A discrete-event simulator for early validation of avionics systems

Denis Buzdalov and Alexey Khoroshilov
{buzdalov,khoroshilov}@ispras.ru

Institute for System Programming of the Russian Academy of Sciences
Moscow, Russia

Abstract. The paper discusses problems arising in development of avionics systems and considers how discrete-event simulation on the base of architecture models at the early stages of avionics design can help to mitigate some of them. A tool for simulation of AADL architecture models augmented by behavioural specifications is presented and its main design decisions are discussed.

Keywords: AADL, discrete-event simulation, early validation

1 Introduction

Nowadays avionics is responsible for control of almost all aspects of aircraft operation. As a result it became a complex system with thousands of sensors and actuators, and hundreds of software components spread across dozens of processor modules. Design and development of such system is a big challenge. And one of the biggest concerns is bugs introduced at the early stages of system design that are usually very expensive to fix in terms of both time and money as long as they often impact a number of components.

Let us demonstrate it on a simple example. A typical prerequisite to guarantee safety of flight is a real-time reaction on hazardous conditions. That leads to a number of safety-critical requirements to avionics subsystems that look like a limitation on time between a moment when a particular sensor reads actual data and a moment when software processes that data and takes appropriate actions, e.g. delivers a control command to an actuator or informs a pilot. If inability to satisfy such requirement is found during integration tests only, it may have catastrophic consequences for the project. For example, if an unexpected latency appears on an overloaded bus, it may require to redesign data flows between components and to reschedule software partitions that leads to redo significant part of safety analysis and other verification activities, i.e. to significant delays and cost increase for the project.

Model-driven system engineering is considered to be the most promising approach that can help to address this concern. The main idea behind it is to represent requirements and architecture decisions in form of models, i.e. in more or less machine-readable form, and then to apply automated verification techniques or even to automatically derive some parts of implementation from these

models. The benefits are manifold. First of all, automated analysis and verification help to identify as much problems as early as possible. But even if changes in system design are introduced, models allow to automate impact analysis and to reduce effort required for repeated verification.

The cornerstone element that enables analysis of various system characteristics is an architecture model of the system under development. One of modelling languages designed for this purpose is Architecture Analysis and Design Language (AADL) [11]. The core of AADL allows to describe an architecture of software-hardware systems. It supports constructions for step-by-step modelling, refinement and integration. Besides the core language, AADL has different extensions, e.g. for constraints definition, fault handling and behavioural modelling. AADL has pretty well-defined semantics that allows different tools to be applied to the same AADL models without significant effort. There are several open source frameworks supporting creation and analysis of AADL models including OSATE [1] and MASIW [7,4].

One way to classify the variety of methods of models analysis is to divide them to *analytical* and *sampling*.

Analytical methods allow to get guaranteed estimations of target characteristics, e.g. the worst-case latency. But the cost of the guarantee is excessively pessimistic estimations. Another drawback of analytical methods is that each of them applicable for a particular tasks only. That means that you usually have to develop a new method to estimate new characteristics or to get estimation in a little bit different conditions.

Sampling methods do not provide a guarantee in getting of worst-case estimations. But their benefit is that they can provide realistic estimations and can be much more easily adopted to new conditions and to new target characteristics.

To apply sampling methods to analysis of architecture models a simulation of the models is required. One of particular kind of simulation — *discrete-event simulation* — is considered in this work. This kind of simulation is naturally suited to the software-hardware systems modelling. In this approach functioning of the modelled system is represented as a sequence of discrete events. Each discrete event is an atomic action of one component's internal state change and interaction with the outer world and other components. All actions of a single discrete event are performed in a single moment of the simulation time.

In the paper we consider possible approaches to model behavioural characteristics of architecture models and discuss design decisions of our AADL simulation engine. Finally we discuss related works and overview possible direction of further improvements.

2 Modelling

In this section we are considering aspects of modelling which are relevant to the discrete-event simulation.

2.1 How Behaviour is Modelled

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

Depending on the type of the supported behaviour representation, different simulation and analysis tools can reach different quality of analysis and can run into different problems during it.

The AADL core does not contain any constructions to model behaviour of components but the language is easily extendable. That is why let us consider some ways of how behaviour can be modelled and some properties of these variants.

Behaviour 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.

There is also a class of behaviour representations which can be called imperative. For example, behaviour 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*. Transition systems can be combined with randomized events flows, particularly as FSMs with the probabilistic non-determinism resolution.

Also the behaviour can be modelled as a *program model*, when model is a code in some programming language.

AADL has a standardized extension called Behavior Model annex [10]. This annex allows to model behaviour of components as an extended timed FSM interacting with its environment.

Finite state machines (in particular, extended and timed) and specialized transition systems are usually naturally suitable 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, these models are not well-suited to modelling of complicated behaviours (in particular, requiring a lot of internal states and events).

By contrast, program models can be pretty conveniently used for modelling of very complicated behaviour but they usually can be used only for execution.

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

Randomized events flow which is a really useful representation sometimes, usually also can be translated to a program model automatically.

2.2 Levels of Abstraction

In this section we consider a model creation process in time. Models get their details during a long-term process but analysis of the models have to be performed from the very early stages. That is why simulation have to work with different levels of abstraction.

There are two dimensions of abstraction levels of system models to be considered.

Structural Abstraction The first one is a *structural abstraction*. Structurally abstract models contain components, internal structure of which is still going to be refined in future. For example, such models may define interconnection interfaces approximately or consider some complex subcomponents as black boxes.

For example, it may be known that some not fully specified device D is connected with a processor block P using some data transmission subsystem T (probably, involving some buses and devices) but it is not decided yet how this subsystem should be implemented. This means neither the connection interface of the device D nor the structure of the subsystem T are known.

Behavioural Abstraction Another dimension is a *behavioural abstraction* that depends on how accurately behavioural characteristics are modelled.

The behavioural characteristics include the following aspects that can be described with different accuracy:

- dependencies between input and output;
- influence of a component on the other ones;
- data which components are working with;
- time intervals between events.

A complexity of a behavioural model accurate by both structural and behavioural dimensions is more or less the same as a complexity of a behavioural model abstract by the both dimensions.

If structurally abstract model is built behaviourally accurate, behavioural models of each component can be very complex both by internal state and by interaction with its environment.

2.3 Analysis

Support of analysis of models represented in different abstraction levels is essential for early model verification and validation. In particular, it is really important to analyze structurally abstract and behaviourally accurate models as long as it allows to check single structure refinements and to expose incorrect ones.

To achieve this goal, the way of how behaviour is modelled have to be convenient for describing complex behaviours. It requires a convenient representation of an internal state and operations with it. This means a simulation system have to support program models.

But still, simple behaviours in structurally accurate models have to be defined in a convenient way, e.g. using formalisms based on transition systems. So combination of program models with other representations should be supported as well.

Another important aspect of usability of a simulation system is familiarity of a formalism (or at least its paradigm) to the users.

Considering the requirements discussed above the best candidate for the main formalism is an imperative high-level programming language which has rich libraries of collections, basic algorithms, etc.

3 AADL Simulator in MASIW

MASIW is an AADL-based framework for development and analysis of avionics and other safety critical systems. It contains various tools for development (text and graphical editors, model importers and generators) and analysis (static structure constrains analyzer, static AFDX latency analyzer, AFDX network simulator).

An AADL-model behaviour simulation tool is a welcome addition to such integrated toolset that enables early verification activities based on dynamic analysis.

In this section we consider the most interesting aspects of implementation of such simulation tool.

3.1 Program Models

As we discussed above, support for program behaviour models is an important feature for a simulator to be used for early validation. But there are several problems that should be resolved.

What Program to Consider a Behaviour Model The first problem is how to organize a program model. On the one hand, it seems to be useful to allow all convenient constructs of programming languages and libraries to be available in the program model. On the other hand, arbitrary program code cannot represent a behaviour model of a component because it have to be compatible with model interface of the component.

Representing behaviour of a component in a system, program models have to be able to model interaction with other components. In the context of event-driven simulation, models also have to be able to explicitly work with simulation time and discrete events.

Specific actions (like sending messages to other components) can be performed using special *simulation library* which is used by a program model. Time management can also be organized by such library. For example, there can be library calls modelling long-term computation or a launch of a long-lasting process.

There are other possible approaches to organize interface with simulation engine. For example, we can interpret the standard output of general executable code as outgoing commands of a component for model-specific activities like sending messages to other components and working with the simulation time. Similarly, standard input of a program can be considered as models-specific component input like incoming events and data. This approach is used in some systems in different areas (one of the widely used examples is FUSE [2]) but it

seems that this approach does not really fit for describing behaviour models of components in AADL-models.

As long as we are not limited by any legacy behavioural models we decided to use the following program models representation. We decided not to limit user in the internal state representation. Consequently, there is no limits for the code that works with it.

A simulation library was implemented to manage everything related to discrete-event nature of the simulation. It is intended to be explicitly used in the program model code. The library has a plenty of different calls for time management and for interaction with other components of a system model.

This model representation allows to express everything needed with maximum freedom in behaviour definition.

Interaction with the Simulation Library The question of how an interface of the simulation library should be organized is not trivial as well.

Two fundamentally different approaches were considered.

The first one expects a program model of any component to have a single entry point that handles all simulation events so that only the program model and not the simulation library determines when external events and data can be managed. Simulation library just provides an interface for getting information of new events. This approach can be called *synchronous*.

The synchronous approach has an advantage that simulation of a model can be organized in a very optimal way. If two parts of a model do not interact for some time, they can be simulated independently. Each independent part can have its own current simulation time.

But the synchronous approach has also a disadvantage: it is not possible in the general case to model a situation when some component launches long-lasting process which is provided by another component, and waits for a result of this process. Necessity of this is known from the practice.

Another approach of interaction with a simulation library can be called *asynchronous*. In this case a behaviour program model has several entry points which are in fact callbacks. These callbacks are called at some moment of simulation time when some specific sort of event arises, e.g. incoming events and data, different requests of other components and system events like simulation start.

Some of callbacks can return a value. In particular, such callbacks can model a response of a component to some other component's request and the mentioned above launching of long-lasting process that returns a value.

But still, asynchronous approach has its own shortcomings. First of all, simulation of different parts of a model cannot be performed independently. The reason of this is the following. Consider a component A having had performed some state changes at the simulation time $t_2 > t_1$ and a component B running at the simulation time t_1. Consider now that B has requested some information related to the internal state of A. If simulation engine lets the component A be simulated independently with B (i.e. including the time t_2), then the component B would be unable to get what it needs.

One of possible solutions of the problem is to prohibit any component A to perform any actions at time $t > t_1$ till other components which may request something from A have finished their execution at t_1. But this approach limits abilities to parallelize simulation activities.

We can try to solve the problem by storing states of faster running components (like A in the example above) at the moments earlier than the current one (for $t < t_2$) accessible by other components (like the state of A at t_1 accessible by B). But this approach still has problems. Consider we are running the component A at the t_2 storing all states of A at moments of time between t_1 and t_2. But consider that B at the moment t_1 can request a change of the state of A instead of just reading it. It means that we need to discard all stored states at the moments of time $t > t_1$ and to rerun simulation for A starting at the time t_1. So, this solution requires a lot of space to store all needed states and a lot of overhead for these copies management.

As a result, simulation of asynchronous models can be less effective comparing to synchronous ones regardless how the asynchronous simulation is organized.

Nevertheless, every synchronous behaviour can be represented as an asynchronous one. This means asynchronous simulation libraries are more universal than synchronous ones. Considering limited abilities of synchronous approach, asynchronous one is more preferred.

Table 1. Comparison of simulation library approaches

	synchronous	asynchronous
code representation	linear (like a single function)	a set of callbacks
effectiveness	easy to parallelize	whole model have to be simulated at a single moment of simulation time
abilities	issues to model long-term processes with a returned value	no limits

A comparison of synchronous and asynchronous approaches is presented in the table 1.

Execution Architecture It is important to consider that code execution of a program model that uses a simulation library 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 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 practice, is a high load to the threading subsystem of an operating system which is used to run simulation. Models can have tens of thousands of active components that requires the same number of threads. It worked well for Linux-based operating systems, but some other operating systems cannot manage such load. So, straightforward multi-thread approach leads to the portability issues.

However, there is another approach to organize program models execution that does not have drawbacks mentioned above. This approach is called *continuations* or *coroutines* in computer science [8,9].

The continuations approach allows to execute several program models in a single system thread. It means that program model code can be suspended and then it can be resumed from the very point is was suspended. Such suspension is performed by a simulation library and no special constructs have to be added to a program model.

This approach runs into a problem of correct error tracing because control flow changes vastly. So some effort is required to make error traces and stack traces looking as if control flow is unchanged.

Some modern and progressive programming languages, that use 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 it requires special handling of simulation start.

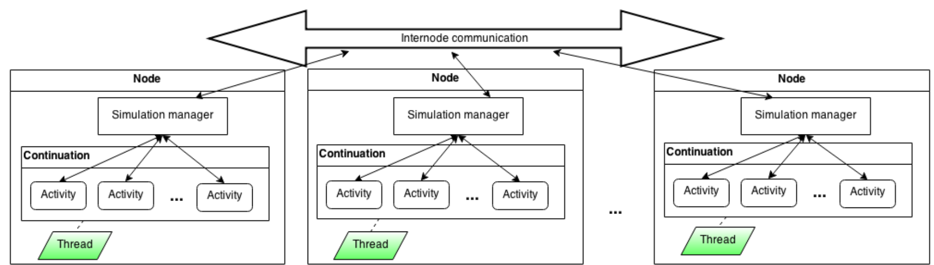

Fig. 1. Continuations approach on multiple nodes

Nevertheless, applying this approach (fig. 1) allows to increase maximal number of model components running on a single node to make it operating sys-

tem independent. It means that more optimal distribution of model components across simulation nodes can be achieved in comparison to the multithreaded multinode approach.

3.2 Behavior Annex

As it was discussed above, support of specialized transition systems as one of allowed types of behaviour model notations is a welcome feature for a simulation tool aimed to analyze models across various stages of development process. AADL has a standardized extension for defining such behaviour models called Behavior Model annex.

AADL Behavior model annex represents behaviour of a component as an extended finite state machine (FSM) of a specific kind. Actions on transitions can contain data state changes, interaction with the outer world and time delays. Transition conditions of such FSMs may depend on data state, external events generated by other components and time events.

Keeping in mind that our program models use simulation library containing operations for communication with environment and time, behavior model annex machines are translated to program models that use the simulation library. This translation is implemented in the MASIW AADL simulator.

Communication actions of the Behavior Model annex are implemented as the simulation library calls. Time-related actions are also mapped to the library calls. State changes (both FSM state and extended data state changes) are implemented naturally in a program model.

3.3 Built-in libraries

MASIW is targeted to the avionics models development and analysis. That is why during simulation we are running into behavioural aspects of different standards widely used in avionics.

For instance, the **ARINC653** standard is widely used for organizing execution of software in the avioncs system. It defines both structural and behavioural aspects of such systems.

Structural information can be represented by special standardized ARINC653 annex of AADL [10] and derived property set.

To ease development of ARINC653-based systems, a standarized behaviours for processors and other components were implemented. They can be used as a part of the behavioural model of a developed system. Moreover, library behaviours can be a base for user-defined behaviours.

AFDX network standard is a very important and widely used standard in avionics systems. AADL standard does not have support for modelling properties related to AFDX networks.

That is why we had to implement our own property set for defining structural aspects of models using AFDX. It have been used in mentioned above MASIW parts: static AFDX latency analyzer and AFDX network simulator.

We have implemented standardized behaviour of AFDX-specific network parts (like AFDX switches and network devices) as library behaviours which can be used in general AADL models simulation in MASIW. Also, some analyzers for these behaviours were implemented such as switches buffers and queues filling, counts of packets for different routes and links, statistics for packets drops and reasons of them and etc.

Such behaviour libraries let the model developer to focus on project features not paying much attention to how to model standard behaviour.

4 Related Works

Marzhin [5] is a proprietary simulator of AADL and AADL Behavior Annex models that mostly targets to analyze schedulability properties. It is based on existing multi-agent simulation kernel and it supports simulation of a subset of AADL and AADL Behavior Annex. The main distinction of MASIW simulator is support not only for Behavior Annex but for program models as well that allows to describe and then to simulate much more complex behaviours. Also, it is pretty easy to configure the MASIW simulator to manage and analyze various properties of a model.

OSATE framework [1] provides several plugins for model development and analysis. One of them called ADeS [12] is dedicated to analysis of behavioural properties using simulation. But unfortunately, its development stopped in 2008 and so this simulator does not support the last version of AADL which really differs from the supported first version.

AADS [13] is a translator of a subset of AADL with Behavior Annex to SCoPE [3] representation. SCoPE implements POSIX-based API that enables it to run appropriate software parts. Also, SystemC [6] is used for hardware simulation. The approach has a lot of benefits. But it is intended to the simulation of pretty accurate models to get accurate estimations. It seems to be not really usable on early steps of the model development.

5 Conclusion

MASIW AADL simulator supports simulation for all stages of the model development using the most appropriate behaviour model for each stage — program models for complex behaviours in structurally abstract models and specialized type of transition system called AADL Behaviour Model annex for other cases. A conclusion from implementation of the simulator is that having the program models support, it is quite natural and easy to implement a support of specialized transition system like AADL Behavior Model annex.

This simulator is integrated to the MASIW framework that supports most steps of the development and analysis process of AADL-models. This integration allowed to perform pretty fast and accurate analysis of avionics models (including models for the early validation).

Chosen behaviour model representation as a program model allows to model errors in models naturally. But support of standardized ways like AADL Error Model annex is a task for the future.

References

1. OSATE 2, `https://wiki.sei.cmu.edu/aadl/index.php/Osate_2`
2. Filesystem in Userspace, `http://fuse.sourceforge.net/`
3. SCoPE, `http://www.teisa.unican.es/scope`
4. Buzdalov, D., Zelenov, S., Kornykhin, E., Petrenko, A., Strakh, A., Ugnenko, A., Khoroshilov, A.: Tools for system design of integrated modular avionics. In: Proceedings of the Institute for System Programming of RAS. vol. 26, pp. 201–230 (2014)
5. Dissaux, P., Marc, O., Rubini, S., Fotsing, C., Gaudel, V., Singhoff, F., Plantec, A., Nguyen-Hong, V., Tran, H.N., et al.: The SMART project: Multi-agent scheduling simulation of real-time architectures. Proceedings of the ERTSS 2014 conference (2014)
6. IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005): IEEE Standard for Standard SystemC Language Reference Manual (Jan 2012)
7. Khoroshilov, A., Albitskiy, D., Koverninskiy, I., Olshanskiy, M., Petrenko, A., Ugnenko, A.: AADL-based toolset for IMA system design and integration. In: SAE 2012 Aerospace Electronics and Avionics Systems Conference. vol. 5, pp. 294–299. SAE Int. (2012)
8. Knuth, D.E.: The Art of Computer Programming vol. 1: Fundamental Algorithms, pp. 193–200. Addison-Wesley, 3 edn. (1997)
9. Reynolds, J.C.: The discoveries of continuations. Lisp and Symbolic Computation 6(3-4), 233–248 (1993)
10. SAE International: Architecture Analysis & Design Language (AADL) Annex Volume 2, SAE International standard AS5506/2 (2011), `http://standards.sae.org/as5506/2/`
11. SAE International: Architecture Analysis & Design Language (AADL), SAE International standard AS5506B (2012), `http://standards.sae.org/as5506b/`
12. Tilman, J.F., Schyn, A., Sezestre, R.: Simulation of system architectures with AADL. In: Proceedings of 4th International Congress on Embedded Real-Time Systems. ERTS 2008 (2008)
13. Varona-Gomez, R., Villar, E.: AADS+: AADL simulation including the behavioral annex. In: Proceedings of the 2010 15th IEEE International Conference on Engineering of Complex Computer Systems. pp. 379–384. ICECCS '10, IEEE Computer Society, Washington, DC, USA (2010)