Simulation of AADL models with software-in-the-loop execution

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

ABSTRACT

In this paper we consider a problem of simulation of avionics models with heterogeneously specified behaviours of components. In particular, we consider a situation when some software components of a model are specified by a program code of ARINC653 partitions but the rest of a model is defined in a very abstract way. One approach of behavioural and time analysis of such models is described in this paper.

1. INTRODUCTION

In the paper we discuss issues related to early validation of avionics and other safety-critical control systems. The problem is important because the cost of mistakes rises exponentially during the development.

One of the ways to manage with rising complexity of modern control systems (used in aviation, transport, medicine, etc.) is modelling them before implementation to perform validation and verification checks.

Different aspects of a system can be represented in different models. For example, one model can be used for analysis of the power consumption of hardware and another model can be used for representation of data flows in a system. We focus on behavioural characteristics of control systems.

To be able to check such kind of characteristics on a model, it should be represented with a modelling languages containing architecture and behaviour definition available for automated analysis. Such models can be used for analysis of behavioural characteristics of the modelled system during virtual integration of the whole system from smaller parts. Languages that are applicable to automated checking and analysis need to be formal enough to be interpreted by tools [7].

Analysis of such models can be *dynamic* and *static*, i.e. involving or not some kind of execution of a model. Formal methods and model-checking of different behavioural and temporal properties are static methods of analysis; simulation of behavioural models is a dynamic method.

1.1 Different abstraction levels

Models of complex systems can be developed by independent groups of people. Developers can use different sources for parts of a single model like projects requirements and documentation, industry standards, legacy developments and etc.

That is why different parts of models can have different level of abstraction, readiness and precision. Some parts of models can be developed fully (for example, in case of reusing them from previous projects), some of them can be at the beginning of development.

For example, for avionics systems, already developed software applications can be represented as existing ARINC653 [1] partitions, and other applications' behaviour is given as abstract behaviour definition, like finite state machines and other transitions systems.

We will address to models containing parts with completely different level of abstraction (in particular, talking about behaviours) as *heterogeneous models*.

Obviously we can manage with such heterogeneous models by modelling existing ARINC653 partitions with some abstract behaviour definitions. But it seems to be very costly and it is a potential flaw because of potential errors that may be introduced at this step. That is why, it would be better to support analysis of heterogeneous models.

1.2 Models reuse

Models appearing on early design stages of a complex control system can be reused after (partial) development of some of software components for analysis of the system.

We can consider not only an application-local correctness but the level of the whole system which a particular application influences on. In this case additional value can be achieved with usage of architecture models during analysis.

The system architecture model can contain *system-wide characteristics*. For example, we can consider time of passing of some data through the model, in particular from some device though decision-taking control application to an actuator. At the architecture design level, we were having some requirements on this property. After implementation of a control application we still are interested whether this property lies in appropriate bounds. We can perform model analysis involving the implemented application (i.e. a heterogeneous model) to see whether this system-wide property still holds.

We can have an additional value of reusing of architecture model in case when we need to estimate or optimize several properties of a system. This can been done by similar analysis of heterogeneous model performed for optimization, not for checking.

> *For example, system developer is free to configure different buffer sizes. He may want to know whether or not some particular values are enough for this or that case. Thus, having an architecture model and particular applications' implementations, we can an-*

[1]This work is supported by the Russian Foundation for Basic Research (grant 14-07-00443).

alyze the whole model to estimate appropriate buffer sizes. The same can be done with other system properties.

2. CONTEXT

2.1 AADL and Behavior Annex

One modelling language supporting both architecture definition and behaviour specification is SAE AADL (Architecture Analysis and Design Language) [11]. It is a standardized language that has several representations for models. It has a textual representation intended to be used both by humans and for automated analysis by tools. For very complex projects and for general overview a graphical representation of AADL exists. It allows to look at the modelled system, its components and interconnections without looking at too much details.

The focus of AADL is on embedded software systems, specifically for representing the software task and communication architecture, the execution platform (OS and hardware), and the physical system interface.

The core of AADL is used for defining the whole structure of the modelled system and for representing main interconnections and relations between components of the model. AADL models can be extended with different additional definitions. For example, a component definition can be extended with description of its possible error states and ways it manages them [12]. Another standardized extension allows you to define behaviour of a component using specialized timed finite state machines (FSM). This extension is called BA, Behavior Model Annex [10].

AADL models with BA definitions allows to describe both architectural and behavioural aspects of a modelled control system.

2.2 MASIW and its universal simulator

MASIW is an open-source framework for development and analysis of AADL-models. It supports several types of analysis, both structural and behavioural, static and dynamic [8].

MASIW provides a universal model simulator that supports AADL models with behaviour specification using BA. Additionally the simulator supports one more way to specify behaviour of model components: one can use Java classes that use special simulation library [6]. Using this simulation library, user can define a component behaviour including computations, input/output (IO) using ports and buses, modelling of time consumption and etc.

Thus, at the moment only abstract behaviour specifications of model components are supported.

2.3 JetOS operating system

JetOS [2] is an ARINC653-compatible open-source operating system. It supports static scheduling of ARINC653 partitions.

For input and output with the outer world (i.e. with partitions on other modules and devices), AFDX [3] network is used mainly. JetOS is configured statically to use appropriate virtual links (VL) settings for this or that partitions' ports.

2.4 QEMU

QEMU [4] is a generic and open source machine emulator. It supports numerous variants of architectures and hardware to emulate. It allows to emulate a single machine running surrounded with particular network servers and services.

Emulation can be controlled by human or by automation. QEMU has an ability to be managed with GDB (GNU debugger tool) and with special machine-oriented command-line interface. In particular, running OS can be stopped, memory and appropriate variables can be observed. Actions allowing to influence on behavior like memory state update can be done also. This means that we can change particular variables of an operating system running inside QEMU.

JetOS supports QEMU as one of target boards it can be run on. It provides a device driver for the standard virtual network card provided by QEMU (virtio interface). Thus, all QEMU facilities for virtual network wiring can be used with JetOS.

3. PROBLEM

Having AADL models of some control system, we need to see how it behaves (both functionally and temporally) to estimate different properties of modelled systems.

For example, we have a requirement on the whole system that a signal from the engine compressor stall detector needs to reach its receivers (pilot indicators and automatic engine power corrector) within known small period of time.

This signal goes though internal connectivity circuits and decision taking program components which can add delays to the signal transferring time.

Estimation of time of moving a signal through the system is needed to check the requirement using the system model.

One of possible solutions is simulation of the AADL model in different cases.

As it was said before, AADL-models can be simulated with MASIW universal simulator which supports two variants of behaviour specification, which both are very abstract. But during development of a control system, some of its parts can be already implemented. In the avionics area, we can consider ARINC653-applications as such implementations.

To be able to perform simulation of such models without extra work, it is good to support the case when behaviour of model components is defined heterogeneously, i.e. for some of them we have only abstract behaviour specification (e.g. using AADL BA timed finite automatons) and for some of them we have code of ARINC653 partitions that is intended to be used on-board.

So, the problem is to organize simulation of heterogeneous models. This implies a need of an easy way to use existing program code of ARINC653-partitions as behaviour specifications. Having this, there is a need to be able to simulate such models to retrieve or estimate behavioural characteristics of such models (like timing of important signals as in the example above and other behavioural properties).

4. SOLUTION

4.1 General scheme

It was decided to reuse the existing simulator's facilities as much as possible. To do so, it was decided to create a special Java-behaviour for ARINC653 operating system component in a model. This behaviour organizes running of JetOS with appropriate applications in QEMU. ARINC653-partitions with appropriate applications become software-in-the-loop which are executed in parallel with the model simulation.

From the simulator's point of view, this behaviour is a usual Java behaviour specification which in fact represents a proxy between one world (simulated by simulator of MASIW) and the other (emulated by QEMU with running JetOS).

This proxy receives/sends messages from/to QEMU using network which is used by JetOS for external world IO. Data coming to a component from a model is translated to binary representation and vice versa.

This behaviour also is responsible for time synchronization between simulated model and running JetOS.

You can see overview of the approach on figure 1.

4.2 Time synchronization problem

Since we are talking about software, we consider a discrete time. This fact imposes upon us to synchronize several independent executions at the right moments.

The simulator considers the functioning of the simulated system as a sequence of discrete events. Each event is associated with a particular moment of time.

Each JetOS instance running in QEMU has its own local time that can differ from simulation time of the main AADL-simulator.

This lack of synchronization leads to wrong time of different events appearing in the system. In particular, moments of appearance of various data in the system can be different for sender and receiver. This can lead to errors in time characteristics estimation and to misbehaving of the modelled system.

So, we should define several points at which each co-simulated component have to be synchronized with the main simulator. These points are related with input and output: all components must be synchronized when software-in-the-loop is sending data to the rest of the model and vice versa.

To organize this, the special proxy Java behaviour knows the current time of JetOS and the current simulation time. It has to know timer frequency of the JetOS instance and to recalculate it to/from simulation time.

The proxy watches the simulation time. When the simulator attempts to move the current simulation time further to t_{next}, it means that no events are going to appear from the simulated model. The proxy previously ensured that JetOS local time equals to the current time t_{curr} of the simulator.

In this case, proxy behaviour pauses the simulator and allows JetOS to run from time t_{curr} till time t_{next}. If JetOS did not send anything till t_{next}, proxy

- pauses JetOS when its local time becomes t_{next};
- resumes the rest simulator;
- waits until simulator's current time reaches t_{next}.

If JetOS has sent something at the moment of time $t_{out} < t_{next}$, the proxy

- stops JetOS at this moment of time (in fact, JetOS stops itself and reports about this to the proxy);
- reads the binary message from JetOS and translates it into the model representation;
- sends a message to an appropriate model component;
- resumes the rest simulator to run until t_{out} to manage the message.

This synchronization allows to receive messages from JetOS by the rest model exactly at the right moment of simulation time. Thus JetOS-to-model IO is synchronized.

Model-to-JetOS IO with this approach is also synchronized. It is easy to see because on every message from some model component to JetOS which is sent at t_{in}, there exists a moment of time t_{curr} when $t_{next} = t_{in}$ and JetOS is not sending anything between t_{curr} and t_{next}. This means that the proxy behaviour has to simply resend a message to JetOS as soon as it is received from the simulator.

4.3 OS changes

To perform control and to synchronize real operating system time and simulation time, couple of things were needed to be changed in JetOS.

We wanted to change as less as we can, that is why no additional control channel was invented. QEMU's memory access was considered to be the best way to manage time management of JetOS.

In JetOS time is represented in ticks where tick time is the time between successive interrupt controller dispatches. Ticks count is managed by JetOS itself and is available for observing with QEMU. ARINC653 calls and other systems calls use only this time measured in ticks for their work.

So, to control time of the OS running, a single additional variable was added. Its meaning was the maximum tick that OS can reach until idling.

Once it becomes less than current time, OS idles and tick counter stops increasing. This means that time has stopped from the applications' point of view.

Once the maximum tick variable becomes more than the current time, OS runs to the tick stored in this variable. After resuming, all application behave in a way as if JetOS was not idling at all.

This variable was intended to be changed by the proxy behaviour though facilities of QEMU (using GDB or special interface, as it was described above). Such control allows the proxy to force the OS time to not to run over the simulation time.

To prevent the simulation time to run over OS time, this variable is changed automatically by JetOS each time it sends something to outside world using network.

Obviously, this approach has a precision of interrupt controller frequency, i.e. we cannot stop JetOS at any moment of its local time, but only on a value of time which is a multiple of clock time. But this is considered acceptable because every message received between successive interrupt controller dispatches will not be managed until the next dispatch. So, this precision is enough to model behaviour correctly.

4.4 QEMU configuration

QEMU allows to emulate environment of emulated OS, for example to create internal LAN with appropriate servers listening and tunneling appropriate connections.

Since networking of JetOS is configured statically and AFDX uses Ethernet frames with IP and UDP headers, it allows us to generate automatically configuration of individual mapping between input or output port of JetOS and appropriate UDP ports in the host machine. Each ARINC653-port of each partition running inside each particular instance

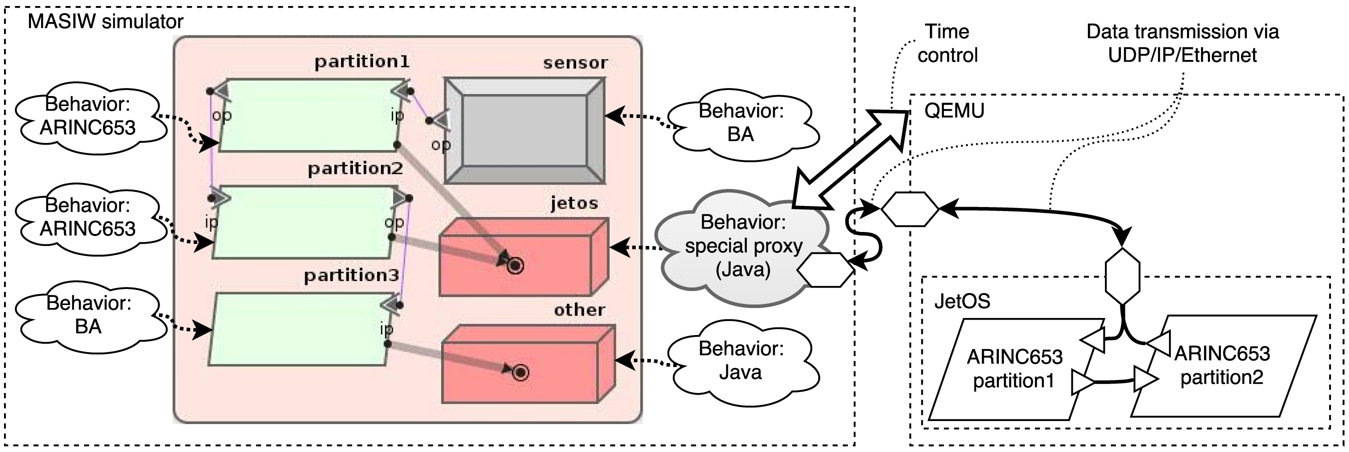

Figure 1: Overview of the simulator with SIL

of JetOS is forwarded to particular UDP ports of *localhost*. This allows to easily receive and send messages by the proxy Java behaviour without much effort. Knowing a mapping between local UDP ports and ARINC653-ports of partitions, we can easily determine to which part of model this or that message is intended to.

4.5 Binary data transformations

One of the specialities that raised during organization of such co-simulation is to prepare and to send data to a co-simulated OS from the model behaviour, we need precisely modelled data with full informations about size of data elements in bytes/bits, endianness, padding and etc.

It was realized that in practice models often do not contain enough data to perform correct binary data generation. This was not very surprising since models usually have narrowed representation of reality not containing details that are not known or not relevant to what the model is targeted to.

But it also was realized that modelling languages sometimes do not provide an ability to put such information into a model. Also, sometimes we can face a problem of refining gradually, i.e. adding only relevant information leaving model to be abstract enough.

As an example, we can look at AFDX frame modelling. Full and structured definition of AFDX frame takes more than one hundred lines of AADL model code and more than 30 fields. But in most cases this full definition is too detailed.

For example, in the basic case (not considering packet fragmentation) for validation of AFDX network routing configuration with simulation we need only four characteristics of AFDX frame (virtual link identity, sequence number, padding size and payload). In this case, model of AFDX frame is pretty simple, it contains only of three numbers and payload. If we consider fragmentation, we need to add only three more numbers and a single boolean value (header size, fragment identity, offset and 'more fragments' flag).

Obviously, modelled parts of the whole AFDX frame data structure are not laying consequently in the binary representation. They are interleaved with other fields that are out of consideration in some particular model.

We have to work around this when performing binary transformation of AFDX frames models to binary values.

Obviously, we can say that we can work only with fully defined data models which can be easily transformed to binary representation. But this approach much limits the area of models that can be analyzed with co-simulation when we sometimes are unable to extend model definition to be full (thus, no early validation and other things that were discussed above). Another point against is that this approach breaks the idea of modelling, i.e. of throwing away of consideration those things that are actually not needed.

But adding only relevant information needed for binary data transformation runs into not readiness of modelling approaches. For example, at the moment AADL does not allow to add offset information conveniently. However it allows to do it with manual addition of paddings of appropriate size between meaningful fields of the data structure. This allows to solve a problem of ability to transform binary data going from AADL model to co-simulated OS and vice versa. But this approach works badly when we need to refine existing model (containing such paddings) to replace paddings with fields (when fields now are needed for this or that model analysis).

5. RELATED WORK

SPADES.

The SPADES simulation engine [9] aims to solve a similar task. It supports discrete-event simulation with software-in-the-loop execution.

SPADES is a multi-agent simulator supporting almost arbitrary behaviours for each agent. Generally, this approach can be used with architecture models by assigning an agent to each component of the model.

In SPADES each agent acts in a simulation time. Actioning of each agent is represented by the following actions:

- *sense*, i.e. receive some signal or stimulus at some moment of time;
- *think*, i.e. consume some simulation time to perform processing;
- *act*, i.e. performing some activity touching other agents.

During *thinking* an agent can perform calculations and request simulation time notification, i.e. agent can say that it is going to continue only at particular moment of time. This is used for modelling of time consumptions when agent behaviour is an abstract behaviour.

Software-in-the-loop feature of SPADES is the following. When some agent is not a model behaviour but a real piece of software, it obviously do not have any time notification requests. To include such software into the loop, SPADES performs CPU time consumption measurement during execution of real software and uses this time as if it was requested by this agent.

This approach uses two notions of time: simulation time and real time system running a simulation. This approach works well when the target executor of a software is the same as one performing simulation.

But it is not working for responsible control systems because usually they use time-deterministic operating systems and deterministic scheduling. Usage of real time of an operating systems that performs a simulation does not give correct estimates on time consumptions of software components. Thus, SPADES-like approach for software-in-the-loop simulation is not applicable for analysis of models of responsible control systems with heterogeneous behaviour specification.

The FALTER case.

A software-in-the-loop simulation approach was used in the FALTER project testing [5].

In the FALTER project an autonomous unmanned aerial vehicle was developed. It was intended to automated indoors exploration of buildings after an accident.

After the software responsible for the sophisticated logic of the vehicle was developed, software-in-the-loop simulation was used to put this software into different interesting cases to see whether it manages the situation or not.

A special test framework was developed for this. This framework contained an environment and platform models and allowed to specify a test case (with definition of particular obstacles and sensors noise). These models were represented as Simulink blocks.

This approach of modelling seems to work very well for small systems with relatively small count of sensors and actuators tested as a whole. It looks like this approach is hard to apply to complex systems with interconnected components when some of them already have their implementation and some other are defined in abstract way.

6. CONCLUSION

Architecture models are widely used in development of complex responsible control systems like avionics. They can be used for the system-wide analysis before implementation of parts of a system.

After partial implementation of software components or during integration of the system, system-wide properties of the system must be checked again.

Methods that are using already developed architecture models with already implemented software components allow to make the analysis of several properties of the system much easier. There is no need to develop additional integration checks because architecture models already contain requirements that need to be checked.

Simulation of architecture models with abstract behaviour definitions supporting software-in-the-loop execution is one of such methods. It allows to check behaviour of the whole system when some software parts are fully implemented while others have only abstract behaviour definition inherited from early stages of development.

One particular implementation applicable in avionics area is considered. AADL and its standard extensions are used as architecture and abstract behaviour modelling language. ARINC653-applications can be used as software-in-the-loop during simulation. This is considered to be useful for purposes of analysis of whole system properties on early stages of development and during system integration.

7. REFERENCES

[1] ARINC 653 standard, http://arinc.com/.

[2] JetOS, an open-source ARINC653-compatible RTOS, http://forge.ispras.ru/projects/chpok/.

[3] ARINC 664 part 7, Avionics Full Duplex Switched Ethernet (AFDX) network, http://arinc.com/.

[4] QEMU, a generic and open source machine emulator and virtualizer, http://www.qemu.org/.

[5] A. Bayha, F. Grüneis, and B. Schätz. Model-based software in-the-loop-test of autonomous systems. In *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation — DEVS Integrative M&S Symposium*, TMS/DEVS '12, pages 30:1–30:6, San Diego, CA, USA, 2012. Society for Computer Simulation International.

[6] D. Buzdalov and A. Khoroshilov. A discrete-event simulator for early validation of avionics systems. In *ACVI 2014 — Architecture Centric Virtual Integration Workshop Proceedings*, pages 28–38, 2014.

[7] D. Buzdalov and A. Khoroshilov. About formal interpretation of architecture models. In *ACVI 2015 — Architecture Centric Virtual Integration Workshop Proceedings*, 2015.

[8] 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*, volume 26, pages 201–230, 2014.

[9] P. F. Riley and G. F. Riley. SPADES — a distributed agent sumulation environment with software-in-the-loop execution. In *Proceedings of the 2003 Winter Simulation Conference*, 2003.

[10] SAE International. *Architecture Analysis & Design Language (AADL), Behavior Model Annex*, 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] SAE International. *Architecture Analysis & Design Language (AADL), Error Model Annex*, 2015.
http://standards.sae.org/as5506/1a/.