An architecture of effective discrete-event simulation engine for early validation of avionics systems

Denis Buzdalov
Institute for System Programming
of the Russian Academy of Sciences
Moscow, Russia
Email: buzdalov@ispras.ru

***Abstract*—Nowadays models which are used in the avionics (aviation electronics) development are large and can contain complex behaviour specifications, especially on early stages. This leads to high requirements to simulators which are used for such models analyses. Existing open-source simulators are not applicable or not effective in application on such models. An architecture of a discrete-event simulator using *continuations* approach for avionics models analysis is suggested.**

***Keywords*—discrete-event simulation, avionics, early validation**

I. Introduction

An area of creation and verification of avionics (aviation electronics) and other responsible systems is considered.

Nowadays design processes of such systems cannot be performed without modelling techniques usage. Such models are used not only for creation of systems but also for analysis and validation of them (including early validation). Such analyses are usually performed in an automated way because of the great size of analyzed models. That is why we consider only automated ways of the model analysis.

There are a lot of kinds of analysis. One of division aspects of functional characteristics analyses is a division to static and dynamic analyses. Both types are important but each one has features that doesn't allow a type to oust another one.

So, static analyses usually give guaranteed estimation of model characteristics. Because such estimations are given for the model in general, they usually are pretty pessimistic. Also, new type of estimated characteristic often leads developing of new static analysis method.

Dynamic analyses are used to determine more optimistic estimations. But important feature of such methods is that such estimations are determined only for particular cases. This means that estimated characteristics are not guaranteed to be the same in different environments (input data and non-determinism resolution). This does not allow to use dynamic analysis to reason about a model in general.

A lot of kinds of dynamic analysis exist. They can differ in data, prerequisites and aspects that system can be analyzed with.

Simulation is a widely used type of dynamic analysis. This approach allows to estimate both timing and functional properties of designed systems in different cases.

One of particular kind of simulation — *discrete-event simulation* — is considered in this work. This kind of simulation is naturally suitable for the computer systems modelling. In this approach the work of the modelled system is represented as a sequence of discrete events. Each discrete event is an atomic action of internal state change and interaction with outer world and other components. All actions of a single discrete event are performed in a single moment of the simulation time.

This paper considers a problem of having or building of an effective and convenient simulation system for avionics systems analysis.

II. Simulation system requirements analysis

There are a lot of ways *how* discrete-event simulation can be performed. They can differ in both which characteristics are taken into account and which input data is required for the simulation system.

A. Events source

One of differences between the discrete-event simulation approaches is in the events source. Some of them consider only *periodic* events which are caused by the time. Also *sporadic* events can be considered. Such events are caused by some internal or external events which have to be modelled appropriately. Also, hybrid approaches which can consider the both event sources types, exist.

B. Model time

Another difference between approaches is in how accurate the time is modelled.

In some approaches time can be only a source of cause-and-effect relationships between discrete events.

Duration of events and processes have to be taken into account for more accurate modelling. This allows to retrieve estimations of timing properties as a result of the model analysis.

C. How behaviour is modelled

Approaches of the discrete-event simulation can differ in the ways of how behaviour of model components is modelled.

Model can, for example, be represented as a *randomized events flow* with given probability characteristics and description of how the component reacts to external events. Model also can be imperative. For example, it can be represented as either a *finite state machine (FSM)*, extensions of FSM which work with extended memory state and time, or some other *transition systems*. Also the behaviour can be modelled as a *program model*, when model is a code in some programming language.

Finite state machines (in particular, extended and timed) and specialized transition systems are usually naturally suit for describing of a behaviour of small components. Also, some of such models are studied well and can be analyzed in some other way except the execution. Such analyses can be used during the whole system analysis. But still, usage of such models is a bad idea for modelling of complicated behaviours (in particular, requiring a lot of internal states and events).

Program models are vice versa: they can pretty conveniently used for modelling of very complicated behaviour but they usually can be used only for execution.

But program models have an additional advantage: any simpler model (e. g. FSM and other transition systems) can be translated automatically to a program model. This means that if a simulation system supports program models, simpler model types can be supported automatically.

Randomized events flow is sometimes a really good behavioural model type for some types of components. And in most cases this model representation also can be translated to a program model automatically.

D. Abstractness

Two independent metrics of abstractness of the system models can be distinguished.

One of them is a *structural abstractness*. Structurally abstract models have components which are going to be refined in the future development but at the moment their structure, number and properties of its subcomponents are not known.

Another factor is a *behavioural abstractness*: the way how accurate behaviour is modelled. This influences on how accurate different aspects are reflected:

- internal state of a component;
- influence of a component to the other ones;
- data which components are working with;
- time intervals between events.

Which model characteristics can be retrieved during the analysis process depend on the behaviour abstractness of a model.

Relative complexity of behavioural models are more or less the same in case when model is accurate by both factors and in case when model is abstract by both factors. If structurally abstract model is built behaviourally accurate, behavioural models of each model component can be very complex (both by internal state and by interaction with environment).

E. General requirements

Support of working with models represented in different abstraction levels is essential for the early model analysis and validation. In particular, it is really important to analyze structurally abstract and behaviourally accurate models. This allows to check single structure refinements and to find out incorrect ones.

To provide this, the way of how behaviour is modelled have to be convenient for complex behaviours. But still, simple behaviours in structurally accurate models have to be able to be defined in a simple way (which is convenient to be done using some formalisms like transition systems).

That's why we consider that in a context of early analysis and validation it is important for a simulation system to support program models. But still, support of translation of other representations is really important too.

The very important aspect of usability of such simulation systems is their main users — designing engineers and integrators — familiarity to program tools and languages or at least to used paradigms.

Moreover it is important to be able to conveniently represent an internal state of a component and to work with it in a behavioural model in a convenient way. Considering ideas from above the best candidate for that is some imperative high-level language which has libraries of collections, basic algorithms and other useful features.

Models of avionics systems can be very big. But nevertheless simulation of such systems have to be performed relatively fast.

Such models usually can be divided into some parts that rarely interact with each other. This makes to think that parallel simulation (with correct distribution of components to nodes) can be performed with an acceptable performance.

Nowadays it is important for a simulation system to be portable. This is good both for users and for the organization process of a parallel simulation. But this requirement can add some restrictions to the way how behaviour is modelled.

III. Related work

There are some work and open-source instruments related to the discrete-event simulation.

Some instruments are based on formalisms that require explicit declaration of all finite states of the modelled component and all transitions between those states (including the timing properties of transitions). For example, instruments adevs [1], PowerDEVS [2] and DEVSimPy

[3] use pretty popular formalism DEVS (Discrete Event System Specification) [4]. Galatea [5] is based on a similar formalism. There are discrete-event simulation systems based on the Petri net, for example CPN Tools [6]. As it is said above, such models can be successfully used in accurate models but do not suit for describing complex behaviours in structurally abstract models.

There are some instruments that are using seldom used in avionics area functional languages (for example, Scala and Haskell used by Facsimile [7] and Aivika [8], [9]) or specific custom languages (for example, jEQN [10] for SimArch [11]).

Some instruments which are not in classes above (for example Tortuga [12], MASON [13], DESMO-J [14] and SimPy [15]) architecturally cannot be parallelized.

JaamSim instrument [16] is aimed only to graphical simulation and that's why it hardly can be used in the automated model analysis.

Thus, for fitting all requirements it is needed to design an architecture of discrete-event simulation which supports program models (using imperative high-level language) with parallel simulation support.

IV. Architecture

A. Interaction with simulation environment

The first question was how to organize interaction between a behaviour model and its environment: simulation time and other model components.

We will call a *basis* a set of action types in a code of a program model which can be used to describe a behavior. A basis has a part intended for an interaction with environment.

Two fundamentally different approaches were considered.

1) Synchronous interaction: Originally a *synchronous basis* was chosen. This approach is used in a number of discrete-event simulation libraries.

The main idea of the approach is that the code of a behaviour model determines moments of time when it is ready in get information from outside. In that case behaviour model can contain the following types of actions (besides actions on the internal state):

- non-blocking data sending to some other model component;
- notification a simulator about the end of the current discrete event with the next event starting at:
  - given moment of the simulation time;
  - external event receiving (with ability to set a timeout).

Program models designed for this approach are linear. Such program model can be defined as a description of an internal state and a single function containing all behaviour. Such way of modelling seems to be natural because a component life-cycle exists explicitly in a model as a single entity (function mentioned above).

Moreover, simulator providing such basis can be implemented to be very effective. In particular, such modelling approach allows to simulate independently rarely interacting parts of a model: until some parts do not send some data to each other they can be simulated in their own simulation time without violation of consistency and with no need of saving and storing of internal states.

But this approach has some problems.

Lets consider a situation when basis of interaction allows ending of a discrete event for receiving external data with a zero timeout. This is used, in particular, to implement a logical expression of kind "if at the current moment there was an external event X we should do one thing else another one". Such kind of expressions are widely used in behaviour modelling when using synchronous basis.

In the considered case results of a simulation really depends on the order of execution of discrete events which are scheduled to the same simulation time. In other words, some messages from one to another model component can be non-deterministically delayed. Obviously, simulation that allows such case cannot be used for accurate analysis of latency and other timing characteristics of models.

Basis can be changed in the following way to not to allow this. Behaviour model have to set a *delta* — positive time which is the minimal time to wait before the next event can arise — each time the external events are processed. The problem of using of such basis can raise when such delta have to be null or depends on external data. In that case delta cannot be given correctly.

One more problem of this basis was discovered. The problem is that the code of behaviour model have to manage external data in the order of their coming. This is not always possible (or, at least, convenient) because sometimes component have to manage some concrete data to make a decision. This leads a basis to have methods for incoming data filtering.

2) Asynchronous interaction: An *asynchronous basis* is an alternative to a synchronous one. The behaviour model have to react somehow to an external data in the very moment it comes. But still, one of the variant of such reaction is ignoring.

It is important that a model represented in such basis is not linear. This leads a rethinking of what the component behaviour model is. Also it influences on an internal state of a component.

It is stated that any model represented in a synchronous basis can be represented in an asynchronous one. This means that the class of possible behaviours described in an asynchronous basic is not smaller (and, in fact, bigger) that class of synchronous ones.

Such basis allows to consider all incoming data and to interrogate other components each moment of time. This makes behaviour modelling much easier comparing to the synchronous approach.

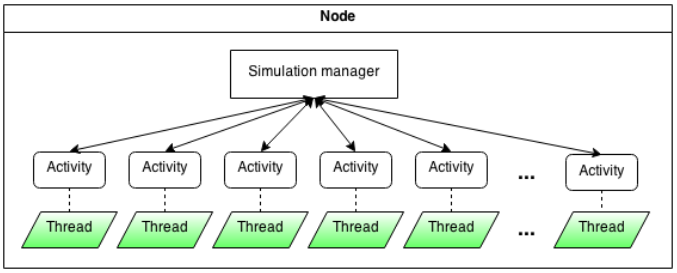

Fig. 1. Threaded approach on a single node

Ability of interrogation allows to model a continuous process initiation in an easy way. One component can initiate a continuous process in another component and is able to retrieve a result of this process as soon as it finishes.

Thus, asynchronous basis contains the following actions list:

- non-blocking data sending to some other model component;
- notification a simulator about the end of the current discrete event with beginning of the next event starting not later than the given moment of time;
- interrogation of the internal state of other component or initiation of a continuous process in it (in that case the current discrete event ending is also performed).

Simulator providing such basis is harder to implement in an effective way (comparing to the synchronous one). Nevertheless, the class of behaviours that can be modelled becomes adequate to requirements. That's why is was decided to use the asynchronous basis.

B. Execution architecture

It is important to consider that program model code execution have to be suspended at the end of discrete events to make other components to able to execute their own events at the same moment of the model time.

One of the simplest and obvious ways of organization of such alternating execution is the usage of a multiple threads. One thread is created for each component and they are suspended by a simulation system when a function of the end of discrete event is called. Thread is suspended until new event is raised for the corresponding component.

This approach is practical and pretty easy to implement. But it has some remarkable drawbacks.

One of them is that a behaviour model writer can easily create a deadlock. This situation can be preserved by using of some conventions for the behaviour model code but these conventions cannot be checked automatically by a simulation system.

But the main drawback, as it is seen from practise, is a high load to the threading subsystem of an operating system: models can have tens of thousands of active components so there are the same count of threads (fig. 1). This worked well for Linux-based operating systems. But some operating systems cannot manage with such load which leads to inability of simulation using them. This approach made the portability to be a problem.

One way of how this problem can be solved is a parallel simulation. This means that if we have enough count of nodes each of them would have small enough count of threads to be managed by any multithreaded operating system (fig. 2).

But having the size of a model as tens of thousands of components and limitation of operating systems to about a hundred of threads we have to use hundreds of nodes. Sometimes it is not really possible to have such count of nodes. Nevertheless usually models cannot be divided to hundreds of parts which rarely interact. This means that overhead of such simulation will be very big.

However, another way of such models execution organization execution exists. This way does not have drawbacks mentioned above but still requires some effort to apply it. This approach is called *continuations* or *coroutines* in the computer science [17], [18].

The continuations approach allows to execute several different program models in a single system thread. This means that program model code can be suspended and after that it can be resumed from the very point is was suspended. In other words, the program model code can be run at the beginning of the current discrete event handling point.

This approach runs into a problem of correct error tracing because control flow changes vastly and some effort is needed to make error traces (including stack traces) to look as if control flow was unchanged. Storing of additional information for that leads having some overheads.

Some modern and progressive programming languages which are using virtual machines for program execution, have the continuations approach built in. Classic languages have libraries implementing this approach but these libraries require an after-compilation program instrumentation.

Instrumentation of library program models is not a hard problem. But instrumentation of user program models can be a problem and require additional organization of the simulation process start.

Nevertheless, applying this approach (fig. 3) allows to increase maximal count of model components running a single node. This count becomes operating system independent. This means that more optimal division of a model to simulation nodes can be achieved. Thereby simulation effectiveness increases.

V. Integration

Simulator having architecture described above has been implemented and integrated as a part of an instrument MASIW [19], [20]. This instrument is dedicated to developing and analysis of avionics models using AADL [21] (architecture analysis and design language), which is

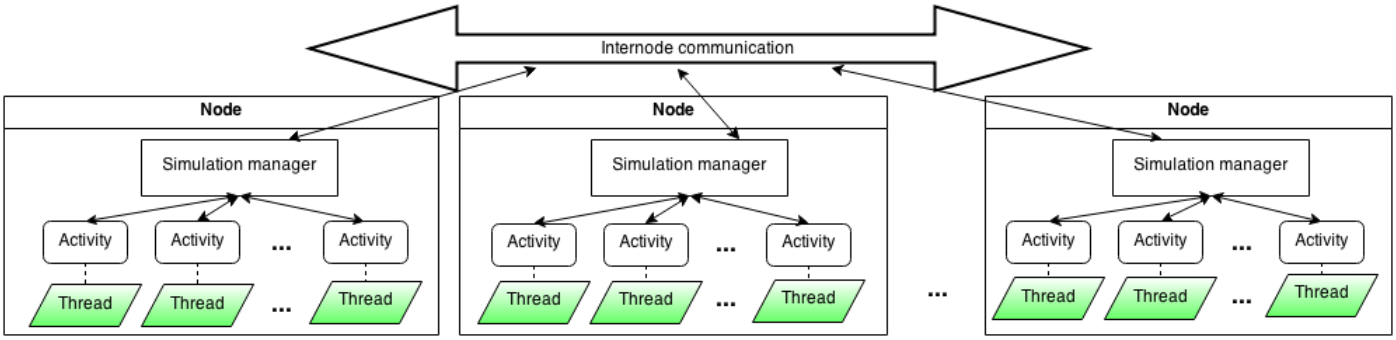

Fig. 2. Threaded approach on multiple nodes

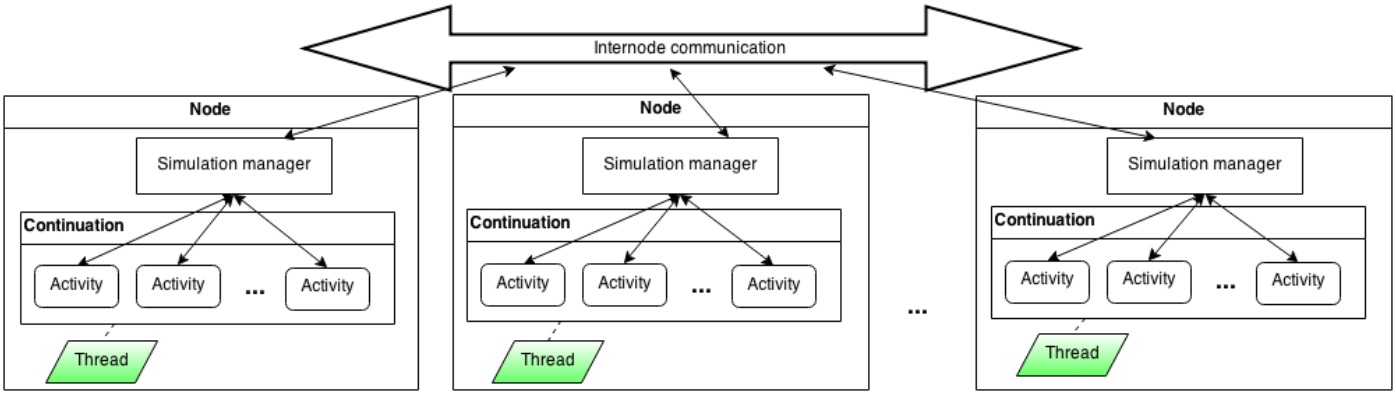

Fig. 3. Continuations approach on multiple nodes

widely used in creation of different responsible systems including avionics systems.

MASIW is a platform of developing of AADL-based models. It supports different representations and analysis of developed systems. The framework of this instrument is an open source.

This instrument has several static and dynamic model analysers. One of its analyzers is a discrete-event simulator. It is based on an architecture above and is used for a general dynamic analysis of AADL-models' behaviour.

VI. Conclusion

The main contribution is the architecture of the discrete-event simulation system that allows to simulate large systems (containing tens of thousands of components) using program behaviour models.

Also interaction of a program model with a simulation environment was investigated. It was shown that some approaches of interaction used in simulation libraries are inapplicable in some cases. An alternative way was suggested.

These architecture solutions were applied in a powerful avionics model design and analysis tool. This allowed to perform pretty fast and accurate analysis of avionics models (including models for the early validation). The architecture allowed to not to require a lot of simulation nodes for such analysis.

References

[1] Adevs library, http://web.ornl.gov/~1qn/adevs/.

[2] PowerDEVS, http://sourceforge.net/projects/powerdevs/.

[3] DEVSimPy, https://code.google.com/p/devsimpy/.

[4] B. Zeigler, *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*. San Diego: Academic Press, 2000.

[5] Galatea, http://galatea.sourceforge.net/.

[6] CPN Tools, http://cpntools.org/.

[7] Facsimile, http://facsim.org/.

[8] D. Sorokin, *An Introduction to Aivika Simulation Library*, 2013, https://github.com/dsorokin/aivika.

[9] A scala port of the Aivika simulation library, https://github.com/dsorokin/scala-aivika.

[10] A. D'Ambrogio, D. Gianni, and G. Iazeolla, "jEQN a java-based language for the distributed simulation of queueing networks," in *Computer and Information Sciences — ISCIS 2006*, ser. Lecture Notes in Computer Science, A. Levi, E. Savaş, H. Yenigün, S. Balcısoy, and Y. Saygın, Eds. Springer Berlin Heidelberg, 2006, vol. 4263, pp. 854–865. [Online]. Available: http://dx.doi.org/10.1007/11902140_89

[11] D. Gianni, A. D'Ambrogio, and G. Iazeolla, "SimArch: A layered architectural approach to reduce the development effort of distributed simulation systems," in *Proceedings of the 11th International Workshop on Simulation & EGSE Facilities for Space Programmes (SESP10)*, Noordwijk, The Netherlands, sep 2010.

[12] Tortuga, https://code.google.com/p/tortugades/.

[13] MASON Multiagent Simulation Tool, http://cs.gmu.edu/~eclab/projects/mason/.

[14] DESMO-J, http://desmoj.sourceforge.net/home.html.

[15] SimPy, http://simpy.readthedocs.org/en/latest/.

[16] JaamSim, http://jaamsim.com/.

[17] D. E. Knuth, *The Art of Computer Programming vol. 1: Fundamental Algorithms*, 3rd ed. Addison-Wesley, 1997, pp. 193–200.

[18] J. C. Reynolds, "The discoveries of continuations," *Lisp and Symbolic Computation*, vol. 6, no. 3-4, pp. 233–248, 1993.

[19] A. Khoroshilov, D. Albitskiy, I. Koverninskiy, M. Olshanskiy, A. Petrenko, and A. Ugnenko, "AADL-based toolset for IMA system design and integration," in *SAE 2012 Aerospace Electronics and Avionics Systems Conference*, vol. 5, no. 2. SAE Int., 2012, pp. 294–299.

[20] D. Buzdalov, S. Zelenov, E. Kornykhin, A. Petrenko, A. Strakh, A. Ugnenko, and A. Khoroshilov, "Tools for system design of integrated modular avionics," in *Proceedings of the Institute for System Programming of RAS*, vol. 26, no. 1, 2014, pp. 201–230.

[21] *Architecture Analysis & Design Language (AADL), SAE International standard AS5506B*, SAE International, 2012, http://standards.sae.org/as5506b/.