DYANA: an integrated development environment for simulation and verification of real-time avionics systems

V.A. Antonenko⋆, E.V. Chemeritsky⋆, A.B. Glonina⋆, I.V. Konnov⋆⋆, V.N. Pashkov⋆, V. V. Podymov⋆, K.O. Savenkov⋆, R.L. Smeliansky⋆, P.M. Vdovin⋆, D.Yu. Volkanov⋆, V.A. Zakharov⋆, D.A. Zorin⋆
⋆Lomonosov Moscow State University
⋆⋆Technische Universität Wien
{juan,dimawolf}@lvk.cs.msu.su

Abstract

In this paper we present DYANA, an HLA-based hardware-in-the-loop simulation tool. This tool is used for Distributed Real-time and avionics systems (RTAS) simulation. RTAS models are described by Unified Modeling Language (UML) statechart diagrams. The statechart diagram is transformed into HLA-based Simulation Model (HSM). After translation into HSM we use CERTI as the simulation runtime. The statechart diagram is also transformed into a network of timed automata (NTA). After translation into NTA we use UPPAAL for RTAS model verification. Results of simulation and verification experiments involving the tool are presented.

1. Introduction

Modern onboard distributed Real-Time Avionics System (RTAS) consist of multiple devices connected by a network of data transfer channels. Each RTAS device runs a specific mix of application and system software tasks. On early stages of RTAS development, hardware and software components of RTAS devices are developed concurrently. On these stages, availability of hardware prototypes for the devices is limited, i.e. prototypes are available only for some of the devices or not available at all. This leads to a variety of problems for RTAS developers. For supporting a process of simulation model-based RTAS software development simulation tool is needed.

This article introduces DYANA, an integrated environment for development, simulation and verification of RTAS. DYANA is the latest simulation tool from the line of instruments developed by the Laboratory of Computer Systems of Moscow State University.

The first system in the line was STAND [1] (1984-1990). The goal of that project was creating an environment for simulation of distributed programs in order to estimate their performance on target systems with different architectures. It employed the first object-oriented distributed operating system in Russia.

The first version of DYANA was developed during 1994-2001. The main aims of that project were to create a mathematical model of the behavior of a distributed RTAS and to implement a generalized approach to simulation of RTAS. The success of DYANA led to proposal of the Model-Driven Approach to RTAS development. The DYANA was used in various industrial projects, including the project DrTesy [4].

In 2001 we started to develop a hardware-in-the-loop simulation (HILS) environment for avionics systems. Hereafter, this system is referred to as HILS STAND. Some subsystems of DYANA were applied to real-time environment. This tool is successfully used for development of nautical embedded systems as well. Since 2010 we are developing a new simulation tool based on the most popular international standards. The first production version was released in 2012, hence the system is also called DYANA.

This paper gives an overview of the following aspects of DYANA. In section 2, we discuss the basic requirements for a modern simulation environment that were considered before designing DYANA. Section 3 gives an overview of the subsystems of DYANA that implement various specific features. In section 4, High Level Architecture standard used in DYANA is introducted. The format in which RTAS are defined in DYANA is discussed in section 5. The verification subsystem is briefly described in section 6. Finally, in section 7 we give examples of the application of DYANA.

Copyright © 2013 by V.A. Antonenko, E.V. Chemeritsky, A.B. Glonina, I.V. Konnov, V.N. Pashkov, V. V. Podymov, K.O. Savenkov, R.L. Smeliansky, P.M. Vdovin, D.Yu. Volkanov, V.A. Zakharov, D.A. Zorin. Published by the EUCASS association with permission.

2. Requirements for simulation environment of distributed RTAS

We have formulated the following key requirements for our RTAS simulation environment. These requirements were established based on the results of the investigations presented in [25, 7], and the personal experience of the authors of this work:

- *Modular structure of the simulation environment.* The simulation environment should actively use existing software components developed in open source projects. Therefore, the environment should have a well-defined block structure.
- *Distributed simulation.* The simulation environment must support distributed simulation of RTAS.
- *Compatibility of modeling and verification tools.* An important task of the RTAS design is to verify if the behavior of the system complies with its specification. Therefore, the model description format must be suitable both for simulation and formal verification of the system being developed.
- *Compatibility of models.* The modern RTAS is a complex computer system, which consists of a large number of interacting devices. Typically, each component of RTAS is described by a single model or group of models, and these models should be consistent with each other.
- *Scalability of models.* In order to check different properties of the RTAS behavior, it may require different models with different levels of abstraction. These models need to be linked with each other. Therefore, the simulation environment should have a tool to carry out the correct scaling of the RTAS model description [13].
- *The ability to create simulation models of RTAS appliances, as well as environment model.* Development of RTAS using simulation consists of several stages. Initially, each component of the RTAS is a simple software model. Then, the component models become more complex to more accurately reflect the behavior of real devices. In the final stages of RTAS development software component models are replaced by the device prototypes up to the complete elimination of software components.
- *Online and offline simulation.* The simulation system should provide developer tools to run models, suspend, resume, and a full stop of the experiment. Also, the model execution environment must have built-in tools to run standalone experiments without operator intervention.
- *Support interaction with the hardware in the model and real-time using full-scale data channels.* Simulation environment must provide the ability to connect devices using suitable full-scale data channels and maintain the speed of the software models execution sufficient to meet the specifications of the data transfer protocols. The accuracy of the model time binding to physical model time should be measured in tens of microseconds. For a correct construction of simulation models with this accuracy runtime environment must have a minimum response time.
- *The possibility of faults injection to data transfer channels.* An important requirement for RTAS is to provide a given level of resilience to hardware failures. This requirement can be partially checked at startup and run the RTAS model, setting the level of random errors that occur during data transfer between the individual components of the system. Thus, the simulation environment should have fault injection module.
- *Registration and processing of the simulation results, including the interaction with hardware monitors of data transfer channels.* The modern RTAS have hundreds of channels to transmit data correctly and on time [2]. In order to check these properties the simulation environment must record all events occurring in the RTAS model and save it in a form suitable for further processing.
- *Easy to adapt third-party simulation environments for use in conjunction with simulation support library.* The developers of the RTAS individual components may have simulation models of their devices. Using of ready simulation models can significantly ease the development of a new DRE. Therefore, the runtime environment must support the ability to connect third-party models.
- *Simulation environment interoperability with external systems.* Some devices in the RTAS can be developed by competing organizations that refuse to give the developers the software model of these components in order to avoid leakage of their technologies in the simulation process.
- *Distributed simulation.* The simulation environment must support the ability to distributed simulation of DRE.

- *Intercomputer time synchronization.* During the distributed simulation it is necessary to ensure correct global order. The models can be run on different processors (computers), so the time synchronization must remain fairly accurate between them.
- *Openness of the simulation environment.* The simulation environment should be open source. This allows you to increase the transparency of its operation, and provides a great opportunity to support and develop the simulation environment.

Analysis of existing simulation environments [7, 2] showed that all of the existing simulation environments, including DYANA [21] and STAND HILS [2], developed with the participation of some of the authors of this work, do not fully satisfy all of these requirements.

3. DYANA architecture overview

Figure 1 shows the main components of the new system.

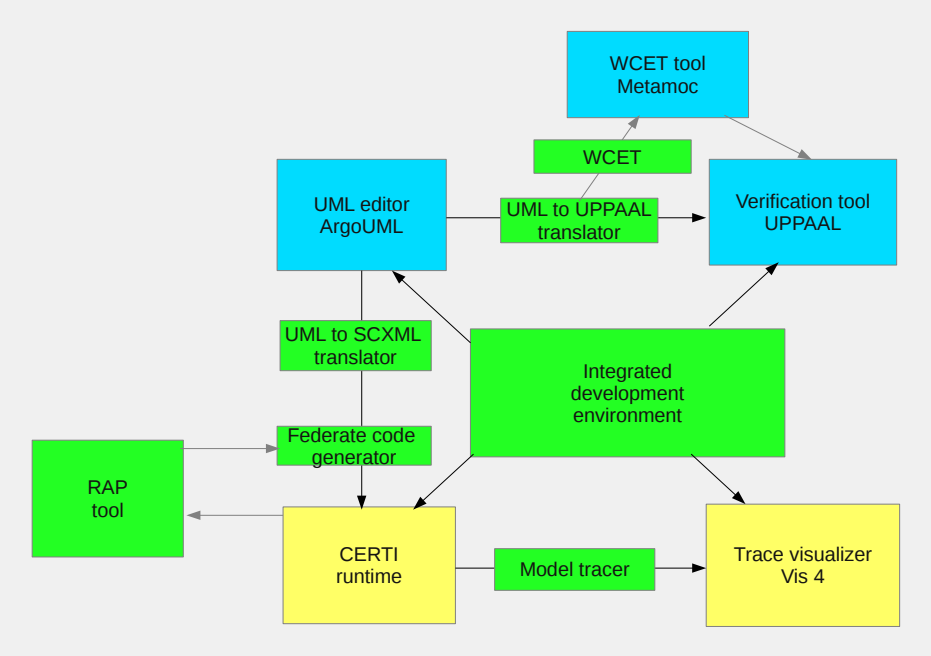

Figure 1: DYANA components

Different colors indicate the degree of reusing open source tools: The blue color designates the tools that were integrated without any modification; The yellow color shows the tools that were substantially modified; The green color highlights the new tools developed exclusively for DYANA.

The user interacts directly with the DYANA IDE. The IDE organizes the files related to the currently developed system, converts the files from one format to another and provides the interface to other components.

We use UML statecharts as the modeling language for real-time systems and ArgoUML [25] as an editor. The integration is done on the level of XMI format, so technically any UML editor that supports XMI can be used instead of ArgoUML.

DYANA is using HLA-based simulation runtime, particularly CERTI [20] for the real-time modeling. As the part of DYANA development efforts we improved CERTI to support multi-thread execution of models [21]. In near future, we are going to contribute the modifications to the CERTI community. Federate Generator produces HLA federates from UML models by a two-step process: first, UML models are translated to SCXML notation, which is providing an intermediate integration point; then, federates in C++ are generated from SCXML representations. Execution traces of models run in CERTI can be visualized in Vis4, the tool based on the tool from [2].

We use UPPAAL [7] as a verification tool for timed automata. UMLToUppaal Tool translates UML statecharts, which represent modeled components, to UPPAAL timed automata as described in [18]. As a byproduct of the translation, the user can check the worst computation estimated time (WCET) by invoking the WCET analysis tool Metamoc [13]. The conversion is applied to the statechart in XMI or SCXML format. The user can also convert the UPPAAL

counterexample to a human-readable format where all states and variables have the same names as in the UML diagram and convert the UPPAAL queries with the names from the UML statechart to the UPPAAL format.

The RAP tool is not a part of the IDE. Its application is discussed in relation to the second case study given in section 7.

The following table gives a summary of the files that can be processed in DYANA and DYANA's capability

Table 1: DYANA file types.

File type	What can be done with it
ArgoUML project	Edit in ArgoUML GUI
XMI file (UML statechart)	Convert to UPPAAL, convert to SCXML
UPPAAL file (Timed automaton)	Open in UPPAAL GUI, verify, convert trace to UML format
SCXML file (UML statechart)	Generate federate code, convert to UPPAAL
CERTI launcher file	Launch simulation
OTF file (CERTI event trace)	Show in Vis
Other file (source code, auxiliary files)	Display contents

4. Simulation runtime

4.1 High Level Architecture simulation standard

HLA is a conventional standard in the field of distributed simulation. HLA introduces its own simulation runtime called the Run Time Infrastructure (RTI). This middleware guarantees the proper functioning of distributed simulation in accordance with the principles and specifications from HLA standard [23]. The actual roots for the HLA stem from distributed virtual environments. Such environments are used to connect a number of geographically separated users. HLA is a conceptual heir of Distributed Interactive Simulation (DIS), which is a highly specialized simulation standard in the domain of training environments [15]. The primary mission of DIS is to enable interoperability among separated simulation systems and to allow the joint simulation of their participation. HLA standard remains relevant to the DIS principles and even extends them.

HLA was introduced in 1993, when the Defense Advanced Research Projects Agency (DARPA) designated an award for developing of an architecture that could combine all known types of simulation systems into a single federation. The HLA standard initially addressed all kinds of as-fast-as-possible, soft and hard real-time, discrete-event and time-driven, fully-synthetic, human- and hardware-in-the-loop distributed simulations. However, hard real-time constraints were not supported until the latest HLA standard version, namely IEEE 1516-2010 (Evolved) released in the very end of 2010. The majority of HLA-based simulation tools were built on the previous HLA standard versions and do not offer a full HLA Evolved support yet.

Thereby, HLA-based HILS became possible quite recently and any researches in this area are innovations in some sense. However, these researches seem to be prospective because of a number of benefits HLA gives. First, HLA strict support by both the runtime and the models provides their guaranteed compatibility. It means that HLA model developed with one runtime can also be used with other runtimes without any modification. In fact, HLA forms an independent market of out-of-the-box simulation models which can be used with any HLA-compatible simulation runtime.

Secondly, HLA is used as an external simulation interface by some non-distributed runtimes. This peculiarity enables joined simulation encompassing diversified runtimes and, consequentially, different model types. For example, a single simulation can include both time-driven fully-synthetic and discrete-event hardware-in-the-loop models simultaneously, and their developers do not have to adjust their models for this cooperation.

In addition, there are a lot of subsidiary runtime-independent HLA-based simulation tools, such as statistic collectors, simulation analyzers, high-level model describing languages and corresponding IDEs. These tools operate at the model level over the HLA API and do not require any additional support from the simulation runtime. Therefore, they can be reused with any runtime implementation.

4.2 Choosing the suitable RTI

There are a lot of off-the-shelf RTI implementations (2) and this fact gives a hope to get some experience from other projects, learning from their mistakes. Thereby, it was decided to explore the area in more details. The study was conducted among the tools, satisfying (at least partially) to the following criteria:

Table 2: RTI Implementations.

RTI	Developer	License type
ARTIS GAIA	University of Bologna	Open Source1
CERTI	ONERA	GPL2 v2 or later
EODiSP	P&P Software	GPL
MAK	MAK Technologies	Commercial
NCWare	Nextel	Commercial
Portico	Portico	CDDL3
pRTI	Pitch Technologies	Commercial
RTI NG	Raytheon	Commercial

1. The description of the architecture and principles of implementation are available;

2. The source code of the product is available;

3. The product is still maintained and developed;

4. The implementation is used for real-time simulation.

Most of the examined tools are commercial, and their source code is unavailable. Thereby, benefits from the use of these implementations are limited to the theoretical base. However, our study found a number of open source systems also, and we decided to build the target simulation system on the basis of the most suitable of them. Unfortunately, all the remaining simulation systems have certain drawbacks in accordance with the purposes of the submitted project. The ARTIS GAIA implementation attracts by its advanced load balancing mechanism supplementation, but the license for this product does not allow the free use of its source code (although it is stated that the project will be fully open in future) [10]. The open source project EODiSP stopped the development in 2006 [9]. Accordingly, there is no one to assist in solving development difficulties encountered. Portico project RTI is implemented using Java and, due to the language specific, it is badly compatible with the real-time simulation that is the primary goal of our project.

Thereby the best base RTI implementation for the development of the considered simulation system a priori is the CERTI one. CERTI is distributed under the GPL license, continues to evolve, and is implemented in C++ (a number of extra bindings including Java, Python, Fortran and even MATLAB is currently available). In addition, CERTI could be deployed on several combinations of platforms (Windows and Linux, Solaris, FreeBSD) and compilers (gcc, MSVS, Sun Studio, MinGW, etc.).

4.3 CERTI

CERTI is an RTI implementation produced by French Aerospace Laboratory (ONERA). The project started in 1996 and its primary research objective was to develop the distributed simulation itself whereas the appeared HLA standard was the project experiment field. CERTI started with the implementation of the small subset of RTI services, and was used to solve the specific applications of distributed simulation theory [20]. Since the CERTI project was open sourced in 2002, a large distributed simulation developer community has been formed around the project. In many ways due to contributions of enthusiasts, the CERTI project has grown from basic RTI into a toolset including a number of additional software components that may be useful to potential HLA users.

The CERTI project has always served a base for researches in the domain of distributed simulation, and a number of innovative ideas have been implemented with its use. For instance, the problem of confidential data leakage was solved in context of CERTI RTI architecture, and the considered RTI guarantees secure interoperation of simulations belonging to various mutually suspicious organizations [8]. The certain interest for the considered project is a couple of application devoted to high performance and hard real-time simulation.

Although HLA is initially designed to support fully distributed simulation applications, it provides a framework for composing not necessarily distributed simulations. Thereby we created an optimized version of CERTI devoted to simulation deployed on the single shared memory platform and composed simulation running on high-performance clusters [1]. Useful experience can also be adopted from ONERA project on simulation of satellite spatial systems. Each federate in this federation is a time-stepped driven one. It imposes an additional requirement of hard real-time: the simulation system should meet the deadlines of each step and synchronize the different steps of the different federates [12].

4.4 The key difference between HILS STAND and CERTI

The key difference in runtimes of CERTI RTI and HILS STAND is the degree of their parallelism. CERTI bases on HLA simulation standard, whose roots stem to distributed virtual environments - games and simulators, which allows geographically separated participants to use a general model of the game world, while providing a sufficient interactivity level [15]. Unlike CERTI, HILS STAND was originally designed as a parallel computer cluster, whose nodes were usually located in the vicinity of each other.

Therefore, the systems were constructed under different requirements, and as a result, they are based on different principles. HILS STAND uses the idea of a common clock. During the simulation hardware clocks of its nodes are synchronized with a high accuracy, and the simulation participants use these local clocks as a system-wide. Thus, the consistency of the simulation model is provided automatically. An assumption of essentially remote nodes does not allow to apply the same approach in HLA-based systems. In accordance with HLA specifications, each simulation participant should use its own clock. The time of this clock is called a logical time of the participant. The logical time of each participant advances independently from the others by means of the RTI time-management services. Consistency of the simulated model is maintained by RTI, whose implementation relies on one of the distributed synchronization algorithms. For instance, CERTI offers a choice of two algorithms for this purpose [11].

Both approaches have their strengths and weaknesses, and choosing the best of them depends on the chosen simulation tasks. Distributed synchronization algorithms usually impose a large overhead. In addition, the careless use of this mechanism can lead to significant performance loss. Imagine a simulation participant, who does not depend on the others and, therefore, is able to advance permanently. Suppose it generates a continuous stream of messages to other participants as fast as it can. The destination participants should handle this stream of messages. But in case their speed is lower than the message-generator one, their logical time is advanced slower, and RTI has to store the unhandled messages in an appropriate local buffer until the right time mark is reached. The bigger buffer grows, the slower it works. However, this fact does not affect the speed of message generation. Thus, the slower buffer works, the bigger it becomes. This situation results into an avalanche growth of total simulation time. However, it can be bypassed by a forced slowing down of the message-generator or size restriction of the buffer.

On the other hand replacing a common system time with a set of independent logical times often allows proactive execution of some simulation participants and may lead to increase in performance in complex real-world simulation problems. Back to our prior example, if the message-generator would perform some complex calculations after the transfer of every five messages, its model time handicap over the other participants would allow it to get more CPU time for its calculations.

5. Using UML to describe RTAS

The models simulated with CERTI are written in C++, however, in practice it is more convenient to design models in specialized modeling languages. We use UML as the most well-known and developed general purpose modeling language. The model is drawn as a UML statechart that can be automatically translated into the internal representation in Python, after that it is converted to .h, .cpp and .fed files, compiled and launched in CERTI.

The statechart consists of simple and composite states and transitions between them. Simple states represent an atomic state of the system.There are two types of composite states: sequential and parallel. Automata residing in a parallel state are executed simultaneously. Composite states have special entry and exit states. Some states are marked with logical formulae called invariants; a system can reside in such state only while its invariant is satisfied.

Each transition between states may be provided with a guard, an action, and a synchronization. Guards express requirements that must be satisfied to enable the transition. Actions are the operations performed after the transition is fired. The syntax of guards, invariants and actions is similar to the syntax of the C language. DYANA also supports including inline code that is added to the federates during code generation, and several other user-friendly features like macros, broadcast signals etc.

We used Cheetah [24], a specialized library of templates, to work with the code generation templates. The main idea of this library is template compiling with cheetah-compile into the representation of patterns in Python. After that Python loader uses this representation for source file generation. Cheetah template consists of a combination of Python source code and code in a special language that looks like Python style code. For federate source code generation (with the RTI interfaces) we developed special templates: separately for .h, .cpp and .fed files. Several cheetah templates for FSM source code generation are used to generate internal logic of the federates. Each state of the FSM is converted to a single C++ class. To control transition from one state to another a special Controller class was developed. Instances of this class are created for each FSM of internal federate logic. This class provides a method to initialize the state and a method to change the state based on newly received events.

6. Verification tools

The verification module of our toolset includes a well-known model-checker UPPAAL [6], a UML-to-UPPAAL translator [18], and a UPPAAL-to-UML counter-example converter. UML-to-UPPAAL translator takes a UML statechart as an input, regards this statechart as a Hierarchical Timed Automaton (HTA), and translates it into a Network of Timed Automata (NTA). In addition the translator also converts the queries addressed to UML statecharts into TCTL formulae that are used as queries in UPPAAL. Then UPPAAL checks an obtained NTA against a TCTL specification. If a NTA does not satisfy a universally quantified TCTL formula then UPPAAL builds a counter-example — a run of NTA which refutes the formula. This run is converted to the corresponding sequence of transitions in UML diagram. In this section we briefly describe the formal models our translator operates with models and outline the idea of translation algorithm.

The concept of *Hierarchical Timed Automata* (HTA) was introduced in [14] to provide a formal operational semantics of UML statecharts. The states of HTA can be nested one into another. There are three types of states, namely, basic, concurrent, and consecutive. A basic state is a primitive of a system and represents a "real" state of the system; no states can be nested into it. A concurrent state include several components nested into it; they represent independent concurrent subcomponents with a standard interleaving semantics. A consecutive state operates as an automaton. Transitions allows HTA to pass control from one state to another on the same nesting level, to activate compound states by entering to their the underlying components, or deactivating compound states by exiting from the underlying components.

States can be marked with invariants of the form $c \leq n$, where c is a real-time clock, and $n \in \mathbb{N}$. A state remains active only unless its invariants hold. Transition between states of the same nesting level are marked with guards and actions. A guard is a Boolean formula over Boolean variables and real-time expressions $c_1 \lhd n$, $c_1 - c_2 \lhd n$, where $\lhd \in \{<, \leq, =, \geq, >\}$. A transition is active if its source is an active state and its guard is satisfied. An action is a set instructions such as assignments to variables, clock resets, sending and receiving of messages via broadcasting channels. Concurrent components of HTA synchronize their behavior by means of shared variables and message exchange. These actions are performed when the transition fires.

At each step of HTA run either time progresses and all clocks increase their values by some amount d, or some set of active transition coherently fire. Formal description of syntax and semantics of HTA can be found in [14].

UPPAAL is a model checker of *Networks of Timed Automata* (NTA) against CTL formulae [6]. A timed automaton (TA) is a set of nodes connected with transitions. Some nodes are distinguished as committed ones: transitions originating from a committed node have the highest priority, and time does not passes until some a transition from an active committed state fires. Nodes of TA are with invariants, and transitions — with guards, synchronizations, and actions. TAs can send and receive messages only via handshake channels.

A network of timed automata (NTA) is a collection of TAs over the same sets of variables, clocks, and channels; this collection can be viewed as a parallel composition of its TAs provided with a usual interleaving semantics. At every step of NTA run either time passes, or some individual active transition which does not involves message exchange fires, or a pair of transitions synchronized by message exchange fire simultaneously. Formal description of syntax and semantics of NTA is given in [6].

Briefly, the *HTA-to-NTA translation algorithm* operates as follows. For every composite state s of HTA A it builds an individual TA P_s intended to simulate activation and deactivation of s by means of message exchanging with the similar automata corresponding to the ambient state s', $s \in s'$, and to the components of s. As soon as P_s receives an activating messages, it sends activating messages to all or to only one of automata (depending on the type of s) that correspond to the components of s. Deactivation of s is simulated in the same way. Committed nodes of TAs make it possible to activate concurrent states simultaneously. The translation algorithm also builds two supervising TA: one of them initializes the whole system of TA, and the other keeps track of the current hierarchy of active compound states of A. The HTA-to-NTA translation algorithm is described in more details in [18]; we improve, complete and extend the original HTA-to-NTA translation technique presented in [14].

7. Case Study

7.1 Traffic lights control system

Just to demonstrate the capability of DYANA to verify and simulate distributed systems designs presented by UML statechrats we use a traffic lights control system as described in [16]. It consists of two traffic lights on a crossroad. The lights are controlled by a processor supplied with some sensors. Lights on the street and on the avenue change colors customary to let cars pass by in both directions. Further, if an ambulance car arrives from any direction, the lights must turn to green on that direction in order to let the ambulance pass as soon as possible. In this case the controller switches

to the mode that opens a fast and safe passage for the ambulance. It is assumed that only one ambulance can arrive at the crossing at a time.

Normally the signals of the traffic light are changed in order allowing cars on both roads to pass: green light on the street lasts 45 time units, then the light turns yellow for 5 units, then 10 units both lights are red and finally the light on the avenue turns green, and so on. There is a sensor that detects the ambulance approaching the crossing. When the ambulance shows up on one of the roads, the sensor sends $appr$ signal to the controller; when the ambulance is close to the crossing, the controller receives the $before$ signal; finally, when the ambulance passes the crossing, the controller receives the $after$ signal. When the first signal is received, the controller turns to safe mode and turns the light red on both roads. When the second signal is received, the light is turned green on the road where the ambulance is. When the third signal is received, the light turns red on both roads and then the normal order is restored.

The model consists of four components — two traffic lights, the ambulance and the controller. The components exchange information via signals and special flag variables. Variables *dir* and *amb* are used to keep track of the ambulance location. Boolean flags *avr*, *avy*, *avg* for the avenue light and *str*, *sty*, *stg* for the street light indicate the current color of the light ($avr = 1$ means that the avenue light is red, $avr = 0$ means that avenue light is not red). The initial state is *ABothRed* (both lights are red).

To verify the model the authors of [16] build its UPPAAL specification manually. We specified this model by means of UML and used our verification system to check the following properties.

1. This property guarantees the absence of deadlocks: $A\Box\neg deadlock$.
2. The lights are synchronized: if the avenue light is green or yellow, the street light must be red and vice versa:
   $A\Box(\neg(stg = 1 \vee sty = 1) \rightarrow avr = 1)$,
   $A\Box(\neg(avg = 1 \vee avy = 1) \rightarrow str = 1)$.
3. This property means that there exists a trace where both lights are green at the same time and it was proved to be invalid: $E\Diamond(stg = 1 \wedge avg = 1)$.
4. The seemingly contrary property is not satisfied too, because there is a situation where one light is red and the other one is yellow: $A\Box(stg = 1 \vee avg = 1)$.
5. Home state for the ambulance car is reachable from the Approaching state, which basically means that the ambulance will always eventually pass the crossing: $Ambulance\_proc.Approaching \rightsquigarrow Ambulance\_proc.Home$.

7.2 Onboard flight controller

The onboard embedded flight control system DrTesy [4] consists of for processors C1, C2, C3, C4 connected with a common bus. Processors C2 and C3 also have shared memory. Each processor works with specific tasks. C1 is the main processor that controls the computations and data flow in the system. Processor C2 calculates the flight parameters and prepares data for sensors. Processor C3 detects the aircraft's position using the data from the sensors and ensures flight on the selected route, and it also controls the sensors. Processor C4 works with the radar data and controls the flight on low altitude.

Our UML model is an abstraction that doesn't deal with exact flight parameters and concentrates on the data flow and communications on the board between processors. The computations are represented with simple states, and their inner structure is treated as a black box.

After initialization, processor C1 works in an endless loop. It has two main states: working mode and timer interrupt mode. All transitions in the main loop have a guard checking that the processor is in working mode, otherwise the loop waits until the interrupt is handled. There are two timers that generate interrupts every 1 and 50 milliseconds respectively.

The main loop contains five sections that describe different behavior scenarios for handling failure of other computers. Sections 1 and 2 load the data and initialize remote computers. In emergency cases when C2 and C3 fail, C1 performs computation on its own. The corresponding algorithms are contained in sections 3, 4 and 5. The whole system can be restarted if a certain flag is set before the beginning of an iteration of the main loop. When the system is restarted, timers are reset as well.

Most states are named SB⟨number⟩ where SB stands for Specific Block. These blocks implement steps of the computational algorithms. The computations that don't affect communication are not specified in the model. Blocks responsible for communication (SB104-SB109) are described in detail.

C1 can be interrupted by timers and when two special flags are set. The first flag signals that it's necessary to send control words to processor C3. The second flag blocks sensors while waiting for data from anti-collision radar.

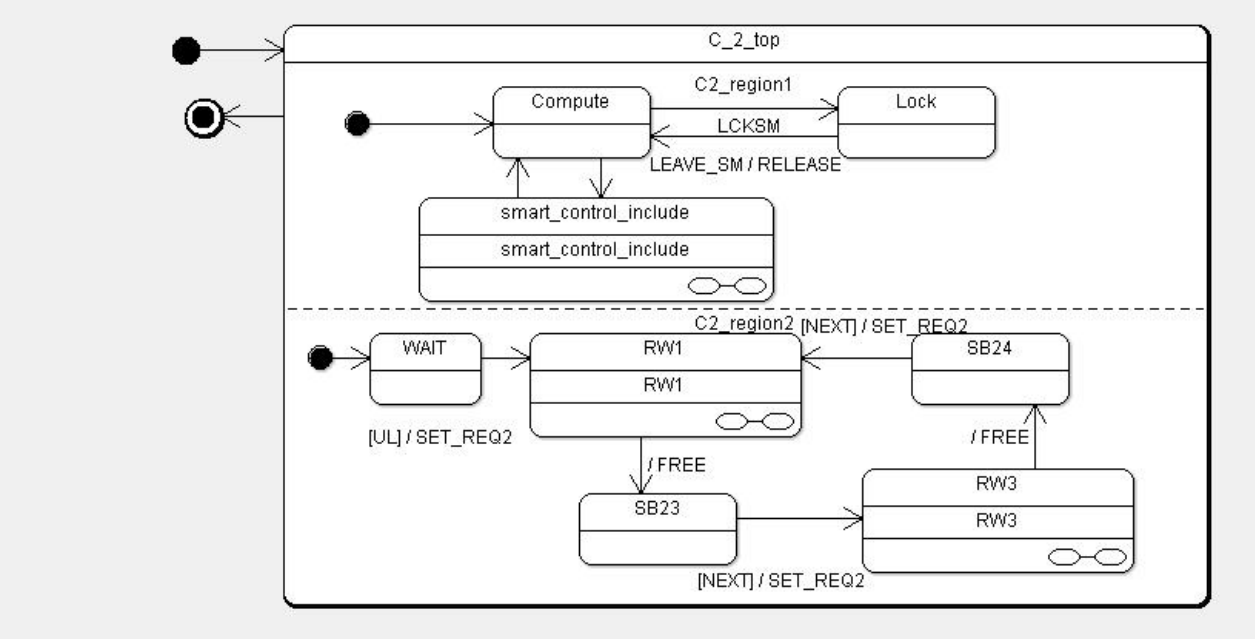

Figure 2: Part of the C2 statechart

New interrupts may appear while C1 is waiting for the bus to transfer data, so the number of received interrupts is counted.

Section 3 is executed if C2 and C3 are working properly. Communication with C4 happens only when the plane is flying at low altitude. Section 4 handles the case when C2 is working and C3 fails, and section 5 handles the case when both C3 and C2 fail.

Processor C2 executes two blocks (23 and 24) in an endless loop. Execution stops when C1 requests data transfer. Processors C2 and C3 have access to shared memory, so the loop can also be stopped if the memory is locked by another processor.

Processor C2 has three states: computation mode, waiting for access to shared memory and interrupt mode. In the computation mode two main tasks are performed: main flight parameters are computed in block 23 and shown on indicators in block 24. Work with shared memory is done between these blocks. Access to the shared memory is regulated with semaphores.

In the interrupt mode the processor C2 exchanges data with C1. There are three cases: initialization of C2, sending flight parameters and receiving control data from C1.

Processor C3 is similar to C2. Its main purpose is computing the parameters related to sensors. Unlike C2, processor C3 has timer interrupt mode when new data is read from the sensors.

Processor C4 works when it's necessary to calculate the parameters of a low altitude flight or when the Collision detection radar is launched. As with other processors, the algorithm to run is determined by the word sent from C1, and the data is sent back during an interruption. There are three states of the processor: idle mode, computation mode and interrupt mode. The processor doesn't use shared memory so there are no peculiarities induced by it.

The following temporal properties should be satisfied due to supposed model behavior of the onboard flight controller model.

- $A\Box notdeadlock$ — a standard deadlock absence property.
- $A\Box(region2.WAIT\_4 \rightarrow A\Diamond C2.SB201)$ — a liveness property that guarantees correct reaction of processor C2 to control words from C1.
- $A\Box(critical\_C2 == false || critical\_C3 == false)$ — a mutual exclusion property both C2 and C3 can't be in the critical section (working with shared memory) at the same time;
- $A\Box((R3\_data == 1) \rightarrow A\Diamond C4.SB30)$ — a liveness property that guarantees correct transition of states in processor C4.

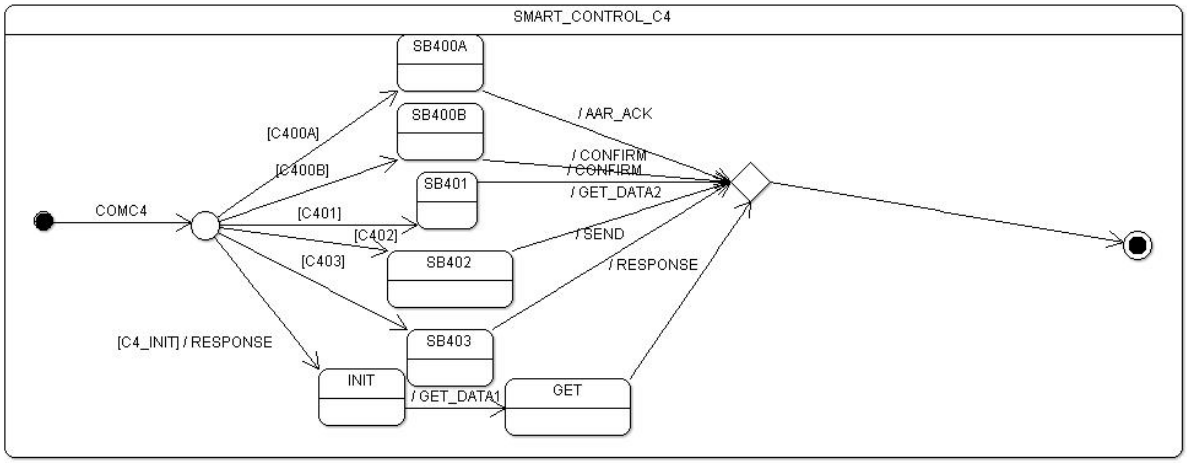

Figure 3: Part of the C4 statechart

- $A\Box((C2\_fault \&\& !C3\_fault) \rightarrow A\Diamond region2.WAIT\_Section3)$ — a liveness property that checks correct behavior of the main loop of C1.

These properties were successfully checked by the verifier. Verification was finished within a few seconds, except for the third property which took several minutes.

This model gives a good example of possible application of DYANA during the design of large-scale embedded systems. The model focuses on a limited functionality related to communication and leaves off the actual computations. The approach of building a model of the slice of the system that contains everything we need allows to observe and check the behavior of the system in a short period of time.

7.3 Integration with RTAS design tools

Another example of using DYANA is a simple RTAS model for measuring execution time of scheduled tasks.

We consider that RTAS consists of a set of processors connected by a network. RTAS program is a set of interacting tasks. During the system design it is necessary to measure tasks execution times repeatedly in order to minimize these times or to verify that time limitations are satisfied. Sometimes tasks execution times can be measured only by simulation or WCET methods. DYANA has a tool for integrating RTAS design programs with the simulation environment for tasks execution times measuring.

Our integration tool creates an RTAS SCXML model from an XML-file containing a schedule. Then SCXML model is converted into HLA federates, which are executed in CERTI. Then simulation output is parsed and an XML file with required times is created.

The RTAS program can be represented with its data flow graph. Each vertex is marked by the time of execution of the corresponding task and each edge is marked by the time of data transfer. A schedule for the program is defined by task allocation, the correspondence of each task with one of the processors, and task order, the order of execution of the task on the processor. [19]

We assume that there may be only one data transfer at any one time. Some real standards, for example MIL-STD-1553, satisfy this restriction.

The main principles of creating RTAS SCXML model from a schedule are described below.

Every processor corresponds to a single state chart. A task execution is represented by a chain of states:

- *wait_data* is the initial state for the task. The chart moves from this state into *work* state after the task has received all required data from other tasks and all previous tasks have finished on this processor.
- *work*. The chart moves from this state to the *wait_channel* state when the task working time elapses.
- *wait_channel*. The chart is being in this state until the data transfer channel is free. Then the chart moves into *send* state and the data transfer channel becomes busy.

- *send*. The chart moves from this state to *end* state and the data transfer channel becomes free when data transfer time elapses. The time of the transfer finish (variable $finish$) is the time which we want to measure.
- *end*. If there is any unexecuted task the chart moves into its *wait_data* state. Otherwise this state is the final state of the chart.

The initial state of the chart is the *waiting_data* state of the first task. *channel_free* — a variable, which indicates if data transfer channel is free. One timer (c) is used.

A fragment of the chart is shown on Figure 4. Note, that SCXML visualization tool doesn't show transfer conditions and triggers, but only transfer events.

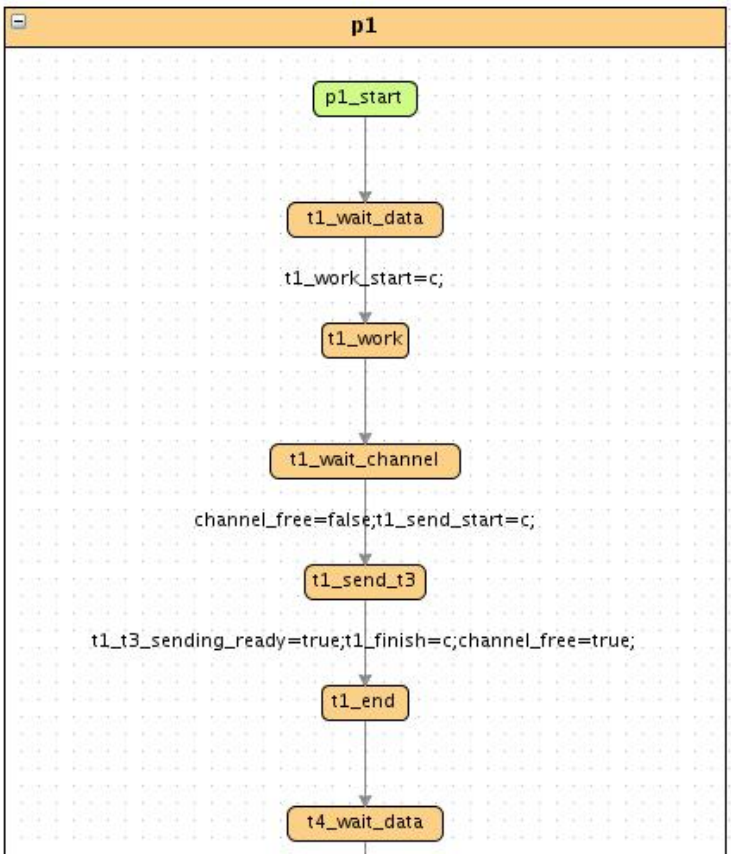

Figure 4: Fragment of the RTAS statechart.

We used this integration scheme with the program which solves reliability allocation problem (RAP). Let us describe this problem informally.

RTES data flow graph is defined. Each vertex corresponds to an RTAS subsystem. Each subsystem consists of a hardware component, a software component and an optional fault tolerance (FT) mechanism. FT is the approach that enables RTAS to continue operating correctly in case of the failure of some of its components. In this study we consider two FT mechanisms: N-version programming (NVP/0/1, NVP/1/1) and recovery blocks (RB/1/1) [22].

For each hardware and software component there is a set of versions of this component. For each version its cost and reliability is defined. We have source code of each version of software component. For each version of hardware component the following characteristics are defined:

- model of CPU pipeline; ARM7 [26], ARM9 [27] and AVR [28] architectures are supported;
- CPU cache characteristics: block size, maximum number of blocks, hit/miss strategies;
- RAM characteristics: access time, block size, number of blocks.

As hardware characteristics and source code are known, an execution time of each software component running on each hardware component can be estimated with WCET tool Metamoc. We can estimate execution times before an optimization algorithm starts. Thus we assume that these times are defined.

Working with WCET estimate consist of the following steps:

1. Creation of the object file using cross-compiler GCC for target architecture. Target architecture is ARM7, ARM9 or AVR.

2. Creation of the executable code for target architecture from the object file. The tool objdump is uses on this step.

3. Translation of the executable code into verification model of the program. The tool Arm-to-uppaal is uses on this step.

4. Adding the model of CPU pipeline to the verification model of the program. The model of CPU pipeline includes CPU cache and RAM characteristics.

5. Execution of UPPAAL verification tool to search the WCET estimation. This step uses the special feature of UPPAAL to find the value of a variable that makes the specified property true. The WCET estimation, therefore, is the maximal value of the execution time variable.

The number of component versions used in the subsystem and number of copies for one version are determined by the FT mechanism chosen for the subsystem. Reliability and cost of RTAS are determined by choice of hardware components, software components and FT mechanisms. There are mandatory restrictions on software components deadlines. Using FT mechanisms increases software components execution times. RTAS which maximizes system reliability under cost and times constraints is to be found. We use an adaptive hybrid genetic algorithm (AHGA) [3] for searching the solution of RAP. It was shown that RTAS configuration in terms of RAP can be represented as a schedule. It allows to compute tasks execution times by simulation using the described integration scheme or by WCET tool Metamoc. Experiments showed that in comparison with AHGA the scheme worked very slow. Therefore we studied some approximation methods in order to decrease required number of simulation experiments or to decrease required number of Metamoc exectution.

8. Conclusion

In this paper we have given an overview of the current features of DYANA, a real-time simulation environment. Key features of the proposed tool:

- Using UML statecharts as the modeling language for real-time systems. We use ArgoUML as an editor of UML statecharts.

- Supporting HLA-based HILS. We improved CERTI to support multi-thread execution of models.

- Converting UML model into timed automata and use UPPAAL as a verification tool for timed automata.

- Checking the worst computation estimated time (WCET) by invoking the WCET analysis tool Metamoc.

- Using of OTF trace format for execution traces of models run in CERTI.

Directions for future research include:

- More thorough exploration of the proposed tool on other examples of RT systems described in literature and on data from real RTES.

- UML-to-UPPAAL translation algorithm can be significantly refined to generate far more state-space saving NTA.

- Creation of tool for automate processing of simulation experiments results.

- Creation of hybrid time synchronization for more fast HILS.

- Research of using metamodels for simulation time optimisation.

References

[1] Adelantado, M., Bussenot, J.L., Rousselot, J.Y., Siron, P., and Betoule, M. 2004. HP-CERTI: towards a high performance, high availability open source RTI for composable simulations. In: *Fall simulation interoperability workshop.*

[2] Bakhmurov, A. G., Balashov, V. V., Chistolinov, M. V., Smeliansky, R. L., Volkanov, D. Yu., and Youshchenko, N.V. 2008. A Hardware-in-the-Loop Simulation Environment for Real-Time Systems Development and Architecture Evaluation. In: *Proc. of the Third International Conference on Dependability of Computer Systems DepCoS-RELCOMEX 2008*.

[3] Bakhmurov, A. G., Balashov, V. V., Glonina, A. B., Pashkov, V. N., Smeliansky, R. L., and Volkanov, D. Yu. 2011. Simulation Modeling Based Method For Choosing An Effective Set Of Fault Tolerance Techniques For Real-Time Avionics Systems. In: *Proceedings of 4th EUCASS European Conference for Aerospace Sciences*.

[4] Bahmurov, A.G., Chistolinov, M.V., Epatko, I.V., Smelyansky, R.L., Winter, K. and Zakharov V.A. 2000. Towards a unified toolset for embedded systems development. In: *Proceedings of the conference UKRPROG-2000 "Problems of Programming"*. 316–322

[5] Bakhmurov, A., Kapitonova, A., and Smeliansky, R. 1999. DYANA: An Environment for Embedded System Design and Analysis. In: *Proc. of 5-th International Conference TACAS'99*. 390–404.

[6] Behrmann, G., David, A., and Larsen, K. G. 2004. A Tutorial on UPPAAL. In: *Lecture Notes in Computer Science.3185*. 200–236

[7] Bengtsson, J., Larsen, K. G., Larsson, F., Pettersson, P., and Yi, W. 1996. UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems. In: *Lecture Notes in Computer Science*. 232–243.

[8] Bieber, P., Raujol, D., and Siron, P. M. 2000. Security architecture for federated cooperative information systems. In: *Proceedings of the 16th Annual Computer Security Applications Conference.*

[9] Birrer, I., Carnicero-Dominguez, B., Egli, M., and Pasetti, A. 2006. EODiSP — an open and distributed simulation platform. In: *International Workshop on Simulation for European Space Programmes.*

[10] Bononi, L., Bracuto, M., D'Angelo, G., and Donatiello, L. 2004. A New Adaptive Middleware for Parallel and Distributed Simulation of Dynamically Interacting Systems. In: *Proceedings of the 8th IEEE International Symposium on Distributed Simulation and Real-Time Applications.* 178–187.

[11] Chaudron, J. B., Noulard, E., and Siron, P. 2011. Design and model-checking techniques applied to real-time RTI time management. In: *Proceedings of 2011 Spring Simulation Multiconference - SpringSim'11*.

[12] D'Ausbourg, B., Siron, P., and Noulard, E. 2008. Running real time distributed simulations under Linux and CERTI. In: *Proceedings of the 2008 Summer Computer Simulation Conference* 43:1–43:9.

[13] Dalsgaard, A. E., Olesen, M. Chr., Toft, M., Hansen, R. R., and Larsen, K. G. 2010. METAMOC: Modular Execution Time Analysis using Model Checking. In: *WCET.OASICS.15*. 113–123.

[14] David, A., Moller, M. O., and Yi, W. 2003. Verification of UML statechart with real-time extensions. IT Technical Report. 009. Department of Information Technology, Uppsala University.

[15] Fujimoto, R. M. 1999. Parallel and Distribution Simulation Systems. *John Wiley & Sons, Inc.*

[16] Furfaro, A., and Nigro, L. 2011. A development methodology for embedded systems based on RT-DEVS. In: *Innovations in Systems and Software Engineering.5.2*. 117–127

[17] Gomaa, Hassan. 2001. Designing Concurrent, Distributed, and Real-Time Applications with UML. In: *ICSE. IEEE Computer Society.* 737–738.

[18] Konnov, I. V., Podymov, V. V., Volkanov, D. Yu., Zakharov, V. A., and Zorin D. A. 2012. On the designing of model checkers for real-time distributed systems. In: *Program Semantics, Specification and Verification: Theory and Applications. The conference materials.*. 72–81

[19] Kostenko, V. A. and Zorin, D. A. 2012. Co-design of Real-time Embedded Systems under Reliability Constraints. In: *Proceedings of 11th IFAC/IEEE International Conference on Programmable Devices and Embedded Systems (PDeS)*. 392–396.

[20] Noulard, E., Rousselot, J.Y., and Siron, P. 2009. CERTI, an open source RTI, why and how. In: *Spring Simulation Interoperability Workshop.*

[21] Smeliansky, R. L., Bakhmurov, A. G., Volkanov, D. Yu., and Chemeritsky, E. V. 2013. An Integrated Environment for Distributed Embedded Real-Time System Design and Analysis. In: *Programming (to be published)*.

[22] Wattanapongsakorn, N., and Coit, D. W. 2007. Fault-tolerant embedded system design and optimization considering reliability estimation uncertainty. In: *Reliability Engineering And System Safety*. 395–407

[23] IEEE. 2000. IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) — Federate Interface Specification.1516-2010.

[24] Cheetah User's Guide. *http://www.cheetahtemplate.org/docs/users_guide.pdf*

[25] ArgoUML Homepage. *http://argouml.tigris.org/*

[26] ARM Ltd. ARM DDI 0210C. Arm7tdmi technical reference manual, 2001. *http://infocenter.arm.com/help/topic/com.arm.doc.ddi0210c/DDI0210B.pdf*, last viewed May 2013.

[27] ARM Ltd. ARM DDI 0180A. Arm9tdmi technical reference manual, 2000. *http://infocenter.arm.com/help/topic/com.arm.doc.ddi0180a/DDI0180.pdf*, last viewed May 2013.

[28] Atmel Corporation. Atmel, avr 8-bit instruction set, 2010. *http://www.atmel.com/images/doc0856.pdf*, last viewed May 2013.