

DYANA

Vitaly Antonenko

Lomonosov Moscow State University
Moscow, Russia

Eugene V. Chemeritsky

Lomonosov Moscow State University
Moscow, Russia

Alevtina B. Glonina

Lomonosov Moscow State University
Moscow, Russia

Igor V. Konnov

Lomonosov Moscow State University
Moscow, Russia

Vasily Pashkov

Lomonosov Moscow State University
Moscow, Russia

Vladislav V. Podymov

Lomonosov Moscow State University
Moscow, Russia

Dmitry Yu. Volkanov

Lomonosov Moscow State University
Moscow, Russia

Vladimir A. Zakharov

Lomonosov Moscow State University
Moscow, Russia

Daniil A. Zorin

Lomonosov Moscow State University
Moscow, Russia

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 Embedded systems (DRE) simulation. DRE 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 for simulation runtime. The statechart diagram is also transformed into a network of timed automata (NTA). After translation into NTA we use UPPAAL for DRE model verification. Results of simulation and verification experiments involving the tool are presented. In the conclusion our future plans are described.

1 DYANA architecture overview

The process of developing development avionic devices and Real-time avionics systems (RTAS) is usually distributed among several teams, each of them located in its own organization. In this paper we

present DYANA, a toolset for modeling and verification of real-time avionics systems, which supports such a distributed development process. This tool is a new revision of DYANA simulation environment (?). The new version of DYANA is designed to support development based on well-known standards such as UML (?) and HLA (?). The following figure shows the main components of the new system: Different

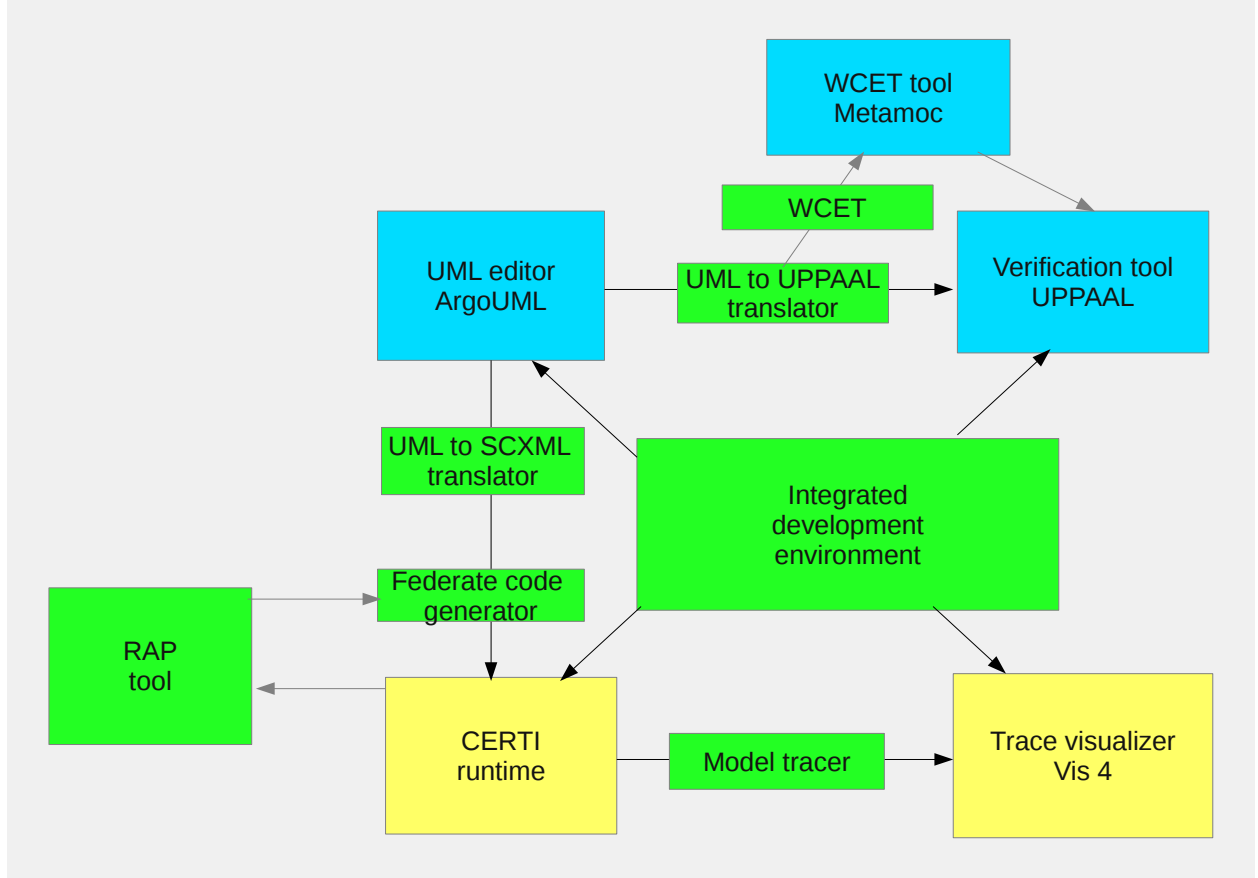


Figure 1: DYANA components

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. DYANA IDE provides the user with a workplace to run different tools within the development system, it also integrates the tools by performing translation between various formats. We use ArgoUML (?) as an editor of UML statecharts, which we use as the modeling language for real-time systems; more details are to be found in Section 2. The integration is done on the level of XMI format, so technically any UML editor that supports XMI can be used instead of ArgoUML. We use UPPAAL (?) as a verification tool for timed automata. UMLToUppaal Tool translates UML statecharts, which represent modeled components, to UPPAAL timed automata as described in (?). As a byproduct of the translation, the user can check the worst computation estimated time (WCET) by invoking the WCET analysis tool Metamoc (?). Verification capabilities are described in Section 3. DYANA is using CERTI (?) as the runtime for the real-time modeling. As the part of DYANA development efforts we improved CERTI to support multi-thread execution of models (?). 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 details on simulation are given in Section 4. In Section 5 we present an RTAS modeling and verification case study using the new version of DYANA.

2 REQUIREMENTS FOR SIMULATION ENVIRONMENT OF DISTRIBUTED EMBEDDED RTS

We have formulated the following key requirements for our DERTS simulation environment. These requirements were established based on the results of the investigations presented in [4, 5], 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 the ability to distributed simulation of DERTS.
- *Compatibility of modeling and verification tools.* An important task of DERTS designing is to verify the compliance behavior of the system with its specification. Therefore the model description format should allow to carry out not only simulation of the developing system, but and its formal verification.
- *Compatibility of models.* The modern DERTS is a complex software and hardware system, which consists of a large number of interacting devices. Typically, each component of DERTS 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 DERTS 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 DERTS model description [7].
- *The ability to create simulation models of DERTS appliances, as well as environment model.* Development of DERTS using simulation is carried out in several stages. Initially, each component of the DERTS 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 DERTS 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 DERTS is to provide a given level of resilience to hardware failures. This requirement can be partially checked at startup and run the DERTS 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 DERTS have hundreds of channels to transmit

data correctly and on time [8]. In order to check these properties the simulation environment must record all events occurring in the DERTS 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 DERTS individual components may have simulation models of their devices. Using of ready simulation models can significantly ease the development of a new DERTS. Therefore, the runtime environment must support the ability to connect third-party models.
- *Simulation environment interoperability with external systems.* Some devices in the DERTS can be developed competing organizations that refuse to give developers the software model of these components in order to avoid leakage of their technologies in the simulation process.
- *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 between them must be maintained fairly accurate time synchronization.
- *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.
- *Distributed simulation.* The simulation environment must support the ability to distributed simulation of DERTS.

Analysis of existing simulation environments [5, 8] showed that the above requirements are not fully satisfy either one of the existing simulation environments, including DIANA [9] and stand HILS [8], developed with the participation of some of the authors of this work.

3 High Level Architecture simulation standard

3.1 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). The RTI is the software implementation of the HLA Interface Specification. This middleware guarantees the proper functioning of distributed simulation in accordance with the principles and specifications from HLA standard (?). 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 (?). 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 appeared 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. At 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.

3.2 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 (?). 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 common clock. During the simulation hardware clocks of its nodes are synched 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 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 is advanced 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 (?).

Both described approaches have their strengths and weaknesses, and choosing the best of them depends on the chosen simulation tasks. Distributed synchronization algorithms usually require a large overhead. In addition, the careless use of this mechanism can lead to significant performance loss. Suppose 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 have 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 of a common system time with a set of independent logical times often allows proactive executing 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.

3.3 Choosing the suitable RTI

There are a lot of off-the-rack RTI implementations (??) and this fact gives a hope to get some developments 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:

1. The description of the architecture and principles of implementation are available;
2. The source code of the product is available.

Table 1: RTI Implementations.

RTI	Developer	License type
ARTIS GAIA	University of Bologna	Open Source ¹
CERTI	ONERA	GPL2 v2 or later
EODiSP	P&P Software	GPL
MAK	MAK Technologies	Commercial
NCWare	Nextel	Commercial
Portico	Portico	CDDL ³
pRTI	Pitch Technologies	Commercial
RTI NG	Raytheon	Commercial

3. The product continues to maintain and develop;
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, taken by the developers of the target simulation system, are limited to the theoretical base. However, the study found a number of open source systems also, and it was decided to build the target simulation system on the basis of the most suitable of them. Unfortunately, all the remained simulation systems have a 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) (?). The open source project EODiSP stopped the development in 2006 (?). Accordingly, there is no one to assist in solving of possible 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 a primary goal of considered project.

Thereby the best base RTI realization 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.).

3.4 CERTI

CERTI is the RTI implementation produced by the French Aerospace Laboratory (ONERA). The project started in 1996 and its primary research objective was 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 concrete applications of distributed simulation theory (?). 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. Thus, the problem of confidential data leak was solved in context of CERTI RTI architecture, and the considered RTI guarantees secure interoperation of simulations belonging to various mutually suspicious organizations (?). The certain interest for the considered project is a couple of application devoted to high performance and hard real-time simulation.

In spite of HLA is initially designed to support fully distributed simulation applications, it provides a framework for composing not necessarily distributed simulations. Thereby there was created an optimized

version of CERTI devoted to simulation deployed on the same shared memory platform and composed simulation running on high-performance clusters (?). Some experience can also be adopted from ONERA project on simulation of satellite spatial system. 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 (?).

4 Code Generation Tool

4.1 SPECIFICATIONS OF SOURCE CODE GENERATION TOOL

One of the main features of DYANA is the generation of HLA-compatible source code from UML statecharts diagrams. The internal federate logic code generation tool has to possess the following features:

- C++ source code generation based on statechart diagrams;
- "inline" code integration in statechart diagram;
- using third-party libraries of simulations components.
- Graphical User Interface;
- developing HLA-compatible models (RTI interfaces for simulation models generation);
- freeware distribution.

4.2 GENERAL FEDERATE CODE GENERATION SCHEME

Our simulation models code generation tool based on UML statecharts was implemented in Python programming language. ArgoUML is used to create and edit UML statecharts. General process scheme of code generation is represented on figure ??.

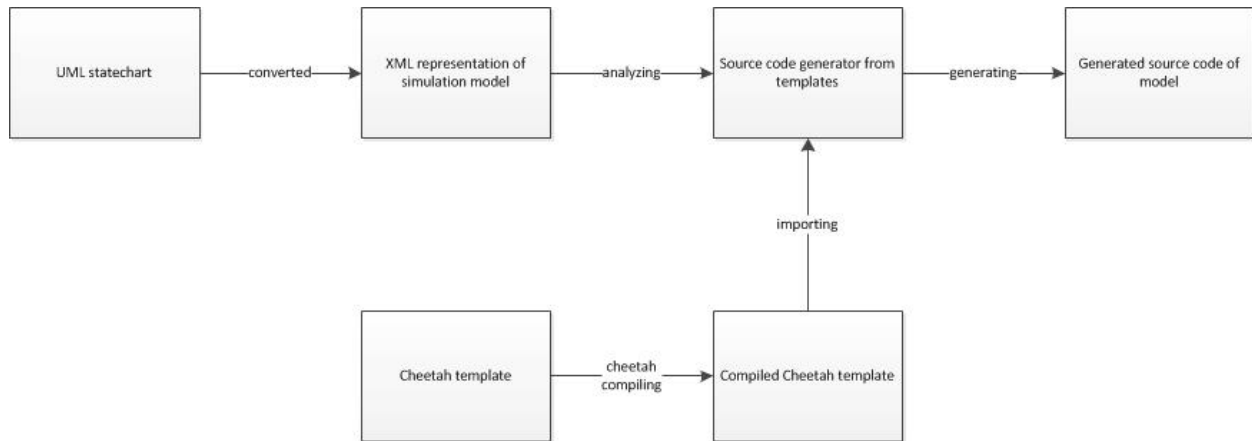


Figure 2: Code generation scheme

Two things are required for code generation: UML statechart representation of simulation model and the templates for code generation. The statechart is translated into the internal representation in Python, after that it is converted to .h, .cpp and .fed files.

As XML representation we use a specialized XML format specifically designed to work with statecharts - State Chart XML (?). This format allows to define finite state machines based on Harel statecharts. Using SCXML we can describe the different types of structures of finite state machines (nesting, synchronization, concurrency etc.).

We used Cheetah (?), a specialized library of templates, to work with the code generation templates. The main idea of this library is as follows: 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. The source code generation scheme for developing HLA-compatible models is shown on figure 2.

Several cheetah templates for FSM source code generation are used to generate internal logic of the federates. The following entities are used:

- UML Note. Notes can be used to explain and to define additional attributes of the objects inside the internal logic.
- UML Generalization. Connects objects and notes containing detailed information about the object.
- UML State Term. Define the initial and final states. These states do not have any names: for the initial state it is not required; for the end it's necessary to add a Note with the name of the state and link with the final state with Generalization.
- UML State. Defines the state of FSM. There is the list of State attributes:
 - entry action — action to do while entering the state. Not execute in internal transitions.
 - do action — action to do after internal transition.
 - exit action — action to do while exiting the state. Not execute in internal transitions.
- UML Transition. Defines the transition from one state to another. There is the list of Transition attributes:
 - trigger — transition condition (event).
 - action — action to do while transition.
 - guard — additional transition condition.

Each state of the FSM is converted to a single C++ class. To control transition from one state to another the 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.

There is a method to add third-party header files to be included in the generated source code. There is also functionality to add source code on statechart diagram ("inline code"). To do this, the user has to link a note with method source code and FSM.

5 Case Study

5.1 Traffic lights control system

The simple example we use to demonstrate the functioning of DYANA in detail is a traffic lights control system as described in (?). The traffic lights control system 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 remember where the ambulance is. 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).

In [4] the authors constructed a UPPAAL model for this system manually to verify its properties. We used our translator and obtained the model automatically.

The following properties were tested.

1. This property guarantees the absence of deadlocks:
 $A\Box\neg\textit{deadlock}$.
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 false. :
 $E\Diamond(stg = 1 \wedge avg = 1)$.
4. At the same time the seemingly contrary property is not fulfilled, because there can be 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:

$$\textit{Ambulance_process_proc.Approaching_active_in_Ambulance} \rightsquigarrow \textit{Ambulance_process_proc.Home_active_in_Ambulance} .$$

5.2 Integration with RTAS design tools

Another example of using DYANA is a simple RTAS model for measuring execution times 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. 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. (?)

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:

- **Waiting For Data** is the initial state for the task. The chart moves from this state into the Working state after the task has received all required data from other tasks and all previous tasks have finished on this processor.
- **Working**. The chart moves from this state to the Waiting For Channel state when the task working time elapses.
- **Waiting For Channel**. The chart is being in this state until data transfer channel is free. Then the chart moves into the Sending state.
- **Sending**. The chart moves from this state to the End state when data transfer time elapses. The time of the transfer finish is the time which we want to measure.
- **End**. If there is any unexecuted task the chart moves into its Waiting For Data state. Otherwise this state is the final state of the chart.

The initial state of the chart is the Waiting For Data state of the first task.

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

RTAS 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 considered two FT mechanisms: N-version programming (NVP/0/1, NVP/1/1) and recovery blocks (RB/1/1) (?).

For each hardware and software component there is a set of versions of this component. For each component its cost and reliability is defined. An execution time of each software component running on each hardware component is defined. 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) (?) 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. 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.

6 Translation algorithm

The core of a verification part of our tool is the translation algorithm, based on the algorithm, presented in (?), which takes a UML statechart as an input and generates a network of timed automata which can be verified with UPPAAL. In this section we give a brief description of input and output models and the main steps of the translation algorithm.

The concept of *Hierarchical Timed Automata* (HTA) was introduced in (?) to provide a formal operational semantics of UML statecharts. HTA operates on a set of states L nested one into another, starting with a set of initial states $L_0 \subseteq L$. To show that s' is nested into s , we will write $s' \in \eta(s)$ using a nesting function η . Some nested states can be marked as exit states and allow to deactivate their ancestors. 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, and no other state can be nested into it. States, nested into a concurrent state, represent independent concurrent subcomponents with a standard interleaving semantics. A consecutive state operates as an automaton.

States can be marked with invariants, which are expressions $c \triangleleft n$, where $n \in \mathbb{N}$, c is a real-time clock, and $\triangleleft \in \{<, \leq\}$. Invariants are necessary conditions for states to be active. States with a common ancestor can be connected with transitions, which can be marked with guards and actions. A guard expresses a

necessary condition for a transition to be performed, expressed by a boolean formula over boolean variables and real-time expressions $c_1 \triangleleft n$, $c_1 - c_2 \triangleleft n$, where $n \in \mathbb{N}$, $c_1, c_2 \in \text{Clock}$, and $\triangleleft \in \{<, \leq, =, \geq, >\}$. An action is a set of assignments to variables and clock resets, i.e. assignments to zero.

System components can synchronize not only with variables, but also with broadcast signals. Transitions can be marked with special actions, namely $!!c$ to send and $??c$ to receive a signal via a broadcast channel c . When a signal is sent, all transitions which can receive it are performed immediately.

An HTA computation is a sequence of steps, and each step defines a set of active states and valuations for variables and clocks. During a step an HTA either makes a time increase, or performs a transition (or several transitions, according to a broadcast synchronization). In the first case active states and variables do not change, and all clocks increase their values by some real number d . In the second case a set of transitions to perform is defined, and after they performed, variables and clocks change their values according to action labels. Strict syntax and semantics of HTA (except of broadcast synchronization) are given in (?).

The outcome of the HTA translation is a *network of timed automata*, which can be processed by UPPAAL verification tool (?) to model-check temporal properties of a given system.

A timed automaton (TA) consists of nodes connected with transitions, with a special initial node. Some nodes of TA can be marked as committed. If a committed node is active, then no time increase is allowed, and a transition originating from a committed node has the highest priority to perform. As in the definition of HTA, nodes of TA can be marked with invariants, and transitions — with guards, synchronizations, and actions. The difference between the labels for HTA and TA is that TA can send and receive signals via handshake channels.

A network of timed automata (NTA) consists of a TA set with common variables, clocks, and channels. Every NTA may be viewed as a parallel composition of its TA with a standard interleaving semantics. As in the case of HTA, a computation of an NTA is a sequence of steps, and each step defines valuations for variables and clocks and a set of active nodes, one for each TA, and on each step either time is increased, or some transitions are performed. Strict syntax and semantics of NTA are given in (?).

The *translation algorithm* generates a TA P_s in the outcoming NTA for each non-basic state s of HTA. Initially P_s has an isolated ordinal node *idle*, which represents the inactivity of the state s .

If s is a concurrent state, then the only other ordinal node in P_s is the state *active*. The node *idle* is connected to the node *active* via the sequence of committed nodes *activate* $[s']$ for each state s' nested in s . The first transition in this sequence receives the activation signal for the TA P_s . All other transitions send activation signals for the nested states. Also, the node *active* is connected to the node *idle* with the transition which receives the deactivation signal.

If s is a consequent state, then nodes *active* $[s']$ are added in P_s for each state s' nested in s . If s' is a non-basic state, then a committed node *aux* $[s']$ is also added and connected to *active* $[s']$ with a transition, which send the activation signal for s' . All HTA transitions within s , which do not originate in a non-basic state, are translated into transitions from *active* $[\cdot]$ to *active* $[\cdot]$ (for basic substates) and *aux* $[\cdot]$ (otherwise) nodes. Initial and final states in such description can be considered as connected to a special s_{idle} state, and transitions, connected with *idle* in P_s , receive activation and deactivation signals. If a transition originates in a non-basic state, then a special set of sequences of committed nodes is added to P_s , one for each set of recursively nested basic states, from which a deactivation of s can be performed. Deactivation signals for recursively nested states are sent during these sequences.

To initialize the NTA, the algorithm performs the initial marking. If a state s is not initial, then the node *idle* of the process P_s is declared initial. Otherwise, either node *active* (for a concurrent state s), or *active* $[s']$ (for a consequent state s and a nested initial state s') is declared initial.

7 Conclusion

Here be conclusion