed after last message it has receved or sent. This issue can be dealt with by direct introduction delay transitions into FSMs and such a model has been called *Timed Finite State Machine*(TFSM) [**?**], [4].

New model rises a problem of new methods for test generation. The obvious solution to this problem is to adopt well-known methods of test generation for regular FSMs. And this, as a consequence, gives a problem of an observable form for a given non-observable TFSM. In our work we give a solution to this problem as an observable form construction procedure.

The rest of the paper has the following structure. In section II all necessary definitions and notions are given. Section III gives some comparison of TFSM with other timed models. Section IV discusses a problem of observable form construction and gives corresponding procedure. Section V concludes the paper.

II. Preliminaries

Here and further $\mathbb{N}$ denotes the set of natural numbers, $\mathbb{Z}_0^+$ denotes the set of unsigned integer numbers (that is $\mathbb{Z}_0^+ = \mathbb{N} \cup \{0\}$).

Definition 1. A *Timed Finite State Machine* (TFSM) is a sixtuple $\mathcal{S} = \langle S, I, O, \hat{s}, \lambda_{\mathcal{S}}, \Delta_{\mathcal{S}}\rangle$, where S is finite, not empty set of states with designated initial state $\hat{s}$, I and O are finite, not empty sets of input and output actions respectively, $\lambda_{\mathcal{S}} \subseteq S \times I \times O \times S$ is a transition relation, $\Delta_{\mathcal{S}} : S \to S \times \mathbb{N}$ is a delay function. We assume, that if $\Delta_{\mathcal{S}}(s) = \langle s', \infty\rangle$, then $s' \equiv s$. □

With every TFSM we associate an internal clock variable χ, which measures time in integer number of ticks passed from the last state change of the TFSM.

If $\langle s, i, o, s'\rangle \in \lambda_{\mathcal{S}}$ (we denote this fact as $s \xrightarrow{i/o} s'$ and call o an *output reaction* of the TFSM in the state s on input action i), then we say, that TFSM $\mathcal{S}$ being in the state s accepts input i, immediately produces output action o and at the same moment changes state to s'. In the state s' clock variable χ is reset to 0.

Function $\Delta_{\mathcal{S}}(s) = \langle s'', t\rangle$ (denoted as $s \xrightarrow{t} s''$) describes delay transitions of the TFSM: if no input action is applied to TFSM $\mathcal{S}$ in the state s within t ticks, then at the moment $\chi = t$ it changes its state to s'' and resets its clock variable to 0. If $\Delta_{\mathcal{S}}(s) = \langle s, \infty\rangle$, then TFSM $\mathcal{S}$ can stay in the state s infinitely long, waiting for an input action.

Definition 2. A *timed input action* is a tuple $\langle i, t\rangle \in \in I \times \mathbb{Z}_0^+$, meaning that input action i should be applied to a TFSM at the moment t, after last state change of the TFSM. A *timed input-output pair* is a tuple $\langle\langle i, t\rangle, o\rangle \in \in I \times \mathbb{Z}_0^+ \times O$, usually denoted as $\langle i, t\rangle/o$. A timed input sequence α is a sequence of timed inputs, that is $\alpha \in \in (I \times \mathbb{Z}_0^+)^*$. □

Definition 3. Given a TFSM $\mathcal{S} = \langle S, I, O, \hat{s}, \lambda_{\mathcal{S}}, \Delta_{\mathcal{S}}\rangle$.

- $\mathbf{out}(s, i) \overset{\text{def}}{=} \{o \in O \mid \exists s' \in S : \langle s, i, o, s'\rangle \in \lambda_{\mathcal{S}}\}$ – set of output reactions of the TFSM $\mathcal{S}$ in the state s on the input i.
- $[\![s]\!]_{i/o} \overset{\text{def}}{=} \{s' \in S \mid \langle s, i, o, s'\rangle \in \lambda_{\mathcal{S}}\}$ – set of states, reachable from the state s under input-output pair $\langle i, o\rangle \in I \times O$.
- $[\![s]\!]_t$ – is such state s', that TFSM $\mathcal{S}$ reaches s' from s after t time units, that is there exist states $s \equiv s_1$, s_2, …, $s' \equiv s_n$ and $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \ldots \xrightarrow{t_{n-1}} s_n$ and $\Sigma_{j=1}^{n-1} t_j \leqslant t < \Sigma_{j=1}^{n} t_j$, where $\Delta_{\mathcal{S}}(s_j) = \langle s_{j+1}, t_j\rangle$, $j = \overline{1, n}$. Note, that due to the fact, that $\Delta_{\mathcal{S}}$ is completely defined function on the set S, such a state s' always exists and is unique.
- $[\![s]\!]_{\langle i,t\rangle/o} \overset{\text{def}}{=} [\![[\![s]\!]_t]\!]_{i/o}$ – set of states, reachable from

the state s under timed input-output pair $\langle i,t\rangle/o$. If $s' \in [\![s]\!]_{\langle i,t\rangle/o}$, then we say, that there is a transition form the state s to the state s' under the timed input $\langle i,t\rangle$ with output reaction o and denote this fact as $s \xrightarrow{\langle i,t\rangle/o} s'$.

- $\mathbf{out}(s,\langle i,t\rangle) \stackrel{\text{def}}{=} \mathbf{out}([\![s]\!]_t, i)$ – set of output reactions of the TFSM S on timed input $\langle i,t\rangle$.

□

Definition 4. Given a TFSM S and timed input sequnce $\alpha = \langle i_1,t_1\rangle \cdot \langle i_1,t_1\rangle \cdot \ldots \cdot \langle i_n,t_n\rangle$. We say, that timed input sequnce α brings TFSM S from the state s to the state s' if there exist such states $s_1, s_2, \ldots, s_{n-1}$ and output sequence $\beta = o_1 \cdot o_2 \cdot \ldots \cdot o_n$, that the following property holds: $s \xrightarrow{\langle i_1,t_1\rangle/o_1} s_1 \xrightarrow{\langle i_2,t_2\rangle/o_2} s_2 \xrightarrow{\langle i_3,t_3\rangle/o_3} \ldots \xrightarrow{\langle i_{n-1},t_{n-1}\rangle/o_{n-1}} s_{n-1} \xrightarrow{\langle i_n,t_n\rangle/o_n} s'$. We denote this fact as $s \xrightarrow{\alpha/\beta} s'$ and call the sequence β *output reaction* of TFSM S on timed input sequence α and the pair α/β *timed input-output* sequence of the TFSM S in the state s. The set of all input-output sequnces of the TFSM S in the state s we shall denote as $\Gamma_{\mathsf{S}}(s)$. □

All notions from Definition 3 are naturally extended to timed input-output sequences.

Definition 5. Two TFSMs S and P are called equivalent, iff $\Gamma_{\mathsf{S}}(\hat{s}) = \Gamma_{\mathsf{P}}(\hat{p})$. □

Definition 6. Given a TFSM S.

- TFSM S is called *deterministic*, iff for any pair $\langle s,i\rangle \in S \times I$ there exists at most one pair $\langle o,s'\rangle \in O \times S$ such, that $s \xrightarrow{i/o} s'$, otherwise it is called *non-deterministic*.
- TFSM S is called *observable*, iff for any triple $\langle s,i,o\rangle$ there exists at most one state $s' \in S$ such, that that $s \xrightarrow{i/o} s'$, otherwise it is called *non-observable*.

□

Definition 7. Given non-observable TFSM S. An observable TFSM P, such that S is equivalent to P we shall call *an observable form* of the TFSM S. □

III. Timed Finite State Machine and Timed Automata

Comparing TFSM with classical timed automaton (TA) [9] one can see several differences. Most obvious is the number of time variables: TFSM always has only one time variable, while a TA can have any finite number of time variables. This prevents from turning an arbitrary TA into TFSM, since it is known [10], that the number of time variables is not reducable in general case.

Another difference is the roles of actions. TA considers just a set of actions, labling transitions, without assigning any roles to them, while in TFSM actions are split into input and output actions and actions always occur in pairs “input/output”. This issue can be overcome by assigning roles to actions and, eo ipso, by consideration of timed atomata with inputs and outputs (TAIO) [11] and restricting TAIO to the form, when output action always comes after input action. TAIO model also overcomes an issue of the time behaviour representation. In a TFSM it is done in a form of time delay transitions. A TAIO analogue of time delay transitions is a transition with unobservable action τ. Howevere, the latter does not mean, that a TAIO is more general model, than TFSM, because in general case TAIO, which has been gotten from TFSM, can occur nondeterministic, even if given TFSM was deterministic and no one has shown yet, that such a TAIO is determinizable. This should be done, since it is known, that in general nondeterministic TA (and TAIO as well) is not determinizable [10].

IV. Building Observable Form of a TFSM

A. A State Stay Time

In the theory non-deterministic Finite State Machines (FSM) to build an observable form G of a given non-observable FSM F one should map subsets of states of the FSM F, which appear due to non-observable transitions (transitions from a given state under the same input-output pair to a diffrent states) to states of G [5]. We shall adopt the same idea to TFSMs with the following modification. Due to delay transitions some set of states can appear with different values of the internal time variable χ for each state in the considering set of states. This leads to the necessity to keep value of internal time variable for each state as it is done for intersection of TFSMs [4]. For example, let us imaging, that we have the following delay transitions: $s_1 \xrightarrow{2} s_1' \xrightarrow{2} s_1$ and $s_2 \xrightarrow{3} s_2' \xrightarrow{2} s_2$ and we consider the set of states $\{s_1, s_2\}$ when $\chi = 0$ for each state. Then we shall get following transitions:

$$\underset{0\ 0}{s_1 s_2} \xrightarrow{2} \underset{0\ 2}{s_1' s_2} \xrightarrow{1} \underset{1\ 0}{s_1' s_2'} \xrightarrow{1} \underset{0\ 1}{s_1 s_2'} \xrightarrow{1} \underset{1\ 0}{s_1 s_2} \xrightarrow{1} \ldots$$

Under each state we write the value of internal time variable χ. As you can see, pair $s_1 s_2$ appears in the sequence twice, but with different times.

B. A Procedure to Build an Observable Form for a Given TFSM

Given non-observable TFSM $\mathsf{S} = \langle S, I, O, \hat{s}, \lambda_{\mathsf{S}}, \Delta_{\mathsf{S}}\rangle$. With each state $s \in S$ we associate set $K_s = \{0, 1, \ldots, k_s - 1\}$, where $\Delta_{\mathsf{S}}(s) = \langle s', t\rangle$ and $k_s = t$, when $t \neq \infty$, or $k_s = 0$, when $t = \infty$. Each state of observable form P for TFSM S corresponds to some subset of the set $U = \cup_{s\in S}\{s\} \times K_s$ – the set of all possible pairs state-stay_time.

The observable form P for TFSM S is constructed as follows.

1) Initial state of P is a pair $\langle \hat{s}, 0\rangle$, that is $\hat{p} = \{\langle \hat{s}, 0\rangle\}$.

2) Let $\{\langle s_1,t_1\rangle \ldots \langle s_n,t_n\rangle\}$ is under consideration. If, according to λ_{S}, the following holds $[\![\{s_1,\ldots,s_n\}]\!]_{i/o} = \{s_1', \ldots s_m'\}$, then transition

$$\{\langle s_1,t_1\rangle \ldots \langle s_n,t_n\rangle\} \xrightarrow{i/o} \{\langle s_1',0\rangle, \ldots \langle s_m',0\rangle\}$$

is in λ_{P}.

3) Let $\{\langle s_1,t_1\rangle \ldots \langle s_n,t_n\rangle\}$ is under consideration and let $\Delta_{\mathsf{S}}(s_1) = \langle s_1', t_1'\rangle, \ldots, \Delta_{\mathsf{S}}(s_n) = \langle s_n', t_n'\rangle$. Then

$$\Delta_{\mathsf{P}}(\{\langle s_1,t_1\rangle \ldots \langle s_n,t_n\rangle\}) = \langle \{\langle s_1'', t_1''\rangle \ldots \langle s_n'', t_n''\rangle\}, t\rangle,$$

where $t = \min\{(t'_1 - t_1), \ldots, (t'_n - t_n)\}$ – time for the earliest delay transition to fire. If $t'_j = \infty$ or $(t'_j - t_t) = t$, then $s''_j = s'_j$ and $t''_j = 0$, otherwise $s''_j = s_j$ and $t''_j = t_j + t$.

Proposition 1. *TFSM P, built with decribed procedure, is an observable form for a given S.* □

It is known [5], that the number of states in observable form for a given FSM F is not greater then $2^{|F|} - 1$, where $|F|$ is the number of states in the given FSM F. This estimation comes from the fact, that the number of all possible non-empty subsets of the state space F is exactly $2^{|F|} - 1$ and only such subsets describe states in observable form. We use the same reasoning to estimate the states number upper bound in observable form P for non-observable TFSM S. Since to describe states in P we use non-empty subsets of the set $U = \cup_{s \in S}\{s\} \times K_s$, then the number of states in P is limited by the number of such subsets, that is $|P| \leqslant 2^{|U|} - 1 = 2^{\Sigma_{s \in S} k_s} - 1$, $k_s = |K_s|$. But this estimation is too rough, since we do not take into account the fact, that due to observable form construction procedure, any state $p = \{\langle s_1, t_1\rangle, \ldots, \langle s_n, t_n\rangle\}$ of the observable form P is so, that at least one $t_j \in \{t_1, \ldots, t_n\}$ equals 0. For example, in extreme case when for any state s of the given non-observable TFSM S holds the following: $\Delta_S(s) = \langle s', t\rangle$ (time of a delay transition is the same for any state), all states of the observable form are subsets like the following $\{\langle s_1, 0\rangle, \ldots, \langle s_n, 0\rangle\}$ and their number is exactly $2^{|S|} - 1$.

For the reasons described above the general estimation for the number of states in observable form for a given TFSM is no reachable (as opposed to the estimation for regular FSMs [5]).

V. Conclusions

In this paper we have considered the problem of the observable form for a TFSM. We have provided the procedure to build an observable form for a given non-observable TFSM and gave some estimation for the number of states in observable form.

The open problem of this paper is the problem of more accurate estimation of number of states in observable form of a given TFSM and the question of its reachability.

References

[1] Евтушенко Н.В., Петренко А.Ф., Ветрова М.В. Недетерминированные автоматы: анализ и синтез. Ч. 1. Отношения и операции: Учебное пособие. — Томск: Томский государственный университет, 2006. — 142 с.

[2] Starke P.H. Abstract Automata // Elsiver, 1972. — 419 pp.

[3] Merayo M.G., Núñez M. and Rodríguez I. Formal Testing from Timed Finite State Machines // Computer Networks. — 2008. — Vol. 52, No. 2. — Pp. 432-460.

[4] Громов М.Л., Евтушенко Н.В. Синтез различающих экспериментов для временных автоматов // поступила в редакцию журнала Программирование.

[5] Трахтенброт Б.А., Барздинь Я.М. Конечные автоматы (поведение и синтез). — М.: Наука, 1970. — 400с.

[6] Bochmann G.v. and Petrenko A.F. Protocol testing: review of methods and relevance for software testing // Proceedings of ISSTA'94. — New York, NY, USA: ACM, 1994. — Pp. 109-124

[7] Petrenko A.F. and Yevtushenko N.V. Testing from Partial Deterministic FSM Specifications // IEEE Trans. Comput. — 2005. — Vol. 54, No. 9. — Pp. 1154-1165.

[8] Hierons R.M. Testing from a Nondeterministic Finite State Machine Using Adaptive State Counting // IEEE Trans. Comput. — 2004. — Vol. 53, No. 10. — Pp. 1330-1342.

[9] Alur R. and Dill D. A Theory of Timed Automata // Theoretical Computer Science. — 1994. — Vol. 126. — Pp. 183-235.

[10] Tripakis. S. Folk theorems on the determinization and minimization of timed automata
Information Processing Letters. — 2006. — Vol. 99, Issue 6. — Pp. 222-226.

[11] Krichen M. and Tripakis S. State Identification Problems for Timed Automata // LNCS. — 2005. — Vol. 3502. — Pp. 175-191.

On EFSM-based Test Derivation Strategies [1]

Aleksandr Nikitin, Natalia Kushik

***Abstract*—Formal model based test derivation is now widely used in software testing. One of the formal models which is very close to software implementations is the model of an Extended Finite State machine (EFSM). Compared with an FSM the EFSM has predicates for condition representation and context variables. However, when deriving tests for an EFSM very complex reachability and distinguishability problem should be solved. For this reason, when deriving tests from an EFSM a number of FSM slices are used. In this paper, we discuss how to derive a test using an FSM slice with limited number of states and how to represent data in the PC memory for fast generation of such slice. Preliminary experimental results with protocol EFSMs are provided.**

***Index Terms*—Software testing, EFSM, computer representation**

I. Introduction

THE complexity of digital systems and devices increases quickly and software is a usual part of almost each system or device. Thus, ad hoc testing of software implementations now is insufficient and a number of methods for formal model based software testing are proposed and widely used, since model based testing provides tests with the guaranteed fault coverage.

One of formal models which is very close to software implementations is the model of an Extended Finite State machine (EFSM). The EFSM model extends the classic FSM model with input and output parameters, context variables, operations and predicates defined over context variables and input parameters. If specification domains of input parameters and context variables are finite then an EFSM can be unfolded to an equivalent FSM (FSM slice) by simulating its behavior with respect to all possible values of context variables and input vectors [1]. A test suite then is derived from the corresponding equivalent FSM. However, the number of states of such corresponding FSM grows exponentially, and thus, it is necessary to limit the maximal number of states and in this case, the corresponding FSM becomes nondeterministic. In this paper, we show how tests can be derived for such nondeterministic FSM slice. In order to derive tests effectively an efficient computer representation of the complex EFSM model is proposed in this paper.

The structure of the paper is as follows. Section II contains preliminaries and a discussion how tests for an EFSM can be derived when the number of a corresponding FSM is limited. Section III is devoted to EFSM computer representation. Section IV discusses preliminary experimental results with protocol EFSMs while Section V concludes the paper.

II. Preliminaries

The EFSM Model

An extended finite state machine [2] A is a pair (S, T) of a set of states S and a set of transitions T between states from S, such that each transition $t \in T$ is a tuple (s, x, P, op, y, up, s'), where:

$s, s' \in S$ are the initial and final states of a transition;

$x \in X$ is an input, and $D_{inp\text{-}x}$ is the set of possible input vectors, associated with the input x, i.e., each component of an input vector is the value of a corresponding input parameter associated with x;

$y \in Y$ is output, where Y is the set of outputs, and $D_{out\text{-}y}$ is the set of possible output vectors, associated with the output y, i.e. each component of an output vector corresponds to an output parameter associated with y;

P, op, and up are functions, defined over input parameters, and context variables, namely:

$P: D_{inp\text{-}x} \times D_V \rightarrow \{\text{True, False}\}$ is a predicate, where D_V is a set of context vectors $\mathbf{v}$;

$op: D_{inp\text{-}x} \times D_V \rightarrow D_{out\text{-}y}$ is an output parameter update function;

$up: D_{inp\text{-}x} \times D_V \rightarrow D_V$ is a context update function.

As in [2], we use the following definitions.

Given an input x and a vector $\boldsymbol{\rho} \in D_{inp\text{-}x}$, the pair ($x$, $\boldsymbol{\rho}$) and vector from $D_{inp\text{-}x}$, is called a *parameterized input*. A sequence of parameterized inputs is called a *parameterized input sequence*. A context vector $\mathbf{v} \in D_V$ is called a *context* of M. A *configuration* of M is a pair (s, $\mathbf{v}$). Given a parameterized input sequence of an EFSM we can calculate the corresponding parameterized output sequence by simulating the behavior of the EFSM under the input sequence starting from the initial state and initial values of the context variables.

As an example, consider the EFSM E in Figure 2 that corresponds to the C function f presented in Figure 1.

[1] This work is partly supported by FCP grant № 02.514.12.402.

```
int f(int *a, int size_a)
{
int i, m;
i = 0;
m = a[0];
while(i < size_a)
{
if(m < a[i]) m = a[i];
i++;
}
return m;
}
```

Figure 1. The function *f*

Function *f* in Figure 1 returns the maximal integer in the array *a* where *size_a* is the cardinality of *a*. To obtain an EFSM that corresponds to the given C function we first determine the set *S* of states. Let *S* be the set $S = \{s_1, s_2, s_3\}$ where s_1, s_2, s_3 are three different points in the C function. The state s_1 corresponds to the beginning of the function *f*; the state s_2 represents the state of the program before comparing *i* with *size_a*; the program moves to the state s_3 if *i* is less than *size_a*. The set *X* of input consists of the array pointer **a* and of the cardinality *size_a* of *a*. Input **a* is a parameterized input, here *index* (item number) is a parameter. Output $y \in Y$ is not parameterized; it corresponds to the variable *m* that is returned by the function *f*. We also add special input (and output) '*NULL*' to specify cases when program accepts (or returns) no external data. The set *P* of predicates consists of P_1 and P_2: P_1 is *true* if *i* is less than *size_a* while P_2 is *true* if *m* is less than *a*[*i*]. The variable *i* is the context variable. The corresponding EFSM *E* is presented in Figure 2.

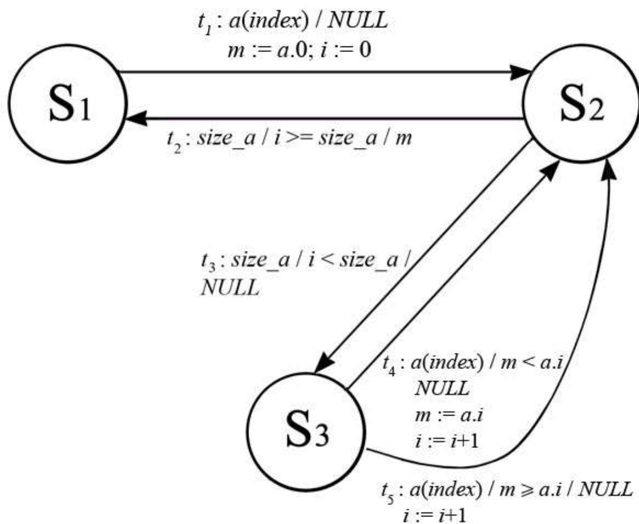


Figure 2. The EFSM *E*

In this paper, we consider deterministic and complete EFSMs, i.e., for each parameterized input sequence there exists a single parameterized output sequence that is produced by the EFSM for the given input sequence.

Unfolding a given EFSM to an equivalent FSM

Given an EFSM *A*, in order to obtain an equivalent FSM *F* we proceed as follows. Given a state *s* of EFSM *A*, a context vector **v**, an input *x* and vector **ρ** of input parameters, we derive the transition from configuration (*s*, **v**) under input (*x*, **ρ**) in the corresponding FSM *F*. We first determine the outgoing transition (*s*, *x*, *P*, *op*, *y*, *up*, *s´*) from state *s* where the predicate *P* is true for input vector **ρ** and context vector **v**, update the context vector to the vector **v´** according to the assignment *up* of this transition, determine the parameterized output *y***ω** and add the transition (*s***v**, *x***ρ**, *y***ω**, *s´***v´**) to the set of transitions of the FSM. The obtained FSM has the same number of states as the number of different configurations (*s*, **v**) of the EFSM *F* that are reachable from the initial state.

Such unfolding can help to detect transfer, predicate, and assignment faults of the given EFSM *A*. However, it is known that the simulation usually leads to a state explosion problem. That is the reason why the maximal number of states of the FSM *F* is limited by integer *B*, for example. In this case, all the states corresponding to configurations (*s*, **v**) with the numbers that are greater then *B* are marked by a special state DNC (don't care state). Two ways are then appropriate for FSM *F* testing.

a) Transitions with DNC states are deleted from *F* and *F* is tested as a partial FSM [3].

b) FSM *F* is tested as completely specified FSM [4] and then the test suit is "cleaned" by deleting all suffixes of test sequences that lead to the DNC state. However, the fault coverage of such a test suite is still unknown.

As an EFSM model is rather complex, a suitable computer EFSM representation can be of a big help when unfolding and deriving tests automatically.

III. EFSM COMPUTER REPRESENTATION

Computer representation of the EFSM *A* uses the following items kept in the PC memory: a number of states of *A*, an array of parameterized inputs, an array of parameterized outputs, an array of context variables, an array of functions *op* and *up*, an array of integers that are used in functions *op* and *up*, an array of predicates, and an array of transitions. We define a set of structures in the C language for the computer representation of an EFSM.

Structure *input* is used for parameterized (or not parameterized) input representation.

```
struct input
{
char *input_name;
int par_quantity;
char **parameters_names;
float *parameters_values;
};
```

The field *input_name* of the structure *input* captures initial input name while the *par_quantity* field is a number of input

parameters. The *parameters_names* array saves initial parameter names and the *parameters_values* array is used for an assignment of the input parameters.

Structure *output* is similar to *input* structure and is used for parameterized (or not parameterized) output representation.

```
struct output
{
char *output_name;
int par_quantity;
char **parameters_names;
float *parameters_values;
};
```

The field *output_name* of the structure *output* captures initial output name; the *par_quantity* field is a number of output parameters. The *parameters_names* array is also used for initial parameters names and the *parameters_values* array keeps an assignment of the output parameters.

We use **structure *variable*** for context variable representation.

```
struct variable
{
char *variable_name;
float value;
};
```

The field *variable_name* of the structure *variable* corresponds to initial name of the variable while value of the context variable is stored in the *value* field.

When simulating behavior of the EFSM A we use integers instead of strings. In other words, we hash inputs, outputs, variables, functions and predicates and use corresponding integer identifiers when deriving tests. We note that such a hashing significantly accelerates the simulation process. Reverse Polish Notation [5] is utilized for faster translation of EFSM predicates and functions into computer representation. That is the reason why the **structure *function*** has two fields.

```
struct function
{
int *rpr;
int rpr_size;
};
```

The Reverse Polish Notation that corresponds to the function is stored in the *rpr* array of *rpr_size* items. The items of the *rpr* array are identifiers of parameterized inputs or context variables. Arithmetic operators are also hashed and stored in the *rpr* array. We use special *rpr* item '–1' to separate operators and operands of the Reverse Polish Notation.

When constructing the Reverse Polish Notation for the predicate P we hash comparison operators and divide an arithmetic expression into two parts: the arithmetic expression that is in the left hand side of the comparison operator is the '*left notation*'. In the right hand side of the comparison operator is the '*right notation*'. Correspondingly, we consider only predicates where left hand and right hand side expressions are separated with one of the following comparison operators {<, >, >=, <=, ==, !=}. Therefore, **structure *predicate*** has four fields.

```
struct predicate
{
int *rpr_left;
int rpr_left_size;
int *rpr_right;
int rpr_right_size;
int sign_op;
};
```

The *rpr_left* array of *rpr_left_size* items corresponds to the 'left notation' while *rpr_left* array of *rpr_left_size* items are used for the 'right notation'. The *sign_op* field corresponds to the comparison operator identifier.

As the EFSM is a pair (S, T) of a set of states S and a set of transitions T, it is necessary to save all the transitions of the set T. Correspondingly we define a **structure *transition*** for $t \in T$.

```
struct transition
{
int s;
int s_prime;
input i;
output o;
int *predicate_numbers;
int *function_numbers;
};
```

Integers *s* and *s_prime* are initial and final states of the transition $t = (s, x, P, op, y, up, s')$ while x and y are parameterized input and output of the transition. Items of the *predicate_numbers* array and *function_numbers* array are identifiers of predicates and functions which guard the transition t.

Therefore, **structure *EFSM*** consists of the following data items.

```
struct EFSM
{
int s_number;
input *input_array;
int input_array_size;
output *output_array;
int output_array_size;
variable *variable_array;
int variable_array_size;
int *integers_array;
int integers_array_size;
function *functions_array;
int function_array_size;
predicate *predicates_array;
int predicates_array_size;
transition *transitions_array;
int transitions_array_size;
};
```

s_number is the number of states of the EFSM;.

input_array (of *input_array_*size items) and *output_array* (of *output_array_*size items) form the sets of parameterized inputs and outputs;

variable_array stores *variable_array_size* context variables.

If predicates or functions use constant integers then these

integers are saved in the *integers_array*. The functions and the predicates are stored in the *functions_array* and the *predicates_array*. The set *T* of transitions is listed in the *transitions_array*.

IV. Experimental Results

We experimented with several protocol EFSMs. The preliminary experimental results show that for several protocol EFSM the unfolding procedure at an appropriate abstraction level can be performed without limiting the maximal number of states of an equivalent FSM. Those protocols are POP3, SMTP, and TIME [6]. The reason is that the number of states of the corresponding protocol EFSMs (at an appropriate abstraction level) is up to four and the number of context variables is less than three while the number of transitions does not exceed 16. The equivalent FSM for POP3 EFSM has six states and 106 transitions while the equivalent FSM for SMTP EFSM has four states and 36 transitions. The TIME EFSM is rather small that is why the number of TIME FSM transitions is 12. More detailed information about performed experiments is presented in Table 1.

Protocol	Number of EFSM states	Number of EFSM context variables	Number of EFSM transitions	Number of equivalent FSM states	Number of FSM transitions
POP3	4	2	16	6	106
SMTP	2	1	8	4	36
TIME	2	0	2	2	12

Table 1. Preliminary experimental results

V. Conclusions

In this paper, we described the EFSM computer representation that is of a big help when automatically unfolding a given EFSM to an equivalent FSM. Such unfolding needs the explicit enumeration of all different configurations reachable from the initial EFSM state. As the enumeration can lead to the state explosion problem, the maximal number of an equivalent FSM is usually limited. We experimented with several protocol EFSMs and our preliminary experimental results show that the unfolding procedure (at an appropriate description level) can sometimes be performed without limiting the maximal number of states of an equivalent FSM. More experiments with different protocol EFSMs are needed in order to estimate the effectiveness of the developed software.

References

[1] A. Faro and A. Petrenko. Sequence Generation from EFSMs for Protocol Testing. In Proc. of COMNET'90, Budapest, 1990.

[2] A. Petrenko, S. Boroday, R. Groz. Confirming configurations in EFSM testing. *IEEE Trans. on Software Engineering*, 2004, 30(1), pp. 29-42.

[3] A. Petrenko and N. Yevtushenko. Testing from Partial Deterministic FSM Specifications. *IEEE Trans. on Computers*, 2005.

[4] R. Dorofeeva, K. El-Fakih, S. Maag, A.R. Cavalli, N.Yevtushenko. Experimental evaluation of FSM-based testing methods. In: Proc. of the IEEE International Conference on Software Engineering and Formal Methods (SEFM05). Germany, pp. 23-32.

[5] V.A. Sibirjakova. Reverse Polish Notation: manual. Tomsk State University Publishers, 1997, 27 p.

[6] N.V. Spitsyna, A.V. Shabaldin. Web-programming: manual. Tomsk State University Publishers, 2002, 50 p.

Comparing GALS Architectures and Communicational Protocols

S. O. Bykov
dept. of Computer Engineering
Vladimir State University
Vladimir, Russia
e-mail: sobykov@gmail.com

S.G. Mosin
dept. of Computer Engineering
Vladimir State University
Vladimir, Russia
e-mail: smosin@vpti.vladimir.ru

***Abstract*—This paper describes the existing GALS (Globally Asynchronous Locally Synchronous) architectures and communicational protocols with respect to their applications for meeting concrete requirements. So it should help in decision-making during designing asynchronous systems.**

Keywords-GALS systems, pausible clocks, handshake protocols.

I. INTRODUCTION

The problem of distributing the global clock in a chip with minimal clock skew is getting difficult to solve due to the increasing complexity of digital circuits. Additionally the integration of complex systems on chip (SoC) requires a multitude of clock frequencies to be integrated on a common die. A fundamentally different synchronization strategy is used in asynchronous design methodologies. So far as synchronous digital design is well understood and the design methodology and flow are established, it is more effective to combine this two strategies and get advantages of both of them. This idea has been realized in the GALS (Globally Asynchronous Locally Synchronous) approach.

A GALS system consists of complex digital blocks operating synchronously. Those blocks are usually developed using standard synchronous CAD tools and design flow. However, the operation of the blocks is not mutually synchronized— that why the term *locally synchronous* is used. The locally synchronous blocks communicate with one another asynchronously; on the block level (globally), the system is asynchronous. A common approach is to add an asynchronous wrapper, which provides an interface from the synchronous environment to the asynchronous one (and vice versa), to every locally synchronous block.

There are three main strategies for implementing GALS systems: pausible clocking, FIFO-based approach, and boundary synchronization [1]. All of them have their own advantages and should be applied in special cases.

The most popular communication protocol is handshake protocol [2]. But also there are some other protocols, for example, protocols based on clock transfers from sender to receiver, and choice of protocol to be used should be done for each concrete design. there are three main requirements for GALS systems: throughput, area consumption, and power consumption. Meeting all of them is often impossible, so one should choose the most important factor and use it for making decisions during design process. This paper presents the solution which might help to meet each requirement.

II. THROUGHPUT

For systems, processing big streams of information and therefore requiring good throughput, the optimal choice is a FIFO-based solution. This approach uses asynchronous FIFO buffers between locally synchronous blocks to hide the synchronization problem. A SoC architecture that uses distinct clock domains connected through bisynchronous FIFO buffers is commonly called a GALS system. Such systems can tolerate very large interconnect delays and are also robust with with regard to metastability - a state that doesn't settle into a stable '0' or '1' logic level within the time required for proper operation. Designers can use this method to interconnect asynchronous and synchronous systems and also to construct synchronous-synchronous and asynchronous-asynchronous interfaces. Figure 1 diagrams a typical FIFO interface.

The advantage of FIFO synchronizers is that they don't affect the locally synchronous module's operation, therefore the FIFO-based approach allows achieving high throughput. However, with very wide interconnect data buses, FIFO structures can be costly in terms of silicon area. Also, they require specialized complex cells to generate the empty/full flags used for flow control [1]. Another disadvantage of FIFO-based solution is high power consumption. That is why this architecture cannot be effective in mobile systems.

In such kind of systems a standard protocol for working with FIFO is generally used: sender writes data to FIFO and receiver read it. But there are some systems, for example the DSP platform described in [3], where data transmission is performed through communication network. Such solutions use communicational protocol based on clock transfers from sender to receiver. Sender sends clock signal with data and these data are written to FIFO, clocked on write side by sent

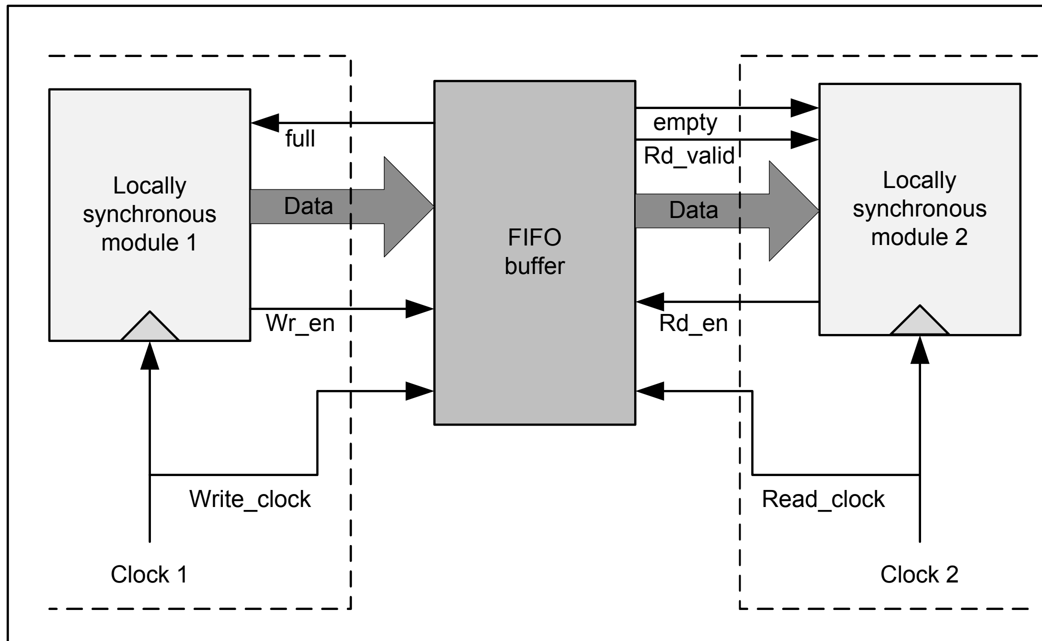


Figure 1. Typical FIFO-based GALS system

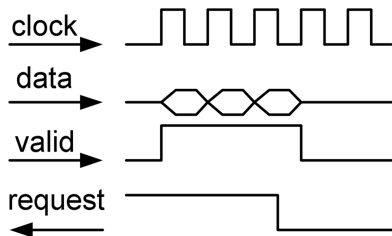


Figure 2. Timing diagram for protocol based on clock transfers

clock. Receiver reads these data from read side of FIFO. Timing illustration for this protocol is presented on Figure 2.

III. AREA OVERHEAD AND POWER CONSUMPTION

Area overhead and power consumption can be considered together, because power consumption is generally related to area if no special techniques are used. The most effective solution for these requirements is pausible clocking. This approach is described in [4]. Figure 3 illustrates the general structure of such system. The basic idea of this approach is transferring data between wrappers when both the data transmitter and data receiver clocks are stopped. This elegantly solves the problem of synchronization between the two clock domains [1]. But throughput of this solution strongly depends on data transfers rate. If the rate is high, frequent clock pauses will practically stop work of synchronous block. This problem is solved in boundary synchronization approach, where data synchronization at the borders of the locally synchronous island performs without affecting the inner operation of locally synchronous blocks and without relying on FIFO buffers. This method can achieve very reliable data transfer between locally synchronous blocks. On the other hand, such solutions generally increase latency and reduce data throughput, resulting in limited applicability for high-speed systems [1].

Pausible clocking and boundary synchronization approach require programmable ring oscillators. This is an inexpensive solution that allows full control of the local clock. However, it has significant drawbacks. Ring oscillators are impractical for industrial use. They need careful calibration because they are very sensitive to process, voltage, and temperature variations. Moreover, embedded ring oscillators consume additional power through continuous switching of the chained inverters.

Solutions, based on pausible clocking, use standard handshake protocol and its different modifications. Existing asynchronous handshake protocols are bundled data protocol, dual-rail data protocol, 1/N data protocol and single-track data protocol [5]. Timing illustration for bundled data protocol is presented on Figure 4. Req and Ack signals need to be changed two times in one transmission cycle, so it is much slower. It allows reusing existing synchronous units, and it can be implemented in a small area, but fail to conquer electromagnetic interference (EMI) [5]. The advantage of dual-rail data protocol is that the data-

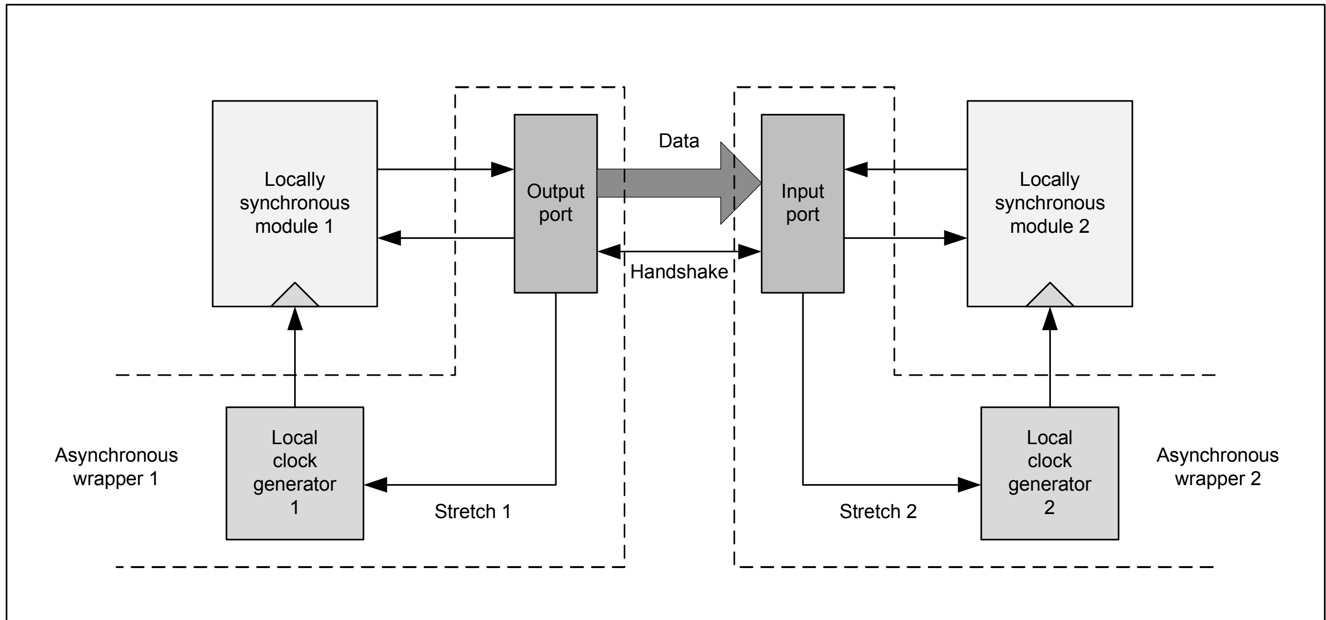


Figure 3. GALS system with pausible clocking

validation information can be carried on data themselves, that there is no need to use Req signal to denote data validation, thus avoiding the delaymatching efforts brought by the complex clocking relationship between req signals and ack signals. Dual-rail data protocol has a better anti-EMI capability due to the fact that two lines represent one data. However, the protocol implementation requires extra chip area (nearly twice as large as the bundle data protocol does) [5].

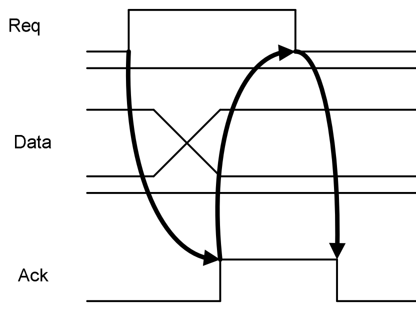


Figure 4. Timing diagram for bundled data protocol

IV. Application of GALS solutions

GALS systems can be used for video coding/decoding. MPEG-4 decoder is example of system, which requires a high throughput. All systems, worked with video, must process big information streams. So, it is necessary to use FIFO-based solution for such designs, because other architectures cannot provide needed throughput.

V. Conclusion

GALS systems are roughly developing direction in a modern science. This paper presented some methods for design such systems. In some cases it may be not optimum, but generally these solutions can be used for various types of designs.

References

[1] Miloš Krstić, Eckhard Grass, Frank K. Gürkaynak, Pascal Vivet, “Globally Asynchronous, Locally Synchronous Circuits: Overview and Outlook”, IEEE Design & Test, v.24 n.5, pp.430-441, September 2007, doi:10.1109/MDT.2007.164

[2] Joep L. W. Kessels, Ad M. G. Peeters, Paul Wielage, Suk-Jin Kim, “Clock Synchronization through Handshake Signalling”, Eighth International Symposium on Asynchronus Circuits and Systems (ASYNC'02), IEEE Computer Society Press, April 2002, doi: 10.1109/ASYNC.2002.10002

[3] Anh Tran, Dean Truong and Bevan Baas, "A GALS Many-Core Heterogeneous DSP Platform with Source-Synchronous On-Chip Interconnection Network," ACM/IEEE International Symposium on Networks on Chip (NOCS), San Diego, CA, USA, May 2009, pp. 214-223.

[4] K.Y.Yun, R.P.Donohue “Pausible Clocking: A First Step Toward Heterogenous Systems”, Proc. International Conference on Computer Design (ICCD), IEEE Computer Society Press. 1996, pp. 118-123.

[5] Xuguang Guan, Duan Zhou, Dan Wang, Yintang Yang, Zhangming Zhu “A Novel GALS Single-Track Protocol Asynchronous Communication Circuits”, Pacific-Asia Conference on Circuits,Communications and System’ 2009(PACCS’09), May 2009, pp. 269 - 272.

Strategy of selecting power reduction technique for energy-efficient semiconductor designs

P.V. Parnevich
dept. of Computer Engineering
Vladimir State University
Vladimir, Russia
e-mail: PVParnevich@gmail.com

S.G. Mosin
dept. of Computer Engineering
Vladimir State University
Vladimir, Russia
e-mail: smosin@vpti.vladimir.ru

Abstract **– There are a lot of techniques, which oriented on power consumption reducing. Unfortunately, there is no such method that is able to meet all the system requirements for energy minimization. This paper presents a strategy of power reduction technique selecting that can be used to make effective decisions in developing energy-efficient semiconductor designs. Techniques operating at synchronous and asynchronous schemes are described, including dynamic voltage and frequency scaling, clock gating, state-retention power gating and others.**

Keywords-energy-efficient technique; clock gating; power gating; dual voltage scheme; GALS

1. Introduction

The design of an electronic system is almost always constrained by power and energy considerations – whether it is battery life for a mobile device, thermal power dissipation in a high-performance processor or ultra-low power consumption for a wireless sensing application. In addition, recent economic forces and increased environmental awareness have changed the landscape for new product design. Now energy efficiency is often the lead discussion as companies formulate new product strategies. However, even though energy budgets are playing a larger role in determining the finished designs, manufacturers realize that the market will not allow them to compromise on performance.

There are some techniques, like GALS (globally asynchronous, locally synchronous) systems [1], DVFS (dynamic voltage and frequency scaling) based techniques [2], clock gating [3], state-retention power gating (SRPG) techniques [4], which oriented on power consumption reducing.

Unfortunately, there is no single power reduction technique that is able to meet all the system requirements for energy minimization. The trick is to effectively combine different techniques to intelligently develop energy-efficient semiconductor designs.

This paper presents a strategy of power reduction technique selecting that can be used to make effective decisions in developing energy-efficient semiconductor designs.

2. Synchronization strategy selecting

The scientific community is showing great interest in GALS solutions and architectures nowadays due to their high performance. However, sometimes it's profitably to use fully synchronous schemes, because synchronous digital design is well understood and the design methodology and flow are established. Thereby it's necessary to care about guaranteeing energy efficiency in both cases.

2.1. GALS systems

A GALS system consists of complex digital blocks operating synchronously. Those blocks are usually developed using standard synchronous CAD tools and design flow. However, the operation of the blocks is not mutually synchronized – hence the term locally synchronous. These locally synchronous blocks communicate with one another asynchronously by using handshaking protocol.

Since each block is in its own frequency domain, it becomes possible to reduce the power dissipation and increase energy efficiency in many ways:

- GALS clocking design allows to utilize simple local ring oscillator for each core, and hence eliminates the need of complex and power hungry global clock trees [5].
- Unused cores can be effectively disconnected by power gating, and thus reducing leakage.
- When workloads distributed for cores are not identical, we can allocate different clock frequencies and supply voltages for these cores either statically or dynamically. This allows the total system to consume a lower power than if all active cores had been operating at a single frequency and supply voltage [6].

However, there are several reasons why standard design practice has not adopted GALS techniques. Ring oscillators are impractical for industrial use. They need careful calibration because they are very sensitive to process, voltage, and temperature variations. Moreover, embedded ring oscillators consume additional power through continuous switching of the chained inverters.

Another significant drawback is that functional test of asynchronous circuits is very difficult because most ATE (Automatic Test Equipment) is cycle based and cannot provide event-based handshake signals. For GALS circuits, the process of arbitration and stretching leads to nondeterministic timing behavior. Therefore, the test result can differ from chip to chip and from test run to test run.

2.2. Synchronous schemes

Like in GALS systems there are many ways to increase energy efficiency in synchronous schemes.

- DVFS allows on-the-fly frequency adjustment according to existing system performance requirements. By lowering the frequency, it is possible to lower the operating voltage (on-the-fly as well), dramatically reducing the power consumption. A drawback of the proposed approach is a reduction in reliability, resulting from the lower link voltage.
- Clock gating is an effective strategy that is widely used to help reduce power consumption while maintaining the same levels of performance and functionality. A circuit uses more power when it's being clocked than when the clock is gated or turned off. Clocks can consume as much as 40 percent of active power. Shutting off the clocks and stopping the data toggling in unused portions of the semiconductor brings sizable energy saving. In this case the organization of clock management becomes a crucial task.
- SRPG is a technique that allows the voltage supply to be reduced to zero for the majority of a block's logic gates while maintaining the supply for the state elements of that block. SRPG can thereby greatly reduce power consumption when the application is in stop mode, yet it still accommodates fast wake up times. Reducing the supply to zero in stop mode allows both dynamic and static power to be removed. Retaining the supply on the state elements allows processing to continue quickly when exiting stop mode.The quality of complex power network is critical to the success of a power-gating design.

3. Energy-efficient technique appliance

The effective combination of energy-efficient techniques was presented in [7]. The main idea of the proposed design is to implement a complete Dynamic Voltage and Frequency Scaling architecture within a complex GALS NOC.

Each units of the dedicated SoC are arranged around a fully asynchronous Network-on-Chip. As shown Figure 1, the NoC units are fully synchronous islands. Synchronization between the communication router and the units is done thanks to a pausable clock mechanism called SAS (containing Synchronous-to-Asynchronous and Asynchronous-to-Synchronous interfaces). A programmable Local Clock Generator (LCG), using a delay line, is implemented within each unit to generate a variable frequency in a predefined applicative range. The power unit manages the local unit voltage, sharing a power switch between a V_{DD} hopping technique and an ultra-cut-off block. The Power Unit uses two external voltages : V_{HIGH} and V_{LOW} to be automatically switched during DVS phases. The Network Interface (NI) is in charge of communications with respect to NoC protocol and is also in charge of local voltage and frequency control for DVFS using a Low Power Manager.

The power management strategy is programmed by the main CPU, through NoC unit attached Network Interface's Low Power Managers, according to required performance and power constraints. DVFS can be executed during IP computation and communication according to their own activity. The only global signals are regarding units' reset and off control. The main CPU is required to directly disable/enable the units for power off mode and the corresponding reset phase.

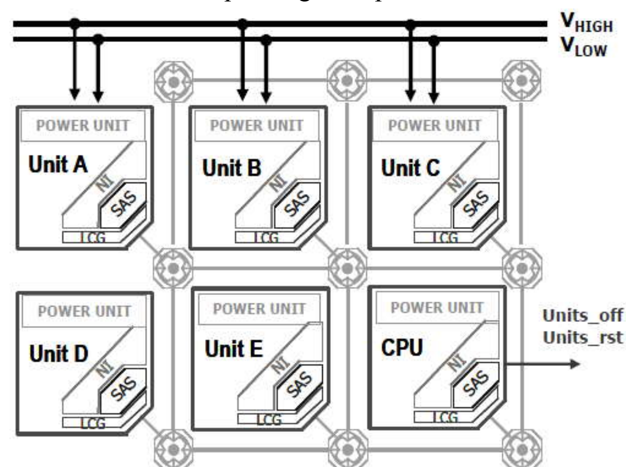


Figure 1. DVFS NOC architecture.

The power efficiency of the proposed system has been evaluated close to 95%.

4. Conclusion

All the described methods are oriented on energy minimization. The trick is to learn how to use and combine them effectively to reach more energy savings. This paper presented a strategy of power reduction technique selecting that can be used to create low energy designs.

References

[1] Miloš Krstić, Eckhard Grass, Frank K. Gürkaynak, Pascal Vivet, "Globally Asynchronous, Locally Synchronous Circuits: Overview and Outlook", IEEE Design & Test, v.24 n.5, pp.430-441, September 2007, doi:10.1109/MDT.2007.164.

[2] Wen-Yew Liang, Shih-Chang Chen, Yang-Lang Chang, and Jyh-Perng Fang, "Memory-aware dynamic voltage and frequency prediction for portable devices", Proc. 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'08), August 2008, pp. 229-236, doi: 10.1109/RTCSA.2008.19.

[3] Jithendra Srinivas, Madhusudan Rao, S. Jairam, H. Udayakumar, Jagdish Rao, "Clock gating effectiveness metrics: Applications to power optimization", Proc. 10th International Symposium on Quality of Electronic Design, pp. 482-487, doi:10.1109/ISQED.2009.4810342.

[4] Kanak Agarwal, Kevin Nowka, Harmander Deogun, Dennis Sylvester, "Power Gating with Multiple Sleep Modes", Proc. 7th International Symposium on Quality Electronic Design, pp.633-637, March 27-29, 2006 , doi:10.1109/ISQED.2006.102.

[5] V. Tiwari, D. Singh, et al., "Reducing power in high-performance microprocessors", Proc. 35th annual Design Automation Conference, pp. 732 – 737, June 1998.

[6] Sebastian Herbert, Diana Marculescu, "Analysis of dynamic voltage/frequency scaling in chip-multiprocessors", Proc. 2007 international symposium on Low power electronics and design, pp. 38 – 43, August 27-29, 2007.

[7] E. Beign´e, F. Clermidy, S. Miermont, and P. Vivet, "Dynamic voltage and frequency scaling architecture for units integration within a GALS NoC," in Network on Chip, Proc. International Symposium, pp. 129–138, 2008, doi:10.1109/NOCS.2008.26.

Contract Specification of Hardware Designs at Different Abstraction Levels: Application to Functional Verification

Mikhail Chupilko and Alexander Kamkin
Institute for System Programming of the Russian Academy of Sciences
25, A. Solzhenitsyn st., Moscow, 109004, Russia
E-mail: {chupilko, kamkin}@ispras.ru

Abstract— **The paper touches upon the issues of functional specification and verification of digital hardware at different abstraction levels. It shows how behavioral models of various degrees of abstraction can be represented by means of the contract paradigm and how contract specifications can be applied to generate test sequences in an automated way. The testing technique under consideration is based on the traversal of FSM derived from specifications. Taking into account that contract specifications are well known to be a high-efficient tool for constructing response checkers and estimating test coverage, we can assuredly report that the contract-based approach is a universal solution for hardware verification at different levels of abstraction.**

I. Introduction

Automation of hardware design verification is feasible only if requirements to hardware are represented in a formal way. This refers not only to formal methods where using mathematical models is one of the integral parts but to simulation-based methods as well. Requirements being represented formally, i.e., in a machine-readable form, are called *formal specifications*. Basing on such specifications one can automatically derive a design model (usually, some kind of automaton) and apply it to functional verification (e.g., to generate stimuli, check reactions, or estimate test adequacy).

When verifying hardware of different size and complexity it is reasonable to use specifications and models of different levels of abstraction. For example, a hardware unit can be specified cycle-accurately, while description of a whole system can be done in a more generalized manner. Choice of an abstraction level depends on the requirements to be specified (they are not always very accurate) and in many cases is conditioned by reusability and maintenance reasons – more accurate specifications are, less reusable they are, and more labor-consuming their maintenance is.

Nowadays, there are lots of specification-based techniques for functional verification of hardware, but there is a lack of unified solutions, which are applicable to various abstraction levels, e.g., contemporary unit and system verification methods are completely different. This paper classifies the modeling levels used in hardware verification and considers how diverse models can be uniformly described with the help of *contract specifications* (i.e., *pre-* and *post-conditions*) and how such specifications can be used to automate simulation-based verification (test sequence generation, in particular).

The testing technique concerned is based on the traversal of FSM constructed from contract specifications. It is obvious that specifications of different size and accuracy produce different automata in terms of determinacy and number of transitions. It is not always possible to make use the derived FSM model due to the huge size or for some other reason. To make the model useable for test generation, one should generalize it. Doing this involves many factors and mostly remains the art of verification. Some of the approaches to construct automata from specifications are described in the paper.

The rest of the article is organized as follows. Section II reviews the papers devoted to specification-based verification of hardware designs. In Section III classification of the abstraction levels used for hardware modeling is given. This section also describes how various models can be represented by means of contract specifications. In Section IV construction of FSM from contract specifications (to be afterwards used for test sequence generation) is considered. Section V gives a concise description of tool support. Finally, Section VI concludes the paper and outlines the directions of our future work.

II. Related Work

There are lots of research and industrial papers on specification-based verification methods. This gives evidence that using formal specifications and models is a right direction for hardware verification. The main question is what kind of specifications and models are preferable. To automate different tasks of testing, distinct types of models are usually used. For example, stimuli generation can be performed on the base of FSM models, while correctness checking can be done by means of temporal assertions. This has a certain disadvantage. Two models require maintenance during the design process to keep up their mutual consistency.

The most of the papers are dedicated to the methods of test sequence generation. Many of them suggest using explicit cycle-accurate models to generate test sequence, e.g., Ur et al. [1] and Mishra et al. [2], [3], [4] use SMV models; Ho

This work was supported by the RFBR (grant 08-01-00889-).

et al. [5] utilize Synchronous Murφ. The main differences between the approaches are concentrated in the following methods: a model construction method (manual development [1], automatic derivation from an RTL description [5], and automatic derivation from specifications [2]) and a test sequence generation method (FSM traversal [1], [5] and model checking [2], [3], [4]).

Manual development of a model is error-prone, while automatic derivation from an RTL description does not scale well on complex hardware designs. In our opinion, the most promising method of model construction is automated extraction from formal specifications. Speaking about test generation, model checking techniques are not intended for full-scale functional verification. They are aimed to verification of a relatively small number of properties. The most usable way of test sequence generation is based on FSM traversal.

In the suggested approach, a model for test sequence generation, so-called generalized FSM model, is almost automatically derived from specifications. The approach uses implicit specifications in the from of pre- and post-conditions and irredundant algorithms for FSM traversal. The distinction feature of the approach is that it does not require two different models for checking design correctness and for test sequence generation. All testing tasks are carried out basing on contract specifications.

III. Specification of Hardware Designs at Different Abstraction Levels

When specifying hardware designs we mostly focus on their behavior (functionality). In other words, hardware is considered to be a black box with the given inputs and outputs, and our goal is to specify the input-output relation of the DUV (Design Under Verification).

It is widely recognized that EFSM (Extended Finite State Machine) is one of the most natural formalisms for hardware modeling [6]. It would be recalled that, roughly speaking, EFSM is an automaton with parameterized inputs (stimuli) and outputs (reactions) where state is divided into control and data parts.

In this paper, it is implied that a functional model of hardware is an EFSM-like automaton (formal definitions are not given not to overload the paper). We distinguish the following abstraction levels for hardware modeling (in order of increasing abstractness):

- cycle-driven models:
  - cycle-accurate models;
  - adaptive cycle-driven models;
- event-driven models:
  - ordered events models;
  - unordered events models;
- operation-driven models.

Let us describe the basic properties of hardware models at each of the abstraction levels given above.

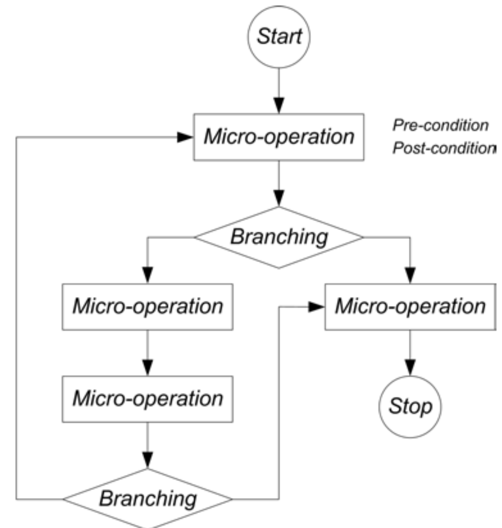


Fig. 1. A control flow graph of an operation.

A. Cycle-Driven Models

Cycle-driven models are the least abstract and most detailed ones. A model is called *cycle-driven* if each of its transitions corresponds to exactly one clock cycle in hardware[1]. To formally describe a cycle-driven model by means of contract specifications, the operations implemented by a DUV should be decomposed into a number of one-cycle *micro-operations* each being described by an individual *contract* (i.e., a *pre-condition* and *post-condition*). It should be noticed that micro-operations of an operation can be connected to each other not only by linear ordering but using more complex control flow relation as well (see Fig. 1, for example).

Summing up, cycle-driven specification of an operation includes its pre-condition (which constrains the situations in which the operation is allowed to start), a set of interconnected micro-operations, and pre- and post-conditions of the micro-operations. Semantics of a micro-operation's pre-condition (which is also called a *guard condition*) differs from the semantics defined for an operation's pre-condition – if a guard condition is not satisfied, it indicates that the micro-operation is interlocked (it will be unlocked, when the guard condition becomes true).

Cycle-driven models are subdivided into two types: cycle-accurate models and adaptive cycle-driven models.

*1) **Cycle-Accurate Models**:* A cycle-driven model is referred to as *cycle-accurate* if for any admissible input sequence it allows *deterministically* identifying (predicting) a DUV's reaction. In other words, it is a self-contained description of the design cycle-by-cycle execution, which can be used *independently* from the implementation (that is the difference between accurate and adaptive models).

In a sense cycle-accurate models are ideal for *co-simulation*. In each cycle a testbench applies an input both to a DUV and

[1]It does not make any serious difference whether a transition or staying in a state corresponds to a clock cycle.

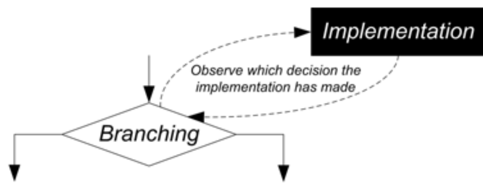


Fig. 2. Using a feedback for choosing a branch.

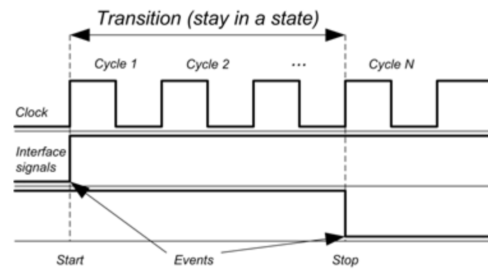


Fig. 3. A transition in an event-driven model.

to a model and compares their outputs for equivalence. There is no problem in determining when to apply a stimulus or when to check a reaction – these actions are performed in each cycle of simulation. However, making a cycle-accurate model is a difficult task which is almost tantamount to writing one more implementation.

*2) **Adaptive Cycle-Driven Models**:* *Adaptive cycle-driven models* are not as self-contained and deterministic as cycle-accurate models are. Sometimes they are not able to determine a DUV's reaction basing only on the input sequence, but it can always decide which transition to perform observing some outputs of the design. In other words, there is a *feedback* from a DUV to a model (that is why such models are called *adaptive*).

Due to the feedback adaptive models can not be used independently from the implementation. Speaking in the terms of contract specifications, micro-operations' pre-conditions and branching conditions can be defined not only over the model variables (control and data state) but over the design's outputs as well (see Fig. 2). The post-conditions do not check the values of the feedback outputs (DUV's outputs the model depends on).

Adaptive models are thought to be a bit more abstract than accurate ones, because they abstract away the way in which feedback outputs are calculated. When using such kind of models there is a tacit assumption that feedback outputs are computed correctly.

B. Event-Driven Models

Event-driven models are the next step in increasing abstractness of the hardware functionality description. The key distinction of that sort of models is that there is no rigid connection between model transitions and clock cycles. Different transitions (or stays in states) can take different amount of time to be completed. Moreover, the same transition being executed several times can take different number of cycles for each execution.

The main concept of event-driven models is an *event*, which is an instantaneous interaction between a design and its environment that can be observed or predicted. If we are talking about verification, an event is an atomic interaction between a DUV and a testbench (e.g., sending of a stimulus, assignment of an output, etc.). A model is called *event-driven* if its transitions are associated with events, not cycles (see Fig. 3).

To specify an event-driven model by means of contract specifications, each operation is decomposed into a number of *sub-operations* (interactions). Further thoughts are very similar to those that are applied to cycle-driven models – all sub-operations are connected into a control flow graph, while each sub-operation is specified by pre-and post-conditions.

Event-driven models are subdivided into two types: ordered events models and unordered events models.

*1) **Ordered Events Models**:* An *ordered events model* is an event-driven model where each transition corresponds to a set of simultaneous events or a linearly ordered set of events happening at different times. It should be noticed that cycle-driven models are a particular case of ordered events models. Indeed, a clock cycle can be considered as a kind of event.

Event-driven models (including models of ordered events) are usually adaptive, because in general case their behavior depends not only on input events (stimuli) but on output events (reactions) as well. To be able to use event-driven models for simulation-based verification, one should develop special testbench components that detect a DUV's reactions and transmit them into a model. Such components are called *catchers*.

*2) **Unordered Events Models**:* In contrast to a model of ordered events, each transition of an *unordered events model* is associated with an unordered set of events. In other words, using such kind of models we know which events have taken place during a transition, but we do not know the linear order of the events (because, for example, some events are not visible or communication medium can change the order).

The distinctive feature of unordered events model is how reactions are checked. Since the events order is not fully known, a testbench tries to create all admissible orders and check the events' post-conditions. This process is called *serialization*. If there is at least one sequence of events such that all the post-conditions are satisfied, the overall reaction is considered to be correct.

C. Operation-Driven Models

No research on hardware verification would be complete that did not consider *operation-driven models*. Such models are the most abstract and described by a vacuous single-state automaton in which each transition corresponds to an operation call. Operation-driven models abstract away from operations' structure (micro- and sub-operations) and cooperative execution of several operations.

Specification of an operation is a classical software contract consisting of a pre-condition (which constrains the situations

in which the operation is allowed to start) and post-condition (which constrains the expected results of the operation).

Operation-driven models are often applied to core-level verification of microprocessors being done with the help of test programs. The same programs are executed on a microprocessor RTL design and its instruction-level simulator (operation-driven model). The results of the two models are compared for equivalence.

D. Unified View on Hardware Specification

It is important that models of various degree of abstraction can be specified in a unified way by means of contract specifications. Contract specifications abstraction/refinement is the topic of a separate research, but we can report that changing abstraction level can be done almost seamlessly. To illustrate the idea, let us consider a simple example. In the example, a operation-driven specification is refined into cycle-driven one by adding timing information without changing the structure of the specification.

The code below (we use the language SeC, specification extension of C) describes the functionality of an address translation operation being implemented by a hypothetical TLB (Translation Lookaside Buffer): if a virtual address is invalid, then the output `error` is set to one; if the address is valid but does not belong to the buffer, then the output `miss` is set to one; if the address is valid and it belongs to the buffer, then the output `pa` is set to the resultant physical address.

```
specification void AddressTranslation(VA va) {
  TLB *tlb = getDUV();
  // Pre-condition of the operation
  pre {
    return true;
  }
  // Post-condition of the operation
  post {
    TLBEntry *entry;
    POST(tlb.out.error == !isValid(va));
    if(!isValid(va)) { STOP(); }
    POST(tlb.out.miss == !isHit(tlb, va));
    if(!isHit(tlb, va)) { STOP(); }
    POST(tlb.out.pa == translate(tlb, va));
    STOP();
  }
}
```

To simplify description, the specification uses two special macroses, `POST` and `STOP`. The first of them checks the predicate given and return $false$ if the predicate is not satisfied. The second macros simply returns $true$.

The code below refines the given specification by adding some timing information. This is done with the help of the constructs `CYCLE` and `PRE`. Calling `CYCLE` leads to a one-cycle delay. `PRE` waits until the condition given is satisfied.

```
specification void AddressTranslation(VA va) {
  TLB *tlb = getDUV();
  pre {
    return true;
  }
  post {
    TLBEntry *entry;
    // Post-condition of the micro-operation 1
    POST(tlb.out.error == !isValid(va));
    if(!isValid(va)) { STOP(); }
    // One-cycle delay
    CYCLE();
    // Post-condition of the micro-operation 2
    POST(tlb.out.miss == !isHit(tlb, va));
    if(!isHit(tlb, va)) { STOP(); }
    // Pre-condition of the micro-operation 3
    PRE(tlb.out.ready == 1);
    // Post-condition of the micro-operation 3
    POST(tlb.out.pa == translate(tlb, va));
    STOP();
  }
}
```

IV. Test Sequence Generation at Different Abstraction Levels

The test sequence generation technique we use for verification is based on the traversal of a control FSM derived from specifications. The distinctive feature of the approach is that it does not require explicit FSM representation – instead, the only information it operates with is a traversed part of a state graph, current state, and set of available stimuli. Using such a technique implies that FSM being traversed is deterministic (in some sense) and has a strongly connected state graph.

Specifications of different abstractness lead to different automata in terms of determinacy and number of states. Obviously, it is not always possible to automatically derive a deterministic FSM with a rather small number of states – sometimes a resultant automaton is nondeterministic; other times it is of a huge size.

Thus, one should choose a right abstraction level and given the abstraction level, construct a deterministic automaton (trying to avoid redundant states, of course). This task is not fully automated, but there are several empiric ideas (testing patterns) that we would like to consider. These ideas basically relate to the current state calculation function of the implicit FSM representation.

A. Cycle-Driven Models

*1) **Stage-Based FSM**:* The most usable pattern of FSM description for cycle-driven specifications is as follows. The control state is represented as a set of simultaneously executing micro-operations (see Fig. 4). This approach is called a *micro-operation-based FSM* or a *stage-based FSM* (*stage* is a generalization of micro-operations and sub-operations).

If there are inter-operation *data dependencies* which have an influence on pipeline interlocks (there are nontrivial micro-

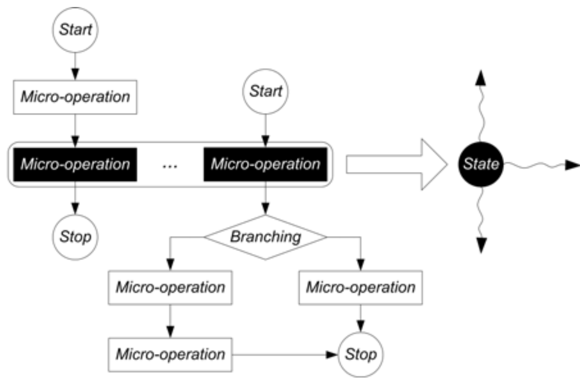


Fig. 4. Constructing a micro-operation-based (stage-based) FSM.

operations' pre-conditions), the pure stage-based approach does not work, because the resultant FSM is most likely nondeterministic. In this case, an extended approach should be used – an automaton state is not only a set of micro-operations, but it also contains some control information describing dependencies between operations. It is also sensible to use *timers* (to make states describe not only a micro-operations but time of their processing as well). More detailed description of the approach is available in [8].

2) ***Resource-Based FSM****:* The other approach is called a *resource-based FSM*. An automaton state represents not currently executing micro-operations but the resources allocated by the micro-operations. This testing pattern abstracts away from the particular operations processing by a design, but it often requires additional work to make an FSM deterministic.

We distinguish some types of resources which hardware designs usually contain: arbiters, FIFO-buffers, RAM-memories, and data transfer channels. Each resource is specified with an individual FSM, and each FSM is described implicitly by defining a way for the resources state calculation. There are some patterns for specifying states. For example, state of an arbiter is often represented as a pair $\langle requests, history\rangle$, where $requests$ are queries having been sent to the arbiter during the previous cycle, and $history$ is a finite sequence of the recent arbiters decisions (it is used to calculate priorities of the requests); state of a FIFO is usually described as a number of entries in the buffer; and so on. Having got known what kind of resources a DUV consists of, one is able to construct a description of a total FSM by concatenating all state calculation functions.

B. Event-Driven Models

The stage- and resource-based approaches can be also applied to event-driven specifications. In this case, sub-operations (instead of micro-operations) are considered. In the stage-based approach, an FSM state is a set of processing sub-operations. In the resource-based approach, an automaton state is a set of the resources allocated by the sub-operations.

C. Operation-Driven Models

Operation-driven specifications are not able to produce an adequate control FSM due to the low informativeness (high abstractness). In this case, combinatorial techniques are used for test generation. The test sequences are generated by systematic enumeration of all feasible combinations of the given operations, test situations (i.e., paths in control flow graphs) and dependencies via shared resources. To reduce number of tests, one can use heuristics, like operation factorization, limitation of the number of dependencies, etc. [7]

V. Tool Support

The suggested approach to specification and test sequence generation is supported by the CTESK toolkit developed at the Institute for System Programming of the Russian Academy of Sciences (ISPRAS) [9]. This toolkit is originally intended for testing software systems written in C, but it has been adapted for verification of hardware designs.

CTESK uses SeC language for development of testbench components. SeC is a C extension, which has additional constructs to define specifications, FSM-based test scenarios, etc. Testbench functionality connected with functional verification of hardware designs is implemented as a library extension of CTESK.

VI. Conclusion

Contract specifications are applicable to a wide rage of hardware including complex parallel-pipeline designs with control flow branching and parallel threads inside individual operations [8]. Their usage allows automating all tasks of simulation-based verification. This simplifies maintenance of functional tests and reduces verification efforts. The important quality of contract specifications is that they can be easily applied to functional verification of hardware at different abstraction levels. Futher we are planning to do a research on smooth changing of specification abstractness. Choosing right abstraction level for verification is a problem which is really hard to formalize. We believe that a tool that could change the abstractness without changing the structure of specifications would be rather useful for verification engineers.

References

[1] S. Ur, Y. Yadin. "Micro architecture coverage directed generation of test programs". Proc. of Design Automation Conference, 1999.

[2] P. Mishra, N. Dutt. "Functional coverage driven test generation for validation of pipelined processors". Proc. of Design, Automation and Test in Europe, 2005.

[3] H.M. Koo, P. Mishra. "Test generation using SAT-based bounded model checking for validation of pipelined processors". Proc. of ACM Great Lakes Symposium on VLSI, 2006.

[4] H.M. Koo, P. Mishra. "Functional test generation using property decomposition for validation of pipelined processors". Proc. of Design, Automation and Test in Europe, 2006.

[5] R. Ho, C. Yang, M. Horowitz, D. Dill. "Architecture validation for processors". Proc. of International Symposium on Computer Architecture, 1995.

[6] A. Petrenko, S. Boroday, R. Groz. "Confirming Configurations in EFSM Testing." IEEE Transactions on Software Engineering, 2004.

[7] A. Kamkin. "Combinatorial model-based test program generation for microprocessors". Preprint of ISPRAS, 2009.

[8] M. Chupilko, A. Kamkin. "Specification-driven testbench development for synchronous parallel-pipeline designs". Proc. of the NORCHIP conference, 2009.

[9] http://hardware.ispras.ru

An Approach to Test Programs Generation for Microprocessors Based on Pipeline Hazards Templates

Alexander Kamkin, Dmitry Vorobyev
Institute for System Programming of the Russian Academy of Sciences
25, A.Solzhenitsyn Street, Moscow, 109004, Russia
E-Mail: {kamkin, vorobyev}@ispras.ru

Abstract — **In this paper we describe an approach to automated test programs generation intended for microprocessor verification. The approach is based on formal specification of microprocessor ISA and description of pipeline hazards templates. The use of formal specifications allows automating development of test program generators and systematizing control logic verification. Since the approach is underlain by high-level descriptions, all specifications and templates developed, as well as the constructed test programs, can be easily reused when the processor's microarchitecture changes. It makes it possible to apply the methodology in early stages of a microprocessor development cycle when the design is frequently modified.**

I. Introduction

Functioning of a modern pipelined microprocessor is implemented in a very difficult way. Pipeline can concurrently process multiple instructions, which, in addition, can interact each other via shared resources. At every cycle of execution a microprocessor makes lots of decisions on hazards resolution, branches processing, exceptions handling, and so on. The microprocessor mechanisms responsible for controlling instructions execution are called *control logic*.

Control logic is a key component of a microprocessor; that is why it should be designed and verified thoroughly, not missing any detail. However, the common practice is to produce tests manually or using random generation techniques, which is obviously inefficient and unsystematic. The other kinds of methods are based on *cycle-accurate models*. Such approaches are aimed at detailed verification of control logic, but the problem is that accurate models are very hard to develop and maintain.

The approach suggested in this work is thought to be somewhere between random generation techniques and techniques based on cycle-accurate models. The approach uses formal specification (modeling) of a microprocessor instruction set (ISA, Instruction Set Architecture). Of course, instruction-level models are less informative in comparison with cycle-accurate ones, but they have a number of practical advantages. First, they are much easier to develop, and second, they can be reused even if the microarchitecture is considerably altered.

This work was supported by the RFBR (grant 08-01-00889-a).

The rest of the paper is organized as follows. Section II is a survey of the related work. Section III introduces the main concepts of the suggested approach. The description of the approach is given in Section IV. Section V considers a case study. Finally, Section VI concludes the paper.

II. Related Work

In the paper [1] two mutually complementary techniques are described. The first one is a test generation technique basing on *model checking*, while the second one uses *template-based procedures*. Source information for the both approaches is a microprocessor specification written in EXPRESSION [2]. Basing on the specification, a generic structure-behavior model in SMV [3] is automatically derived. For this model, a test developer defines a *fault model*. For each element of the fault model the negation of the corresponding property is produced. Then, a counter-example is generated using the SMV model checker. As the authors say, model checking does not scale on complex designs. So, the technique employing templates is used as an addition.

Templates are developed by hands and describe sequences of instructions that create special situations in microprocessor behavior (in the first place, pipeline hazards). Generation is performed with the help of the graph model extracted from the specification. The template-based technique requires greater efforts, but it scales well. It should be noticed that both approaches are based on rather accurate specifications, and it is better to apply them in late design stages, when the microarchitecture is stable. Otherwise, there would be a need to modify the specification to keep up its consistency.

In the approach [4], pipeline structure is formally specified in the form of state machine, which is called OSM (Operation State Machine). OSM describes control logic at two levels, called *operational* and *hardware* levels. At the first level, "movement" of instructions through the pipeline stages is described (every operation is described by a separate state machine). At the second level, hardware resources are modeled using so-called *token managers*. An operation state machine changes states by capturing and moving tokens. A pipeline model is defined as a composition of operation state machines and resource state machines. The goal of testing is to traverse all the transitions of the joint

automaton. Like the previous techniques, this approach is based on accurate specifications.

The paper [5] considers the test program generation tool Genesys-Pro (IBM Research). A generator is composed of two main components, an *engine* (which does not depend on target architecture) and a *model* (which describes microprocessor-specific knowledge). A verification engineer develops *templates*, which specify structure of test programs and properties they should satisfy. Genesys-Pro transforms each template into a set of constraints and builds a test program using *constraint solving* techniques. The tool is quite universal and applicable to different microprocessor architectures. However, so far as development of test templates is done by hand, tests maintenance is hard enough.

In the paper [6] a technique for test generation basing on FSM traversal is described. In one of the stages the technique uses Genesys (the previous version of Genesys-Pro). A test developer creates a microprocessor model using the SMV language. After this, a set of paths covering all the edges of the state graph extracted from the model is built. Each path (so-called *abstract test*) is translated into a Genesys template. The technique allows achieving good coverage of control logic, but it has two principal shortcomings. First, one needs a skilled expert to develop a microprocessor model. Second, to be able to map abstract tests into Genesys templates, a complex description has to be done.

Summing up, all the techniques reviewed can be divided into two classes: *techniques based on accurate models* [1,4,6] and *techniques based on templates* [5]. The techniques of the first class allow achieving high quality of verification. However, it is not practical to use them in early design stages. Template-based techniques do not have this shortcoming, but they are unsystematic and cannot guarantee high quality of verification.

III. Foundations of the Suggested Approach

The suggested approach is based on *combinatorial model-based generation* [7]. It uses *formal specifications* of microprocessor ISA, which describe instructions regardless of their processing on a pipeline. Description of each instruction includes a mnemonic, list of operands, precondition, latency, and semantics in the imperative form. Besides instructions, *test situations* and *dependencies between instructions* are formally described. Test programs are generated automatically by combining test situations and dependencies for finite sequences of instructions.

A. Structure of a Test Program

Test program is a sequence of *test cases*. The key part of a test case is a *test action*, which is a specially prepared sequence of instructions intended to create a certain situation in microprocessor behavior (hazard, exception, and so on). A test action is prepared by *initializing instructions* and followed by a *test oracle* (sequence of instructions that checks correctness of a microprocessor state after execution of the test action). Thus structure of a test program can be described by the expression:

$$Test = \{\langle Pre_i, Action_i, Post_i\rangle\}_{i=0,n-1},$$

where Pre_i is the initializing instructions of the ith test case, $Action_i$ is the test action, and $Post_i$ is the test oracle. In elementary case a test program consists of a singular test case without a test oracle ($Test = \langle Pre, Action\rangle$).

The assembler code below is a test program fragment that contains one test case (we use MIPS ISA [8] to illustrate ideas of the approach).

```
// Initialization of sub[0]: IntegerOverflow=true
// s5[rs]=0xffffffffc1c998db, v0[rt]=0x7def4297
lui s5, 0xc1c9
ori s5, s5, 0x98db
lui v0, 0x7def
ori v0, v0, 0x4297

// Initialization of add[1]: Exception=false
// a0[rs]=0x1d922e27, a1[rt]=0x32bd66d5
...
// Initialization of div[2]: DivisionByZero=true
// a2[rs]=0x48f, a1[rt]=0x0
...

// Dependencies: div[2].rt[1]-sub[0].rd[0]

// Test action: 2010
sub a1, s5, v0 // IntegerOverflow=true
add t7, a0, s3 // Exception=false
div a2, a1     // DivisionByZero=true
```

In this fragment, *Action* is represented as a sequence of three instructions: `sub`, `add` and `div`. There is a register dependency between the `rt` operand of the `div` instruction and the `rd` operand of the `sub` instruction. This dependency implies using the same register for each of the operands. Initializing instructions *Pre* load values into the independent input registers of all instructions of the test action (see preparation of the instruction `sub`, for example). *Post* is empty here.

B. Test Templates

Test template is an abstract representation of a test action where constraints (test situations and dependencies) are specified instead of concrete instructions and their operands' values. Generally speaking, each template defines a testing purpose (situation that should be tested). The goal of test program generation is to construct a representative set of test templates. One of the possible templates for the example above is shown below.

```
IADDInstruction R, ?, ? @ IntegerOverflow=true
IADDInstruction ?, ?, ? @ Exception=false
IDIVInstruction ?, R    @ DivisionByZero=true
```

The template is composed of three instructions. The first two instructions belong to the equivalence class `IADDInstruction`, while the third one belongs to `IDIVInstruction`. For the first instruction the situation `IntegerOverflow=true` is given (instruction should throw the integer overflow exception). The situation of the second instruction is `Exception=false` (absence of exceptions). The third instruction should raise division by zero (`DivisionByZero=true`). In addition, there is a dependency between the first and the third instructions (the first register of the first instruction is equal to the second register of the third instruction). Independent operands of the instructions, i.e., operands that are not revolved by the dependency are denoted as `?`.

Test templates are allowed to be *parameterized*. The example below demonstrates a template with four parameters: `$FirstInstruction` (equivalence class of the first instruction), `$Situation` (test situation of the first instruction), `$ThirdInstruction` (equivalence class of the third instruction), and `$Dependency` (dependency of the third instruction on the first and the second instructions).

```
$FirstInstruction @ $Situation
IADDInstruction   @ IntegerOverflow=false
$ThirdInstruction @ $Dependency
```

C. Test Situations

Speaking about control logic verification, test situations related to execution of instructions on a pipeline are of interest. As a rule, processing of an instruction is performed in the same way for all values of the operands (of course, if exceptions are not taken into account). Thereby, for an instruction that can raise N exceptions, N+1 test situations are usually defined: *Exception=false*, $Exception_0=true$, …, $Exception_{N-1}=true$. For an instruction which handling depends on the operands values one should specify all possible paths of its execution.

Branch instructions are examined in a particular way. A test situation of a branch instruction includes a target address and truth values of the condition (if the branch instruction is conditional). In general case, a test situation is as follows: *Target=Label*, $Trace=\{C_0, \ldots, C_{M-1}\}$, where *Label* is a target label (address) and C_i is a truth value of the branch condition for the ith time of the instruction execution.

D. Dependencies between Instructions

Dependencies between instructions are thought to have a key role in creation of pipeline hazards. There are two main types of dependencies: *register dependencies* and *address dependencies* (*data dependencies*). Register dependencies are expressed as equality of registers being used as operands of two instructions. Such dependencies can be of the following types:

- *read-read* — both instructions read from the same register;
- *read-write* — the first instruction reads from a register, while the second one writes into it;
- *write-read* — the first instruction writes into a register, while the second one reads from it;
- *write-write* — both instructions write into the same register.

Address dependencies have more complex structure and are related to internal organization of memory management units [9]. Some examples of the address dependencies are itemized below.

- *VAEqual* — equality of virtual addresses;
- *TLBEqual* — equality of TLB entries;
- *PAEqual* — equality of physical addresses;
- *CacheRowEqual* — equality of cache rows.

IV. The Suggested Approach

Basing on documentation analysis, a verification engineer marks out situations "interesting" from the control logic point of view (different types of pipeline hazards). For each hazard type its *generalized specification* is developed, which is a parameterized template creating the corresponding hazard situation. Such templates are also called *pipeline hazards templates* or *basic templates*. Basic templates are usually of a small size, because hazards between instructions occur if instructions are close to each other. To be able to generate tests, one should define *iterators* of templates' parameters. After this, a generator constructs test programs using different values of the parameters and combining templates together.

A. Specification of Hazards

Let us examine a general scheme of pipeline hazards specification. All of the situations derived from the documentation are classified by their types (see Fig. 1). The four main types are *exceptions*, *data hazards*, *structural hazards*, and *control hazards*.

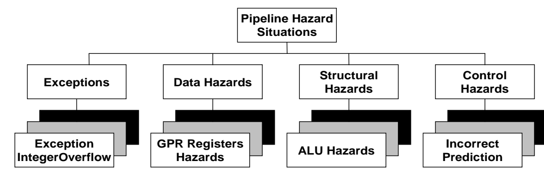


Figure 1. Classification of pipeline hazard situations

Generally, all situations of the same type are described by one basic template. The difference between two single-type specifications is connected with various constraints for template parameters – parameters domain is divided by a verification engineer into a number of equivalence classes, and this serves as a basis for the further construction of iterators (see Fig. 2).

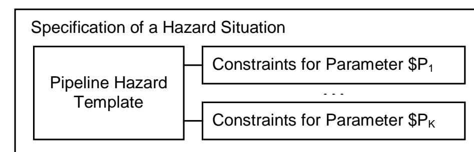


Figure 2. Specification of a hazard situation

1) Specification of Exceptions

Exception is a special event that signals that something goes wrong during instruction execution. When an exception is raised, control flow is switched to a special routine, called *exception handler*, and all the instructions loaded after the instruction throwing the exception are flushed. The typical errors related to exception handling are incorrect setting of an exception signal (incorrect calculation of an exception condition) and incorrect flushing of loaded instructions.

There are two main strategies for exception handling in test programs. First, if an exception is raised, execution is switched to the next instruction of the test action. Second, execution is switched to the test oracle passing the rest

instructions of the test action. To check pipeline flushing mechanisms, the second strategy is preferable. Generalized specification of an exception is given by the template below.

```
$PreInstructions
$ExceptionInstruction @ $ExceptionType
$PostInstructions
```

The template uses the following parameters:

- `$PreInstructions` — a sequence of instructions that precedes an exception (pre-instructions should not raise exceptions);
- `$ExceptionInstruction` — an instruction that raises an exception;
- `$ExceptionType` — an exception type;
- `$PostInstructions` — a sequence of instructions that succeeds an exception (post-instructions should be flushed).

Here is an example of a concrete test action that corresponds to the template given.

```
dadd   r25, r30, r7
lb     r22, 0(r4) // TLBInvalid=true
daddiu r5,  r18, 13457
```

The sequence `$PreInstructions` consists of the only instruction `dadd`. `$ExceptionInstruction` is instantiated by the instruction `lb` that raises the exception `TLBInvalid` (`$ExceptionType`). The sequence `$PostInstructions` includes the only instruction `addiu`.

2) *Specification of Data Hazards*

Data hazards are situations in which different instructions try to access the same data and at least one instruction tries to write them. Thereby, to describe data hazards, one should use “read-write”, “write-read”, and “write-write” types of dependencies. The typical error related to data hazard resolution is incorrect implementation of pipeline interlocks resulting in data flow integrity violation. Generalized specification of a data hazard is given by the template below.

```
$PreInstructions
$FirstInstruction
$InnerInstructions
$SecondInstruction @ $Dependency
$PostInstructions
```

The template uses the following parameters:

- `$PreInstructions` — a sequence of instructions that precedes a dependency (pre-instructions should not raise exceptions);
- `$FirstInstruction` and `$SecondInstruction` — a pair of dependent instructions that causes a data hazard;
- `$Dependency` — a dependency between instructions that causes a data hazard;
- `$InnerInstructions` — a sequence of instructions between dependent instructions (inner-instructions should not raise exceptions and produce hazards);
- `$PostInstructions` — a sequence of instructions that succeeds a data hazard (post-instructions are usually suspended with the dependent instruction).

Here is an example of a concrete test action that corresponds to the template given.

```
madd.s   $f18, $f6,  $f28, $f10
add.s    $f8,  $f17, $f3
ceil.l.s $f2,  $f18  // Data hazard
div.s    $f23, $f13, $f24
```

3) *Specification of Structural Hazards*

Structural hazards occur when several instructions try to access the same unit of a microprocessor (or some other resource). Usually, such kinds of hazards happen when two similar multi-cycle instructions are located closely to each other. In some cases an additional data dependency is required to create a structural hazard between instructions. The typical error related to structural hazard resolution is the same as for data hazards (incorrect implementation of pipeline interlocks). Generalized specification of a structural hazard is absolutely the same. A concrete example is given below.

```
div.s $f11, $f27, $f3
add.s $f28, $f7,  $f30
div.d $f23, $f1,  $f20 // Structural hazard
add.d $f18, $f2,  $f25
```

4) *Specification of Control Hazards*

Control hazards are related to branch instructions. Depending on microprocessor organization, execution of a branch instruction can result in pipeline *stalling* or *flushing*. Errors of control hazard resolution relate to pipeline interlocks, branch prediction and other control logic mechanisms. Generalized specification of a control is given by the template below.

```
$PreInstructions
$BranchInstruction @ $Target, $Trace
$DelaySlots
$PostInstructions
```

The template uses the following parameters:

- `$PreInstructions` — a sequence of instructions that precedes a branch instruction;
- `$BranchInstruction` — a branch instruction;
- `$Target` — a target address of a branch instruction;
- `$Trace` — an execution trace of a branch execution (sequence of truth values of a branch condition);
- `$DelaySlots` — instructions in delay slots;
- `$PostInstructions` — a sequence of instructions that succeeds a branch instruction.

Here is an example of a concrete test action that corresponds to the template given.

```
L:addi r1, r1,  1
  beq  r1, r0,  L // Target=L, Trace={1, 0}
  dadd r7, r12, r23
```

B. *Test Programs Generation*

Let us consider how test programs are generated on the base of test templates. Test actions are divided into two types: *simple test actions* (which correspond to a single basic template) and *composite test actions* (which are constructed by composition of several basic templates).

1) *Constructing Simple Test Actions*

Simple test actions are targeted at creation of one hazard situation. A technique for their construction is easy and based on using *basic templates* and *iterators*. For each situation derived from documentation, a set of test actions is built. Test actions are constructed by iterating parameters

values and combining them to each other (commonly, all possible combinations are used) (see Fig. 3).

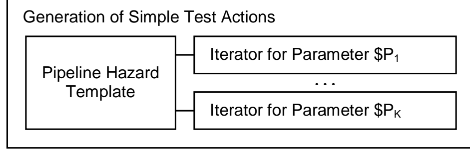


Figure 3. Generation of simple test actions for a pipeline hazard

Let us examine simple test action construction for a structural hazard on FPU (Floating Point Unit) being described by the following basic template:

```
$PreInstructions
$FirstInstruction
$InnerInstructions
$SecondInstruction @ $Dependency
$PostInstructions
```

Some constraints on parameters values are defined for this hazard. For example, the hazard occurs only between instructions being executed more than one cycle. It is also obvious that size of `$InnerInstructions` should not exceed latency of `$FirstInstruction` subtracted by two. In addition, equivalence classes of dependent instructions can be given. Assume that the template's parameters are setup with the following values.

```
FMULInstruction    : {mul.s, mul.d}
FDIVInstruction    : {div.s, div.d}
IADDInstruction    : {add, sub}

$PreInstructions   : {}
$FirstInstruction  : FMULInstruction, FDIVInstruction
$SecondInstruction : FMULInstruction, FDIVInstruction
$Dependency        : class($FirstInstruction) ==
                     class($SecondInstruction)
$InnerInstruction  : {IADDInstruction}
$PostInstructions  : {}
```

Given the parameters values, two test actions are generated:

```
$PreInstructions   →
$FirstInstruction  →  mul.d $f12, $f3,  $f21
$InnerInstructions →  sub   r6,   r15,  r3
$SecondInstruction →  mul.s $f9,  $f23, $f7
$PostInstructions  →

$PreInstructions   →
$FirstInstruction  →  div.s $f18, $f28, $f4
$InnerInstructions →  add   r25,  r13,  r27
$SecondInstruction →  div.s $f5,  $f12, $f10
$PostInstructions  →
```

2) Constructing Composite Test Actions

The aim of composite test actions as opposed to simple test actions is creation of several "simultaneous" pipeline hazards. Composite actions allow testing complex situations in microprocessor behavior (*parallel hazards*, *nested hazards*, *parallel exceptions*, and so on). Construction of composite actions is performed by composition of several basic templates.

Let T be a template of an arbitrary type, T_E be a template of an exception, T_H be a template of a data or structural hazard, and, finally, T_C be a control hazard template. The main composition operations are given below.

a) Overlapping: $T=T_{H1}|T_{H2}$

$T_H.PreInstructions = T_{H1}.PreInstructions \equiv T_{H2}.PreInstructions$
$T_H.FirstInstruction = T_{H1}.FirstInstruction \equiv T_{H2}.FirstInstruction$
$T_H.SecondInstruction = T_{H1}.SecondInstruction \equiv T_{H2}.SecondInstruction$
$T_H.Dependency = T_{H1}.Dependency \ \& \ T_{H2}.Dependency$
$T_H.InnerInstructions = T_{H1}.InnerInstructions \equiv T_{H2}.InnerInstructions$
$T_H.PostInstructions = T_{H1}.PostInstructions \equiv T_{H2}.PostInstructions$

b) Shift: $T_H=T_{H1}\downarrow T_{H2}$

$T_H.PreInstructions = T_{H1}.PreInstructions$
$T_H.FirstInstruction = T_{H1}.FirstInstruction$
$T_H.SecondInstruction = T_{H1}.SecondInstruction$
$T_H.Dependency = T_{H1}.Dependency \ \& \ T_{H2}.Dependency$
$T_H.InnerInstructions = \{T_{H1}.InnerInstructions, T_{H2}.FirstInstruction, T_{H2}.InnerInstructions\}$
$T_H.PostInstructions = \{T_{H1}.PostInstructions, T_{H2}.SecondInstruction, T_{H2}.PostInstructions\}$
$T_{H1}.PostInstructions \equiv T_{H2}.PreInstructions$

c) Concatenation: $T=T_1\rightarrow T_2$

$T.PreInstructions = T_1.PreInstructions$
$T.MainParameters = T_1.MainParameters$[2]
$T.PostInstructions = T_2$
Type of a template T matches with the type of a template T1

d) Nesting: $T_H=T_{H1}[T]$

$T_H.FirstInstruction = T_{H1}.FirstInstruction$
$T_H.SecondInstruction = T_{H1}.SecondInstruction$
$T_H.Dependency = T_{H1}.Dependency$
$T_H.PreInstructions = T_{H1}.PreInstructions$
$T_H.InnerInstructions = T$
$T_H.PostInstructions = T_{H1}.PostInstructions$

To clearify the semantics of composite templates, let us consider an example where the overlapping is used to compose a data hazard and a structural hazard:

```
$PreInstructions1,2                 → add.s  $f28, $f7,  $f30
$FirstInstruction1,2                → div.s  $f11, $f27, $f3
$InnerInstructions1,2               → dsub   r25,  r30,  r7
$SecondInstruction1,2
  @ $Dependency1 & $Dependency2     → div.d  $f23, $f11, $f20
$PostInstruction1,2                 → lb     r22,  0(r4)
```

It is intuitively obvious how to iterate test actions for a given composite test template. Some parameters of different basic test templates are identified. After this, iterators are specified for resultant parameters (commonly, iterators developed for simple templates are used). Generation of composite test templates is perform by enumeration of different syntactical structures consisting of a small number of basic templates connection by composition operations.

V. CASE STUDY

The suggested approach was applied to verification of control logic of two arithmetical coprocessors, floating point coprocessor (CP1) and complex arithmetic coprocessor (CP2). Coprocessors have common control flow with CPU and use three execution channels (functional pipelines):

- channel of floating point arithmetic;

[2] *MainParameters* is set of template parameters excluding *PreInstructions and PostInstructions*.

- channel of RAM operations;
- channel of on-chip memory operations.

The control logic supports solving of different types of hazards. In both coprocessors memory exceptions can occur. In addition, in CP1 arithmetic exceptions can be raised. The CPU implements static branch prediction and speculative execution mechanisms.

The test actions were composed of four instructions of different types. Structure of the test situations and dependencies was very much the same as it is described in the paper, but some particular features of the microarchitecture were taken into account and additional data dependencies were included.

TABLE I. CASE STUDY INFORMATION

CPU revision	Code volume (LOC)	Number of instructions	Affected instructions	Affected code (LOC)
Coprocessor CP1				
8	28500	113	—	—
20	28650	114	94	485 (1.7%)
Coprocessor CP2				
8	4950	15	—	—
20	11550	59	5	45 (0.9%)
29	14350	100	18	165 (1.4%)

Table I shows code volume (including ISA specifications and pipeline hazards templates), number of implemented instructions, number of instructions affected by the revision and volume of the affected code. As it is seen in the table, when the microprocessor is modified, a small part of the generated code has to be changed. It took us less than half an hour to alter the code.

The generated test programs detected a considerable number of errors in both coprocessors which had not been found by randomly generated test programs.

VI. CONCLUSION

Verification of a pipeline microprocessor is a very difficult task that cannot be carried out without using automation techniques. In the paper the technique for automated test programs generation is described. As opposed to the common approaches, like manual development and random generation, the suggested methodology has a high level of automation and allows systematically testing control logic of a microprocessor. At the same time, the methodology differs from the approaches that use accurate models by the possibility of using it in early design stages when microarchitecture is frequently revised.

Using accurate models of control logic is reasonable in late stages of a microprocessor design cycle when control logic is stable. Due to the high informativeness of accurate models, such approaches allow finding errors which are really hard to detect. Moreover, accurate models make it possible to create more compact set of tests. In the future we are planning to extend the approach to support accurate models as well. This would make the generator to be more flexible – it would be applicable to both early and late design stages unifying the verification process.

REFERENCES

[1] P. Mishra, N. Dutt. *Specification-Driven Directed Test Generation for Validation of Pipelined Processors*. ACM Transactions on Design Automation of Electronic Systems, 2008.

[2] P. Grun, A. Halambi, A. Khare, V. Ganesh, N. Dutt, A. Nicolau. *EXPRESSION: An ADL for System Level Design Exploration*. Technical Report 1998-29, University of California, Irvine, 1998.

[3] www.cs.cmu.edu/~modelcheck/smv.html.

[4] T.N. Dang, A. Roychoudhury, T. Mitra, P. Mishra. *Generating Test Programs to Cover Pipeline Interactions*. Design Automation Conference, 2009.

[5] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, A. Ziv. *Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification*. Design and Test of Computers, 2004.

[6] S. Ur, Y. Yadin. *Micro-Architecture Coverage Directed Generation of Test Programs*. Design Automation Conference, 1999.

[7] A. Kamkin. *Test Program Generation for Microprocessors*. Institute for System Programming of RAS, 2008. (in Russian)

[8] *MIPS64^{TM} Architecture For Programmers. Revision 2.0.* MIPS Technologies Inc., 2003.

[9] D. Vorobyev, A. Kamkin. *Test Program Generation for Memory Management Units of Microprocessors*. Institute for System Programming of RAS, 2009. (in Russian)

Database index for approximate string matching

Alexander Korotkov
National Research Nuclear University "MEPhI"
Moscow, Russia
email: aekorotkov@gmail.com

***Abstract* — In this work the database index for approximate string search is proposed. In particular the task of finding strings from some data domain which have a distance from given string less than given number is considered. Some kind of editorial string distance is used in capacity of string distance. Some subclass of regular expressions is used in the capacity of tree node predicates. The analysis of performance tests was performed and the areas of further researches were surveyed.**

***Index Terms* — database systems, approximate string searching, search tree, dictionary search**

I. Introduction

Traditional database queries support a limited number of predicates. The limitation of standard search queries is caused in part by lack of implementation of required data types and search predicates in DBMS. This limitation is also caused by finite data structures on which database indexes are based. Search predicate support by database index frequently is required in order for a database to be scalable in terms of data amount incensement. The traditional search query predicates which are supported by database indexes are equality and linear range predicates [1].

Modern database applications tend to extend their functionality. Part of this functionality can be implemented without extension of DBMS, but the other part is not. Modern database applications frequently require the support of nonstandard data types and nonstandard search query predicated from a DBMS. An example of an application of nonstandard data types and nonstandard search query predicates is geographical informational systems (GIS). In these systems the geometrical data types which are nonstandard in DBMS are used. Also, the search predicates like overlap and containment are non-standard on DBMS. The spatial indexes are used for search optimization of these predicates [2].

This work considers the implementation of a database index for approximate stirng searching. The search predicate is based on the editorial distance between strings. The database index was implemented as an extension of GiST, which is a universal framework for database index implementation.

II. Approximate string search

An approximate string search is implied as a string search when search pattern or search domain can suffer from some kind of distortion. Some examples of approximate string search are finding DNA subsequences after possible mutations [3, 4, 5] and searching for typing and spelling errors in text [6, 7, 8].

In this work the searching data domain is the set S of strings s_i; $S = \{s_1, s_2, \dots , s_n\}$. The search predicate is the assertion that the editorial distance from the element of domain s_i to the search string p is less or equal than fixed number d, i.e. $ed(s_i, p) \le d$. The editorial distance between string s_1 and string s_2 is the minimal number of editorial actions required to transform s1 to s_2. In this work the Levenshtein distance [9] is used in the capacity of editorial distance. In the Levenshtein distance there are three editorial actions: character insertion, character deletion, and character replacement. This type of search predicate can be applied to search for a misspelled word in the dictionary. However, the application of this search predicate is not limited by described case.

Various implementations of database indexes for approximate string search in this definition already exist [10]. In this work the search index implementation based on a generalized search tree is presented. The pg_trgm module which is an implementation of approximate string search indexes already exists for GiST [11]. However in the pg_trgm the amount of matching trigrams is used in the capacity of string distance whereas in this work the Levenshtein distance is used.

III. Generalized search tree

Generalized search tree presents a very general solution of the generalization of database access methods. GiST is the data structure which is extensible in terms of search queries as well as in terms of indexing data types. GiST defines the set of interface functions, for which implementation defines search indexes. These interface functions only depend on indexing data type and search predicates, but these functions are abstracted from data pages, records, query processing, etc. Thus to implement a search index using GiST it is not

required to write a code which maintains data structure [12]. Additionally, GiST generalizes the majority of currently existing search trees. For example B+-tree and R-tree can be implemented as GiST extensions [13].

At this moment GiST is fully implemented in open source postrelational DBMS PostgreSQL, though the result of GiST researches is used in the majority of commercial DBMSs such as Oracle and DB2. Several reasons for implementation of search index based on GiST in this work can be noted:

- To provide open source and license free solutions for approximate string searching
- To see completely new application of GiST
- Simplicity of GiST extension implementation

IV. USING GIST FOR APPROXIMATE STRING SEARCH

As it was noted before the Levenshtein distance is used in the capacity of string distance. The Levenshtein distance is the minimum number of elementary operations needed to transform one string to another one. There are the following elementary operations:

- Insertion of arbitrary character to arbitrary position of string
- Replacement of arbitrary character of string with another arbitrary character
- Deletion of arbitrary character of string

The two sequences alignment algorithm [14, 15] can be used in order to calculate the distance between strings **a** and **b**. The two modifications of this algorithm were introduced in this work. Next let's consider this algorithm in detail.

The $\mathbf{a} = a_1a_2\ldots a_n$ and $\mathbf{b} = b_1b_2\ldots b_m$ are two strings of length n and m. The alignment is produced when a null character «-» is inserted into the strings; the new strings must have the same length L. After insertion of «–» the $\mathbf{a} = a_1a_2\ldots a_n$ becomes $\mathbf{a}^* = a_1^*a_2^*\ldots a_n^*$ and $\mathbf{b} = b_1b_2\ldots b_m$ becomes $\mathbf{b}^* = b_1^*b_2^*\ldots b_m^*$. The alignment is the two sequences which are written one over the other.

$$\begin{matrix} a_1^* & a_2^* & \ldots & a_L^* \\ b_1^* & b_2^* & \ldots & b_L^* \end{matrix}$$

The distance between strings **a** and **b** is introduced as:

$$D(\mathbf{a},\mathbf{b}) = \min \sum_{i=1}^{L} d(a_i^*, b_i^*)$$

The d(a,b) represents the distance between characters a and b. In the case of Levenshtein distance d(a,b) is defined below:

$$d(a,-) = d(-,a) = 1$$

$$d(a,b) = \begin{cases} 0, a = b \\ 1, a \neq b \end{cases}$$

The matrix D is introduced as the distance between prefixes of strings **a** and **b**.

$$D_{i,j} = D(a_1a_2\ldots a_i, b_1b_2\ldots b_j)$$

There are following rules of matrix filling:

$$D_{0,0} = 0$$

$$D_{0,j} = \sum_{k=1}^{j} d(-,b_k) \quad D_{i,0} = \sum_{k=1}^{i} d(a_k,-)$$

$$D_{i,j} = \min\{D_{i-1,j} + d(a_i,-), D_{i-1,j-1} + d(a_i,b_j), D_{i,j-1} + d(-,b_j)\}$$

TABLE 1. THE MATRIX OF ALIGNMENT

	–	b_1	b_2	…	b_m
–	$D_{0,0}$	$D_{0,1}$	$D_{0,2}$	…	$D_{0,m}$
a_1	$D_{1,0}$	$D_{1,1}$	$D_{1,2}$	…	$D_{1,m}$
a_2	$D_{2,0}$	$D_{2,1}$	$D_{2,2}$	…	$D_{2,m}$
…	…	…	…	…	…
a_n	$D_{n,0}$	$D_{n,1}$	$D_{n,2}$	…	$D_{n,m}$

The bottom right element of the matrix represents the distance between strings.

$$D_{n,m} = D(a_1a_2\ldots an, b_1b_2\ldots bm) = D(\mathbf{a},\mathbf{b})$$

A. *Search predicate*

In this work the optimizable search predicate is P(x) = (levenshtein (s,x) ≤ d), where levenshtein – the function of Levenshtein distance calculation, s – given string, d – given nonnegative number. Thus this predicate is true for strings which have Levenshtein distance to a given string less than or equal to a given number. Considering Levenshtein distance as metrics (it is possible because this distance has metrics properties) set of strings satisfying this predicate can be represented as solid sphere with center in s string and d radius.

B. *Tree node predicate*

The selection of tree node predicate is critical for GiST extension implementation. All the characteristics of the resulting tree generally depend on selected tree node predicate. In this work the predicate of matching to some class of regular expressions was selected. The description of a selected class of regular expressions is below. Each expression of a selected class can be represented as a concatenation of n (n is nonnegative integer) sub-expression. Each sub-expression can be defined in one of the ways below:

- One character from set of m characters (format of sub-expression is "[$a_1a_2\ldots a_m$]")
- One character from set of m characters or empty string (format of sub-expression is "[$a_1a_2\ldots a_m$]?")
- Any character or empty string (format of sub-expression is ".?")

In this work when the term "regular expression" is used this class of regular expressions is mentioned.

C. *GiST interface methods implementation*

The GiST interface consists of 7 methods. The purpose of these methods is considered below.

1) compress and decompress – these two methods are responsible for key compression and decompression (in keys should be suitable to work with them but it is frequently reasonable to compress a key before storing it to disc)
2) consistent – this method calculates compatibility of

tree node key and search query (The search optimization performs at the expense of this method. If the predicate of the tree node is incompatible with the search predicate then all the sub-tree should be skipped)

3) union – this method returns the union of two keys (all the values which conform to any of source keys should conform to the resulting key)
4) penalty – this method returns the measure of growth of the source key after addition of another key to it (this value should represent the measure of growth of values set which conforms to the key predicate)
5) picksplit – this method splits an array of keys into two arrays. It is desirable that union keys of the resulting two arrays have a minimal size (the size of the key is assumed to be the size of set of values which conform to the key predicate)
6) same – this methods checks if two keys are the same

In this work the compression of keys before writing them to the disc is not used. This is why the implementation of the compress and decompress methods was trivial. The implementation of same method also was trivial because all the regular expressions are stored in same manner. The penalty and picksplit methods were implemented using the keys union function and key size measurement function. The penalty method calculates keys union and calculates the difference between keys union size and source key size. The picksplit method is based on the Guttmann's clusterization algorithm. The union and consistent methods use the modification of two strings alignment algorithm.

1) Consistent method

The consistent method implementation uses the modification of two strings alignment algorithm[14] which makes it possible to find the minimal Levenshtein distance between any string which conforms to regular expression and the search query string. The resulting minimal distance can be represented by the expression:

$$d = \min\{\text{levenstein}(\mathbf{s},\mathbf{x}) | \mathbf{x} \sim \mathbf{r}\}$$

where **s** – search query string, **r** – regular expression, "~" – operator of regular expression conformance.

The decision on compatibility of search query and regular expression is made by comparing the resulting value and maximum distance of the search query.

The modification of two strings alignment algorithm is used in calculations of minimal distance. In this modification alignment between $\mathbf{s} = s_1 s_2 \ldots s_n$ and $\mathbf{r} = r_1 r_2 \ldots r_m$ is produced. The minimal distance between a string which conforms to **r** and **s** is calculated by expression:

$$D(\mathbf{s},\mathbf{r}) = \min \sum_{i=1}^{L} d(\mathbf{s}_i^*, \mathbf{r}_i^*)$$

There is following definition d(s,r):

$$d(s,r) = \begin{cases} 0, \text{if r allows s} \\ 1, \text{if r doesn't allow s} \end{cases}$$

$$d(s,-) = 1$$

$$d(-,r) = \begin{cases} 0, \text{if r allows empty string} \\ 1, \text{if r doesn't allow empty string} \end{cases}$$

TABLE 2. THE EXAMPLE OF ALIGNMENT MATRIX FOR FINDING MINIMAL DISTANCE BETWEEN STRING AND REGULAR EXPRESSION

	-	[dk]	[uzm]	[oc]	.?
-	0	1	2	2	2
d	1	0	1	1	1
o	2	1	1	1	1
m	3	2	1	1	1

Other parts of this algorithm is similar to the original algorithm. Let's consider an example. Let's find minimal distance between "dom" word and "[dk][uzm][oc]?.?" expression.

2) Union method

In the union method some other modification of the two strings alignment algorithm was used. The following distance function between two sub-expressions was used:

$$d(r_1, r_2) = \frac{u_1}{c + u_2} + \frac{u_2}{c + u_1},$$

There u_1 – the number of unique characters in the first sub-expression (the number of characters which are allowed by the first sub-expression and are not allowed by the second sub-expression), u_2 – the number of unique characters in the second sub-expression and c – number of common characters in sub-expressions. The empty string is assumed to be a separate character.

The case of equality of one sub-expression to ".?" should be considered separately (when the both sub-expressions are equal to ".?", it is evident that distance should be assumed as zero). In this case following measure was used:

$$d(".?", ".?") = 0$$

$$d(".?", r_2) = \frac{n - c_2}{n}$$

There c_2 – the number of characters of second sub-expressions and n – the total number of characters in the alphabet used.

In the case of one sub-expression being skipped, the following measure was used:

$$d(r,-) = d(-,r) = 1 + \frac{u}{u + c},$$

There u = 0, when an empty string was allowed by sub-expression, u = 1, otherwise; c – the number of characters in sub-expression.

In this modification of alignment it is not only necessary to calculate the distance but also to find the union expression. Let's consider alignment of two expressions $\mathbf{a} = a_1 a_2 \ldots a_n$ and $\mathbf{b} = b_1 b_2 \ldots b_m$.

$$\begin{matrix} a_1^* & a_2^* & \ldots & a_L^* \\ b_1^* & b_2^* & \ldots & b_L^* \end{matrix}$$

The resulting expression $\mathbf{c} = c_1 c_2 \ldots c_m$ can be calculated by $c_i = u(a_i, b_i)$, where u – the function of two subexpressions

unification.

$$u(".?", a) = u(a, ".?") = ".?"$$

$$u("[a_1a_2\ldots a_n]","[b_1b_2\ldots b_m]") = "[a_1a_2\ldots a_nb_1b_2\ldots b_m]"$$

$$u("[a_1a_2\ldots a_n]","[b_1b_2\ldots b_m]?") = u("[a_1a_2\ldots a_n]?","[b_1b_2\ldots b_m]") = u("[a_1a_2\ldots a_n]?","[b_1b_2\ldots b_m]?") = "[a_1a_2\ldots a_nb_1b_2\ldots b_m]?"$$

In the operation of unification of sub-expressions if the number of characters in the final sub-expression exceed the threshold value k then this sub-expression is replaced by “.?”. This replacement is performed in order to decrease the length of sub-expression and to improve the performance.

Let’s consider the process of unification of “[abc][def][hg]?.?” and “.?[ad][bef]?h?h?” expressions as an example. The final matrix is presented below.

TABLE 3. THE EXAMPLE OF ALIGNMENT MATRIX UNIFICATION OF TWO REGULAR EXPRESSION

	-	.?	[ad]	[bef]?	h?	h?
-	0,00	1,00	2,33	3,33	4,33	5,33
[abc]	1,25	0,88	1,88	2,88	3,88	4,88
[def]	2,50	2,13	1,75	2,75	3,75	4,75
[hg]?	3,50	3,13	2,75	2,63	3,63	4,63
.?	4,50	3,50	3,75	3,59	3,53	4,53

The resulting alignment is

expression 1	[abc]	[def]	[hg]	.?	–
expression 2	.?	[ad]	[bef]?	h?	h?
union	.?	[adef]	[befhg]	.?	h?

The union expression is “.?[adef][befhg]?.?h?”.

V. THE PERFORMANCE TESTING

Two tasks should be completed in order to perform synthetic testing of a database index. These tasks are to prepare test data domain and to prepare the test set of queries. The English dictionary with a volume of 61 505 words was used as test date domain.

After that the test was generated. There are two kinds of generated tests. The first kind of test is with random generated words. The second kind of test is with random distortion in existing words.

In the tests with random generated words the sequence of random characters of English alphabet with a length between 3 and 18 was generated. After that a random number between 1 and $\lceil n/5 \rceil$ was generated. This number was used as the radius of the search query. The expression $\lceil n/5 \rceil$ was used as the upper boundary in order to prevent the radius of the search query from being too high in comparison with word length.

In the tests with random distortion in existing words the random word from the dictionary was selected. Let’s assume the length of this word as n. After that the random distortions (insertion, replacement and deletion of character) with number between 1 and $\lceil n/5 \rceil$ was applied to the selected word. Eventually the random number between 1 and $\lceil n/5 \rceil$ is selected as the radius of search query.

The results of the tests are presented in the tables.

TABLE 4. THE RESULTS OF INDEX TESTING WITH RANDOM DISTORTIONS IN EXISTING WORDS

	Search query radius									Average		
	1			2			3					
Dist.	S	WOI	WI	S	WOI	WI	S	WOI	WI	S	WOI	WI
0	2,29	124	66	1,36	141	115	1,529	186	132	1,72	150	104
1	2,91	124	58	1,54	143	108	1,751	180	126	2,07	149	98
2	3,52	142	63	1,62	142	102	1,851	184	121	2,38	156	95
3	10,7	180	36	6,66	187	063	2,549	182	116	6,64	183	72
Avg.	4,85	142	56	2,80	153	097	1,920	183	124	**3,19**	**160**	**92**

In the table 4 the results of testing of search queries with random distortion in existing words. The dependence of average speedup (S), average search time without index (NI) and average search time with index(I) on radius of search query and number of distortion in source word is presented. The speedup (S) calculates as $S = T_{woi} / T_i$, where T_{woi} and T_i are the time of search without using of index and the time of search with using of index respectively. As it is shown in the table the quotient of NI and WI is less then S as the rule. There is no contradiction because the average of quotient is not the quotient of average. This argues that the faster queries have higher speedup than slower ones.

In the table 5 the results of testing of search queries with random generated words are shown. In this table the same data as in the table above is presented but it depends on the length of generated word and radius of search query. As it is shown in the table the speedup increases as the length of generated word increases, and speedup decreases as the search radius increases.

TABLE 5. THE RESULTS OF INDEX TESTING WITH RANDOM WORDS

	Search query radius												Average		
	1			2			3			4					
Len.	S	NI	I	S	NI	I	S	NI	I	S	NI	I	S	NI	I
3	5,3	80	16										5,3	80	16
4	4,5	89	24										4,5	89	24
5	4,6	99	29										4,5	99	29
6	4,6	109	33	1,8	109	68							3,2	109	51
7	5,7	119	30	2,2	119	71							4,0	119	50
8	6,6	128	27	2,7	128	63							4,7	128	45
9	8,4	139	22	3,2	138	58							5,8	138	40
10	11	148	16	6,1	148	35							8,5	148	26
11	12	157	14	7,3	156	28	3,4	157	58				7,7	157	33
12	16	166	11	9,8	167	20	5,4	167	47				10,5	166	26
13	21	175	9,4	13,2	174	14	8,5	175	25				14,1	175	16
14	28	183	7,5	15,4	184	13	9,3	184	23				17,6	184	14
15	49	193	4,9	20,3	192	10	12,7	194	17				27,4	193	11
16	88	201	2,8	32,3	202	7,4	16,1	200	13	10,58	202	20	36,7	201	11
17	201	211	1,6	63,6	210	4,4	25,5	209	10	13,74	211	16	76,1	210	7,9
18	353	220	0,8	116	218	2,2	43,6	218	5,8	17,90	220	13	132	219	5,4
Avg.	51,3	151	16	22,6	165	30	15,6	188	25	14,07	211	16	**25,9**	**179**	**22**

VI. CONSLUSION

In this work the development of a new search index for the approximate string search based in GiST was considered. The new search index which allows searching in the domain of

strings $S = (s_1, s_2, \dots s_n)$ such s_i that levenshtein(si, p) $\leq$ d was developed. The index testing with the English dictionary with the volume of 61 505 words in the capacity of a data domain was performed. The average speedup in the tests with random distortions in existing words was 3.19 times. The average speedup in the tests with random words was 25.88 times.

There are following directions of further researches:

- To research the developed index behavior on the different data domains. To understand which domain can be used with considerable performance improvement and which is not.
- To improve the performance of the index. There are two ways to improve the performance. The first way is to change the implementation of some GiST interface methods (in particular the PickSplit method). The second way is to change the class of regular expressions used in the capacity of tree node predicates.
- To apply the developed index for other search predicates. These predicates are the following: predicate based on the editorial distance different than Levenshtein distance, the regular expressions in the capacity of search predicate.

REFERENCES

[1] Douglas Comer, “The Ubiquitous B-Tree”, Computing Surveys 11(2), June 1979, pp. 121–137.

[2] Antonin Guttman, “R-Trees: A Dynamic Index Structure For Spatial Searching”, In Proc. ACM SIGMOD International Conference on Management of Data, June 1984, pp. 47–57.

[3] Miller , and DJ Lipman, “Gapped BLAST and PSI-BLAST: a new generation of protein database search programs”, Nucleic Acids Res., Sep. 1997, pp. 3389-3402.

[4] Zemin Ning, Anthony J. Cox, and James C. Mullikin, “METHODS: SSAHA: A Fast Search Method for Large DNA Databases”, Genome Res., Oct. 2001, pp. 1725-1729.

[5] Maria B. Chaley, Eugene V. Korotkov, and Konstantin G. Skryabin, “Method Revealing Latent Periodicity of the Nucleotide Sequences Modified for a Case of Small Samples”, DNA Res, June 1999, pp. 153-163.

[6] Sun Wu, Udi Manber, “Fast text searching: allowing errors”, Communications of the ACM, Oct. 1992, pp. 83 – 91.

[7] James L. Peterson, “Computer programs for detecting and correcting spelling errors”, Communications of the ACM archive, Dec. 1980, pp. 676 – 687.

[8] Fred J. Damerau, “A technique for computer detection and correction of spell-ing errors”, March 1964, pp. 171 – 176.

[9] V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals”, Soviet Physics 10, 1966, pp. 707–710.

[10] G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio, “Indexing methods for approximate string matching”, IEEE Data Engineering Bulletin 24, April 2001, pp. 19-27.

[11] R. C. Angell, G. E. Freund, P. Willett, “Automatic spelling correction using a trigram similarity", Information Processing and Management 19(4), 1983, pp. 255-262.

[12] M. Kornacker, C. Mohan, J.M. Hellerstein, “Concurrency and recovery in generalized search trees”, In Proceedings of the ACM-SIGMOD Conference, May 1997, pp. 62-72.

[13] Paul M. Aoki, “Generalizing “search” in generalized search trees” in Proc. 14th Int'l Conf. on Data Engineering, Feb. 1998.

[14] Robert A. Wagner, Michael J. Fischer, “The String-to-String Correction Problem”, Journal of the ACM, Jan. 1974, pp. 168 – 173.

[15] Michael S. Waterman, “Introduction to computational biology: maps, sequences and genomes”, 1995.

Adaptation of Hierarchical clustering by areas for automatic construction of electronic catalogue

Fedor Vladimirovich Borisuyk
N.I. Lobachevsky State University of Nizhny Novgorod
Nizhny Novgorod, Russia
fedorvb@gmail.com

Vladimir Ivanovich Shvetsov
N.I. Lobachevsky State University of Nizhny Novgorod
Nizhny Novgorod, Russia
shvetsov@unn.ru

Abstract

This paper explores adaptation of Hierarchical clustering by areas algorithm, presented in [1], for automatic construction of electronic catalogue of text documents. Electronic catalogue is traditionally maintained and built by hand, significant research was done to classify the documents to existing hierarchical structures, but little was done about to build the electronic catalogue from scratch. We describe enhancements of Hierarchical clustering by areas algorithm and compare it (to be applied in the field of automatic catalogue construction) with other researches. This paper also proposes an algorithm of feature space reduction for preparation of document representations for text clustering.

Keywords - electronic catalogue, rubrication, informational retrieval, text clustering, area tree

I. INTRODUCTION

At present in different stores of knowledge (electronic and traditional) have accumulated vast amount of information. At the same time because of large volumes of information, their weak structuring and reporting non-electronically, the obtaining of relevant and complete information on a specific topic is quite complex, and majority of the accumulated information resources becomes useless because of it immense. It can be noted that the search of solution for a specific scientific objectives requires high labor costs to find and analyze information on the topic. Therefore, in connection with the stated above, there is the problem of efficient structuring, storing, processing and retrieval of information in the data arrays. To solve this problem people use different thematic classifiers, rubricators, electronic catalogues, which allows finding (either automatically or manually) the documents in a small subset of the document database corresponding to the interesting for the user topic. Electronic catalogues is usually a set of headings grouped into a hierarchy (directory). For each category there is set of documents, which is assigned corresponding to category subject. Currently there are available are two types rubricators - manual and automated. In case of manual rubricator, each new document must be manually analyzed and expert have to define which sections of directory of electronic catalogue it belongs to, after that document becomes available for search. Also there are automated categorization systems, which store sets of characteristics for each category of catalogue and automatically determines corresponding category for the analyzed document. At most the list of attributes for each entry is made by the expert. The disadvantage of the existing automated systems is their static character and the inability to automatically, without the participation of expert rebuild formed earlier catalogue.

In this paper we propose the way of automatic construction of an electronic catalogue for text documents as one of the most effective ways to access the necessary data. The urgency of this problem is increasing as the number and volume of electronic texts is constantly increasing. To automatically construct electronic catalogue the adaptation of novel Hierarchical clustering by areas algorithm is presented. The proposed hierarchical structure of electronic catalogue supposes a quick and efficient way to find the relevant information.

II. RELATED WORK

Much of the previous work concentrates on the classification of text documents, either on flat classification or on hierarchical classification to predefined categories. In case of flat classification of text documents each category treated individually and equally so that no structures exist to define relationships among them [2]. For example, good work in the hierarchical classification to two layer hierarchical structure described by Dumais and Chen in [3], they used SVM classifiers and explored small advantage for accuracy for hierarchical model over flat models.

Only a little research was done in the field of automatic construction of hierarchical catalogue structures. There can be observed two qualitative articles.

Tao Li and Shenghuo Zhu [4] have used linear discriminant projection approach for transformation of document space onto lower-dimensional space and then cluster the documents into hierarchy using Hierarchical agglomerative clustering algorithm [5]. They have found good benefit from using linear projection approach as according to the paper it preserves underlying class structure relation (semantic relations between clustering objects). The paper investigates the affect of using generated hierarchical structure for text classification. They have used LIBSVM [13] as a classifier, which is library for support vector classification, regression and support multi-class

classification. Taken experiments showed that generated hierarchies improve classification performance in most cases, the most significant gain in accuracy (growth up to 53 %) they have reached on the Reuters-top10 collection.

A promising work was done by O. Peskova [6], which presents some improvements for clustering feature selection referred as selective feature space reduction and develops a modification of layerwise clustering method of Ayvazyan [7]. Suggested method of selective feature space reduction decreases feature space in 3.5 times, decreases a computation time, and improves accuracy of clustering algorithm. Presented method was tried on small collections of text documents generated from electronic library on informational technologies http://citforum.ru. Author of the [6] found a 4% advantage in average f-measure of the developed clustering method over Hierarchical agglomerative clustering algorithm [5].

Our work explores the way of automatic construction of an electronic catalogue for text documents as one of the most effective ways to access the necessary data. We propose adaptation of Hierarchical clustering by areas algorithm, which is described in [1].

In next sections we will describe clustering feature selection, description of datasets used for testing of the proposed approach, description of the adapted hierarchical clustering by areas algorithm and evaluation of the proposed approach.

III. Clustering Feature Selection

For the clustering purpose each text document is presented as the vector of keywords. Universal of document keyword vectors presents keywords feature space. To reduce the feature space stop words (words that usually does not used for search, for example, conjunctions and prepositions) are eliminated. Also modified TFxIDF [8] metric is used to calculate the weight of the word in relation to the document. Finally no more than top 300 features with the highest weight of the word in relation to the document are selected to represent the document. Hereby the following algorithm for keywords extraction from the documents is used:

1) *For all words of the document stem is extracted using Porter algorithm[9]. Number of occurances TF_i of each stem in the document D is counted.*
2) *Stop words are removed from list of extracted words.*
3) *Remove words, which have frequency more than predefined max frequency or less than predefined minimum frequency.*
4) *Weight of the $stem_i$ in the document D is calculated using modified TFxIDF formular, and if to denote - max frequency between all stems as $MaxStemFreq_D$, total number of documents in collection as TDN, number of documents where this stem occurs as DN_i, then we have these formulars to compute weight of the stem:*

$$IDF_i = \log(1+TDN/DN_i) \quad (1)$$

$$Weight_D(stem_i) = ((0.5+0.5*TF_i)/MaxStemFreq_D)*IDF_i \quad (2)$$

5) *No more than 300 stems with the highest weight are selected as keywords to represent the document (document representation).*

To improve the determinant behavior of features of document representations, we use the idea of Selective feature space reduction presented in [6]. Our implementation is different to [6]. Firstly we cluster the documents collection using modified algorithm of Hierarchical clustering by areas (see section IV.C). Each area in the resulted hierarchical area tree has vector of keywords, which describes it. If to assume areas collection similarly to the documents collection, we can execute keywords extraction algorithm described above on the areas collection to select the vector of most significant keywords of each area in relation to other areas in the areas feature space: this way we reduce the number of keywords used for describing of areas in the area tree (see section IV). Keywords, which are not present in the areas feature space, area removed from document vectors; this way we reduce document vector sizes. The quality of clustering rises two times (for Hierarchical clustering by areas algorithm) in comparison with the clustering without using this technique.

IV. Modified Hierarchical clustering by areas algorithm

For construction of the electronic catalogue we propose adaptation of Hierarchical clustering by areas algorithm, which is described in [1]. Hierarchical clustering by areas algorithm builds hierarchical tree of the areas, which consists of the documents of initial collection. Characteristics of the areas are calculated during algorithm execution. Final clusters of the algorithm are placed in the nodes of the tree. Node of the hierarchical tree contains objects, which is most closed to each other. Hierarchy of the tree reflects relations between areas.

A. *Initialization of the hierarchical by areas algorithm*

Lets we have incoming stream of the documents, which are supposed to be integrated into the hierarchy. Each document is represented by vector of the keywords. Primarily all incoming documents are put into the recycle bin area of the tree until the number of the documents exceeds a predefined limit, noted as KMax. When the number of the documents of the recycle bin area surpasses KMax, it divides into subareas. Hereby the list of root areas appears.

B. *Improvements of the hierarchical by areas algorithm*

To improve the quality of the hierarchical structure we propose the enhancement of hierarchical by areas algorithm with three additional techniques:

1) *Limitation of depth of the tree.*
2) *Recycle bin at each level.*

Traditionally electronic catalogue has limited number of layers. For example, catalogue of Yandex Corp. has up to 6 layers in depth [10]. Therefore clustering hierarchical algorithm should provide possibility to manage the number of layers in the hierarchy tree. This feature makes the catalogue

to be observable for the user and to locate necessary information easily.

To improve the quality of clustering we introduce the instance of recycle bin at each level. The idea of recycle bin is that all those documents which do not meet entry criteria of the areas of the certain level should be temporary stored in the special area on the same level. Areas on the same level should be as much different from each other as possible, and the idea of recycle bin helps in this question. When the number of objects in the recycle bin surpasses the predefined limit, it is divided and a new detached area is connected to the current level.

C. Phase of processing of incoming document flow

This section describes in detail modified Hierarchical clustering by areas algorithm.

In the capacity of data for the phase of processing there is document, which is represented by vector of keywords, and tree of areas. On the first step of algorithm there is a verification of possibility of correct insertion the document to the area tree. The possibility of correct insertion is determined by measuring of the closeness between document and areas of root level. If closeness does not exceed the dynamically calculated limit, which is defined as minimum of closeness between already processed documents, then document is temporary stored in the recycle bin of the root level. If document has closeness, which is more than predefined limit then, it is directed along the tree to the closest subareas. On the next steps of the algorithm document moves until it meets the closest area of the tree. Document is accommodated in the found area. In case if number of elements of area exceed predefined limit then area is divided into subareas. If number of subareas exceeds the predefined limit then operation of integration of subareas executes. The operation of integration of subareas consists of two basic operations:

1) Partitoining of subareas in two groups of most close to each other.

2) Aggregation of subareas under one area from the elements of the group.

For the notations of modified Hierarchical clustering by areas algorithm see Table I.

TABLE I. NOTATIONS OF THE HIERARCHICAL BY AREAS ALGORITHM

Notation	Description
KMax	Maximal number of elements in the area
RootArea	Root of the tree which points to the list of first level areas of the hierarchical tree
MinProximity	Minimum proximity for the document (to be inserted) that should be between document and node of the area tree.
Divide (area)	Operation of division of area to subareas
proximity(A,B)	Operation of calculation of the nearness between objects A and B
getChildren (area)	Operation of building the list of descendants of area
ConnectToTree (Area)	Connect new area as a child to its parent area from which it was derived (Divide operation) or to the parent of Recycle bin from which was derived.
Integrate (area)	Operation of integration of subareas of the area
RecycleBin	Recycle bin - special area, which stores declined objects.

Take a look at algorithm of insertion of the document in the tree of areas:

```
1 step.  New document Doc has been supplied.
2 step.  areaList=getChildren(RootArea);
3 step. FOR EACH area IN areaList: find area which is the
  most close to Doc.
4 step.  Verify if document can be inserted in the hierarchy:
   IF (proximity (Area, Doc) < MinimumProximity)) {
      RecycleBin.Add (Doc);
      IF (RecycleBin.size() > KMax) {
           Result = Divide (RecycleBin);
           ConnectToTree (Result);
      } End of algorithm;
   }
5 step. areaList=getChildren(Area);
  IF (areaList.size() == 0) GOTO 8 step.
 6 step. FOR EACH area IN  areaList: find area NArea
        – which is maximally close to Doc.
7 step.  IF (proximity (Area, Doc) <
             proximity (NArea, Doc))  {
        Area = NArea; GOTO 5 step;
  } ELSE {
     Area.add (Doc);
    IF (Area.size() > KMax)     {   divide (Area);
      IF (number of descendants of Area is over limit) {
                 Integrate (Area);
      }
    }
    GOTO 8 step.
  }
8 step. Update vectors of keywords of areas, which is
located on the path to the resulted area.
```

V. RESULTS

For the verification of the presented approach we have used two datasets and compare proposed algorithm with Hierarchical agglomerative clustering algorithm.

A. Datasets selection

For testing purposes we have used two datasets. One is subset of 20Newsgroups (20000 articles), which contains 2000 articles evenly divided among 20 Usenet newsgroups (http://www.cs.cmu.edu/afs/cs.cmu.edu/project/theo-20/www/data/news20.html). Second dataset NNSU8 was prepared by us from the scientific articles of Nizhny Novgorod state University, because in future the approach described in this article is supposed to be integrated in the environment of internet portal of Nizhny Novgorod State University as scientific catalogue. NNSU8 contains 1302 scientific articles in 8 scientific areas.

B. Evaluation

For evaluation of proposed algorithm of electronic catalogue construction we have used standard external metrics. In the context of clustering tasks, the terms true positives, true negatives, false positives and false negatives are used to compare the given clustering of the documents with the desired, "sample" partitioning of documents to the groups, which is given a priori. This is illustrated by the Table II below:

TABLE II. EVALUATION METRICS

For each pair of the documents D_i and D_j	D_i and D_j contain in one cluster of "sample" partitioning	D_i and D_j contain in different cluster of "sample" partitioning
D_i and D_j contain in one cluster of automatic clustering	tp (true positive)	fp (false positive)
D_i and D_j contains in different clusters of automatic clustering	fn (false negative)	tn (true negative)

We have used these metrics to evaluate the clustering [12]:

- Recall (1) is the fraction of number of correctly grouped documents in automatically generated cluster to the number of documents in "sample" cluster:

$$\text{Recall} = \frac{tp}{tp + fn} \quad (3)$$

- Precision (2) is the fraction of number of correctly grouped documents in automatically generated cluster to the number documents in generated cluster:

$$\text{Precision} = \frac{tp}{tp + fp} \quad (4)$$

- F-measure (3) that combines Precision and Recall is the harmonic mean of precision and recall:

$$F = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}. \quad (5)$$

We compared the modified Hierarchical clustering by areas algorithm with Hierarchical agglomerative clustering algorithm (Single link algorithm), implementation of which was taken from the Yooreeka project [11]. The results of computational experiments, which are presented in Table III, show good advantages of Hierarchical clustering by areas algorithm in comparison with Hierarchical agglomerative clustering by means of precision, recall and f-measure characteristics, and computation time. There is a 27% advantage in average f-measure of Hierarchical clustering by areas algorithm over Hierarchical agglomerative clustering algorithm on the NNSU8 collection and a 38% advantage in average f-measure on the 20NewsGroups collection.

TABLE III. EVALUATION OF AVERAGE METRICS OF HIERARCHICAL BY AREAS ALGORITHM IN COMPARISON WITH HIERARCHICAL AGGLOMERATIVE CLUSTERING ALGORITHM

Metric	Algorithms and datasets			
	Hierarchical by areas		**Hierarchical Agglomerative**	
	20News groups	***NNSU8***	***20News groups***	***NNSU8***
Recall	0.79	0.66	0.1	0.40
Precision	0.35	0.59	0.11	0.38
F-measure	0.48	0.6	0.1	0.33
Time (msec)	2505	2391	2896	45116

Top levels of catalogue generated by Hierarchical by areas clustering for NNSU8 collection and 20Newsgroups are presented in Table IV and Table V accordingly.

TABLE IV. TOP LEVELS OF CATALOGUE GENERATED BY THE HIERARCHICAL BY AREAS CLUSTERING FOR NNSU8

Areas	Members
1	Law, philosophy;
2	Mathematics;
3	Sociology;
4	Economics;
5	Physics
6	Biology, Chemistry;

Addition of new documents to the ready-built catalogue do not need rebuilding of the whole catalogue structure, it supposes the same insertion algorithm, which is presented in section IV.C.

TABLE V. TOP LEVELS OF CATALOGUE GENERATED BY THE HIERARCHICAL BY AREAS CLUSTERING FOR 20NEWSGROUPS

Areas	Members
1	talk.politics.mideast, talk.politics.guns, talk.politics.misc
2	comp.graphics, comp.os.ms-windows.misc
3	rec.sport.baseball, rec.sport.hockey, rec.autos, rec.motorcycles
4	sci.crypt, sci.med, sci.space
5	comp.sys.ibm.pc.hardware; comp.sys.mac.hardware
6	soc.religion.christian
7	sci.electronics; misc.forsale; comp.windows.x
8	talk.religion.misc

CONCLUSION

The research presented in this paper explores adaptation of Hierarchical clustering by areas algorithm for automatic construction of electronic catalogue of text documents. The computational experiments showed advantages of Hierarchical by areas algorithm for automatic electronic catalogue construction in comparison with Hierarchical agglomerative clustering by means of quality and time for computation. In this paper we also present algorithm of feature space reduction for preparation of document representations for clustering, which substantially increases quality of clustering. The addition of new documents to ready-built Hierarchical electronic catalogue, in other words – classification of new documents to the hierarchy, does not need rebuilding of catalogue. Presented in this article approach can be used for construction of web electronic catalogues.

REFERENCES

[1] F.V Borisyuk. and V.I. Shvetsov “New search method based on hierarchical clustering by areas of text documents,” Vestnik of N.I. Lobachevsky State University of Nizhny Novgorod, 2009, # 4, pp. 165–171.

[2] Yiming Yang and Xin Liu. “A re-examination of text categorization methods”, Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval, Berkeley, California, United States, 1999, pp. 42 – 49.

[3] S. T. Dumais and H. Chen (2000). “Hierarchical classification of web content”. Proceedings of SIGIR'00, 2000, pp. 256-263.

[4] Tao Li and Shenghuo Zhu. “Hierarcical document classification using automatically generated hierarchy”, Journal of Intelligent Information Systems, V. 29 , Issue 2, 2007, pp. 211 - 230.

[5] A.K. Jain and R.C.Dubes, (1988). “Algorithms for clustering data”, Prentice Hall, 1988, 320 p.

[6] O.V. Peskova, “Automatic full-text documents classifier building”. Electronic Libraries: perspective methods and technology, electronic collections:Proceedings of 9th all-russian scientific conference «RCDL'2008» . Russia, Dubna, 2008, pp. 139-148.

[7] “Applied statistics: Classification and dimintion reduction”: Sprav. izd. S.A. Ayvazyan, V.M. Buhshtaber, I.S.Enyukov, L.D. Meshalkin; under the editorship of S.A. Ayvazyan. Moscow: Finance and statistics, 1989. 607 p.

[8] Kelleher D., Luz S. Automatic Hypertext Key phrase Detection // Proceedings of the Nineteenth Inter national Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK. 2005. P. 1608–1610.

[9] The Porter stemming algorithm. http://tartarus.org/~martin/PorterStemmer/

[10] Yandex catalogue: http://help.yandex.ru/catalogue/?id=873432

[11] The Yooreeka project. A library for data mining, machine learning, soft computing, and mathematical analysis. http://code.google.com/p/yooreeka/

[12] Stein, B., S. M. Eissen, F. Wissbrock. On Cluster Validity and the Information Need of Users. In: Proc. 3-rd IASTED Intern. Conf. on Artificial Intelligence and Applications (AIA'03), Acta Press, 2003, pp. 216–221.

[13] Koller D., Sahami M.(1997).Hierarchically classifying documents using very few words. Proceedings of the Fourteenth International Conference on Machine Learning, 1997, pp. 170 – 178.

\

The method of programs compression based on the frequency characteristics of programs behaviour

Alexander Shalimov
Lomonosov Moscow State University, Russian Federation
Email: ashalimov@lvk.cs.msu.su

***Abstract*—This article presents a method of programs compression based on the frequency characteristics of programs behaviour. The proposed method allows us to keep in the compiled form only frequently executed portions of programs and to store infrequently executed portions of programs in the compacted interpreted form, and to dynamically unpack and load them into the memory for execution only when they are requested. The method also allows us to control the growth of the compacted program execution time. The theoretical and experimental results of the research prove the possibility of using the proposed method for programs compression in embedded real-time controlling systems.**

***Index Terms*—code compression; code compaction; program compaction; decompression; program analysis; execution frequency; embedded systems.**

I. Introduction

Embedded real-time controlling systems is one of the main areas of the computer engineering and software development [1].

Let's list the characteristics of such systems:

1) The necessity to fulfill the requirements of high reliability and safety of an embedded system operation.
2) The programs must be executed in their deadlines, i.e. program's execution time must not exceed a given time limit.
3) Memory limitation. Such systems have small amount of main memory (for example, in modern aircraft available approximately 10MB of RAM [2], [3]).
4) Systems space and weight limitations.

The complexity of tasks for embedded real-time controlling systems is increasing, which results in consuming more memory resources. Therefore the most important characteristic of a program for such systems is the program's runtime memory size.

Programs compression methods reduce the program's runtime memory size (memory footprint). Hence it will enable to create major amount of services configured the system functionality. Besides, the modern tendency of preferential using of low-level programming languages for designing embedded systems (due to the fact that using of high-level programming languages leads to generation of exceeding program code) may serve as an additional argument for necessity of using programs compression methods in embedded systems.

In fact compressed programs run slower than original programs (overhead on execution speed). But, as mentioned above, for embedded real-time controlling systems it is important that the programs should be executed in its deadlines. This fact leads to the following main requirement for the compression methods: *the compressed program must not exceed the execution time limit set for the original program*. As a matter of fact there is always some gap between the time limit set for a program and its real time execution. This gap can be used as a time resource for compression.

Generally all program compression methods can be divided into two main groups: without decompression process [4], [5], [6] and with decompression process [7], [8], [9]. Programs compression methods without decompression can be used in the embedded real-time control systems, because they practically don't increase programs execution time. But its compression ratio strongly depends on the program (for example, how often program uses the different libraries, how many duplicated code are contained in the program, etc). Programs compression methods with decompression have a high compression ratio, but methods lead to considerable increasing execution time without possibility to control this process and the exists method can't be used in real-time controlling systems.

That is why it is necessary to create a program compression method with decompression which would allow us to control the program execution time depending on the compression ratio.

The method presented in this article guaranties that the compressed program execution time will not exceed on average a given time limit (Sections 2, 3). Also the article contains the mathematical dependencies for determining the possibility of using the proposed method for a given embedded system (Section 4). The experimental results of the research prove the possibility of using the proposed method for programs compression in embedded real-time controlling systems (Section 5).

II. Method Description

The idea of the proposed method rises from [10], [11], [12] and is based on the following two facts:

1) For a consequent program, execution of 15-20% of the program usually takes 80% of total program execution time [10], [11].
2) Program presentation in the interpreted form is usually smaller than it is in the compiled form [12].

These two facts served as a starting point for making a decision to research a method, that would allow us to keep in the compiled form only frequently executed portions of programs and to store infrequently executed portions of programs in the compacted interpreted form, and to dynamically unpack and load them into the memory for execution only when they are requested. The basic compression/decompression scheme was first described in [9].

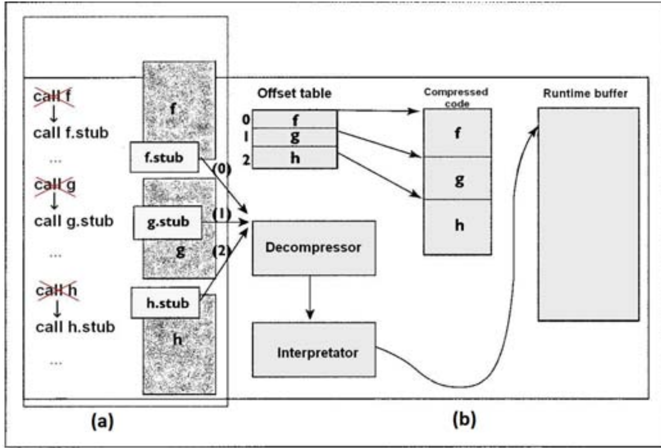


Fig. 1. Main scheme

The developed method of programs compression consists of two main parts: program compression and the compressed program execution. Figure 1 shows the basic principles of the method. Consider a program with three infrequently executed fragments of code, f, g and h, as shown in Figure 1(a). The structure of the code after compression is shown in Figure 1(b).The code for each of these fragments of code is replaced by a stub (a very short sequence of instructions) that invokes a decompressor whose job is to decompress the interpreted code for a fragment into the runtime buffer and then to transfer control to the interpreter for this decompressed code execution. A fragments offset table specifies the location within the compressed code where the code for a given fragment starts. The stub for each compressed fragment passes an argument to the decompressor that is an index into this table; this argument is indicated in Figure 1(b) by the label ((0), (1), ...) on the edge from each stub to the decompressor. The decompressor uses this argument to index into the fragment offset table, retrieves the start address of the compressed code for the appropriate fragment, and starts generating uncompressed interpreted code into the runtime buffer. The decompressor then transfers control to the interpreter for the generated interpreted code execution. When this decompressed code finishes its execution, it returns to its caller in the usual way.

The proposed idea of programs compression provides some possibilities and advantage over the existing programs compression methods.

1) Using "80-20 rule". Using compressor/decompressor scheme allows us to exploit 80/20 aspect of programs [10], [11]. For a consequent program the greater part of its execution time is usually spent on a smaller program part execution. Therefore the infrequently executed code compression will not lead to significant increasing program execution time.
2) This method's organization allows us to manage the program compression level depending on requirements to a program execution time and amount of available memory. This will allow us for embedded real-time controlling systems to consider requested time limits of programs execution time.
3) Using software implementation of the compressor/decompressor scheme enable to use programs, which can not be fully loaded into the main memory, due to keeping infrequently executed code fragments in the auxiliary memory.
4) This method does not require any hardware and hence great expenses and time losses related to adding hardware to an embedded system.

III. Method Details

The proposed method is intended to solve the following tasks:

1) Determining frequency characteristics of programs behavior;
2) Determining infrequently executed portions of programs.

A. Determining execution frequency of programs basic blocks

To solve this task the author has developed the method of determining execution frequency [13].

Given $\Pi(x_1, \ldots, x_p) = \{V, E\}$ - the original sequential program with p input parameters $(x_1, \ldots, x_p)$. The program is presented in the form of a control flow graph where the vertices $V = \{b_j\}$ $(j = \overline{1, m})$ represent basic blocks and edges $E = \{(b_{j_1}, b_{j_2})\}$ represent possible transfer of control flow from one basic block to another.

For each input parameter $x_1, \ldots, x_p$ we know a finite set of admissible values and the distribution function for these values. The program $\Pi(x_1, \ldots, x_p)$ does not get caught in an endless loop on admissible sets of input parameters (i.e. each basic block is executed a finite number of times).

Lets use the following notations:

1) $T(x_i)$ — the set of admissible values of input parameter x_i;
2) $\hat{x}_i$ — the value of the input parameter from $T(x_i)$;
3) $M_p = T(x_1) \times \ldots \times T(x_p)$ — the set of all inputs of power $|M_p|$ and of dimension p;
4) $\Pi_j(\hat{x}_1, \ldots, \hat{x}_p)$ — the number of basic block executions while program running on the $(\hat{x}_1, \ldots, \hat{x}_p)$.

Each input parameter can be treated as a random variable with a given distribution function. Lets assume $X_1, \ldots, X_p$ random values for input parameters $(x_1, \ldots, x_p)$.

Then, the frequency of b_j we will consider as $e(b_j) = \sum\limits_{(\hat{x}_1, \ldots, \hat{x}_p) \in M_p} \Pi_j(\hat{x}_1, \ldots, \hat{x}_p) \cdot P((X_1, \ldots, X_p) = (\hat{x}_1, \ldots, \hat{x}_p))$ - the value of mathematical expectation of b_j execution count.

Calculation of $e(b_j)$ requires an enormous computational outlay comparable with an outlay for running a program on all input values. In the paper [13] it was proposed to calculate the frequency $e(b_j)$ with a given precision ε and reliability γ, i.e. to find such estimated value N_j that $P(|e(b_j) - N_j| \leq \varepsilon) = \gamma$.

The idea of proposed approach is to use the Monte Carlo method. In the beginning of each basic block we add a special counter which is incremented each time when a control flow goes into that basic block. The modified program is being iteratively re-run. On each iteration new values for input parameters are generated using their distribution functions. After n program runs we will have n values of execution counter for each basic block $\Pi_j^1, \Pi_j^2, \ldots, \Pi_j^n$. It is proved that *Law of Large Numbers*, *Central Limit Theorem* and *the Berry-Essen theorem* are applicable for analysis of these numbers.

According to the Law of Large Numbers, $N_j = 1/n \cdot \sum_{i=1}^{n} \Pi_j^i$ — the average of the values of basic block execution counter obtained from a large number of program runs should be close to the mathematical expectation $e(b_j)$, and will tend to become closer as more program runs are performed ($N_j \to e(b_j)$ when $n \to \infty$).

Both Central Limit Theorem and the Berry-Essen theorem allow us to estimate a number of program runs to get the execution frequency with a given precision and reliability.

See the next algorithm for evaluation basic block execution frequency.

1) Set ε, γ.
2) Set counter of program runs to zero, $n = 0$.
3) Run modified program on a generated set of input data. Π_j^n – the value of b_j execution counter. Increase the number of program runs, $n = n + 1$.
4) If $n > 30$ then calculate the following values (we assume that after 30 iterations we can trust to sample characteristics). Else go to step 2.
   a) $N_j = 1/n \cdot \sum_{i=1}^{n} \Pi_j^i$ — the average,
   b) $s_j^2 = \frac{1}{n-1} \cdot \sum_{i=1}^{n} (\Pi_j^i - N_j)^2$ — the sample variance,
   c) $m_j^3 = \frac{1}{n-1} \cdot \sum_{i=1}^{n} (\Pi_j^i - N_j)^3$ — the sample third central moment.
5) If the $\frac{0.5 m_j^3}{s_j^3 \sqrt{n}} \leq \frac{1-\gamma}{10}$ and $n > \left(\frac{u_{\frac{1+\gamma}{2}}}{\varepsilon} \right)^2 \cdot s_j^2$ then N_j evaluates $e(b_j)$ with a given precision and reliability ($u_{\frac{1+\gamma}{2}}$ - quantile of order $\frac{1+\gamma}{2}$ of the standard normal law). Else go to step 2.

Note, that this algorithm is not applicable to basic blocks with constant execution frequency (i.e. if a basic block execution counter does not depend on input data or probability of its execution is closer to zero). Therefore if during the program runs the execution counter of some basic block remains the same, then the final decision about the execution frequency of such basic block should be taken by a programmer, i. e. a programmer should decide to continue programs reruns or to stop.

B. Determining infrequently executed code

Lets use the following notations:

1) threshold θ – the part (quota) of a total program execution time that infrequently code can account for. I.e. if execution time of any program's code is less than θ, then this code is called infrequently executed (how to choose threshold are described in Section 4). Note further we assume that execution times of instructions are the same and take one unit of time (program execution time is measured in the number of instruction executed at program runtime).
2) $weight(b_j) = e(b_j) \cdot |b_j|$ — the weight of a basic block be the number of instructions in the block multiplied by its execution frequency.
3) $T_{av} = \sum_{j=1}^{m} weight(b_j)$ — the average number of instructions executed at program runtime.

We consider all basic blocks in the program in increasing order of execution frequency until the sum of their weights will not exceed $\theta \cdot T_{av}$. All selected basic blocks are considered to be infrequently executed.

IV. Method Application

This section is about the mathematical dependencies for determining the possibility of using the proposed method for a given embedded system. Use the following notations:

1) $\tau \geq 1$ – the coefficient of admissible increasing of a program execution time. This coefficient is necessary for using a gap between actual and requested execution time.
2) $\lambda(\theta)$ — the compression ratio achieved by using the proposed method. It is calculated empirically for a given implementation of the method.
3) I – the number of instructions used for execution of a single interpreted command.
4) M – the amount of additional memory for using the proposed program compression method.

It is important to note that last three parameters are the characteristics of a given implementation of the method (see next section).

The proposed method stores infrequently executed portions of programs in compacted interpreted form and dynamically executes them. According with infrequently code definitions, no more than $\theta \cdot T_{av}$ executed instructions are transformed. So, the count of executed instruction growths on $\theta \cdot T_{av} \cdot I$ instructions and the total execution time of compacted program is equal to $(1-\theta) \cdot T_{av} + \theta \cdot T_{av} \cdot I = (1+\theta \cdot (I-1)) \cdot T_{av}$. This time should be not more than $\tau \cdot T_{av}$ and $(1+\theta \cdot (I-1)) \cdot T_{av} \leq \tau \cdot T_{av}$. Therefore, in order to compacted program execution time not to exceed a given time limit, the threshold θ should be no more than $\frac{\tau-1}{I-1}$. Note that it is only guaranteed on average, because determining of infrequently executed code based on average execution frequencies of basic blocks.

The memory overhead resultant from a programs compression method use must not exceed the amount of memory saved

due to code compression. Therefore it is necessary to select programs with total memory footprint exceeded $\frac{M}{1-\lambda(\theta)}$.

Then for using the proposed method it is necessary:

1) Choose parameter $\theta \leq \frac{\tau-1}{I-1}$.
2) Choose for compression programs whose total size more than $\frac{M}{1-\lambda(\theta)}$.
3) If the above conditions (1, 2) can not be concurrently met or a greater compression ratio should be gained, then it is possible to add the requirement to increase the system performance in $\frac{1+\theta\cdot(I-1)}{\tau}$ times.

If to follow the above recommendations on use of the proposed code compression method, it is guaranteed, that programs execution time will increase on average no more than in τ factor with the compression ratio $\lambda(\theta)$.

V. Method Implementation

The system of programs compression implemented the proposed method is written on the C++ language. It consists of two parts: program compression and compressed program execution (see Figure 2). The input for the system is a program for compression (written on C language), a distribution functions of program's input parameters, and maximum quota the program total execution time. The output is the compressed program and files with an offset table and compressed interpreted code.

The compressor transforms infrequently executed program code into the interpreted presentation and compresses it as a text. Based on distribution functions of input parameters the compressor determines average execution frequency of program basic blocks. The threshold and average frequencies calculated for a given program are used to determine fragments of infrequently executed code. The infrequently executed code are grouped that the memory overhead due to their compression would not exceed the code size reduction that can be achieved.

The decompressor consists of two main parts: the interpreter and a service of decompression of compressed interpreted code.

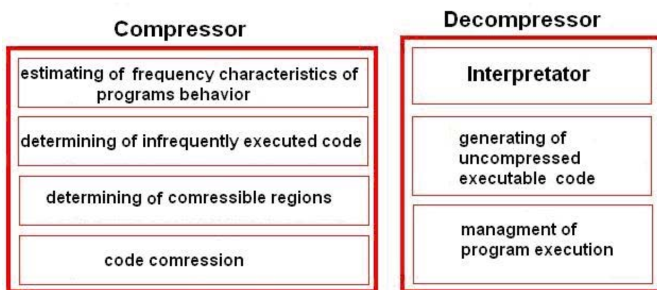


Fig. 2. Implemetation scheme

Programs for an on-board aircraft computer system were used for testing the proposed method implementation [14]. The aim of the work was to determine dependency between input parameter θ and the compression ratio achieved as a result of using the proposed method.

On each test program the system runs 10 times with different values of θ $(0.1, 0.2, \ldots, 1)$. The compression ratios $\lambda(\theta)$ obtained for each run were saved and averaged after finishing of all the experiments. This resulted in getting the following dependency presented in the Figure 3.

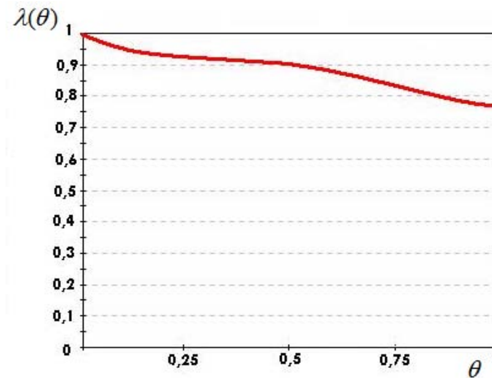


Fig. 3. Compression Ratio

So, for $\theta = 1$ the best compression ratio was achieved $\lambda(\theta) = 77\%$. For $\theta = 0.5$ the compression ratio $\lambda(\theta) = 89\%$ is achieved. For $0.2 < \theta < 0.4$ the grade of the compression ratio decreases. Therefore it is necessary to choose the input parameter from the recommended (given, specified) range.

VI. Conclusion

This article represents a program compression method based on the frequency characteristics of programs behaviour. The implementation of the proposed method has been written on the C++ language. Testing of this implementation was aimed to determine the dependency of the compression ratio on the input parameter for the system implemented the proposed method. For testing the system were used the real programs for on-board aircraft computer systems. The testing results prove the possibility of using the proposed method for on-board embedded systems.

The proposed compression method allows us to control the program execution time depending on the compression ratio. The mathematical dependencies guarantee that compressed program execution time will not exceed on average a given time limit.

It should be noted, that the proposed method is universal and can be used not only in embedded systems.

References

[1] Embedded Computing Design [HTML] (www.embedded-computing.com)
[2] *K. Kolpakov* History of onboard embedded systems in Russia // PCWeek, N32, 1999
[3] *A.M. Pavlov* Principles of organization of advanced onboard computing systems [HTML] (http://www.mka.ru/?p=41177)
[4] *B. Bus, D. Kastner, D. Chanet, L. Put, B. Sutter* POST-PASS Compaction Techniques // Communications of the ACM August 2003/Vol. 46, No.8
[5] *Sheayun Lee, Jaejin Lee* Selective code transformation for dual instruction set processors // ACM Transactions on Embedded Computing Systems (TECS), Volume 6, Issue 1, May 2007

[6] *B. Sutter, K. Bosschere* Software techniques for Program Compaction // Communications of the ACM August 2003/ Vol. 46, No.8
[7] *T.M. Kemp, R.M. Montoye* A Decompression Core for PowerPC // IBM Journal of Research and Development, Volume 42 Number 5/6, September, 1998
[8] *S. Seong, P. Mishra* Bitmask-Based Code Compression for Embedded Systems // IEEE Transactions on computer-aided design of integrated circuits and systems, 2007
[9] *S. Debray, W. Evans* Profile-Guided Code Compression. // Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, 2002
[10] *R.L. Smeliansky, D.E. Guryev, A.G. Bahmurov* About one mathematical model for calculation of programs behavior. Programming, N6, 1986
[11] *R.L. Smeliansky, T. Alanko* On the calculation of control transition probabilities in a program Inform. Processing Letters N.3, 1986
[12] *P. Brown* Macros without tears // Software: Practice and Experience. Volume 9, Issue 6, 1979
[13] *A.V. Shalimov* Method of determining execution frequency of programs basic blocks // Modeling and analysis of information systems, Volume 18, Number 2, 2010.
[14] DrTesy [HTML] (http://lvk.cs.msu.su/index.php/articles/65)

Metrized Small World Approach for Nearest Neighbor Search

Andrey Logvinov, Alexander Ponomarenko, Vladimir Krylov, Yury Malkov
MeraLabs, Nizhny Novgorod, Russia
alogvinov@meralabs.com, aponom@meralabs.com, vkrylov@meralabs.com, ymalkov@meralbs.com

Abstract

In different areas attempts are made to organize data into multi-linked structures which are well suited for information search, in particular the nearest neighbor search where the result data items are metrically close to a given data item. These structures often take the form of trees (M-Tree, cover tree, KD-tree, GNAT) or networks (M-Chord, VoroNet, RayNet) built over a set of data items.

In this paper we give the regular approach to the construction of links between data items which provides logarithmical time complexity of the nearest neighbor search in the structure. According to this approach, data items are organized into an undirected graph with Small World properties, which ensure the existence of a short path between any two data items regardless of the graph size.

We propose different construction and search algorithms depending on the properties of the metric which determines the proximity of data items. The types of metric we consider are abstract metric and ordered metric. Further we extend the ordered metric approach to compound data items in the form of attribute-value pair sets to enable inclusion search by an arbitrary subset of attribute-value pairs.

Finally we provide simulation results for the structure with compound data items.

1. Introduction

The nearest neighbor search problem is defined as follows: given a set S of n points in some metric space (X, d), build a data structure on S so that for a given query point $p \in X$ one can efficiently find a point $q \in S$ which minimizes $d(p, q)$.

Different approaches exist for building such a structure. The works [4, 5, 11] suggest hierarchical tree structures constructed using information about metric proximity of the elements. One notable shortcoming of this approach is the presence of the mandatory root node in tree-like structures which makes building totally distributed implementations problematic.

There are also ways to build a distributed structure over the set S. The works [12] suggest distributed hash table as the data structure using the pivot-based metric space indexing approach.

The work [6] discusses the VoroNet distributed data structure. The elements of S are two-dimensional Euclidian space points. Each point from S is linked to all of its neighbor points on Voronoi diagram (Delaunay graph) plus additional distant points to give the structure Small World properties. Greedy search algorithm is used.

The following work [7] by the same authors considers the structure where the elements are points in a n-dimensional Euclidean space. The main difference from the previous work is that every point is connected with only a subset of the Voronoi neighbors to avoid exponential dependence of complexity on the number of dimensions. But this link set reduction leads to inexact search results, i.e. the result point is not always the nearest neighbor of the query point although number of such result can be made insignificant. Another drawback of this approach is that it can only be applied to the points of Euclidian space with a fixed number of dimensions.

In this paper we propose a regular approach to the construction of links between data elements in the form of an undirected graph with Small World properties [9, 10] to provide logarithmical complexity of the nearest neighbor search. We called the resulting structure Metrized Small World [1] (MSW).

We propose different construction and search algorithms depending on the properties of the metric which determines the proximity of data items.

The rest of the paper is structured as follows. Section 2 describes the construction of MSW structure based on abstract semi-metric. Section 3 describes MSW structure construction algorithms for ordered metrics. In the section 4 we extend the ordered metric approach to compound data items in the form of attribute-value pair sets to enable inclusion search by an arbitrary subset of attribute-value pairs. Finally we

provide simulation results for the structure with compound data items in the section 5.

2. Metrized Small World data structure

Metrized Small World data structure on the set of data items S is expressed by the graph $G(V,E)$. Each vertex $v \in V$ corresponds to a single element of the set S. Each edge $e \in E$ is associated with a link between two data items from the set S. Assume that $d(v,p)$ equivalent to $d(s,p)$ where s is the data item which corresponds to the vertex v. Then the search of the nearest neighbor of the query point $p \in X$ comes to finding the vertex $v \in V$ with the minimal distance to p.

In the work [1] we gave the construction and search algorithms for that structure. In the paper [2] we also suggested a distributed storage architecture based on the proposed structure. Here we re-cite those algorithm according to the notation assumed for this paper.

We provide the algorithm which adds v_{new} vertex to the graph $G(V,E)$, where V is the set of previously added vertices. Thus the parameters of the algorithm are V — the set of previously added vertices, v_{new} the vertex being added, $v_{start} \in V$ – an arbitrarily selected vertex from V (the starting point of the search) and two integer numbers m and n.

Algorithm: $add_metric(V, v_{new}, v_{start}, n, m)$

1. Arbitrarily select an element $v_{curr} \in V$
2. Let *VisitedList* be the set of visited elements.
3. Let *CandidateList* be the set of candidate elements for link establishment sorted by value of semi-metric to v_{new} in ascending order.
4. Assume that *CandidateLists* initially contains only v_{start}.
5. For i =1 to n do
   5.1. Sort *CandidateList* by value of semi-metric to v_{new} in ascending order.
   5.2. Select the first element p from *CandidateList* not contained in *VisitedList*. If no such element exists then break.
   5.3. Add p to *VisitedList*.
   5.4. Add the set of p neighbor elements to *CandidateList*.
6. Mutually connect the element with m arbitrary elements from *VisitedList*.

We shown that the structure constructed using this algorithm provides the necessary condition for the existence of effective search algorithm, because the Small World properties of the graph $G(V,E)$ ensure the existence of a short path between any two vertices. But this structure requires search algorithms which are more complex than the greedy algorithm due to the existence of metric local minimums.

An advantage of this approach is that the proximity measure M can be any function which is a general metric or even semi-metric defined over the set S.

3. Single-attribute Distributed Metrized Small World Data Structure

In the paper [3] we gave the algorithm for constructing the similar structure for a narrower class of metrics, i.e. for the metrics for which the order between data items is defined. If any data item will be linked with its direct predecessor and successor with regard to the metric, there will be no local minimums. The condition of the data item being linked to its direct successor and predecessor ensures the existence of the Delaunay graph which in its turn provides for correctness of the greedy search algorithm which attempts to minimize the distance from the query on each step.

Algorithm:
$add_ordered_metric\ (V, v_{new}, v_{start}, m)$

1. Let $v_{cur} = v_{start}$.
2. For each neighbor v_i of v_{cur} calculate $d_i = d(v_i, v_{new})$.
3. If $\min(v_i) < d(v_{cur}, v_{new})$ let $v_{cur} = v_i$ for which $d_i = \min(d_i)$ and go to step 2.
4. If $v_{cur} < v_{new}$ let $v_{pre} = v_{cur}$ and let v_{succ} be the direct successor of v_{new} chosen from the neighbors of v_{cur}.
5. If If $v_{cur} > v_{new}$ let let $v_{succ} = v_{cur}$ and let v_{pre} be the direct predecessor of v_{new} chosen from the neighbors of v_{cur}.
6. Mutually connect v_{new} with v_{pre} and v_{succ} if they exist.
7. Repeat m times:
   7.1. If v_{pre} exists, let v'_{pre} be the direct predecessor of v_{pre} chosen from its neighbors.
   7.2. If v_{succ} exists, let v'_{succ} be the direct successor of v_{succ} chosen from its neighbors.
   7.3. If none of v'_{pre} and v'_{succ} exist then break.
   7.4. If only v'_{pre} exists or $d(v'_{pre}, v_{new}) < d(v'_{succ}, v_{new})$ mutually connect v_{new} and v'_{pre} and let $v_{pre} = v'_{pre}$.
   7.5. If only v'_{succ} exists or $d(v'_{succ}, v_{new}) < d(v'_{pre}, v_{new})$ mutually connect v_{new} and v'_{succ} and let $v_{succ} = v'_{succ}$.

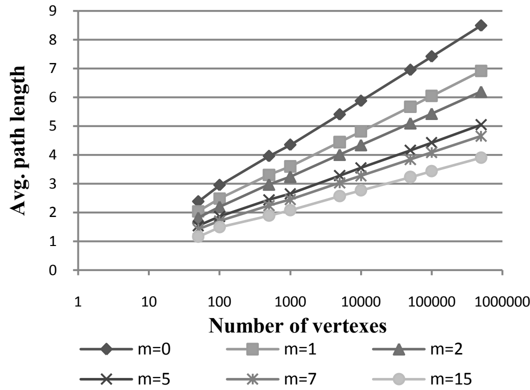


Figure 1. Average shortest path length between two vertexes

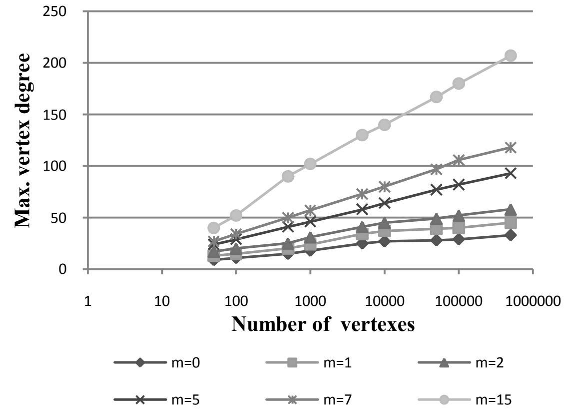


Figure 2. Maximum vertex degree

The nearest neighbor search is performed by following links from one element to another in the direction of the minimal metric.

The Small World properties of the graph ensure the logarithmical search complexity for a random data set. The absence of the root element and the construction of the structure on the data item level provides for creating a completely distributed implementation of the structure. As can be seen on Fig. 1 and 2, both average shortest path length and maximum vertex degree scale logarithmically with the number of vertexes. Therefore the structure is suitable for storing very large amounts of data.

The nearest neighbor search is reduced to finding the minimum of the metric from the query to a data item. If the distance between the query and the found data item is lower than the query radius than the fond data item is the result, otherwise there is no result. If we must find all data items inside the query radius, we perform a sequential search in both directions from the first found data item.

The proposed data addition algorithm is incremental, i.e. the addition of a new data item affects only a small number of existing data items.

4. Multi-attribute Distributed Metrized Small World Data Structure

In the two previous sections we considered the elements as atomic entities relative to the metric. Now we want to extend our approach to composite data items. We will consider the composite objects which are represented by an unordered set of atomic objects for all of which one common ordered metric is defined.

Then we define the search problem as the search of at least one of all of the composite objects which include the given set of atomic objects. This data model is often used for describing application domain entities with a set of tags or keywords, e.g. images, hyperlinks, musical tracks, blog posts etc. This model can also represent objects consisting of non-fixed set of attribute-value pairs.

Therefore for convenience we will consider arbitrary strings (or tags) as atomic objects. Hence the composite objects will be represented as unordered sets of tags.

Our main idea was to construct the graph $G(V,E)$ in a way that objects with any matching subset of atomic objects T_{fix} would constitute the sub graph (layer) $L_{T_{fix}} \in G(V,E)$ consisting of a single connected component which in its turn would form the MSW structure described in the previous section. Then the search for an element containing the given set of tags $T_q = \{t_1, t_2, \dots, t_m\}$ would be performed by first finding object from sub graph (layer) $L_{T_{t1}}$ consisting of objects containing the tag t_1. After that, inside this subgraph-layer $L_{T_{t_1}}$ another element from the subgraph-layer $L_{T_{t_1,t_2}} \subset L_{T_{t_1}}$ is recursively searched for. The subgraph-layer $L_{T_{t_1,t_2}}$ consists of objects containing both tags t1 and t2. The process continues until an object form the subgraph-layer $L_{T_{t_1,t_2,\dots,t_m}}$ is found which consists of objects containing all the given tags $\{t_1, t_2, \dots, t_m\}$.

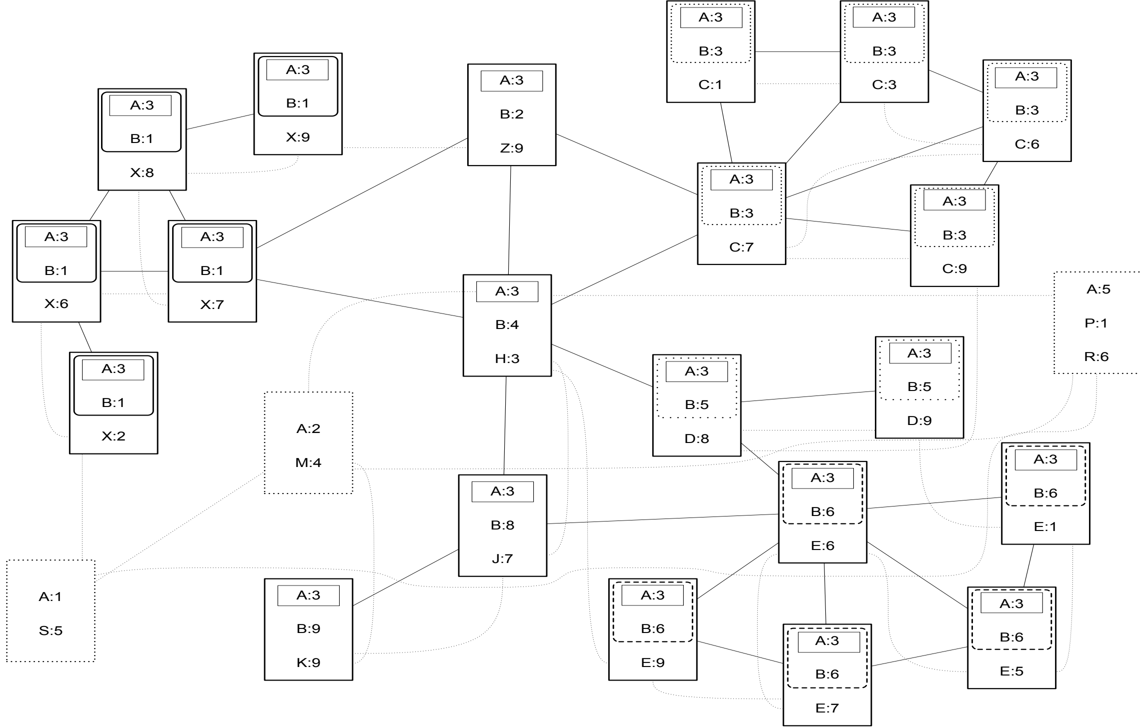


Figure 3. Example Multi-attribute Distributed Metrized Small World Data Structure. The dashed lines represent the edges in the $L_{\emptyset}$ layer. Solid straight lines show the links between objects having a common subset of tags.

For demonstration purposes we provide the example of the network of objects almost all of which contain three tags. Dashed curved lines show the links between objects which contain tags which are neighbors in lexicographical order. Solid straight lines show the links between objects having a common subset of tags.

Further we give a more formal description of the construction and search algorithms for this structure Let $T = \{t\}$ be the set of all possible tags which are distinct string values.

For each data element a let there be the unordered set $T_a \subset T$ of tags associated with the object. Given a query set $T_q \subset T, T_q \neq \emptyset$ we must find the set A_r of resulting data elements such that $\forall a \in A_r \; T_q \subset T_a$, i.e. all data elements which have all of the tags specified in the query.

Let the set $MSW_x = \left\{\left(t_{a_i}^k, t_{a_j}^l\right)\right\};\ a_i, a_j \in X, t_{a_i,a_j}^{k,l} \in T_{a_i,a_j}$ be the MSW structure built over a set of elements X. Every element of MSW_X represents a link between pair of tags in data elements (it can be the same element). If there is no element corresponding to a pair tags, there is no link between them. Two identical tags on the different items cannot have links simultaneously in one MSW_X. We consider a tag t being a member of the MSW_X if $\exists t_j, (t, t_j) \in MSW_X$. We can use our algorithm described in the section 3 of this paper to search for given tag in MSW.

Let $L_{T_{fix}} = \left(T_{fix}, MSW_X\right)$; be the MSW layer built over a set of tags T_{fix}. For every tag that is a member of $L_{T_{fix}},\ t \in T_a, T_{fix} \subset T_a$.

Let the $a = search_single(L_{T_{fix}}, t_{start}, t);\ a \in X$ be the operation of searching for a single element, member of $L_{T_{fix}}$ for which $t \in T_a$. The tag t_{start} (member of $L_{T_{fix}}$) is the entry point of the algorithm described in the second section of this paper.

Let $add_partial(L_{T_{fix}}, t_{start}, a, t_a)$ be the operation of addition of the tag t_a of the element a to the MSW layer $L_{T_{fix}}$. The tag t_{start} is used as the entry point. The time complexity of the $add_partial$ operation is logarithmic to the number of tags in $L_{T_{fix}}$. We consider an element a being a member of the MSW layer $L_{T_{fix}}$ if it has been partially added to $L_{T_{fix}}$ at least once.

Let $add_complete(L_{T_{fix}}, t_p, a)$ be the operation of complete addition of the element a to the MSW layer $L_{T_{fix}}$. The $add_complete$ operation is performed using the following algorithm:

Algorithm: $add_complete(L_{T_{fix}}, t_p, a)$

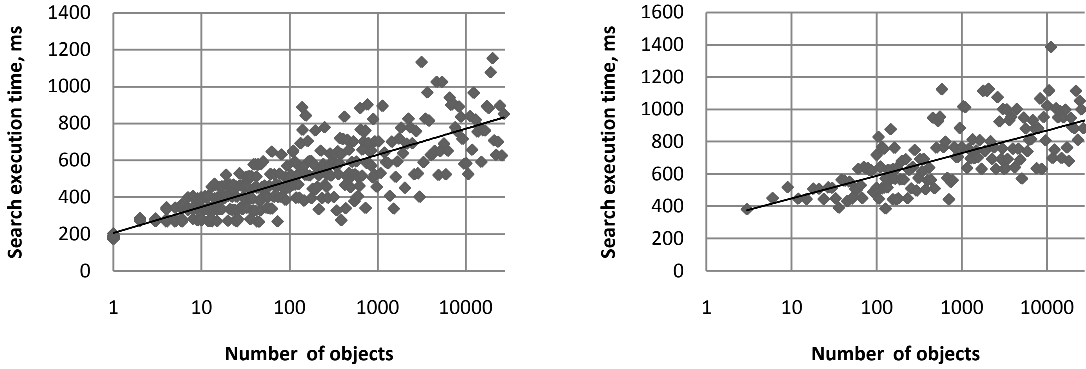


Figure 4. Experimental results. Left: two common tags. Right: three common tags.

1. Let $T_{free} = T_a \setminus T_{fix}$ be the set of all tags associated with the element a but not contained in T_{fix}.
2. For each $t_a \in T_{free}$ do $add_partial(L_{T_{fix}}, t_p, a, t_a)$.

Let $S = \{L_{T_{fix}}\}$ be the set of all MSW layers (the structure being described). An arbitrary member t_{global_start} of the MSW layer $L_\emptyset$ can serve as a global entry point for addition process.

Let $add_recursive(S, L_{T_{fix}}, t_{start}, a)$ be the operation of addition of the element a to the structure S.

The $add_recursive$ operation is performed using the following algorithm, assuming that the initial values are $T_{fix} = \emptyset$ and $t_{start} = t_{global_start}$.

Algorithm: $add_recursive(S, L_{T_{fix}}, t_{start}, a)$

For each $t \in T_a \setminus T_{fix}$

1. Find $a_{next} = search_single(L_{T_{fix}}, t_{start}, t)$
2. **If** p_{next} exist, perform $add_recursive(S, L_{\{T_{fix}, t\}}, t_{a_{next}}, a)$, where $t_{a_{next}}$ is a random tag of a_{next}, $t_{a_{next}} \notin \{T_{fix}, t\}$ **else** $add_partial(L_{T_{fix}}, t_{start}, a, t)$

Let $A = search_recursive(S, L_{T_{fix}}, t_{start}, T_{query})$ be the operation of searching all elements A for which $T_{query} \subset T_a$.

The $search_recursive$ operation is performed using the following algorithm, assuming that the initial values are $T_{fix} = \emptyset$ and $t_{start} = t_{global_start}$.

Algorithm: $search_recursive(S, L_{T_{fix}}, t_{start}, T_{query})$

1. If $T_{query} = \emptyset$ then return all elements in layer $L_{T_{fix}}$
2. for random $t_q \in T_{query}$ find.
3. $a_{next} = search_single(L_{T_{fix}}, t_{start}, t)$
4. Remove t_q from T_{query}
5. $search_recursive(S, L_{\{T_{fix}, t\}}, t_{a_{next}}, a)$ where $t_{a_{next}}$ is a random tag of a_{next} $t_{a_{next}} \notin \{T_{fix}, t\}$

Constructing link using the above approach is to a certain degree equivalent to indexing by all possible combinations of columns in a relational database. The main advantage of this approach is the possibility to quickly find an object or a set of objects with any given set of tags without regard to the quantity of objects with a certain subset of tags (atomary objects).

Further we give the experimental data obtained on the structure prototype to confirm the theoretical assumptions regarding the advantages of our approach.

5. Experimental data

The experiments were set up as follows.

In the first experiment a set of N objects was generated half of which contained the single common tag “X”, other half contained the single common tag

“Y” and a single object with both “X” and “Y” tags. The objects were added to the structure in random order. We measured the time of search for the object containing “X” and “Y” tags. The measurement was repeated many times for different values of N, the set of random objects was regenerated each time. See the left graph.

In the second experiment the test set contained N random objects containing equal amounts of object containing two common tags “X”, ”Y”; “Y”, ”Z”; “X”, “Z” and the single object containing all three tags “X”,”Y”,”Z”. See the right graph.

The results are shown on Figure 4. The graphs show that in both cases the object search time depends logarithmically on the number the objects in the structure which confirms our theoretical assumptions.

6. Conclusion and future work

We believe that the key to the building of search-oriented distributed systems is the construction of multilinked structures similar to social networks. But the metric distance between data items must be correlated to the number of links which separate them. In this paper we described the methods of construction of such structures for certain data types. The necessary and sufficient condition of correctness of the greedy search algorithm is the inclusion of Delaunay graph into the structure graph. Failure to satisfy this particular condition was the obstacle for using the greedy search algorithm with the structure described in the section II. The condition of existence of Delaunay subgraph has been satisfied in the structures described in sections III an IV. But supporting the correct Voronoi tessellation as in [6] or in section IV requires large overhead with the number of dimensions greater than two. For this reason we intend to focus our further research on finding the compromise between search accuracy and calculation overhead.

7. References

[1] V. Krylov, A. Logvinov, A. Ponomarenko, D.Ponomarev “Metrized Small World Properties Data Structure”, Proc. Software Engineering and Data Engineering (SEDE 2008).

[2] V. Krylov, A. Logvinov, A. Ponomarenko, D.Ponomarev “Active Database Architecture for XML Documents”, Proc. Computer applications in Industry and Engineering (CAINE 2008).

[3] V. Krylov, A. Logvinov, A. Ponomarenko, D.Ponomarev, “Single-attribute Distributed Metrized Small World Data Structure”, Proc. IEEE International Conference on Intelligent Computing and Intelligent Systems 2009 (CAS)

[4] CIACCIA, P., PATELLA,M., AND ZEZULA, P. 1998. A cost model for similarity queries in metric paces. In Proc. 17th ACMSymp. on Principles of Database Systems (Seattle), 59–67.

[5] BRIN, S. 1995. Near neighbor search in large metric spaces. In Proceedings of the 21st conference on Very Large Databases (VLDB’95), 574–584.

[6] Beaumont, O. and Kermarrec, A.M. and Marchal, L. and Riviere, E., VoroNet: A scalable object network based on Voronoi tessellations, in IEEE IPDPS, 2007

[7] O. Beaumont, A.-M. Kermarrec, and E. Rivire. Peer to peer multidimensional overlays: Approximating complex structures. In OPODIS,11th International conference on principles of distributed systems, 2007.

[8] J.-D. Boissonnat and M. Yvinec. Algorithmic Geometry. Cambridge University Press, 1998.

[9] D.J. Watts “Small Worlds”, Princeton, New Jersey: Princeton University Press, 1999.

[10] R. Albert and A.-L. Barabasi “Statistical mechanics of complex networks.” Rev. Mod. Phys., 74(1): pp. 47-97, January 2002.

[11] A. Beygelzimer, S. Kakade, and J. Langford. “Cover trees for nearest neighbor”. Proceedings of the 23rd International Conference on Machine Learning, pages 97–104, 2006

[12] D. Novak and P. Zezula. M-Chord: A scalable distributed similarity search structure. In Proceedings of First International Conference on Scalable Information Systems (INFOSCALE 2006), Hong Kong, May 30 June 1 . IEEE Computer Society, 2006.

An approach to on the fly activation and deactivation of virtualization-based security systems

D. Yefremov
Moscow State University
Moscow, Russian Federation
Email: yefremov.denis@gmail.com

Scientific Advisor:
P. Iakovenko
Institute for System Programming RAS
Moscow, Russian Federation
Email: yak@ispras.ru

***Abstract*—We report on work in progress which allows on the fly activation and deactivation of hardware virtualization based security systems intended for protecting applications running under the control of untrusted operating system. We present an approach for reserving hardware resources from the operating system for the hypervisor and additional virtual machines that may be required by the security system. We also consider that hypervisor is launched from the untrusted environment that may try to fool the user during the startup and shutdown of the hypervisor.**

***Keywords*-VMM; hypervisor; virtual machine monitor; virtualization; secure activation; on the fly load.**

I. Introduction

Hardware virtualization technology is widely used for consolidating hardware resources, reducing power consumption, simplifying datacenter administration and improving system's reliability. Recently it has got spread into the computer security area. Among the problems that are being solved with it are malware analysis [1], reliable host-based intrusion detection systems [2], securing applications in the untrusted operating systems [3] and others.

Virtualization technology provides the means for executing unmodified operating system (OS) inside a hardware virtual machine (VM) under the control of a relative small system program — virtual machine monitor (hypervisor) [4]. Depending on the requirements hypervisor may assign a device to a particular VM or share a device among virtual machines multiplexing their accesses to it. In the former case VM has exclusive access to the device and no other virtual machines may use it. In the later case the amount of virtual machines that may access the device is not limited but hypervisor must virtualize it providing each VM with the device software model. Such models may constitute significant amount of overall hypervisor code size and require having device driver inside the hypervisor or priviledged control virtual machine [5].

The common property of the existing virtualization-based systems that secure applications inside the untrusted operating system is the absence of necessity for such systems to stay activated all the time the computer is up and running. They are needed only for that periods of time when user executes *trusted applications*. The rest of time the execution of OS inside the VM may introduce restrictions into the user workflow. In particular the performance of virtualized system is lower than of real one and some features of specific devices may become unavailable.

Overshadow [3] secures applications by keeping their executable and data files encrypted in the file system. When user launches such applications Overshadow dynamically decrypts their content in the memory. Overshadow is idle for the time when only untrusted applications are running. The same case applies to Proxos [6] which runs trusted applications in the dedicated virtual machines. All other applications execute in one and the same untrusted VM. In [7] hypervisor prevents data leakage through the network connection by running OS in the VM that doesn't have network adapter at all. Then it provides network access for trusted applications by delegating their socket-related system calls to other network-enabled VM. The same level of isolation may be ensured without the hypervisor if network cable is disconnected from the computer.

Obviously when user activities are restricted due to system is virtualized he may switch between the bare hardware and virtualized configurations by rebooting the computer. During the boot time he selects the desired configuration: bare system with high performance or virtualized system with ability to run trusted applications. However such workflow may be inconvenient for the user and lead to declining using particular virtualization-based security solution.

Therefore we deem that that the problem of dynamic activation and deactivation of virtualization-based security system is topical. The solution should allow on the fly launch of the hypervisor which moves the up and running OS into the virtual machine environment and if necessary launches additional virtual machines. Then some time later user should be able to bring the system to the original state without rebooting the computer. Meaning that hypervisor stops all additional virtual machines, brings back OS to the bare environment and finally terminates itself. It is also important to include robust attestation procedure into such solution since hypervisor is launched from the untrusted environment.

Dynamic load of the hypervisor is described in [8]. Authors discuss the consequences of using hardware assisted virtualization for implementing malware and present BluePill system that loads on the fly into the memory putting operating system into the virtual machine. Authors

claim that malware hypervisor may fool OS inside the VM that it runs on bare hardware. Since hypervisor has unrestricted access to VM resources it may bypass built-on OS security mechanisms and perform intended malicious actions. Our work follows the same approach for on the fly hypervisor loading however we propose to use our approach for virtualization-based security systems that normally do not need to hide from the OS. This gives more options for reserving hardware resources from the OS for the hypervisor own usage (for example, legally disabling OS access to some devices).

The attestation of a software stack launched from the untrusted environment is discussed in [9]. Authors present an approach and describe attestation protocol that allows safe usage of public computer by providing secure initialization of trusted software stack. The protocol uses Trusted Platform Module (TPM) chip that must be installed on the machine and provides convincing evidences to the user that actually initialized software stack is exactly the same as the expected by the user one. Our approach to attestation of the dynamically loaded hypervisor is also based on the TPM chip however unlike the system in [9] we do not need rebooting the computer which in turn allows achieving simpler attestation procedure.

We will describe our approach with regards to the security system described in [7]. However, we believe that suggested approach may be used for other virtualization-based security systems that 1) assign hardware devices to virtual machines for exclusive usage 2) allow their late launch (i.e. after OS has been booted) without sacrificing security requirements.

We assume that computer has CPU supporting AMD-SVM technology [10] including memory virtualization (Nested Page Tables) [11]. Computer is also equiped with Trusted Platform Module chip supporting TPM specification version 1.2 [12] whose public certificate is known to the user. However our choice of using AMD virtualization technology is merely based on the hardware available to us and the approach described in this article may be applied to the computers equiped with Intel processors supporting Intel-VT [13] virtualization technology.

We will briefly describe the security architecture presented in [7]. Hypervisor executes two virtual machines: primary (called *private* VM) and service (called *public* VM). User works in the primary VM which controls all hardware devices except network adapter. Hypervisor runs primary VM without network adapter since OS in the primary VM is not trusted and may use network connection to leak sensitive data from the VM. Service VM may run in the background since the only purpose of this VM is to serve socket-related system calls executed by the trusted processes in the primary VM. System call requests are delivered by the hypervisor to the service VM through the tamper-proof inter-VM channel. It should be noted that 1) primary VM must run on one CPU (one CPU core) and 2) the only device that is controlled by the service VM is network adapter while primary VM must not have access to it at all.

Service VM is required only for providing network access for the trusted applications. During the time intervals when these applications are not executing the OS in the primary VM may run directly on the bare hardware and even may control network adapter as long as the user ensures that network cable is *disconnected* from the computer. Additionally primary and service virtual machines do not share any peripheral devices.

This article is organized in the following way. In section 2 we describe the steps taken during the on the fly security system activation. In section 3 we discuss the attestation of activated system. In section 4 we describe the steps taken to deactivate the security system in a fool-proof way. In section 5 we conclude presented approach.

II. Security system activation process

Initially user boots operating system installed on the computer (hereafter we call it primary OS) ensuring that network cable is disconnected from it, i.e. computer is physically isolated from the network. Hypervisor is not started during the boot process however it is possible that there is a malware hypervisor (or any other kind of malware) running on the computer.

The file system contains hypervisor image, image for OS in the service VM (hereafter we call it service OS) and a secure loader. Secure loader is a special piece of software required for SKINIT instruction. OSLO secure loader may be used for it [14]. It is worth mentioning that all these files are not required to be encrypted however user knows their checksums (hash codes) and these checksums were calculated on a trusted machine.

There is a driver preloaded into the operating system kernel which exposes an interface to the user space applications via a device file in the file system. The driver's task is to reserve specific hardware resources from the operating system using built-in OS kernel interfaces, load images from the file system into the memory and start secure system initialization. Driver is also responsible for printing informational messages coming from hypervisor on the display. Hypervisor notifies driver about pending messages by injecting interrupt into the virtual machine which passes control to the handler installed by the driver.

To start security system activation user launches an application that sends "activate" command to the driver using *ioctl* system call passing the file names of the images. The driver then:

1) Unplugs all CPU cores, except the boot strap core (BSP), using the public Linux API for hot plugging CPUs.
2) Reserves required amount of physical memory. This may be done in two ways. The preferred way is to unplug memory banks using the public Linux API for hot plugging memory modules. Alternatively memory may be reserved using `get_free_pages()` function. Unplugging memory banks gives less fragmented memory areas (in ideal case not fragmented at all). The advantage for having contiguous memory is described below.

3) Loads hypervisor image, service OS image and the secure loader into the reserved memory.
4) Shutdowns network interface, unloads network driver and installs "pci-stub" driver, that is available in Linux, instead of it. This stub driver reports to OS that it controls network adapter without actually initializing the device.
5) Executes SKINIT instruction which securely transfers control to the secure loader which in turn passes control to the hypervisor.

Upon receiving the control hypervisor prepares VMCB structure filling it with the current hardware state, creates nested page tables [11] for the VM and starts VM execution. From the user perspective nothing has changed in the environment except the decrease of active CPU cores and memory amount. Since network cable was originally disconnected completely removing network adapter from the system does not harm user space applications.

Before hypervisor starts VM execution it re-initializes released CPU cores and starts idle loop on them while executing inside virtual machine primary OS cannot regain control over released resources. Attempt to access memory outside its guest physical address space, re-initialize unplugged CPU cores or reload original network adapter driver will lead at worst to the VM crash without getting access to the hardware resources.

One of the released CPU cores is used for running service VM. Hypervisor virtualizes local APIC so it presents this core to the service OS as a BSP core of a single core processor. Service OS completely runs inside memory. The OS kernel is preconfigured by the administrator in such a way that it does not require access to any peripheral devices except network adapter.

Both virtual machines execute on separate CPU cores: primary VM on the BSP core (zero core), service VM on the other core. Such approach does not require having VM scheduler in the hypervisor that distributes CPU time between virtual machines. Hypervisor itself does not require dedicated CPU cores for execution since it gains control only on virtual machine exit. We assume that CPU has at least two cores which is common for most of modern CPUs.

Mapping of the guest (virtual machine) physical address space into the host address space (machine memory) is resolved using nested page tables (NPT) provided by the hardware. These page tables add additional layer to the hardware address translation so every memory access made from inside the virtual machine is translated with NPT to evaluate machine address. It is desirable to reserve physical memory from the primary OS using memory hotplug feature available in Linux since having large contiguous memory blocks makes the structure of NPT rather simple. However success of memory unplug operation depends on proper kernel configuration, kernel boot options ("movablecore" option) and kernel runtime state.

With nested page tables the hypervisor can control accesses to the machine memory made from CPU context. However if OS in VM owns DMA capable device then it may use it to modify arbitrary physical memory areas. This constitutes potential threat to the security system since malware may use DMA operations to subvert hypervisor [15]. Physical address space visible to a DMA capable device may be limited by the IOMMU device [16]. IOMMU sits between PCI bus and system memory and allows specifying page tables that are used to translate every memory access originated from the devices connected to PCI bus. Whenever an address does not have valid translation in IOMMU page tables the device receives master abort and memory access is rejected.

Currently hardware supporting IOMMU is not wide spread so we also consider possibility to use Device Exclusion Vector (DEV) feature of AMD CPUs. The major difference between IOMMU and DEV is that DEV does not support address translation. It simply allows to specify bit mask (one bit per each physical page) that marks DMA write-protected pages. Since DEV may not perform address translation it may be used for one VM only whose guest physical memory is one-to-one mapped to the machine memory starting from zero address. That is the case for the primary VM. DEV bit mask for that VM write-protects all physical pages outside primary VM memory.

The use of DEV for the service VM is not that straightforward. However service OS kernel is configured by the administrator and he may include paravirtualized driver for the network adapter into it. Such driver asks hypervisor to translate address for every DMA operation. So DEV may still be used for the service VM with the bit mask write-protecting all machine memory outside service VM address space.

It is worth mentioning that DEV does not provide ability to read-protect pages so every VM may read data from any machine address. This may violate security requirements if malware in service VM would use DMA operations to read sensitive data from the primary VM memory. Therefore the use of DEV for DMA protection requires service OS to be *trusted*. Service OS is merely used to execute socket-related system calls so a microkernel OS (e.g. Minix 3) may be used in service VM instead of Linux.

III. Security system attestation

We use Trusted Platform Module (TPM) [12] to perform attestation of the activated security system. TPM is a chip normally attached to the motherboard that provides set of security primitives. TPM has several platform configuration registers (PCR) that are intended for accumulating SHA-1 hash-codes. These hash-codes (measurements) represent hardware installed or software running on the machine. The contents of a PCR register may be updated by executing `TPM_Extend` operation only (there is no way to write to PCR directly). `TPM_Extend` operation concatenates current value of the PCR register with the provided data, hashes the result and updates PCR with it. CPU is integrated with the TPM to perform secure late launch of the software using SKINIT instruction. Upon

executing SKINIT CPU asks TPM to reset PCR 17 register to zero value. This is the only way how PCR 17 may receive zero value.

Contents of the PCR register may be cryptographically signed with Attestation Identity Key (AIK) using `TPM_Quote` operation. Private part of the AIK is sealed inside the TPM while public part is exposed to the user. Signed PCR value gives creditable and believable proof to the user that PCR value originates from the legal TPM. A random number (nonce) may be provided to the `TPM_Quote` operation as a parameter in order to protect signed PCR value from the replay attacks.

Activation of security system starts from executing SKINIT instruction passing address of secure loader (SL) to it [14]. CPU disables interrupts, blocks DMA writes to the memory area occupied by the SL, asks TPM to measure (hash) SL and finally transfers control to the SL. On a multi processor (multi core processor) SKINIT must be executed on the BSP core with all other cores put into the idle state. SL in turn performs the same operations with regards to the hypervisor image and then transfers control to the hypervisor code. Hypervisor measures service OS image and asks TPM to sign the PCR 17 contents using the `TPM_Quote` operation to which it passes user-provided nonce. The signed PCR 17 value is delivered to the user (e.g. displayed on the screen). The necessity of having SL in this chain is caused by the hardware size limitations for the SL. It should be at most 64K which may be insufficient to fit all the hypervisor code.

The resulting PCR 17 value contains measurements of the SL, hypervisor, service OS image and the nonce. User knows hash-codes of these components so he may repeat all calculations performed by TPM on a separate trusted device (e.g. mobile phone). By comparing hash-code displayed on the screen against the hash-code calculated on the trusted device he checks whether all security system components have been activated in the proper order. And by validating TPM signature using the public part of AIK (known to the user) he gets assurance that displayed measurement represents the actual state of the system and is not a fake measurement generated by the malware that has intercepted activation process and tries to fool the user. Only after validating displayed measurement the user is safe to connect network cable to the computer.

We consider that execution of SKINIT instruction may be trapped by the malware hypervisor running on the machine. This malware may perform all above-mentioned measurements without actually passing control to the security system or it may start security system inside the virtual machine hence maintaining control over it in a hidden way. However `TPM_Quote` operation may be executed only against PCR register which in turn may be updated using the `TPM_Extend` operation only. Therefore malware must use TPM to perform all measurements. However since PCR 17 is not reset to zero (this may be done by executing SKINIT instruction only) the resulting PCR value will not match the one calculated by the user on the trusted device.

If malware executes SKINIT instruction to reset PCR 17 to zero value then hardware securely passes control to the SL which in turn securely transfers it to the hypervisor thus not allowing malware to interpose on the control transfer. Malware can modify SL or hypervisor image to regain control after security system has been activated but then these modifications will be caught by the user upon validating the final measurement displayed on the screen.

IV. Security system deactivation process

The deactivation of security system implies removing hypervisor from the memory, bringing back primary OS to the bare hardware and returning all reserved resources to it. So this procedure may start only as long as hypervisor has a proof that deactivation command originates from the user that has disconnected network cable from the computer. We propose the following protocol for the deactivation procedure:

1) User disconnects network cable and executes a program that requests hypervisor to deactivate security system.
2) Hypervisor asks TPM to generate nonce using `TPM_GetRandom` operation which is then passed to the user.
3) User signs this nonce on a trusted device using his own private key and returns signed nonce to the hypervisor.
4) Hypervisor validates users signature with the public user key known to him, compares original nonce with the one received from the user and as long as they are equal deactivates security system.

While completing deactivation procedure hypervisor simply stops service VM. There is no information in this VM that must be persistently stored. Then hypervisor upon next VM exit event of the primary VM rewrites CPU registers with the VM state stored in its VMCB structure and resumes primary OS execution without entering virtual machine mode. Finally driver in the service OS returns previously reserved hardware resources back to the primary OS by hot plugging CPU, memory banks and reloading original network adapter driver.

V. Conclusion

In this article we have presented an approach to on-the-fly activation and deactivation of virtualization based security systems. We target this approach on the systems that allow discrete functioning without violating stated security properties. Among such systems are hypervisor-based solutions that secure user applications running under the control of untrusted operating system. Both activation and deactivation procedures consider the presence of malware on the computer and provide means for attesting security system being activated and for performing safe deactivation correspondingly.

In our approach the hypervisor is started on the fly from the up and running operating system upon the user request. Operating system and all running user applications are transparently moved inside the hardware virtual machine

where they continue executing but from now under the control of the launched hypervisor and security system implemented inside it. If necessary additional virtual machines may be started at this moment. The security system shutdown procedure is performed upon the user request too with hypervisor being unloaded from the memory and operating system brought back to the bare hardware. Both activation and deactivation procedure do not require rebooting the computer.

Currently we are working on implementing proposed approach for the security system described in [7].

References

[1] R. Riley, X. Jiang, and D. Xu, "Multi-aspect profiling of kernel rootkit behavior," in *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*. New York, NY, USA: ACM, 2009, pp. 47–60.

[2] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[3] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems," in *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2008, pp. 2–13.

[4] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2006, pp. 2–13.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2003, pp. 164–177.

[6] R. Ta-Min, L. Litty, and D. Lie, "Splitting interfaces: making trust between applications and operating systems configurable," in *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*. Berkeley, CA, USA: USENIX Association, 2006, pp. 279–292.

[7] I. Burdonov, A. Kosachev, and P. Iakovenko, "Virtualization-based separation of privilege: working with sensitive data in untrusted environment," in *VDTS '09: Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems*. New York, NY, USA: ACM, 2009, pp. 1–6.

[8] J. Rutkowska, "Subverting vistaTMkernel for fun and profit." BlackHat, 2006. [Online]. Available: http://www.blackhat.com/presentations/bh-jp-06/BH-JP-06-Rutkowska.pdf

[9] S. Garriss, R. Cáceres, S. Berger, R. Sailer, L. van Doorn, and X. Zhang, "Trustworthy and personalized computing on public kiosks," in *MobiSys '08: Proceeding of the 6th international conference on Mobile systems, applications, and services*. New York, NY, USA: ACM, 2008, pp. 199–210.

[10] *AMD Architecture Programmer's Manual Volume2: System Programming*, Advanced Micro Devices Inc., November 2009.

[11] *AMD-VTMNested Paging*, White paper, Advanced Micro Devices Inc., July 2008.

[12] "Trusted platform module," Trusted Computing Group. [Online]. Available: http://www.trustedcomputinggroup.org/resources/tpm_main_specification

[13] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide*, Intel Corporation, March 2010.

[14] B. Kauer, "Oslo: improving the security of trusted computing," in *SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2007, pp. 1–9.

[15] R. Wojtczuk, "Subverting the xen hypervisor." BlackHat, 2008. [Online]. Available: http://blackhat.com/presentations/bh-usa-08/Wojtczuk/BH_US_08_Wojtczuk_Subverting_the_Xen_Hypervisor.pdf

[16] *AMD I/O Virtualization Technology (IOMMU) Specification*, Advanced Micro Devices Inc., February 2009.

The Modern Educational Course on Agile Software Development

Evgeny Sorokin
Lobachevsky State University of Nizhni Novgorod, Russia
e-mail: evgeny.sorokin@inbox.ru

Kirill Kornyakov
Lobachevsky State University of Nizhni Novgorod, Russia
e-mail: kirill.kornyakov@gmail.com

***Abstract* — the article presents new educational course dedicated to Agile Software Development. The course consists of theoretical and practical parts. Theoretical part gives an overview of classical agile methodologies, widely accepted by industry practices and some important programming principles and patterns. The second part of the course includes several labs, aimed for practical usage of agile development principles and patterns. Major feature of the course is its practical orientation and focus on agile ideas accepted by the industry.**

Keywords – agile; software development; software engineering education.

I. INTRODUCTION

Agile methodologies are a trend nowadays. The most appropriate way to work in startup companies is to adopt software process, which makes team responsive to market. The same situation is observed within large companies sometimes [1, 2, 3]. Heavy software processes are usually good for long-term projects with well-defined requirements. Small projects and teams prefer lightweight techniques, which can help them work effectively in uncertain conditions.

It is most likely that young software engineer will participate in an agile software project. Regardless of this, every developer need to now basic agile development principles and practices, which help to create high-quality, maintainable and flexible software systems. For example, *refactoring* and *continuous integration* practices are de-facto standard for now, SOLID object-oriented design (OOD) principles are also very important for quality of software.

We have created a modern Agile Development educational course. The goal of the course is to create a set of materials suitable for training students (and developers) on agile concepts and practices. This is the list of major course features:

- General overview of Agile methodologies;
- Detailed discussion of the Scrum and eXtreme Programming (XP) methodologies;
- Overview of architectural styles;
- Introduction to Domain-Driven Design (DDD);
- Practical explanation of SOLID principles;
- Practical exercises on Test-Driven Development (TDD) technique.

The rest of the paper is organized as follows. In section 2 we discuss the purpose of the course. Section 3 and 4 give course overview and information about course evaluation. Benefits to the Software Engineering community are discussed in section 5, section 6 concludes the paper.

II. COURSE PURPOSE

Agile methodologies are a growing trend for software companies nowadays. Many and many projects use Scrum, XP, Kanban etc. But there is no complete course in academic program, which covers agile methodologies in such a way so that student can learn key principles and practices of that domain. At present time we can see numerous materials about Software Engineering (SE) and Agile Development particularly. Most of them can be subdivided into the following categories:

- Classical Software Engineering courses [4, 5, 6].
- Pioneering books about Agile Development [7, 8, 9] and their followings.
- Commercial training courses by consulting companies for developers and managers.
- Extensive research on applicability of agile practices in various environments.

We can see strong attention to the Agile methods from both industry (software development and consulting companies) and academy. But there is still a lack of free Agile Development courses with strong practical orientation. There is now way for beginner developer to understand agile programming practices except joining an agile team.

Such situation results in mutual disappointments of employer and employee during the first years of software engineer's career. It also results in a number of failures in startup projects because of the ignorance of ideas and recommendation of agile evangelists.

Another point is that classic publications about Agile don't reflect corrections made by industrial usage and patterns of adaptation. There are lots of materials in the Internet, which explains concepts of Agile but there are no free stand-alone workshops aimed for complete overview of

Agile world and strong practicing of agile techniques in software projects.

So, there is a demand for a modern and open course, which can be used for training students and software developers.

III. Course Description

To simplify and facilitate the entering into industrial software development the agile software development course has been elaborated. Below we combine the description of the course and the experience report of its conduction. The important point is that we encourage senior engineers from Nizhni Novgorod Hi Tech companies to participate in that course as guest lecturers. This year we have invited IT specialists from Intel [10] and Itseez [11], who have explained materials from their day-to-day work. This has a serious value to the audience because it, on the one hand, demonstrates that the material is being applied in the real world and, on the other hand, exposes the tactical and practical aspects.

The course provides complete overview of the agile software development world: Agile Manifesto, detailed description of the most authentic methodologies — Scrum and XP, brief introduction to software adaptation of Kanban (Toyota Production System) and Lean philosophy.

The second part of the course contains introduction to software architecture and design. Again, the course approaches this topic from the point of practice: instead of UML diagrams lectures demonstrate the examples of real-life software and/or scalable approaches of architectural decision making. The specifics of the materials chosen from the Software Engineering are based on real industry demand. Many IT companies in Russia provide outsourcing of business application development. So as authors of that course we made the analysis of most popular practices adopted by the IT community. We have conducted several interviews with representatives of different software companies of Nizhni Novgorod to define their needs.

The stories about success and failures of real adoption of agile methodologies from practitioners are supported by set of handouts and homework. The practical exercises are dedicated to the different programming practices (refactoring, Test-Driven Development (TDD), SOLID principles of OOD, application patterns, Domain-Driven Design (DDD)).

DDD is a modern approach to software development by a structure of practices and terminology. DDD helps to make architectural and design decisions for software projects dealing with complex domains.

TDD is a software development technique which requires automated unit test to be written *before* producing the code which passes that test. This technique influences design solutions (makes them simple and flexible) and provides validation of correctness.

The practical exercises helped students to feel deeply do's and don'ts of practical coding of application in uncertain conditions. The requirements to the lab work were general and amended based on concrete solution with intent — to practice refactoring to patterns, to let students see the value of TDD and to educate dependency management.

Another point is that students used the same tools, which usually being used in software companies. The labs were created in Microsoft Visual Studio with C# programming language. For refactoring audience used ReSharper from JetBrains company [12]. It matters because it helped audience to increase their productivity and leave mechanical routine of working with syntax to the tools. It is extremely important for adopting TDD for example.

The tasks of lab work were designed in the way that students could complete the product. For example, as a result of TDD task students have DLL, which implements something useful and a set of unit tests, which supports the implementation. As a result of lab on Passive View pattern [13] students have complete Windows- or Web- application with architectural layers and another set of unit tests for validation of the presentation logic.

IV. Course Evaluation

Currently we have a strong support from lecturers, engineers and managers from UNN [16], Itseez, Intel, ITLab [14].

- Nizhni Novgorod State University facilities are used as a basis for initial course approbations. The course has been conducted at the Master's course of Faculty of Computational Mathematics and Cybernetics of State University of Nizhni Novgorod.
- The course will be tested at Winter and Summer Schools (annual sessions) at ITLab.
- Members of ITLab Seminar on Software Engineering [15] offer their help for course materials review and discussions.
- Several IT companies of Nizhni Novgorod suggested their free consulting services and opportunity to test effectiveness of course materials for their beginner developers training.

Among the results of the course we can emphasize the following:

- Students acknowledge and appreciate the part of the course, which covers process practices. Examination has demonstrated that key concepts of agile software development were easy to perceive.
- TDD lab work has been the one of the most "heavy to learn" practices. Students have often considered TDD as just writing of unit tests.
- Students who have applied all lab works to a single software project appeared higher results than those who have created new projects for every lab work.
- The materials of the engineering part of the course, which haven't been covered by lab work (such as DDD) were not clear to the audience.

V. BENEFITS TO THE SOFTWARE ENGINEERING COMMUNITY

The course brings the following benefits to the SE community:

- Course materials will be available online for free. Community will get the tool for educating students and developers for agile concepts and techniques.
- Graduates will have strong agile development knowledge, adequate skills for fast integration into software development process, ability to adopt agile methodologies and build up a unique software process for small and medium sized software projects. We expect to see developers with deep understanding of what Agile is, and what it's not. We hope that students will be ready for joining agile teams and initiating their own agile projects.
- This course will be valuable for the whole Russian-speaking software development community. Especially because there is very few information on Agile in Russian language.

VI. CONCLUSION

Initial version of modern Agile Development course have been created and evaluated as Master's course at the State University of Nizhni Novgorod, Faculty of Computational Mathematics and Cybernetics. Feedback was positive in general both from students and evaluating experts.

We are looking forward to hear from all whom this course may concern, especially lecturers who also would like to develop this course together with us, IT companies who would like to conduct it for their employees (or to participate as guest lecturers) and for anyone who would like to review our course and help us to improve it.

REFERENCES

[1] The Dancing Agile Elephant: IBM Software Group's Transition to Agile and Lean Development http://www.infoq.com/presentations/dancing-agile-elephant

[2] Agile adoption at Google: Potential and challenges of a true bottom-up organization http://www.agile2007.org/agile2007/index.php%3Fpage=sub%252F&id=713.html

[3] The Microsoft Solutions Framework: An Integrated Approach to Agile or Formal Software Development Process http://blogs.msdn.com/askburton/articles/330974.aspx

[4] Ian Sommerville, Software Engineering, 8 ed., Addison Wesley, 2006.

[5] R. Pressman, Software Engineering: A Practitioner's Approach,McGraw-Hill, 2001.

[6] S. McConnell, Rapid Development: Taming Wild Software Schedules, Microsoft Press Books, 1996.

[7] A. Cockburn, Agile Software Development. Reading, Massachusetts: Addison Wesley Longman, 2001.

[8] K. Schwaber and M. Beedle, Agile Software Development with SCRUM. Prentice-Hall, 2002.

[9] K. Beck, Extreme Programming Explained: Embrace Change, Second ed. Reading, Massachusetts: Addison-Wesley, 2005.

[10] http://www.intel.com

[11] http://www.itseez.com

[12] http://www.jetbrains.com

[13] http://martinfowler.com/eaaDev/PassiveScreen.html

[14] http://itlab.unn.ru

[15] http://groups.google.ru/group/itlab_se?hl=ru

[16] http://unn.ru

Programming as a part of the Software Engineering education

Maksimenkova Olga
State University Higher School of Economics
Moscow, Russia
e-mail: omaksimenkova@hse.ru

Vadim Podbelskiy
State University Higher School of Economics
Moscow, Russia
e-mail: vpodbelskiy@hse.ru

Abstract — **Programming for the first-year undergraduates starts as a part of "Computer science" academic subject. Some traditional methods of teaching programming are popular in higher education in Russia nowadays. A different method, which is used as a part of Software Engineering education in our University, is described in this article.**

Keywords: software engineering; software engineering education; computer science

I. Introduction

Software engineering is a modern and complicated field of knowledge. Following the SWEBOK it contains such knowledge areas as software requirements, software design, software construction, software engineering process and so on. A place of our teaching interests laying in the software construction part. The term software construction refers to the detailed creation of working, meaningful software through a combination of coding, verification, unit testing, integration testing, and debugging [1]. As we can see coding is associated with the other fields of knowledge. In any training course, which is connected with software engineering, a teacher should provide and explain this connection at any stage of teaching.

The academic subject "Computer science" within the bounds of Software engineering education is a significant brick to train specialists in this field.

II. Structured programming languages as a base

"Computer science" as a course is traditionally widely connected with coding, algorithms and data structures [2]. Traditional higher education in Russia supposes one of the structured programming languages such C or Pascal to be taught to the first-year students.

Of course, such languages as C play a significant part in further education, because of that the most of modern programming languages (Java, C++, C# etc.) are based on theirs syntax. Some students, however, have an experience in structured programming from schools and special lyceums, but they should repeat lots of basic concepts at first time. Therefore, they lose interest to the main course and, for example, teachers need to prepare problems for all and for the advanced listeners to keep their interest. If advanced listeners had to solve easy problems in the beginning they couldn't have concentrated further on the new for them concepts of course.

III. Object-oriented programming languages as a base

The main idea of our course is to teach first-year students to use theory in the field of information technologies and programming skills to describe algorithms using one of the up-to-date programming languages. Graduating student should have "live" knowledge, which he can successfully use in his day-to-day activity.

Object-oriented technology is widely used in software design and development nowadays. So it should have been quite reasonable to use one of modern object-oriented programming languages as a base for course. That's why we chose C# for our first-year students. C# specification supposed C# to be a simple, modern, general-purpose, object-oriented programming language [3].

IV. Target audience

"Computer science" is lectured during an academic year to the first-year undergraduates. The most of our students come to the University just after graduation from secondary schools, so we expect basic school knowledge in the field of Math and Computer science. In reality, only mathematical skills are more or less equal. As for basic skills of our students in computer science, they are quite different. Mainly just because of absence government curriculum for the Computer science in schools. Some of our students, for example, lyceums-graduates or programming competition winners have an experience in programming. The others have basic school knowledge or haven't got any special knowledge in "Computer science" at all. Anyway, each other should be involved and should be given a chance to be a success.

V. Theory and practice

The whole course of studies consists of two widely connected parts. First of them is the course of lectures and the second one is practical training.

Course of lectures is given to students during an academic year, which is approximately 86 academic hours, as it postulates in the curriculum. Students also take practical trainings, which are given during 88 academic hours following the curriculum.

Curriculum also specifies self-instruction during 204 academic hours for the "Computer science" course.

Course of lectures contains not only theoretical material but general practical examples with significant algorithms

and data-structures, as well. Theoretical concepts of structured programming and basic rules of object-oriented programming are also included into the course of lectures.

Practical trainings consist of set of C# examples and a list of problems to self-instruction. Theoretical material from the course of lectures meets its practical application on the practical trainings. Students consolidate knowledge and gain programming experience during these lessons.

We use multimedia means to represents material and make our lessons more interactive and efficient. Materials of practical trainings with a home task are sent to the students every week after lessons.

VI. TEST CHECKS

Test checks are represented by class written tests, home written tests, computer-based module-tests and yearly project.

Written tests involve one or two short problems to be done using Microsoft Visual Studio 20xx at class. Students write their programs for 40-60 minutes, it depends on difficulty of an introduced written test. We appraise written tests using special criteria. In them such characteristics as correspondence of a program with a task, functionality of a program, its failure stability and so on are taken into consideration. So, students not only take skills in programming and algorithmization, but learn basic maxims of development like constructing functionality programs in limited time.

Home written test is unassisted work of students. In which they should implement a program and provide it with simple documentation. Variant of a home written test is more complicated as opposed to class written test variant. It can contain, for example, wider class structure, more difficult collaboration between classes, user's interface design and data processing layer development. Results, which are reached during the home work, are a great help for students in yearly project (see "Yearly project" topic).

Our school year is divided into five modules – 7 weeks each. So students take a computer-based module-test five times a year during a test period. Each test consists of 30-40 questions of the different types. Usually we use multiple choice questions, multiple response questions and short-answer questions. Students are tested during 40 minutes. Topics of the test problems involve theoretical and practical questions of a current module.

VII. YEARLY PROJECT

Yearly project isn't completely included into the academic subject "Computer science". But it should represent a working application with a documentation package. In yearly project our University carries out an interdisciplinary approach in education. Such a project, obviously, do a lot of good for our students, because they gain not only programming experience, but obtain experience in such fields of software engineering as software requirements, software design, software construction, etc.

Within a "Computer science" academic subject we offer our student a simple method of how to develop a program easily. This method has grown up from an Agile software development methodology, to be more precise, from Extreme Programming. Extreme Programming, XP for short, is an Agile software development methodology that is made up of a collection of core values, principles, and practices that provide a highly efficient and effective means of developing software [4].

For sure, we didn't have a goal to teach first year students such complicated methodologies. We only used some main ideas of them to answer a hidden question: "How to develop a yearly question?" We provided our first-year undergraduates with a method which helps them to develop a huge application without lots of mistakes. We for ourselves called this method "Evolutional approach".

VIII. EVOLUTIONAL APPROACH

Every beginner programmer, like our first-year undergraduates, has a variety of troubles in making programs. Efficient but complicated software development methodologies are not suitable for him. Troubles in development may lie even in choosing algorithm or standard library. Sometimes, beginner programmer starts to develop an application without realizing what kind of means or structures will be needed for its implementation. Each beginner programmer is a software architect, developer and tester at the same time. Our main task, as tutors and teachers, is to help beginner programmers to solve all the problems in a way of implementation their applications.

Evolutional approach, following the traditional software development methodologies, gives next recommendations:

1. Development should be carried out by steps, each of those contains: design (including interface design), coding (including syntactic and semantic debugging) and testing.
2. Stages of development informally divide into two types: research stages and development stages, which give software new functionality according to a requirements specification.
3. In spite of type, the first stage provides creation of working software with the minimum functionality.
4. The analysis of a current version of software is carried out at the end of the each stage of development. A goal of analysis is to assess possibility of adding constructions for implementation new, additional requirements according to a requirement specification.
5. On basis of analysis results refactoring of a current version is taken place or project decisions are accepted, coding and testing are carried out.
6. Software, staying in up state, obtains new, or changes current functionality at the end of each stage.

Thereby, software every time stays in up state, and changes makes permanently in it. In the other words, software is evolving from step to step and its functionality is growing from minimum to wishful (given in requirement specification).

IX. Marks

It should be said that in our University we use ten marks from 1 to 10. Marks less than 4 are unsatisfactory, 4 and 5 are satisfactory, 6 and 7 are good, 8, 9 and 10 are excellent marks.

Modular and total marks are accumulative. They are calculated using special weighting coefficients for each current mark. As far as all the marks are quite complicated we are giving, as following, formulas we have used this year:

A. Module 1, 2, 5, current marks:

$$M_i = min(\kappa_i, \tau_i), \text{ if } \kappa_i < 4 \text{ or } \tau_i < 4,$$

else

$$M_i = 0.5 * \kappa_i + 0.5 * \tau_i,$$

there i =1, 2, 5 – number of modules; κ – written test mark, τ – computer-based test mark.

B. Total mark after module 2:

$$\Theta_2 = 0.4 * M_1 + 0.6 * M_2,$$

there M_i – modular current mark, i = 1, 2.

C. Module 3, 4, current marks:

$$M_i = min(\kappa_i, \tau_i, \zeta_i), \text{ if } \kappa_i < 4 \text{ or } \tau_i < 4 \text{ or } \zeta_i < 4,$$

else

$$M_i = 0.3 * \kappa_i + 0.4 * \tau_i + 0.3 * \zeta_i$$

there i = 3, 4, κ – written test mark, τ – computer-based test mark, ζ – home written test mark.

D. Total mark after module 5:

$$\Theta_5 = 0.3 * M_3 + 0.35 * M_4 + 0.35 * M_5,$$

there M_i – modular current mark, i = 3, 4, 5.

E. Total yearly mark:

$$\Theta_y = min(\tau_t, \kappa_t), \text{ if } \tau_t < 4 \text{ or } \kappa_t < 4,$$

else

$$\Theta_y = 0.3 * \Theta_5 + 0.35 * \tau_t + 0.35 * \kappa_t,$$

there Θ_5 – total mark after module 5, τ_t – total computer-based test mark, κ_t – total written test mark.

As it is shown in formulas, each mark has a special threshold value of current marks. Usually written test mark is one of them, the others are depended on a structure of a module. If one of threshold values is unsatisfactory the whole accumulative mark should be unsatisfactory, even though all the others are satisfactory.

X. Results and Statistics

Described way of teaching programming within the bounds of “Computer Science” is taught by authors for almost three years.

Some statistics for second, fifth modules and yearly project of the last two years is following. We don’t give statistics for a current year because of it incompleteness.

We didn’t use computer-based modular test in 2007-2008 academic year Fig. 1.

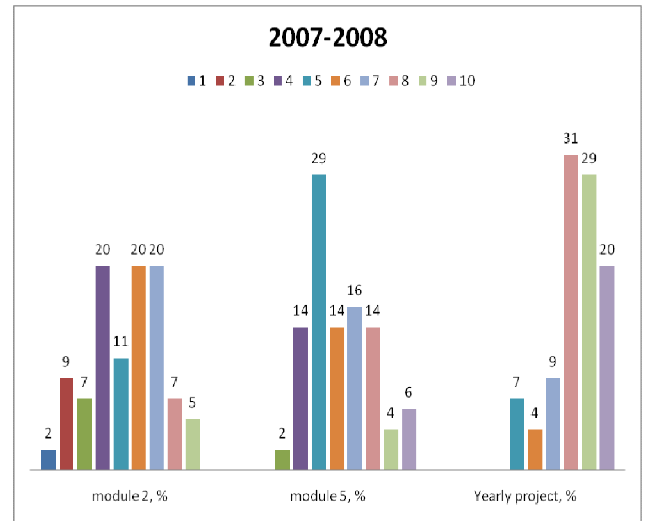


Figure 1. Statistics 2007-2008 acad. year.

We started using computer-based modular test as a test checks in 2008-2009 academic year Fig. 2.

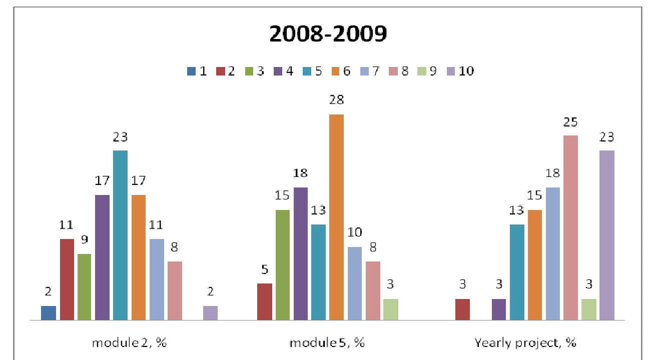


Figure 2. Statistics 2008-2009 acad.year

Conclusion

This work describes educational measures in the field of teaching programming as a part of Software engineering education within the bounds “Computer science”. All these measures are successfully taken for three years in State

University Higher School of Economics by Software engineering department.

REFERENCES

[1] SWEBOK. Software Engineering – Guide to the Software Engineering Body of Knowledge (SWEBOK). First edition, - Geneva (ISO/IEC 19759: 2005(E)). – 173 pp.

[2] Lethbridge, T.C., Leblanc Jr., R.J., Kelley Sobel, A.E., Hilburn, T.B., Diaz-Herrera, J.L. SE2004: Recommendations for undergraduate software engineering curricula (2006) *IEEE Software*, 23 (6), pp. 21-25

[3] ECMA-334. C# Language Specification. 4th Edition / June 2006, - Geneva (ISO/IEC 23270:2006). – 553 pp.

[4] G. Pearman, J. Goodwill, “Pro .NET 2.0 Extreme Programming”, Berkeley: apress, 2006, pp. 3 – 17.