Generating environment model for Linux device drivers

Ilja Zakharov
ISPRAS
Moscow, Russian Federation
Email: ilja.zakharov@ispras.ru

Vadim Mutilin
ISPRAS
Moscow, Russian Federation
Email: mutilin@ispras.ru

Eugene Novikov
ISPRAS
Moscow, Russian Federation
Email: novikov@ispras.ru

Alexey Khoroshilov
ISPRAS
Moscow, Russian Federation
Email: khoroshilov@ispras.ru

***Abstract*— Linux device drivers can't be analyzed separately from the kernel core due to their large interdependency with each other. But source code of the whole Linux kernel is rather complex and huge to be analyzed by existing model checking tools. So a driver should be analyzed with environment model instead of the real kernel core. In the given paper requirements for driver environment model are discussed. The paper describes advantages and drawbacks of existing model generating approaches used in different systems of model checking device drivers. Besides, the paper presents a new method for generating model for Linux device drivers. Its features and shortcomings are demonstrated on the basis of application results.**

Keywords—operating system; Linux; kernel; driver; model checking; environment model; Pi-processes

I. INTRODUCTION

Linux kernel is one of the most fast-paced software projects. Since 2005, over 7800 individual developers from almost 800 different companies have contributed to the kernel. Each kernel release contains about 10000 patches - work of over 1000 developers representing nearly 200 corporations [1]. Up to 70% of Linux kernel source code belongs to device drivers, and more than 85% errors, which lead to hangs and crashes of the whole operating system, are also in the drivers' sources [2] [3].

A. Linux device drivers

The Linux kernel could be divided into two parts - core and drivers (look at Fig. 1). Drivers manage devices and the kernel core is responsible for a process management, memory allocation, networking and et al.

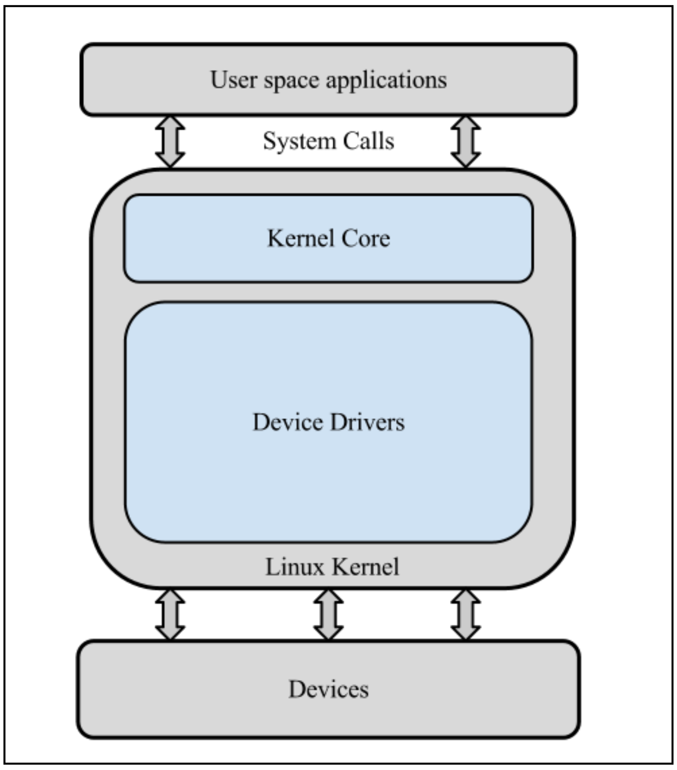

Fig. 1. Device drivers in the Linux kernel.

Most of drivers can be compiled as modules that can be loaded on demand. Drivers differ from common C programs. Drivers do not have a main function and a code execution order is primarily determined by the kernel core. Let us describe driver organization by considering a simplified example of a driver in Fig. 2:

- **Driver initialization function** (the function *init* below). A module of a driver is loaded on demand by the Linux kernel core when the operating system starts or when a necessity to interact with a corresponding device occurs. A module execution always begins with an invocation of a driver initialization function by the kernel core. In the Fig. 2 the initialization driver function is *usbpn_init*.

- **Driver exit function** (the function *exit* below). An interaction with the device is allowed until the module is unloaded. This happens after an invocation of a driver *exit* function by the kernel core. The function *usbpn_exit* is such function in Fig. 2.
- **Driver Handlers**. Various driver routines are usually implemented as callbacks to handle driver-related events, e.g. system calls, interrupts, et al. There are two handlers in the example in Fig. 2: *usbpn_probe* and *usbpn_disconnect*.
- **Driver structures** (we will call them just "structures" from now on). Most of handlers that work with common resources consolidated in groups. Each handler in such a group implements certain functionality defined by its role in this group. Usually pointers to handlers from one group stored in fields of a special variable with complex structure type. That is why we identify such groups as "driver structures". In example in Fig. 2 *usbpn_driver* is the driver structure with *usb_driver* type. It has two fields "*.probe*" and "*.disconnect*" initialized with pointers to *usbpn_probe* and *usbpn_disconnect* handlers.
- **Registration and deregistration of handlers**. Before the kernel core can invoke handlers from the module, they should be registered. The typical way to register driver handlers is to call a special function. The function registers the driver structure with handlers and since the structure is registered, its handlers can be called. The driver structure registration takes place in the driver initialization (in *init* function body) or in an execution of a handler from another structure. An example of registration of *usb_driver* structure is illustrated in Fig. 2: *usb_register* is called in *usbpn_init* function body and it registers *usbpn_driver* structure variable. Also similar deregistration functions are implemented for the handler deregistration.

Even this simplified example illustrates the complexity of device drivers. A lot of driver methods are called by the kernel core such as handlers, *init* and *exit* functions and there are routines from the kernel core that are invoked by the driver such as register and unregister functions and other library functions. Besides, interaction of the kernel core and the driver depends on system calls from the user space and interrupts from devices. Their large interdependency with each other leads to availability of almost arbitrary scenarios of handler calling. But in all of such scenarios rules of correctness are taken into account such as restrictions on order, parameters and context of handler invocations.

B. *Model checking Linux device drivers*

Nowadays it is not easy to maintain the safety of all device drivers manually due to complexity of drivers, high pace of the Linux kernel development and huge size of source code. That is why an automated driver checking is required. There are various techniques for achievement this goal and a model checking approach is one of them.

```
static int usbpn_probe(struct usb_interface *intf, const
struct usb_device_id *id){
…
}
static void usbpn_disconnect(struct usb_interface *intf){
…
}
static struct usb_driver usbpn_driver = {
            .name =             "cdc_phonet",
            .probe =            usbpn_probe,
            .disconnect =       usbpn_disconnect,
};
static int __init usbpn_init(void){
            return usb_register(&usbpn_driver);
}
static void __exit usbpn_exit(void){
            usb_deregister(&usbpn_driver);
}
```

Fig. 2. A simplified example of a driver drivers/net/usb/cdc-phonet.c[1] (compiled as cdc-phonet.ko module).

As illustrated before, a driver execution depends on the kernel core. But analysis of a driver together with the kernel core is rather difficult nowadays for tools due to complexity of kernel core source code and its huge size. That is why a driver environment model is required for analyzing device drivers. The model can be implemented as C program that emulates interaction of the driver with the kernel core. In general the model should emulate the interaction with hardware too, but this aspect is not considered in this paper. The driver environment model should provide:

- Invocation of the driver initialization and exit functions.
- All available in the real interaction of the kernel core and the driver scenarios of invocations of handlers taking into account:
    - Limitations on parameters of handler calls.
    - A context of handler invocation such are interrupts allowed or not.
    - Limitations on order and number of invocations of handlers for:
        - Handlers from a driver structure.
        - Handlers from different driver structures.
- Models for kernel core library functions.

An incorrect model often causes a false positive verdict from a verification tool (*verifier* below) or a real bug skipping

[1] http://lxr.free-electrons.com/source/drivers/net/usb/cdc-phonet.c?v=3.0

[4]. An example of environment model for the driver considered above is shown in Fig. 3:

```
void entry_point(void){
        // Try to initialize the driver.
        if(usbpn_init())
                goto final;
        // The variable shows usb_driver device is probed
        // or not.
        int busy = 0;
        // For call sequence of handlers of any length.
        while(1){
                // Nondeterministic choosing
                switch(nondet_int()){
                        case 1:
                           // The device wasn't probed.
                           if(busy == 0){
                              res = usbpn_probe(..);
                              if(res == 0){
                                  busy = 1;
                              }
                           }
                        break;
                        case 2:
                           // The device was already
                           // probed.
                           if(busy == 1){
                              usbpn_disconnect(..);
                              busy = 0;
                           }
                        break;
                        case 3:
                           // Try to unload the module
                          // if the device wasn't probed.
                           if(busy == 0){
                              goto exit;
                           }
                        break;
                        default: break;
                }
        }
        // Unload driver.
        exit: usbpn_exit();
        final:
}
```

Fig. 3. Environment model for the driver from Fig. 2.

A verifier starts the driver analysis from the function *entry_point*. First the driver should be initialized by the function *ubpn_init*. If it returns success result "0", then *usbpn_probe* and *usbpn_disconnect* are invoked. The variable *busy* is needed for calling handlers in the proper order. The handler *usbpn_probe* should be called the first and if it returns success result, then *usbpn_disconnect* should be called. Operator *while* is needed for call sequences of handlers of variable length. Operator *switch* with the function *nondet_int* returning random *int* provides non-deterministic handler calling for various scenarios of handler invocations covering in the. At the end of a driver work *usbpn_exit* should be called, but it can take place only if the device wasn't probed, either when *usbpn_probe* wasn't called or when *usbpn_probe* returned an error value or when *usbpn_probe* and *usbpn_disconnect* were already called one or more times. After the *exit* invocation a verifier finishes analysis.

II. RELATED WORK

There are several verification systems for device drivers, but only Microsoft SDV [5] is in industrial use. For modeling driver environment various approaches are used.

- **Microsoft SDV.** SDV provides a comprehensive toolset for analysis of source code of device drivers of Microsoft Windows operating system. These tools are used in the process of device driver certification, and have been included in Microsoft Windows Driver Developer Kit since 2006. SDV's driver environment model is based on manually written annotations of handlers. SDV provides a kernel core model that contains simplified stubs of some kernel core routines. Microsoft SDV is specifically tailored for analysis of device drivers of Microsoft Windows. Unfortunately, it is proprietary software, which prohibits its application to other domains outside Microsoft.
- **Avinux** [6]. This project was developed in University of Tubingen, Germany. Its environment model is based on handwritten annotations of each handler. Authors paid attention to the problem of proper initialization of various resources and uninitialized pointers in the environment model [7].
- **DDVerify** [8]. The project was developed in Oxford and Carnegie Mellon universities. Authors implemented a partial kernel core model for a special kernel version for verifying drivers of several types. But the model is handwritten and maintaining it manually is complicated while the kernel is under continuous development.
- **LDV** [9]. LDV framework for driver verification is developed in Institute of System Programming of Russian Academy of Sciences. This project took a high pace of the Linux kernel development. An environment model generation process is fully automated and does not need manual annotations in code. It is based on an analysis of the driver source code and on a configuration. The configuration consists of handwritten specifications for several driver structures and a heuristic template for other cases. Generated model provides nondeterministic handler call sequences, interrupt handlers invoking. A model can be generated for a driver module from any subsystem and in most cases it correctly describes interaction of a driver with the kernel core.

The lack of such sufficient handicaps as a demand for handwritten annotations or the difficulty of model maintaining allows to efficiently using LDV for verifying all kernel drivers from a lot of kernel releases. However, a driver environment generator has considerable limitations:

- Only linear handler call sequences are available, where each handler can be called only once.
- Driver structures registration and deregistration are not taken into account in model generation process.
- Source code analysis is based on regular expressions. This approach leads to syntactic mistakes in a model due to changes in the kernel and complexity of the kernel source code.
- Not all needed restrictions on handler calls can be described in the configuration.
- For a driver module that consists of several files the tool generates separate models for each ".c" file but not the one for the whole module.

Such shortcomings lead to incorrect verdict from a verifier or real bugs missing. And in most cases the tool doesn't provide any capabilities for overcoming model imperfectness. This paper suggested a new approach for generating driver environment model.

III. Suggested Approach

The main goal of this research was to develop a new tool for automatically generating environment model for kernel driver modules which contain one or several files. An approach suggests environment model that should take into account:

- All available in real driver handler call scenarios from a one driver structure.
- Limitations on order of calling handlers from several driver structures.
- Association of handlers invocation with registrations and deregistration of driver structures.
- Restrictions on handler call parameters.

A new generator should provide full automated generating of environment model for a kernel driver module and facilities for describing restrictions on handler calls in the model.

Moreover the tool should provide additional capabilities for driver environment model debugging, altering generated code and understanding scenarios available in generated C code.

IV. Architecture of the New Driver Environment Model Generator

The design of the new driver environment model generator (DEG) is illustrated in Fig. 4. An input file for DEG is a LDV command stream. This file contains information on build options for compiler, paths to driver and kernel source code, et al. LDV components connects with each other through this file and DEG transforms it during its work. The environment model generation process consists of 3 steps: driver source code analysis, generating of the model in the intermediate representation and printing of corresponding C code. We shall consider these steps below in details.

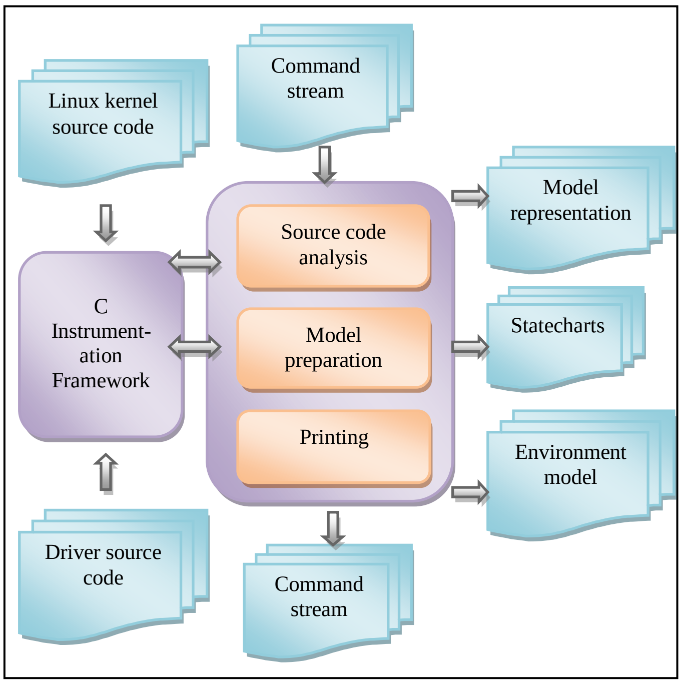

Fig. 4. Design of the driver environment generator.

A. Driver source code analysis

Linux kernel source code is sophisticated and often changing. That is why using analysis based on regular expressions leads to various bugs in generated code of the model or lack of information on source code for model generation. For solving this problem DEG uses C instrumentation framework (CIF below) for source code querying [10]. DEG requests information from this tool about driver source code like initialization and exit procedures, driver structures, library function invocations, et al. Querying process can be divided into two steps:

1) Querying for handlers and driver structures used in the driver.

2) Querying for functions used for registration of these driver structures and other queries based on information extracted at a first step.

After source code analysis it is needed to get additional information on handler call order, handler return values and handler arguments before environment model can be generated. Such the information is stored in a configuration.

B. Internal driver environment model representation construction

Paper [11] designed a formal driver environment model based on Robin Milner's Pi-processes [12]. The model is considered as a parallel composition of Pi-processes. A group of handlers from one driver structure corresponds to a Pi-process. Interactions between such processes are implemented by signals exchanging. Driver structure registration and deregistration are modeled via these signals too. Also this work proposed a method of translating such driver environment model into a multi-threaded C program. And it showed that the translated sequential program reproduces the same traces as available in the initial model via Pi-processes. This result is important because nowadays model checking verifiers don't support multi-threaded C programs analysis but drivers can be executed in several threads.

A DEG configuration is developed for specifying Pi-processes of environment model. The configuration consists of two parts: manually written specifications for several driver structures and patterns for automatically generating such specifications for other driver structures. This design of configuration allows generating Pi-process description for difficult cases using manually written specifications and for other cases using patterns.

New DEG constructs a model representation on the basis of the configuration and data extracted from the source code. The following algorithm for constructing the representation is used:

1) First of all presence of manually written descriptions in configuration is checked for each driver structure that was found in the driver.

2) If such specification for a driver structure is found, it will be adopted for this driver structure taking into account its handlers and registration methods that were founded in the driver source code. This adopted specification is used for modeling the driver structure in the model representation.

3) If a description is not found, then a suitable pattern will be chosen from the second part of the configuration. This pattern will be adopted using heuristics and taking into account the driver structure.

As a result of this stage DEG provides a driver environment model representation with Pi-processes descriptions for each driver structure found in the driver source code and signals that are used for interaction between processes. Representation format almost coincides with the format of the configuration.

For debugging purposes DEG can generate statecharts with available handler invocation scenarios in the generated code. Each statechart illustrates the call handler order for the corresponding Pi-process. Simplified examples of such charts showed in Fig. 5 and Fig. 6. These figures illustrate two graphs for order of calling *init* and *exit* functions and for order of calling handlers from the driver structure with the *usb_driver* type. In the Fig. 5 there are 3 states: "state 0" in which driver wasn't been initialized yet, "state 1" in which driver normally operates and "state 2" in which it is already unloaded. In "state 1" other handlers can be called like it is illustrated in the charts in the Fig. 6: after *init* execution, *usb_driver* structure is become registered, after this event handler *probe* can be called. If the device was probed successfully handler *disconnect* can be called. After *disconnect* invocation *exit* can be called that unregisters *usb_driver* structure.

C. Printing C code of driver environment model

On the last stage DEG translates the driver environment model representation based on Pi-processes into a C code.

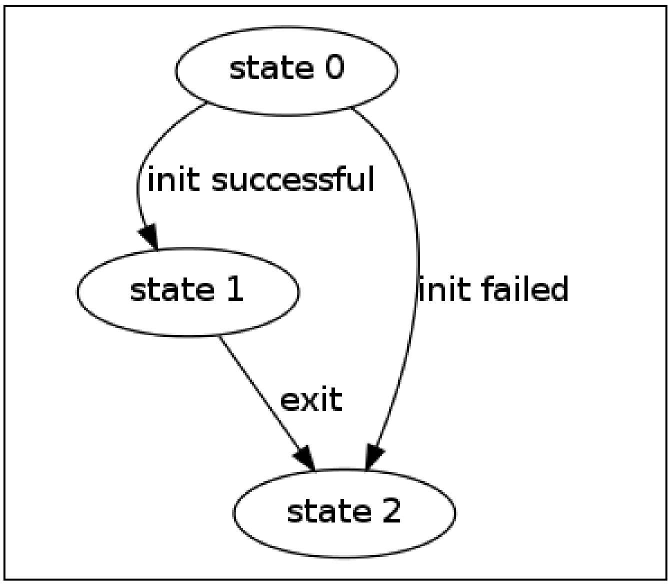

Fig. 5. Simplified example of the statechart for *init* and *exit* functions.

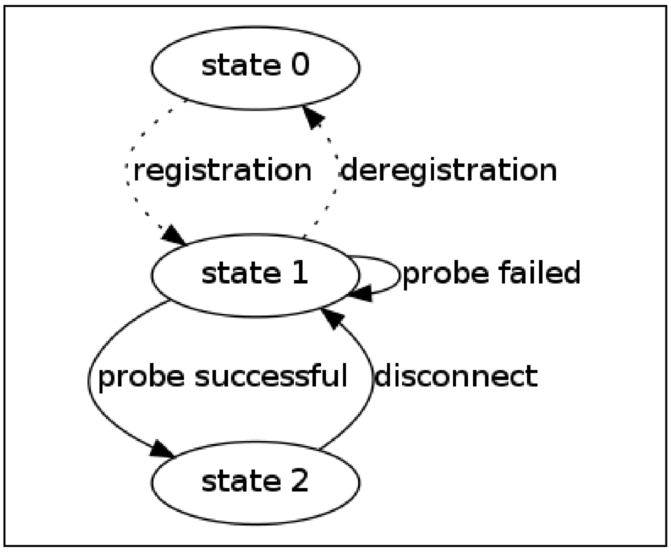

Fig. 6. Simplified example of the statechart for handlers from *usb_driver* driver structure.

Then DEG represents this C code in form of aspect files, which are used by another LDV component Rule Instrumentor. After applying these aspect files by that component, code of the generated model is added to the driver source code and some driver routines are also changed. Added code includes various auxiliary routines, variables and *entry_point* function in which handlers are invoked and from which a verifier starts its analysis.

V. RESULTS

New DEG is still under development, but some results have been obtained already. For a comparison of the new tool with the old one 2672 drivers from the Linux kernel 3.8-rc1 were analyzed. As a model checking verifier BLAST tool was used [13].

Table I illustrates transitions of verification verdicts after switching to the new driver environment model generator. Columns show results of checking of modules for a corresponding error type connected with: blk_requests executing (1); classes, chrdev_regions and usb_gadgets allocating (2); pairing of *module_get* and *module_put* routines (3); using locks (4). The first table line contains the numbers of modules without any exposed errors for both old and new DEG. One of the main goals of development of the new tool was to decrease the number of false positives from a verifier due to incorrect environment model. Progress in this direction is illustrated in the second line. The number of transitions isn't as much as expected due to incorrect work of other LDV components and BLAST tool or time and memory limits (because more resources are needed for proving safety of a driver than for finding an error). Next three lines demonstrate cases with true and false positives from the verifier. The first of these lines illustrates the number of modules whose environment model becomes better. In the next line there are cases with still incorrect environment model. And the last of these 3 lines stores true positives or false positives with the incorrect verdict occurred not due to environment model (for example due to an imperfect pointer analysis by BLAST). Next 2 lines contain the number of absence of the verdict with a new model. In 20% of these cases a reason is limit of memory or time because in some cases new DEG generates a sophisticated and large model. In other cases reasons are various bugs in LDV or in the verifier. Next two lines contain number of cases when an old model had syntax errors in the contrast to the new one. The last line shows cases with LDV or verifier fails despite both new and old environment model because these modules are too huge or just due to bugs in verification system or in the verifier.

TABLE I. VERIFICATION VERDICTS AFTER SWITCHING TO THE NEW DRIVER ENVIRONMENT MODEL GENERATOR.

Transitions		**1**	**2**	**3**	**4**
Safe → Safe		2469	2441	2414	2444
Unsafe → Safe		0	2	5	7
Unsafe → Unsafe	***Model becomes better***	6	3	6	4
	Model is still incorrect	0	1	6	5
	Unsafe is not due to model	0	15	18	5
Safe → Unknown		43	44	46	45
Unsafe → Unknown		0	1	16	4
Unknown → Safe		12	13	10	16
Unknown → Unsafe		0	1	4	1
Unknown → Unknown		142	151	149	141

Proper environment model is one of necessary conditions for obtaining true verdicts. Despite a minor number of transitions from false positives, an experience of using the new generator showed that such incorrectness of an environment model often hides various problems in other LDV components or in verifier. Switching to the new generator explored such problems and allowed to increase quality of driver verification in general.

VI. FURTHER DEVELOPMENT DIRECTIONS

The suggested approach increased quality of generating of driver environment models, but there are the following shortcomings in the current tool that should be solved in future:

- **Configuration extension**. For several types of drivers specifications for driver structures should be written manually in the configuration. The number of driver structures is estimated as two hundreds in the whole kernel. There are 15 described already in the configuration and about 15 are needed to be specified.
- **Interrupts, timers, tasklets modeling**. New DEG doesn't invoke interrupt handlers, timer routines or tasklet callbacks yet. For increasing coverage of code analysis they should be invoked in the new model.
- **Generating model for several modules**. Sometimes an analysis of only one module leads to sophisticated or incorrect environment model, because drivers can contain several modules or common routines from several drivers are picked out to a library module. Thus environment model should be generated for groups of interacting modules rather than for separate modules of these groups.

VII. CONCLUSION

The paper describes the new approach for automatically generating driver environment models for model checking Linux kernel drivers. Also it demonstrates the new version of the component of LDV framework called Driver Environment Generator implementing this approach. The new DEG provides:

- Fully automated environment model generating for drivers that can be compiled as Linux kernel modules. Generating process is based on source code analysis performed by C Instrumentation Framework [10].
- The new configuration for generating process management. This configuration consists of specifications for driver structures and patterns for invoking handlers from other driver structures having an unknown type. The configuration is based on Pi-processes and allows setting various restrictions for handler invocation including restrictions on order and parameters of calling handlers from one or several driver structures.
- Facilities for simplifying work with generated environment models by its representation in

configuration format or statecharts that illustrate order of handler calls.

- Driver environment model as a set of aspect files for applying to the driver source code by LDV component Rule Instrumentor.

Initial experience of the new tool application demonstrated that the new approach allows increasing quality of generated environment models and decreasing the number of false positives from verifiers. Also usability of DEG tool was improved.

The new DEG will replace soon the old one and will be available as component of LDV framework. Information on LDV framework is available on the site of the project http://linuxtesting.org/project/ldv.

References

[1] J. Corbet, G. Kroah-Hartman, A. McPherson., "Linux kernel development. How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It," http://go.linuxfoundation.org/who-writes-linux-2012, 2012.

[2] A. Chou, J. Yang, B. Chelf, S. Hallem, and DR Engler, "An Empirical Study of Operating System Errors," Proceedings of the 18th ACM Symp. Operating System Principles, 2001.

[3] M. Swift, B. Bershad, H. Levy, "Improving the reliability of commodity operating systems," Proceedings of the nineteenth ACM symposium on Operating systems principles, 2003.

[4] D. Engler, M. Musuvathi, "Static analysis versus model checking for bug finding", Proceedings of the 16th international conference CONCUR 2005, San Francisco, CA, USA, 2005.

[5] T. Ball, E. Bounimova, V. Levin, R. Kumar, J. Lichtenberg, "The Static Driver Verifier Research Platform," Formal Methods in Computer Aided Design, 2010.

[6] H. Post, W. Kuchlin, "Integrated Static Analysis for Linux Device Driver Verification," Proceedings of the 6th international conference on Integrated formal methods, Germany, 2007.

[7] H. Post, W. Kuchlin, "Automatic data environment construction for static device drivers analysis," Proceedings of the conference on Specification and verification of component-based systems, USA, 2006.

[8] T. Witkowski, N. Blanc, D. Kroening , G. Weissenbacher, "Model Checking Concurrent Linux Device Drivers," Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, ACM, USA, 2007.

[9] M. Mandrykin, V. Mutilin, E. Novikov, A. Khoroshilov, P. Shved, "Using linux device drivers for static verification tools benchmarking," Programming and Computer Software September 2012, Volume 38, Issue 5, pp 245-256.

[10] A. Khoroshilov, E. Novikov, "Using Aspect-Oriented Programming for Querying Source Code," Proceedings of the Institute for System Programming of RAS, volume 23, 2012.

[11] V. Mutilin, "Verification of Linux Operating System Device Drivers with Predicate Abstractions," Phd's Thesis, Institute for System Programming of RAS, Moscow, Russia, 2012.

[12] R. Milner, "A Calculus of Communicating Systems," Springer-Verlag (LNCS 92), ISBN 3-540-10235-3, 1980.

[13] D. Beyer, T. Henzinger, R. Jhala, R. Majumdar, "The Software Model Checker Blast: Applications to Software Engineering," International Journal on Software Tools for Technology Transfer (STTT), vol. 5, pp. 505-525, 2007.