A Toolsuite for the Verification of Real-Time Systems in Eclipse[*]

Lucía Cavatorta, Guido de Caso, Andrés Ferrari, Víctor Braberman, Diego Garbervetsky, Nicolás Kicillof, Fernando Schapachnik

Departamento de Computación, FCEyN, Universidad de Buenos Aires, Buenos Aires, Argentina

{lcavat, gdecaso, aferrari, vbraber, diegog, nicok, fschapac}@dc.uba.ar

Alfredo Olivero

Centro de Estudios Avanzados, FIyCE, Universidad Argentina de la Empresa, Buenos Aires, Argentina

aolivero@uade.edu.ar

ABSTRACT

In this work we present an Eclipse plug-in for the VInTiMe (Verifier of INtegrated TImed ModEls)[1] suite of tools that combines high-level expressive power, unassisted property-preserving model reduction and distributed model checking to describe and verify complex real-time system designs and their properties.

Keywords

Eclipse plug-in, timed model checking, verification, timed automata, Lapsus, ObsSlice, VTS, Zeus

1. INTRODUCTION

Real-time systems are often complex and critical. Their formal verification has gained attention in the last few decades, specially due to several cases in which the usage of automated tools unveiled design errors not found by humans [12, 13, 16]. However, adoption of these techniques has faced some reluctance from practitioners, mainly because they require significant mathematical skills and verification consumes expensive computational resources.

The response from academia has been to develop easier to use formalisms and new tools to handle more complex systems and to verify their properties in less time. VInTiMe is a step forward in this direction, consisting of an integrated approach to solve the problems of usability and scalability.

In order to verify real world applications, usually the following steps should be followed:

- The system is precisely described.
- System requirements are specified.
- A (time consuming) verification is executed.
- If requirements do not hold, appropriate counterexamples are obtained in order to fix the system.

The VInTiMe toolset covers the full specification and verification cycle. This is actually the result of a decoupled approach in which autonomous tools with clear interfaces were developed independently. They were unified under a single framework once they became mature enough, using timed automata as a *lingua franca* between them.

Timed automata [3] (TA) are finite automata extended with positive *real* valued clocks that record the dense time elapsed between events. Clocks can be used to guard transitions, and to force control locations to be abandoned in case an invariant does not hold.

Automated Software Engineering tools must have a solid mathematical background and yet provide a simple and intuitive interface to users. In the next section we will show how the VInTiMe tools hide complexity from the user by providing a smooth interaction with our kernel formalism.

2. INTEGRATED TIMED MODELS VERIFICATION

VInTiMe comprises the integration of the tools Lapsus, VTS, ObsSlice and Zeus as described in [2].

Lapsus [6, 5] translates real-time system designs based on fixed-priority scheduling into timed automata. Succinctly, it uses Worst-Case Completion Times (WCCT) and Best-Case Completion times (BCCT) –which can be calculated using analytical techniques provided by these theories– to build an abstract and analyzable model of the system as a collection of timed automata. We use those timed automata to analyze complex properties involving coordination of a set of tasks.

Once Lapsus' job is done, VTS [1] comes into play. Using simple graphical patterns, the designer can define interesting properties over the system under analysis (SUA). More precisely, VTS is a notation for expressing real-time requirements in a visual and friendly –yet powerful– language, by means of negative scenarios. A VTS scenario is basically an annotated partial order of relevant events, denoting a (possibly infinite) set of matching time-stamped executions.

[*]Supported by IBM Eclipse Innovation Grants 2005.

[1]The tool homepage is http://dependex.dc.uba.ar/vintime

VTS is meant to existentially predicate on system executions. That is, it is used to state a simple though relevant family of questions of the form "Is there a potential run that matches this generic scenario?". When interpreted as negative scenarios, these questions can express infringements of safety or progress requirements, which turn out to be decidable. The tool translates a scenario into a timed automaton (observer) that recognizes matching runs. This automaton is composed with the SUA in order to check whether a violating execution is reachable in that behavioral model.

It should be noted that the tools presented so far tackle the usability issue by simplifying the construction of formal models of both timed systems and properties over them. There is still a scalability problem to deal with, and that is the goal of the two remaining components of the suite: they deal with the state explosion problem in orthogonal ways.

ObsSlice [7] is an optimization tool suited for the verification of networks of timed automata using virtual observers, the same kind of observers that VTS produces. It automatically discovers the set of modeling elements that can be safely ignored at each location of the observer by synthesizing behavioral dependence information among components. ObsSlice is fed with a network of timed automata and generates a transformed network which is equivalent to the one provided, as far as the properties are concerned.

ObsSlice automatically derives a subset of components which is enough to perform the verification process. That is, one of the most appealing aspect of this approach is that generally only a small subset of a system's components is really needed to perform model checking, dramatically reducing the cost of this step. Hence, the complexity of our approach mainly depends on the number of components involved in the requirement instead of the size of the complete model, which is the case in previous works on automatic verification of real time applications.

Contrary to other approaches that use schedulers, Lapsus modeling of real-time tasks helps ObsSlice reduce the amount of components that are relevant to verify a given property. If schedulers were used, the resulting system would be highly coupled, making optimization harder [5].

Finally, the model checker Zeus [9, 10, 11] tackles the state explosion by performing a distributed computation. Even though timed model checking is a problem ill-suited for distribution, Zeus makes good use of multiple processors using interesting rebalancing techniques.

3. PLUGGING VINTIME INTO ECLIPSE

The development of real-time systems requires specification and implementation frameworks, usually realized as separate software tools, which forces designers to constantly switch between them. The integration of VInTiMe as an Eclipse plug-in greatly simplifies this process. Designers can have their systems both specified and implemented entirely within the Eclipse platform.

Eclipse also provides VInTiMe users with additional features such as the ability to have their verification projects saved in CVS repositories or exported to ZIP files.

3.1 Features

In order to support the integration of the tools described in section 2, the VInTiMe Eclipse plug-in offers a set of editors for each of the formalisms.

The verification process begins with the creation of a new VInTiMe project to contain the different elements necessary to model the SUA and to guide its verification.

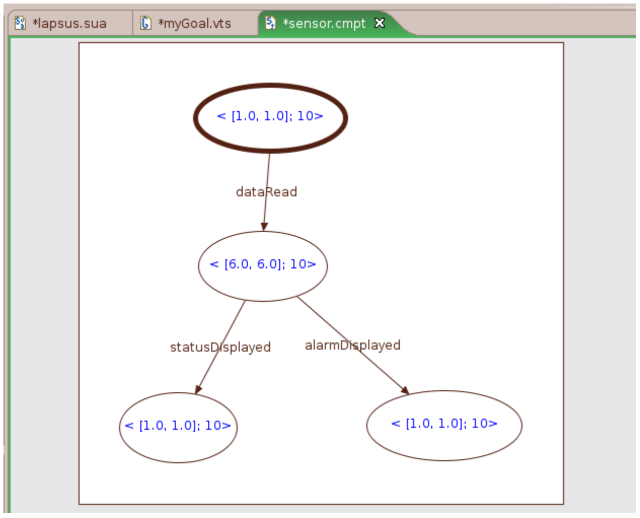


Figure 1: A Lapsus component

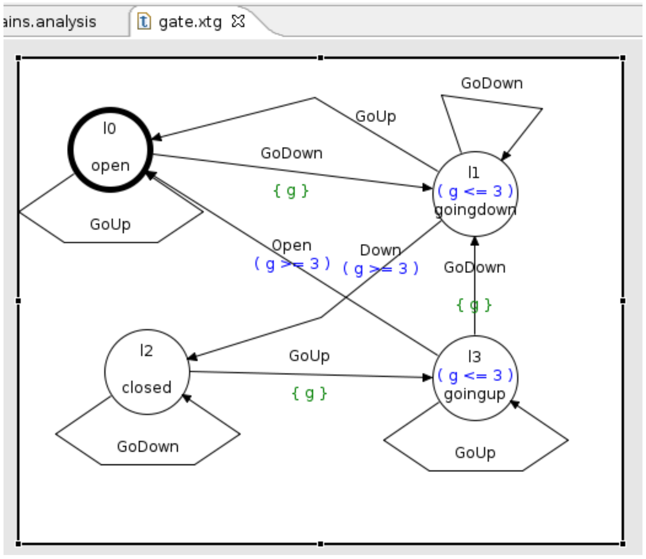


Figure 2: A TA component

Since real-time systems are usually made up of several interacting components, the designer starts by constructing each of them independently. Main system components are modeled using Lapsus tasks (Figure 1), while environment and user defined connectors can be directly modeled with TA (Figure 2).

Once the components are defined, the user can create a SUA by dragging together instances of them. Making component construction and composition separate steps of the process maximizes reusability of system parts.

After the SUA has been defined, users can express requirements over it in the form of VTS patterns, as shown in Figure 3. Since these scenarios can contain variables, an instantiation phase is required: users drag goals and SUAs together while a wizard helps them bind variables.

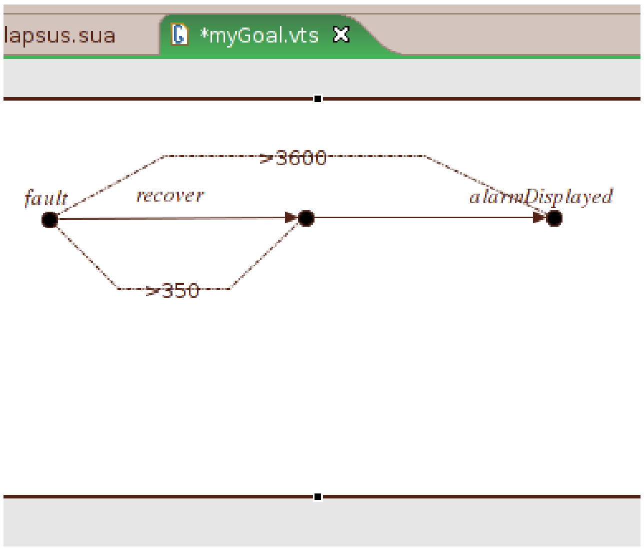
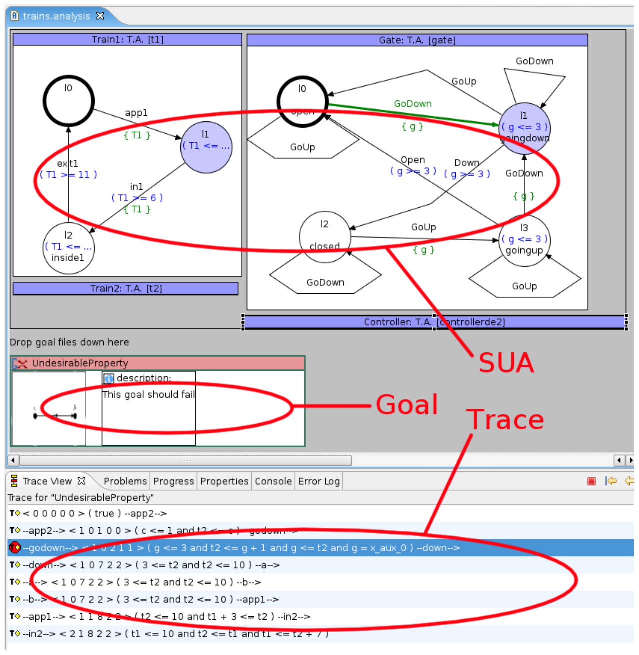


Figure 3: A VTS scenario

This binding of goals to SUAs is called an *Analysis*. Figure 4 shows the Analysis Editor.



description:
This goal should fail

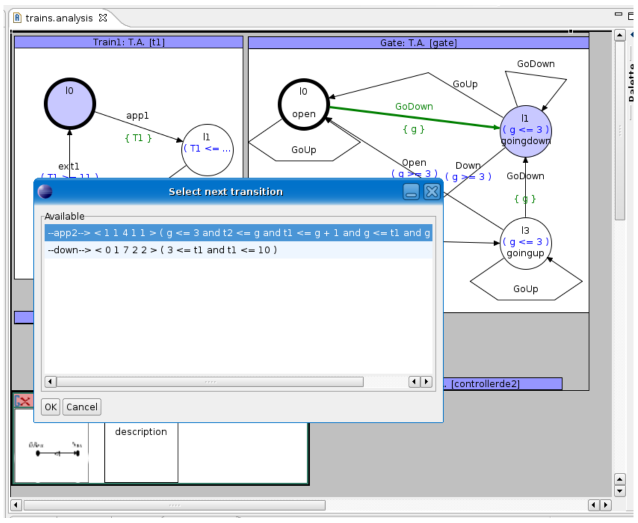
```
Trace for "UndesirableProperty"
< 0 0 0 0 0 > ( true ) --app2-->
--app2--> < 1 0 1 0 0 > ( c <= 1 and t2 <= c ) --godown-->
--godown--> < 1 0 2 1 1 > ( g <= 3 and t2 <= g + 1 and g <= t2 and g = x_aux_0 ) --down-->
--down--> < 1 0 7 2 2 > ( 3 <= t2 and t2 <= 10 ) --a-->
--a--> < 1 0 7 2 2 > ( 3 <= t2 and t2 <= 10 ) --b-->
--b--> < 1 0 7 2 2 > ( 3 <= t2 and t2 <= 10 ) --appl-->
--appl--> < 1 1 8 2 2 > ( t2 <= 10 and t1 + 3 <= t2 ) --in2-->
--in2--> < 2 1 8 2 2 > ( t1 <= 10 and t2 <= t1 and t1 <= t2 + 7 )
```

Figure 4: Analysis Editor and Trace View

This is the final step in the design stage, now the verification phase can begin. From the Analysis Editor users can call Zeus to perform a distributed model check of a goal on a cluster[2].

If the SUA can produce a behavior expressed by the (negative) goal, a trace showing how it can be reproduced is presented in a Trace View, as shown in Figure 4. Otherwise, the goal is valid and no trace is found. As a visual aid, a color cue is used to mark goal states.

[2]A cluster is a network of workstations that can be graphically defined in the bundled Cluster Editor.



```
Available
--app2--> < 1 1 4 1 1 > ( g <= 3 and t2 <= g and t1 <= g + 1 and g <= t1 and g
--down--> < 0 1 7 2 2 > ( 3 <= t1 and t1 <= 10 )
```

Figure 5: Users select how they want the trace to continue

The SUA can be traversed by following a trace. While doing so, current states and transitions are highlighted appropriately. As a result of applying ObsSlice during verification, SUA components are dynamically enabled or disabled in a trace. Our tool automatically minimizes disabled SUA components during trace traversal and restored when they are enabled again. This greatly simplifies the task of understanding error traces, as only relevant components need to be analyzed. Traces can also be manually modified, or generated from scratch in order to allow the user to simulate additional interesting behaviors, as shown in Figure 5.

3.2 Architecture

The software architecture of the VInTiMe plug-in, shown in Figure 6, can be divided into three different parts: Basic Editors, Composition Editors and External Tools.

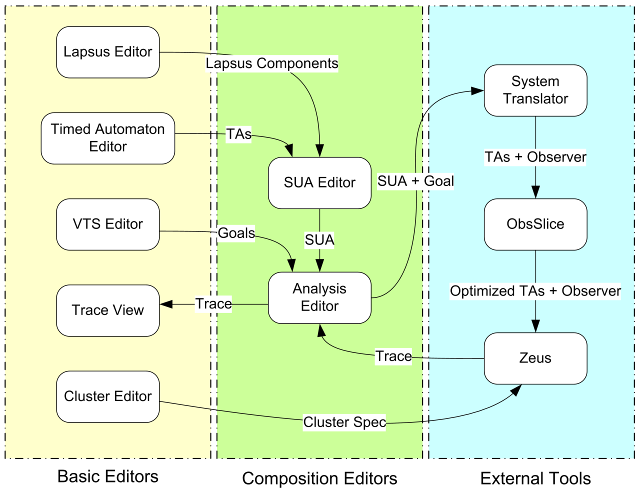


Figure 6: Plug-in architecture

The Basic Editors are used in the first steps of the verification process. Designers can use them to specify the different components that make up a system. The Trace View is a special case because it extends `ViewPart`, but it is similar to an editor, as it can import, modify and export traces.

Components specified with the Basic Editors are then dragged into the Composition Editors to create larger and more complex systems. The Analysis Editor is an essential piece of the plug-in. It allows users to launch ZEUS, open the generated traces in the Trace View, traverse them, watch the system evolve, etc.

The backend components that are used to translate the system, optimize it and verify it are treated as external tools. The system translator is written in Java and consists of three parts: one handles the translation of the graphical TA models to a file format understandable by ZEUS, the other two are in charge of translating LAPSUS components and VTS scenarios to timed automata. OBSSLICE is also a Java application, only minor adjustments to its original code were necessary. Finally ZEUS is written in pure C; interfacing with this tool is described in the next section.

3.3 Design and Implementation

Several challenges had to be faced during the implementation of the VINTIME Eclipse plug-in.

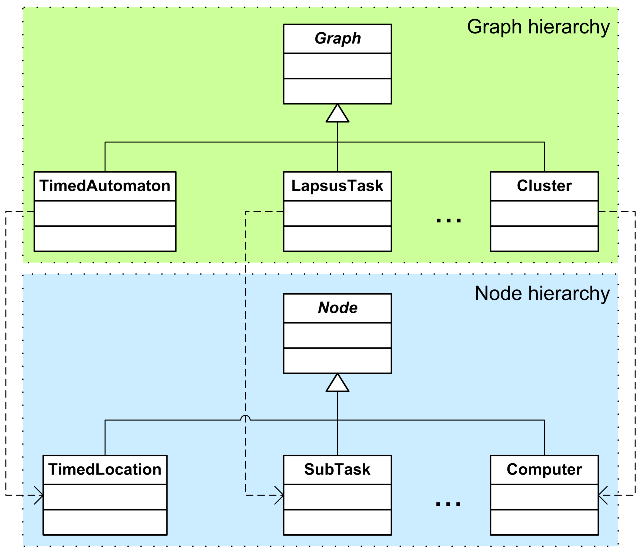


Figure 7: Graph and node hierarchies

3.3.1 Graph Editors

First we needed to create several graphical editors: one for each notation and one to edit workstation clusters. They have different features, but are based on common grounds: they are all graph editors. We have created an abstract Basic Graph Editor managing nodes and edges, to contain the shared functionality. The editor for each notation extends this Basic Graph Editor and adds the needed capabilities.

Following the same spirit, as shown in Figure 7, each kind of node for a specific notation overrides a basic node class and each kind of transition overrides a basic edge class. Each editor allows only certain kinds of nodes and edges into its graph. In the figure this is shown with dotted lines between the nodes and the graphs that accept them.

3.3.2 Integration with ZEUS

Another issue to solve was ZEUS integration. Being a distributed tool written in C, communication was achieved following the scheme in Figure 8.

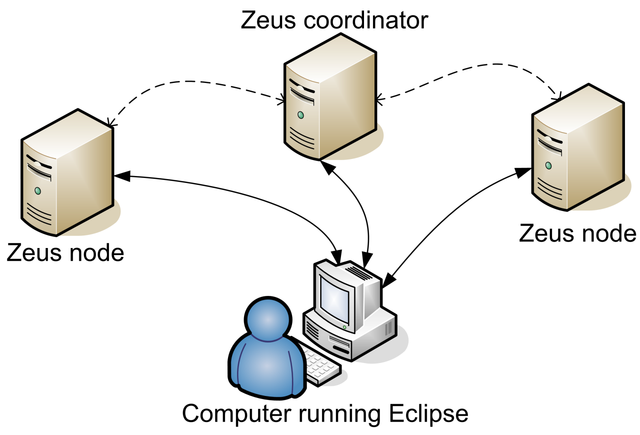


Figure 8: Cluster integration

Solid lines represent SSH connections that link the computer running the VINTIME plug-in with every computer in the cluster. Using this protocol, the plug-in transmits the (platform dependent) ZEUS executable, the configuration and TA files, launches the process itself and finally fetches the generated results. Prior to the introduction of our tool, all these tasks required manual intervention.

Dotted lines denote socket connections between the ZEUS coordinator and the rest of the computers in the cluster.

3.3.3 Synchronization issues

We have mentioned that complex models can be created by gluing together several components. In the implementation, we had to decide whether to use references or copies for shared components. References have the advantage to easily reflect component changes in every location where they appear. On the other hand, copies avoid synchronization conflicts that could arise if one component is modified while a SUA using it is being edited in another window.

Since the VINTIME plug-in uses references, a mechanism to deal with synchronization was required. Using file timestamps to establish if a component has changed presents a problem: graphical changes (e.g., node positions) as well as some operating system activities would force unnecessary dependent-file reloads. To avoid this, we implemented a hash function that depends only on the semantic properties of the models. Each time a SUA editor gets focus we compare each component hash with the one stored in its corresponding file header. If they are the same, then the reloading process is not necessary.

4. CONCLUSIONS AND FUTURE WORK

We managed to integrate into Eclipse a verification toolset originally made up of several different –although compatible– command line tools. The end user is now provided with a simple, yet powerful, interface to solid Software Engineering tools. The project's main goal has been achieved: via Eclipse, we offer the real-time system designer an integrated specification and implementation platform.

As future work, we wish to continue developing the different individual applications such as ZEUS and OBSSLICE. As long as they keep using the same interface the VINTIME plug-in will not need to be modified. We would also like to improve the user experience by adding more wizards and keeping up to date with Eclipse guidelines and improvements, as well as being able to interchange data with other tools like Uppaal [4], OpenKronos [15], etc.

A current line of research is to generate code from models. We would like to incorporate such a feature as it will fit naturally into the Eclipse platform. Adding UML-RT [14] formalisms to our tool suite is also in our research agenda.

Finally, we are also evaluating support for conditional scenarios in the VTS language [8]. Essentially, conditional scenarios state that, whenever a matching for a sub-scenario (the antecedent) is encountered in a given trace, the same matching must be extensible to cover one in a set of consequent scenarios. Consequents are not necessarily subsequent (future) events as in other approaches: they can predicate about previous occurrences of events (e.g., "if *ack* is found, then *request* must have been issued at least 10 t.u. before"), and they can even add constraints to the antecedent event-pattern (e.g., "if two events *a* and *b* are encountered in a trace, then either no *c* event can be found in between, or the distance is greater than 5 t.u."). Support for this kind of scenarios requires few modifications of the VTS syntax as well as integrating an existing tool that translates them into a set of antiscenarios (the ones already supported by the toolsuite).

5. ABOUT THE AUTHORS

Lucía Cavatorta, Guido de Caso and Andrés Ferrari: CS students and main developers of the VINTIME plug-in. Víctor Braberman: PhD in CS, Assistant Professor working on modeling and verification of real-time and distributed systems. Diego Garbervetsky: PhD candidate in CS and Assistant Professor working on program analysis and verification. Nicolás Kicillof: PhD candidate in CS and Assistant Professor working on modeling and verification of software systems. Alfredo Olivero: PhD in CS and Professor working on modeling and verification of real-time and distributed systems. Fernando Schapachnik: PhD candidate in CS and Lecturer working on modeling and verification of real-time and distributed systems.

6. REFERENCES

[1] A. Alfonso, V. Braberman, N. Kicillof, and A. Olivero. Visual timed event scenarios. In *Proc. of the 26th ACM/IEEE International Conference on Software Engineering.* ACM Press, 2004.

[2] A. Alfonso, D. Garbervetsky, V. Braberman, A. Olivero, N. Kicillof, and F. Schapachnik. Vintime: Combining high-level finesse with low-level muscle to verify real-time systems. In *First International Conference on Principles of Software Engineering, PRISE 2004*, Buenos Aires, Argentina, November 2004.

[3] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[4] J. Bengtsson, K. Guldstrand Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In *Hybrid Systems*, pages 232–243. Springer-Verlag, 1995.

[5] V. Braberman. *Modeling and Checking Real-Time Systems Designs.* Ph d. thesis, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, 2000.

[6] V. Braberman and M. Felder. Verification of real-time designs: Combining scheduling theory with automatic formal verification. In *Software Engineering - ESEC/FSE'99: 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1687 of *LNCS*, pages 521–525, Touluse, France, September 1999. Springer-Verlag.

[7] V. Braberman, D. Garbervetsky, and A. Olivero. ObsSlice: A timed automata slicer based on observers. In *Proc. of the 16th Intl. Conf. CAV '04*, LNCS. Springer Verlag, 2004.

[8] V. Braberman, N. Kicillof, and A. Olivero. A scenario-matching approach to the description and model checking of real-time properties. *IEEE Transactions on Software Engineering*, 31(12):1028–1041, 2005.

[9] V. Braberman, A. Olivero, and F. Schapachnik. Zeus: A distributed timed model checker based on Kronos. In 1st *Workshop on Parallel and Distributed Model Checking, affiliated to CONCUR 2002 (*13th *International Conference on Concurrency Theory)*, volume 68 of *ENTCS*, Brno, Czech Republic, August 2002. Elsevier.

[10] V. Braberman, A. Olivero, and F. Schapachnik. Issues in Distributed Model-Checking of Timed Automata: building ZEUS. *International Journal of Software Tools for Technology Transfer*, 7:4–18, feb 2005.

[11] V. Braberman, A. Olivero, and F. Schapachnik. Dealing with practical limitations of distributed timed model checking. *Formal Methods in System Design*, 2006.

[12] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. In *Proceedings of the* 16th *IEEE Real-Time Systems Symposium (RTSS'95)*, pages 66–75, Pisa, Italy, December 1995. IEEE Computer Society Press.

[13] M. Bozga, O. Maler, A. Pnueli, and S. Yovine. Some progress in the symbolic verification of timed automata. In O. Grumberg, editor, *Proceedings of the* 9th *International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 179–190, Israel, June 1997. Springer-Verlag.

[14] B. Selic, A. Moore, M. Woodside, B. Watson, M. Bjorkander, M. Gerhardt, and D. Petriu. *UML Profile for Schedulability, Performance and Time Specification.* Object Management Group, 2005.

[15] S. Tripakis. *L'Analyse Formelle des Systemès Temporisés en Practique.* PhD thesis, Univesité Joseph Fourier, 1998.

[16] S. Tripakis and S. Yovine. Verification of the fast reservation protocol with delayed transmission using the tool KRONOS. In *Proceedings of the* 4th *IEEE Real-Time Technology and Applications Symposium (RTAS'98)*, pages 165–170, Denver, Colorado, USA, June 1998. IEEE Computer Society Press.