An experience on using simulation environment DYANA augmented with UPPAAL for verification of embedded systems defined by UML statecharts

Alevtina B. Glonina[1], Igor V. Konnov[2], Vladislav V. Podymov[1], Dmitry Yu. Volkanov[1], Vladimir A. Zakharov[1] and Daniil A. Zorin[1]

[1] Lomonosov Moscow State University, Moscow, Russia
alevtina@lvk.cs.msu.su, valdus@yandex.ru, dimawolf@cs.msu.su, zakh@cs.msu.su, juan@lvk.cs.msu.su
[2] Technische Universität Wien, Vienna, Austria
konnov@forsyte.at

Abstract

DYANA [1] is an environment designed for development of distributed systems. We demonstrate that DYANA augmented with UPPAAL is well suited for simulation and model checking of real-time embedded system designs defined with UML statecharts. To this end, we present the case studies of onboard systems for cars and aircrafts as well as of a general-purpose fault-tolerant system for running parallel programs. For each case study we give its informal description, specify a formal model defined with UML diagrams and report on the simulation and model checking results.

1 Introduction

Designing of embedded systems is a very difficult task, since such systems usually are highly heterogeneous, operate concurrently with numerous real-time computing constraints. Usually an embedded systems includes processing cores that are either microcontrollers or digital signal processors, sensors, control units, monitors, etc. Nowadays, due to significant achievements in micro-electronics, embedded systems become so much complex that their full-grown development without special tool support is impossible.

For this purpose in 1994-2001 in Moscow State University a software environment for simulation of distributed real-time systems DYANA (DYnamic ANAlyser) has been developed [1]. The initial targets of the project were elaboration of mathematical models for dynamic distributed computer systems, implementation of a comprehensive approach to simulation of distributed systems which includes both a qualitative analysis of system behavior and quantitative analysis of its parameters and characteristics. This project brought one of the first successful implementations of the Model Driven Approach to the designing of real-time distributed systems. Since that time DYANA has been successfully used in the designing of a number of airborne and shipborne computer systems.

Since the end of 90-s the developers of DYANA understood that the simulating software environment does not cover the whole spectrum of tasks unless verification tools are involved. Upon making some attempts to integrate various verification tools in DYANA [5], it became clear that formal verification of embedded systems yields the most benefits at the earliest

[1]This research is supported in part by the Skolkovo Foundation Grant N 79, Russian Ministry of Education and Science grant, and RFBR grant 12-01-00706

stage of their development. At this stage the embedded system can be specified by means of statecharts. Since the behavior of a statechart is somewhat similar to that of a net of timed automata, verification of statecharts can be performed with the help of any model checking tool for real time automata. Thus, we arrived at the conclusion to supply DYANA with a verification module for the model checking of UML diagrams by means of UPPAAL. Verification tools are integrated into the general DYANA GUI. The main use cases that can be triggered by the user are: conversion of a UML statechart to a Network of Timed Automata (NTA), verification of the NTA in UPPAAL [9], and browsing the counterexample.

The scheme of the architecture of DYANA is shown on figure 1

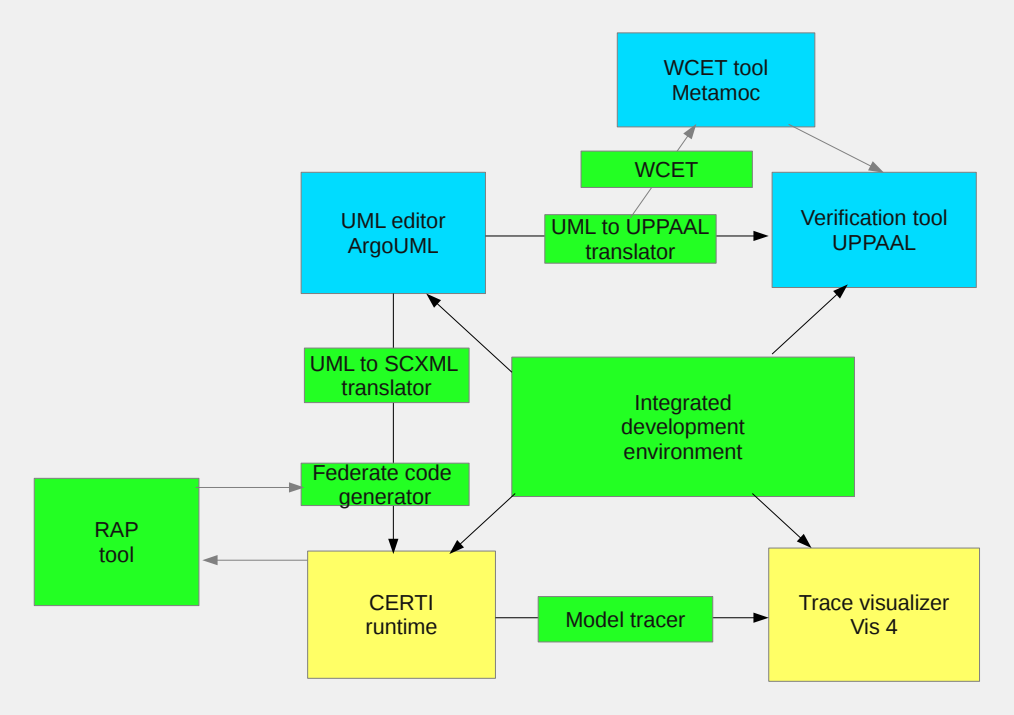

Figure 1: DYANA components

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.

ArgoUML [2] is an editor of UML statecharts, which we use as the modeling language for real-time systems. 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 CERTI [17] 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 [18]. 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 [6].

Our UML-to-UPPAAL translator operates as follows. It takes UML statecharts in the widespread XMI format as input, transforms it into the intermediate form — a Hierarchical Timed Automaton (HTA) [11], — and finally apply the algorithm described in [15] to transform a HTA into a NTA. The translator also converts the queries addressed to UML statecharts into TCTL (Timed Computational Tree Logic) formulae that are used as queries in UPPAAL. Then UPPAAL checks an obtained NTA against a TCTL specification. If an 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.

DYANA works in cooperation with analysis tool Metamoc [13] for checking the worst computation estimated time (WCET). The method implemented in Metamoc, Modular Execution Time Analysis using Model Checking, is a static method providing safe and tight WCET bounds, but utilising real-time model checking to establish WCETs. The use of model checking in Metamoc provides a very modular approach for dealing with these techniques: the model to be analysed comprises an abstract model of the program, and similarly for the component models for the hardware platform, which include caches, pipelines and memories. Metamoc works in cooperation with UPPAAL verification tool [8].

The goal of this paper is to demonstrate that our verification tool for UML diagrams is applicable to various embedded systems. In the next section we briefly describe the basic ideas of the algorithm for translation of HTA to NTA. Then in sections 3-5 we describe in some details the case studies of an onboard system for traffic control and an airborne navigation system as well as of a general-purpose fault-tolerant system for running parallel programs. For each case study we give its informal description, specify a formal model defined with UML diagrams and report on the simulation and model checking results. Finally, we outline some directions for future research.

2 UML to UPPAAL translation

The verification subsystem of our toolset includes a well-known model-checker UPPAAL [9], a UML-to-UPPAAL translator [15], 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 [11] 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 includes several components nested into it; they represent

independent concurrent subcomponents with a standard interleaving semantics. A consecutive state operates as an automaton. Transitions allow HTA to pass control from one state to another on the same nesting level, to activate compound states by entering to their 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 such instructions 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 [12].

UPPAAL is a model checker of *Networks of Timed Automata* (NTA) against TCTL formulae [9]. 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 pass until some transition from an active committed state fires. Nodes of TA are marked with invariants, and transitions are labelled 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 [8].

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 provide simultaneous activation of concurrent states. 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 [15]; we improve, complete and extend the original HTA-to-NTA translation technique presented in [12].

3 Onboard car controller

3.1 Description

The model [14] describes an open intersection of two roads (without circular motion). The intersection is represented by several sections, namely four sections "before" the intersection (which can be labeled as west, east, north, and south), four sections "after" (west, east, north, and south also), and four critical sections (north-west, north-east, south-east, and south-west).

Each critical section can be occupied by at most one car at the same time, while any number of cars can occur at the other sections.

To explain the meaning of sections, consider a car that is going to pass the crossing from south to west. First it arrives at the south-before section. Then it moves to the south-east, north-east, and north-west sections consequently. Finally it reaches the west-after section and leaves the crossing.

A car is modeled by two sets of parameters, namely physical parameters and a control part. Physical parameters describe the car's position, speed, and closest section to stop. To take into account control delays, we divide a control part into decision component and processor. The first component instantly decides either to stop, or to accelerate, or to move normally, according to current road situation. The second component implements the first component's decision after delay. Also, we assume that each car has fixed initial and target directions.

We call a traffic situation *standard* if the car follows traffic rules to avoid accident situations. Following rules are taken into account:

- A car should give way to cars moving from its right;
- If a car is moving to the left, then it should give way to cars moving from the opposite lane to the left or straight ahead;
- If a car has no chance to stop in time to meet previous rules, then it should be allowed to pass.

The decision component takes these rules into account, as well as the rules of common sense (e.g. to slow down if the next section is occupied).

To analyze the system, we assume that there is an upper limit N for a number of cars that can arrive at the crossing simultaneously. Then the whole system can be expressed in terms of UML statechart diagrams. The system consists of the crossing and N cars working independently.

The crossing is divided into two parts. The first part (Figure 2) counts the number of cars that have arrived on "before" section ($cars\_b$), and the number of cars that should, but cannot keep the first rule ($cars\_n2$) from each side. To provide it, broadcast signals (arr, oc, $stop$) sent by cars are being processed by this component. The second part (Figure 3) handles critical sections represented by boolean variables. If two cars appear on a critical section, then the diagram changes its state to *accident*. Here and further indices i and j that occur in transition labels mean several copies of the transition with all possible values for i and j.

A car is modeled by the set of diagrams representing its characteristics. The states of the position diagram (Figure 4) represent current section occupied by the car and the way the sections are changing. The states *arriving* and *changeij* are auxiliary and passed with no time delay. The speed diagram (Figure 5) consists of three states representing zero, nonzero, and the highest possible speed of the car. The stop (Figure 6) diagram handles the counter *cou_stop*, which stores the number of the closest section at which the car can perform a full stop. The driver diagram (Figure 7) represents the current strategy of the car: either to stop, or to reach the highest possible speed, or to pass with any nonzero speed according to the traffic rules. The last diagram (Figure 8) describes a processor, which reads the current strategy stored in *cou_drv* and copies it to *cou_act* — a variable, which represents an actual strategy.

3.2 Experiments

The following temporal properties have been checked against the crossing model:

- $A\Box notdeadlock$ — a standard deadlock absence property that also allows to estimate a state space size;
- $A\Box(position.before \rightarrow A\Diamond position.passed)$ — a liveness property that should guarantee that once a car reached the crossing, it will eventually pass it;
- $A\Box(from == 0 \&\& from\_car2 == 1 \rightarrow not(position.pos\_1 and position\_car2.pos\_2))$ — a mutual exclusion property: two cars moving from the east and south cannot appeat at the north-east section;
- $A\Box notplaces.accident$ — informally, it is a shorthand for mutual exclusion properties for all critical sections; however, this property is formulated in different terms and can be checked faster than the whole scope of mutual exclusion properties.

Truth, time, and space results for the properties are given in Table 1.

Table 1: Experimental results.

Property	1 car			2 cars			3 cars		
	Time (sec)	States checked	Satis-fied	Time (sec)	States checked	Satis-fied	Time (sec)	States checked	Satis-fied
Deadlock absence	<1	1161	yes	410	743303	yes	t.-o.	$17 * 10^6$	-
Liveness	<1	15	no	<1	41	no	<1	67	no
Mutual exclusion	-	-	-	400	808945	yes	t.-o.	400000	-
Accident absence	<1	1325	yes	390	808945	yes	t.-o.	400000	-

A counterexample to the liveness property can be described as follows: a car appears at the crossing, and then it continuously slows down, but doesn't stop. This is a strategy of a timid and overcautious driver who will never pass any crossing. Although this way of driving looks curious, it does not viloate any traffic rule.

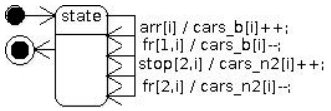

Figure 2: Counters diagram.

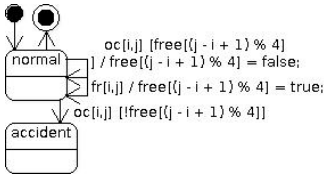

Figure 3: Places diagram.

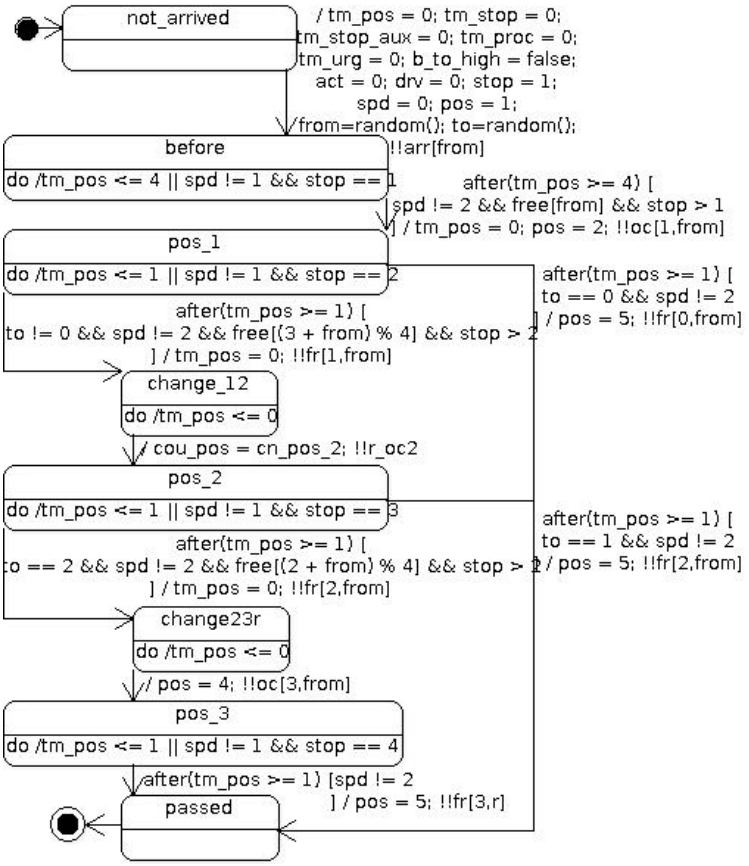

Figure 4: Position diagram.

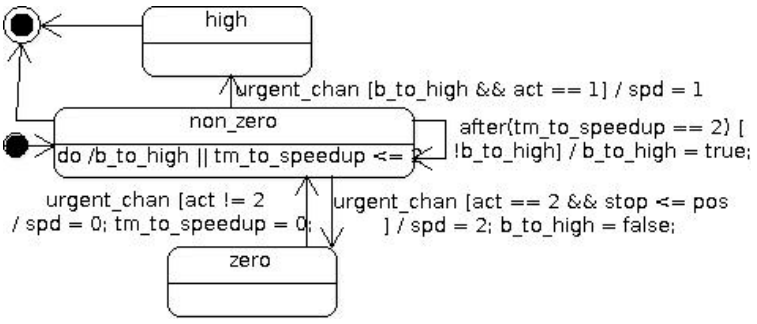

Figure 5: Speed diagram.

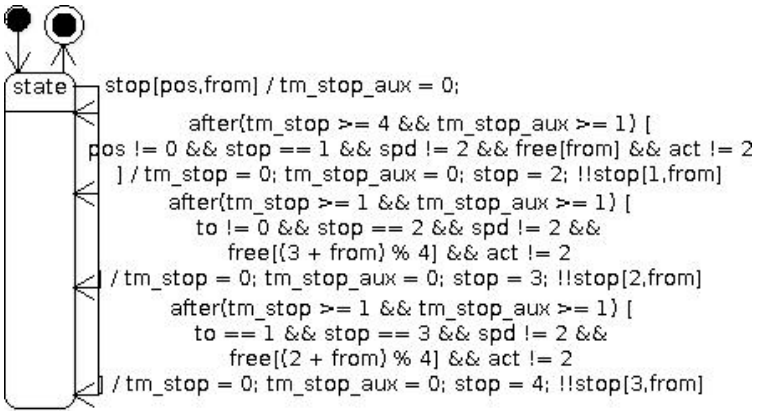

Figure 6: Stop diagram.

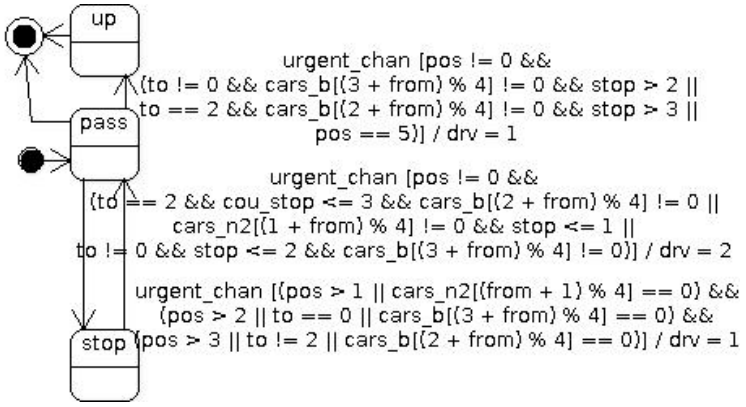

Figure 7: Driver diagram.

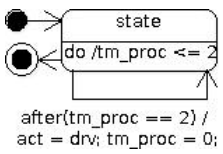

Figure 8: Processor diagram.

4 Onboard flight controller

4.1 Description

The onboard embedded flight control system DrTesy [5] 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. 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

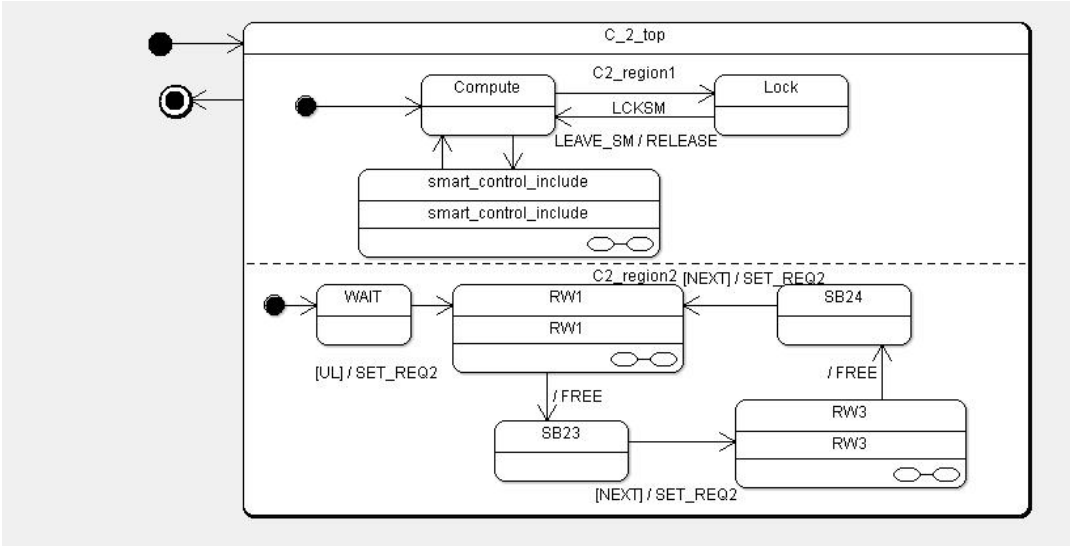

Figure 9: Part of the C2 statechart

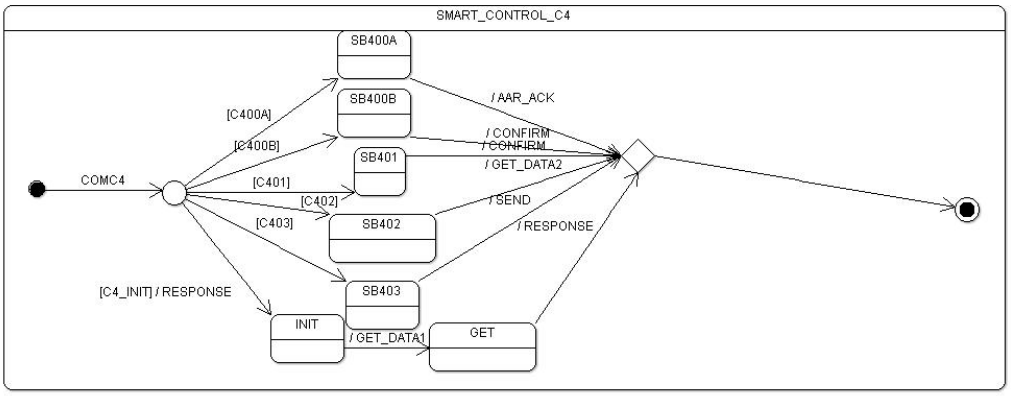

Figure 10: Part of the C4 statechart

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.

4.2 Experiments

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.
- $A\Box((C2\_fault \&\& !C3\_fault) \rightarrow A\Diamond region2.WAIT\_Section3)$ — 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.

5 Integration with RTES design tools

Another example of using DYANA is a simple Real-time Embedded System (RTES) model for measuring execution time of scheduled tasks.

We consider that RTES consists of a set of processors connected by a network. RTES 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 RTES design programs with the simulation environment for tasks execution times measuring.

Our integration tool creates an RTES 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 RTES 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. [16]

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 RTES 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 11. Note, that SCXML visualization tool doesn't show transfer conditions and triggers, but only transfer events.

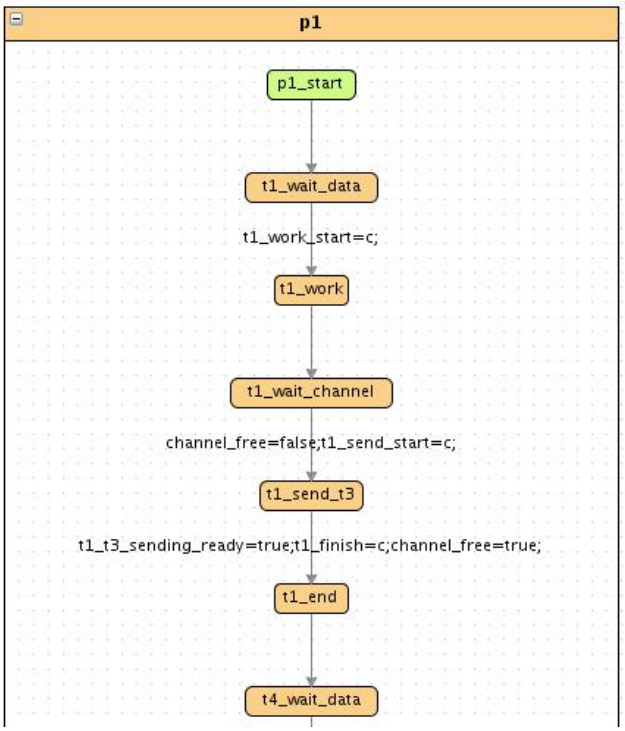

Figure 11: Fragment of the RTES statechart.

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 RTES subsystem. Each subsystem consists of a hardware component, a software component and an optional fault tolerance (FT) mechanism. FT is the approach that enables RTES 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) [19].

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 [4], ARM9 [3] and AVR [10] 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 RTES 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. RTES which maximizes system reliability under cost and times constraints is to be found. We use an adaptive hybrid genetic algorithm (AHGA) [7] for searching the solution of RAP. It was shown that RTES 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.

6 Conclusion

The case studies presented in this paper show that the combined model-checking tool is well-suited for verification of a wide spectrum of UML model of embedded systems. The only

limitation we encounter with is the capability of UPPAAL model checker. Now we are working on further improvement and enhancement of the verification module. First, we think that UML-to-UPPAAL translation algorithm can be significantly refined to generate far more state-space saving NTA. Another possible direction of research is to combine a model-checker with the simulation tools of DYANA. Such integration could increase the effectiveness of both the simulation and verification of embedded systems. We are searching also for new case studies our toolset could be applied to.

The authors are grateful to the reviewers for their comments that helped us to improve the paper.

References

[1] Bakhmurov A.G. and Smeliansky R.L. Dyana - the pilot project of investigation of distributed programs and computer systems. In *Proc. of the 2nd Russian-Turkish Seminar New High Information technologies*, pages 72–81, Gebre, Turkey, 1994.

[2] ArgoUML. Argouml homepage. Accessed April 8, 2013. `http://argouml.tigris.org/`, 2013.

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

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

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

[6] A. G. Bakhmurov, V. V. Balashov, M. V. Chistolinov, R. L. Smeliansky, D. Yu. Volkanov, and N.V. Youshchenko. 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*, 2008.

[7] A. G. Bakhmurov, V. V. Balashov, A. B. Glonina, V. N. Pashkov, R. L. Smeliansky, and D. Yu. Volkanov. 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*, St. Petersburg, Russia, 2011.

[8] G. Behrmann, A. David, and K. G. Larsen. A tutorial on uppaal. In *Lecture Notes in Computer Science*, volume 3185, pages 200–236. 2004.

[9] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal - a tool suite for automatic verification of real-time systems. In *Lecture Notes in Computer Science*, volume 1066, pages 232–243. 1996.

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

[11] A. David and M. O. Moller. *From HUPPAAL to UPPAAL: a Translation from Hierarchical Timed Automata to Flat Timed Automata*. Research Series RS-01-11. 2001.

[12] A. David, M. O. Moller, and W. Yi. Verification of uml statechart with real-time extensions. Technical report, Department of Information Technology, Uppsala University, Uppsala, 2003.

[13] Andreas E.Dalsgaard, Mads Chr. Olesen, Martin Toft, Rene Rydhof Hansen, and Kim Guldstrand Larsen. Metamoc: Modular execution time analysis using model checking. In *WCET'10*, pages 113–123, 2010.

[14] Osipov G.S. *Artificial intelligence methods*. Fizmatlit, Moscow, Russia, 1st edition, 2011.

[15] I. V. Konnov, V. V. Podymov, Volkanov D. Yu., V. A. Zakharov, and Zorin D. A. On the designing of model checkers for real-time distributed systems. In *Program Semantics, Specification and*

Verification: Theory and Applications. The conference materials., pages 72–81, Nizhny Novgorod: Publishing house of the Nizhny Novgorod State University, 2012.

[16] V. A. Kostenko and D. A. Zorin. 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)*, pages 392–396, Brno, Czech Republic, 2012.

[17] E. Noulard, J.Y. Rousselot, and P. Siron. Certi, an open source rti, why and how. In *Spring Simulation Interoperability Workshop*, San Diego, USA, 2009.

[18] Ruslan L. Smeliansky, Anatoly G. Bakhmurov, Dmitry Yu. Volkanov, and Eugene V. Chemeritsky. An integrated environment for distributed embedded real-time system design and analysis. In *Programming*. 2013. To be published.

[19] Naruemon Wattanapongsakorn and David W. Coit. Fault-tolerant embedded system design and optimization considering reliability estimation uncertainty. *Reliability Engineering And System Safety*, pages 395–407, 2007.