Safety Critical Systems Based on Formal Models

Lars Asplund
Uppsala University
Dept. of Information Technology
P.O. Box 325
SE-751 05 Uppsala, Sweden
lars.asplund@it.uu.se

Kristina Lundqvist
Massachusetts Institute of Technology
Dept. of Aeronautics and Astronautics
77 Massachusetts Avenue
Cambridge, MA 02129, USA
kristina@mit.edu

Abstract

The Ravenscar profile for high integrity systems using Ada 95 is well defined in all real-time aspects. The complexity of the run-time system has been reduced to allow full utilization of formal methods for applications using the Ravenscar profile. In the Mana project a tool set is being developed including a formal model of a Ravenscar compliant run-time system, a gnat compatible run-time system, and an ASIS based tool to allow for the verification of a system including both COTS and code that is reused.

1. Introduction

The number of processors is steadily increasing, and although most of the common interest is focused on personal computers and Internet technology, more than 98% of all processors are embedded computers [EGH00]. This high percentage will most probably not decrease in the future, on the contrary the use of electronics in all kinds of products will at least keep this high number.

The range of applications for embedded computers vary in many different aspects, and the size of the used processors vary in size from 4 or 8 bit micro-controllers to networks of powerful 64 bit processors. An issue that is very important for application designers is the aspect of the application's safety level. For applications, where safety is obvious like in avionics or nuclear industry, there are different authorities, like TÜV and FAA, that maintain certification standards. With an increasing awareness of the criticality of embedded software, we can foresee an increased use of modern methods such as verification tools for all kind of safety critical systems.

A characteristic feature of embedded systems is that they are real-time systems and the major solution used is based on cyclic executives. With the increased complexity of applications it is becoming more difficult to design and maintain embedded systems. Modern process scheduling theory can rigorously evaluate the performance of process based real-time systems. One of the critical parameters for real-time tasks, is the response time, which heavily depends on execution time analysis. A tight estimate of the *worst case execution time* is required to meet the demands of high utilisation of the cpu and high rates of task invocations. These execution times are a necessary prerequisite in a system based on cyclic executives, and hence there is a possibility to fully utilize a schedulability analysis.

Safety critical systems are today often implemented using the Ada programming language [Cha98], and systems at the highest level of criticality have so far been implemented using different Ada subsets, e.g., the SPARK Ada subset, that do not include any tasking. The definition of the Ravenscar Ada subset [DB98] opened the possibility for using Ada including tasking in applications at the highest level of criticality.

A research area that is tightly connected to safety critical systems, is *formal methods*. The most prominent use of formal methods is in the railway signalling industry. The reason, is that these systems contain complex logic, and are therefore good candidates for automatic verification by the use of a commercial tool [Bar97]. Formal methods have had difficulty gaining acceptance in the industrial sector. Some complaints are that formal methods are considered impractical to use and need a long time to be learned. Adding to this scepticism is the fact that some formal methods not yet has proven to be able to handle systems of realistic complexity. Formal specifications and mathematical analysis theoretically present a way out of the dilemma that we can test only a very small part of the available state space. They have the potential for both increasing safety and decreasing the cost of certifying safety critical systems. While formal methods have been applied to hardware in industry, the results of formal methods research for software has only rarely reached beyond the research lab, and there has as yet been only limited success with applying formal methods to complex systems.

The use of formal methods for process based Ada systems including full tasking is not feasible due to the complex semantics of full Ada, which with full coverage will result in an unmanageable state explosion. The definition of Ravenscar however opens up the possibility of using formal methods for verification of not only the application, but also the complete run-time system. Our previous work, on modelling of the complete run-time system for Ravenscar, is a complement to SPARK-Ada, and allows for the verification of real-time aspects of applications [LA99a, LA99b, LA99c, LAM99, Lun00, LA00].

As for all appliance of theory and methods to real-world problems there is a balance between time and complexity. Fine grained and thorough analysis is not feasible. The usefulness of these methods increases both due to more powerful computers, but also in refinement of the underlying theories.

In this paper we will discuss the use of formal methods in the area of high integrity systems using Ada as the implementation language. The tool used for formal verification is the real-time model checker UPPAAL [LPY97]. UPPAAL is a tool-suite that can be used to model, validate and verify real-time systems. It uses networks of timed automata extended with data variables with finite domains, and arrays of such variables. Using timed automata it is possible not only to check the properties that correspond to the behaviour and interaction of the finite state machine, but also timing properties, such as a detailed response time analysis. The basic understanding is that all processes are described in terms of finite state machines. The states in the formal model can correspond to states or variables in the application. Transitions are either explicit, where one or several automata take part in a transition or silent where the transition has a probability to occur. The verification engine is based on the on-the-fly and symbolic constraint solving techniques presented in [YPD94]. The tool is quite user-friendly in that it has graphical interfaces and automatically produces a diagnostic trace, that explains why, or why not, a verification of a property is fulfilled. Thus, it is well suited for practical use in real-world projects. We have in previous work [LA99a, LA99b, LA99c, LAM99, Lun00] successfully used UPPAAL to verify a Ravenscar compliant run-time kernel together with applications, and the model of the run-time system (RTS) has in [LA00] been shown to fully behave according to the semantics of Ada 95.

2 Safety Critical Systems and their Implementations

Safety can be defined as freedom from accidents or losses, and a safety critical system can thus be seen as a system where a failure would result in actual system hazards [Lev95]. A safety critical system often has some requirements that are of such importance that particular high levels of assurance may be needed, and testing alone is insufficient to establish those levels. Such a system may benefit from using technologies that can be too costly to use on a system as a whole, e.g., formal verification. A safety critical system also needs an approval from a licensing bureau or by a government agency such as TÜV or FAA.

The Ada language has a number of features valuable while implementing safety critical systems, e.g., low level access to hardware. Other properties of the language such as strong typing, minimal number of implementation-dependent and unspecified behaviour, can help in the development of provable correct programs. A problem with full Ada and its run-time system (RTS) is that it is too complicated to allow for formal validation of applications. To avoid this problem, a number of subsets of Ada have been defined, e.g., the SPARK subset [Bar97], AVA (A Verifiable Ada) [Smi92] and the Penelope environment [GMP90] showed that this was the way to go for Ada 83 applications, although the proof technologies and Ada 83 itself made the provable subsets somewhat limited. Both SPARK and AVA have in newer versions adopted to the Ada 95 standard, but common for all these subsets is that they still do not contain any tasking facilities.

The goal when formulating Ravenscar was to make possible the design of safety critical systems using a safe subset of Ada 95 including tasking. One objective was to give system developers at the highest degree of safety, e.g., implementing systems aiming at level A in DO-178B, a rich enough language profile where concurrency can be expressed. It was achieved by eliminating all tasking constructs that contain implementation dependencies and nondeterministic behaviour resulting in a simplified kernel.

The computational model that the kernel supports is made up of a fixed set of concurrently executing tasks, which can communicate asynchronously by means of protected objects. Each task has to be structured as an infinite loop and is not allowed to terminate. The tasks are either time triggered periodic tasks using the *delay until* or asynchronous tasks making calls to the entry of protected objects, which are released by time triggered tasks or external interrupts.

The kernel supports five different operations. Four of the operations corresponds to the four operations an application can invoke (i.e., call to a protected entry, a protected function, or a protected procedure, and the *delay until* operation.). The fifth kernel operation is in response to external interrupts.

In order to model interrupts a model of the world is required. When the application interacts with the real world, actions from tasks make changes in the real world. These changes can directly or indirectly give rise to an interrupt. The real world can of course give rise to interrupts without any actions from the application. In any case the actual behaviour of the real world can be modelled using UPPAAL. This model can capture both the regular input and output control between the tasks in the application as well as generating interrupts.

The specification of the Ravenscar tasking model significantly advances the possibilities to use strict and mathematical based formal methods to verify systems including tasking.

Former verifications of kernels have tried to make a general verification that should fit any application using the kernel [Hut94, Tol95, FW96, FW97]. After such a verification it is possible to state that “the kernel is perfect”. Our attempt, however, aims at obtaining a tool-set that allows the verification of the complete application, which includes the full run-time kernel. One can after a verification with our tool set not claim that the kernel will work for any general application, but it is possible to claim that the complete application is correct.

2.1 The Ravenscar Profile

The Ravenscar profile lists a number of features (primarily dealing with tasks, protected types, and delay statements) in Ada 95 which are forbidden for applications that require deterministic and schedulable behaviour. The profile does however permit a number of real-time and concurrency features that still can be used with statically deterministic behaviour [AJLB99, BDR98].

Restrictions of the language are specified through the use of *pragmas*. A pragma is a directive to the compiler and the pragma `Restrictions` specifies the restrictions that a program can take upon itself. At the 9th IRTA Workshop [AJLB99] 15 new pragma identifiers were introduced and accepted for the profile. If there exists an appropriate pragma `Restrictions` for the forbidden language constructs, then they are mentioned together with respective item in the list below.

2.1.1 What is restricted?

- No tasks and protected object declarations other than at the library level, i.e., no hierarchy of tasks.
  `Restrictions (No_Task_Hierarchy)`,
  `Restrictions (No_Local_Protected_Objects)`

- No dynamic allocation of task or protected objects.
  `Restrictions (No_Protected_Type_Allocators)`
  `Restrictions (No_Task_Allocators)`

- No unchecked deallocation of protected and task objects.

- Can assume tasks do not terminate.
  `Restrictions (No_Task_Termination)`

The four items above result in a program that at start of execution consists of all tasks and POs that will exist during the program execution. This results in a task set that is suitable for schedulability analysis.

- No requeue. The `requeue` statement introduces a significant run-time overhead and static analysis of the program is made complicated.
  `Restrictions (No_Requeue)`

- No forms of select statements. The semantics of the select statement is inherently non-deterministic.
  `Restrictions (No_Select_Statements)`

- No asynchronous transfer of control (ATC).
  `Restrictions (No_Asynchronous_Control)`

- No abort.
  `Restrictions (No_Abort_Statements)`
- No dynamic priorities.
  `Restrictions (No_Dynamic_Priorities)`
- No Calendar package. The Real_Time package is mandatory.
  `Restrictions (No_Calendar)`.
- No relative delays. Absolute delay (*delay until* in Ada) is allowed, and the delta delay is not included.
  `Restrictions (No_Relative_Delay)`
- No protected object with more than one entry. Simplifies servicing of entry queues since the non-determinism associated with selecting the next task to execute the protected operation is avoided.
  `Restrictions (Max_Protected_Entries => 1)`
- No attempts to queue more than one task on a single protected entry.
  `Restrictions (Max_Entry_Queue_Depth => 1)`
- No protected entries with barriers other than a single boolean. Results in bounded execution time for entry barrier evaluation.
  `Restrictions (Simple_Barrier_Variables)`
- No user-defined task attributes. Reduces run-time complexity and overhead.
  `Restrictions (No_Task_Attributes)`
- No task entries.
  `Restrictions (Max_Task_Entries => 0)`
- No locking policies other than Ceiling locking
  `Locking_Policy (Ceiling_Locking)`
- No scheduling policies other than FIFO within priorities
- No interrupts are mapped onto task entries, since Ravenscar does not allow task entries, only protected procedures are allowed as interrupt handlers.

The Raven project [DR99] experience led to several interesting observations. A number of additional pragma `Restrictions` identifiers were proposed (included in the list above), and also some tasking related `Restrictions` identifiers were proposed as useful in memory-size determinism and shedulability analysis.

- Static_Storage_Size

The Raven project also found it necessary to place a suitable set of restrictions in some of the sequential (non-tasking) features of Ada.

- No_Exception_Handlers
- No_Standard_Storage_Pools
- No_IO
- No_Nested_Finalization

These 5 last `Restrictions` were all accepted at the 9^{th} IRTA Workshop [AJLB99].

3 Modelling a Real-Time System with Timed Automata

There exists many formalism description techniques for describing time-constrained systems, e.g., presented in [HNSY92, Yi91, Han91]. One of the most successful formalisms is the model of timed automata introduced in 1990 by Alur and Dill [AD90]. Properties of timed automata can be verified using reachability analysis and model checking. Model checking requires tool support and there exists a number of tools all with a little different semantics of timed automata, e.g., CMC [LL98], HyTech [HHWT97], KRONOS [Yov97], SGM [HW98], and UPPAAL [LPY97]. The UPPAAL tool has been used in the work presented here.

3.1 Tool for Modelling and Verification

There exists several classes of techniques to perform formal verification of real-time systems. Reachability analysis determines the set of states reachable from a given set of initial states. The analysis allows proving that, e.g., an erroneous state never is reached, or that certain timing constraints are always satisfied. Formal verification also consists of real-time symbolic model checking, i.e., checking the correctness of a formal model against requirements expressed in a real-time temporal logic, e.g., timed Computation Tree Logic (CTL) is used in UPPAAL to specify properties to verify.

UPPAAL is a tool box for modelling, simulating and verifying real-time systems, developed jointly by Uppsala University and Aalborg University [LPY97]. It is useful for verification of systems that can be modelled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels and (or) shared variables.

UPPAAL consists of three main parts: a description language, a simulator, and a model-checker. The description language is a non-deterministic guarded command language with data types. It serves as a modelling or design language to describe system behaviour as a network of timed automata extended with data variables. It is also possible to create and modify models using a graphical interface to the tool. The simulator and the model-checker are designed for interactive and automated analysis of system behaviour. The simulator enables examination of possible dynamic executions of a system during early modelling or design stages and thus provides an inexpensive means of fault detection prior to verification by the model-checker which covers the exhaustive dynamic behaviour of the system. It is possible to generate a counter example trace if a required property turns out to be false. This trace can be input to the simulator to help debug the system.

4 Model of a RTK and Application

As described in a previous section, the Ravenscar compliant kernel is a small kernel, which only handles the protected operations, *delay until* instructions and external interrupts. To model this, automata for the protected object (handling protected procedure, entry and function calls), the delay queue, ready queue, system clock and the idle task is needed.

The functionality of the protected object (PO) is modelled by two automata, one handling the protected entry and one for handling both the protected procedure and the protected function. They can be modelled using the same automaton, since the protected function can be modelled as a protected procedure with a restricted behaviour, i.e., the protected function cannot change the value of the barrier whereas the protected procedure can. A PO usually has a lock associated to it, and at the start of a protected procedure or entry call, the calling task has to seize the lock before evaluating barriers, executing protected operation bodies, or manipulating entry queues. Instead of modelling this lock with a separate automaton and variables, we have chosen to implement the locking mechanism exclusively by priority boosting (imposed by the Ravenscar profile to follow the immediate inheritance protocol). When a task calls the PO, the task will immediately inherit the ceiling priority of the PO. The ceiling priority will be higher than the highest priority of any task calling the PO (also known as the *eggshell* model), this way it is assured that all pending requests for a PO entry is finalised after release by PO-procedures. The delay queue and system clock is modelled by one automaton each.

The scheduling policy used in this work is preemptive scheduling, which allows for the priority ceiling and inheritance protocol [BW90]. All scheduling is handled within a single ready queue automaton, and task dispatching is specified in terms of the ready queue, task states, and task preemption. Since the modelled system is a single processor system, only one ready queue is needed. The ready queue is modelled as an array with one field for every task id. If there would be no task ready for execution, then task T0 (the *null process*), with priority zero, would be scheduled to execute. At startup of the system, task T0 will execute, as soon as another task is runnable task T0 will be preempted.

4.1 Verification of an Application

Incompleteness or other flaws in software requirements, rather than coding errors, are usually the major factor behind accidents or major losses where computers are involved [Lev95]. This fact makes it natural to try and use formal methods to prove that a design is free from errors. Although this may be the obvious way to design a system according to the academia, it may not always be the case within commercial software development. Usually it is not all of the code in a project that is developed directly from the design. There may already be a lot of code written, so reuse in general and COTS in particular, is often used for large portions of commercial systems software. In the future, especially within safety critical applications, all of the system software should be developed from a formal design, and therefore make it possible to fully use and verify the requirements-design-implementation path.

In the following we will discuss how to use our model of the run-time for Ravenscar in industrial applications, where there exists no formal design, but mere a working system.

To verify an application one can analyze the sequential part of the program, and such a verification captures the complete trace through the program. The analysis in such an approach will result in a graph with all paths captured. The graph will not only contain the paths but also the values of all data objects have to be taken into account. One can easily see that a program does not have to be large or contain a lot of data objects before the state explosion will make it impossible to do a complete coverage of the program. Even if it is proven that it is impossible to do such an analysis for any program the approach is still very valuable. With the SPARK-tool [Bar97] several large scale applications have successfully been analyzed.

Another approach for a formal verification concentrates on the real-time behaviour such as periodic scheduling of tasks and the interaction between tasks (in Ravenscar the use of Protected Objects).

It is of particular interest for applications where cyclic executives are being used today to use formal verifications to prove the predictability of the program.

In this kind of verification we first reduces the complexity by making various paths in a task that do not contain any calls to the underlying kernel (i.e. no calls to a protected object or to a delay until-statement) be denoted in a graph for the task as a single execution node. After this reduction the graph for each task captures the interaction between the tasks, the protected objects and the underlying run-time system. Since the following analysis will prove the predictability it is important to capture the complete timing in the program. The execution time for a node is set to be between the lowest BCET (Best Case Execution Time) for the different paths, and the highest WCET (Worst Case Execution Time) for all paths.

In the reduction of the various paths it may result in a less tight estimation of the execution time. A tight estimation is where BCET does not differ significantly from WCET. An example of a reduction that will result in a less tight estimation is an if-else-path with execution time of (2,3), i.e., a BCET of 2 and WCET of 3 for one path and (100,105) for the other path. Both paths are quite tight but if neither of these two paths contain a call to a protected object or a *delay until*, they can be combined to a single execution node with the execution time (2,105). Such a reduction is worse than keeping the two original paths. Not only do we have a very bad estimation of the execution time, but we will also have to analyze the task for the interval 4..99, which will never occur in practise. There is no problem still of doing this ’reduction’, since if we can prove that the model of the program is predictable with this range for the execution time it also holds for the actual program.

After all that code has been reduced we are left with all calls to POs and *delay until*. For each of these calls a series of UPPAAL nodes are inserted that will interact with the Ravenscar compliant run-time system.

The two different ways to verify a program described here complement each other, and for high integrity systems we can assume that both methods will be used.

5 Conclusion

The Ravenscar profile was designed with predictability in mind, and our work on the formal definition of a Ravenscar compliant run-time kernel has shown that the definition is not larger than it is possible to capture the full semantics of the kernel using formal methods.

This paper has discussed the use of this formal model in conjunction with a formal model of the application, and especially how the application can be modelled to reduce the complexity, but still keep the formal description rich enough to capture the real-time behaviour of the program and thus make possible a verification of the predictability.

References

[AD90] R. Alur and D. Dill. Automata for Modelling Real-Time Systems. In *Int. Colloquium on Algorithms, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer-Verlag, 1990.

[AJLB99] L. Asplund, B. Johnson, K. Lundqvist, and A. Burns. Session Summary: The Ravenscar Profile and Implementation Issues. *The 9th International Real-Time Ada Workshop (IRTAW9), Ada Letters*, XIX(2):12–14, 1999.

[Bar97] J. Barnes. *High Integrity Ada, The SPARK Approach*. Addison-Wesley Publishing Company, Inc., 1997.

[BDR98] A. Burns, B. Dobbing, and G. Romanski. The Ravenscar Tasking Profile for High Integrity Real-Time Programs. In Lars Asplund, editor, *Reliable Software Technologies – Ada-Europe'98*, volume 1411 of *Lecture Notes in Computer Science*, pages 263–275. Springer-Verlag, 1998.

[BW90] A. Burns and A. Wellings. *Real-Time Systems and their Programming Languages*. Addison-Wesley Publishing Company, Inc., 1990. ISBN 0-201-17529-0.

[Cha98] P. Chapront. Ada+B The Formula for Safety Critical Software Development. In Lars Asplund, editor, *Reliable Software Technologies – Ada-Europe'98*, volume 1411 of *Lecture Notes in Computer Science*, pages 14–18, 1998.

[DB98] B. Dobbing and A. Burns. The Ravenscar Tasking Profile for High Integrity Real-Time Programs. In *SIGAda'98*, 1998.

[DR99] B. Dobbing and G. Romanski. The Ravenscar Tasking Profile – Experience Report. *Ada Letters*, XIX(2):28–32, June 1999.

[EGH00] D. Estrin, R. Govindan, and J. Heidemann. Embedding the internet. *Communications of the ACM*, 43(5):39–41, May 2000.

[FW96] S. Fowler and A. J. Wellings. Formal analysis of a real-time kernel specification. In B. Jonsson and J. Parrow, editors, *4th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, 1996.

[FW97] S. Fowler and A. J. Wellings. Formal development of a real-time kernel. In *18th Real-Time Systems Symposium*, Dec. 1997.

[GMP90] D. Guaspari, C. Marceau, and W. Polak. Formal Verification of Ada Programs. *IEEE Transactions on Software Engineering*, 16(9):1058–1075, Sept. 1990.

[Han91] H.A. Hansson. *Time and Probability in Formal Design of Distributed Systems.* PhD thesis, Department of Computer Systems, Uppsala University, 1991.

[HHWT97] T. A. Henzinger, P-H Ho, and H. Wong-Toi. HyTech: A Model Checker for Hybrid Systems. *Software Tools for Technology Transfer*, (1):110–122, 1997.

[HNSY92] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. In *IEEE Symp. on Logic in Computer Science*, 1992.

[Hut94] A. Hutcheon. Safe Nucleus Formal Specification, Aug. 1994. Project reference CI/GNSR/27: The Design and Development of Safety Kernel.

[HW98] P.-A. Hsiung and F. Wang. A state-graph manipulator tool for real-time system specification and verification. In *5th International Conference on Real-Time Computing Systems and Applications, RTCSA'98*, Oct. 1998.

[LA99a] Lundqvist and Asplund. A formal model of a ravenscar-compliant run-time kernel and application code. Technical Report 111, Department of Computer System, Uppsala University, May 1999.

[LA99b] K. Lundqvist and L. Asplund. A formal model of a run-time kernel for ravenscar. In *Sixth International Conference on Real-Time Computing Systems and Applications — RTCSA'99*, pages 504–507, 1999.

[LA99c] K. Lundqvist and L. Asplund. A formal model of the ada ravenscar tasking profile; delay until. In *ACM SIGAda Annual International Conference (SIGAda'99)*, pages 15–21, 1999.

[LA00] K. Lundqvist and L. Asplund. A ravenscar-compliant run-time kernel for safety-critical systems. *Real-Time Systems, Kluwer Academic Publishers Group*, 2000.

[LAM99] K. Lundqvist, L. Asplund, and S. Michell. A formal model of the ada ravenscar tasking profile; protected objects. In *Reliable Software Technologies – Ada-Europe'99*, number 1622 in LNCS, pages 12–25. Springer, 1999.

[Lev95] N. G. Leveson. *Safeware: system safety and computers*. Addison-Wesley Publishing Company, Inc., 1995. ISBN 0201119722.

[LL98] F. Laroussinie and K. G. Larsen. CMC: A tool for compositional model-checking of real-time systems. In *IFIP Joint Int. Conf. Formal Description Techniques & Protocol Specification, Testing, and Verification (FORTE-PSTV'98)*, pages 439–456. Kluwer Academic Publishers Group, Nov. 1998.

[LPY97] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct. 1997.

[Lun00] K. Lundqvist. *Distributed Computing and Safety Critical Systems in Ada*. PhD thesis, Uppsala University, Sweden, Department of Computer Systems, Information Technology, Uppsala University, P.O. Box 325, SE-751 05 Uppsala, Sweden, April 2000.

[Smi92] M. K. Smith. *The AVA Reference Manual: Derived from ANSI/MIL-STD-1815A-1983*. Computational Logic Inc., 1992.

[Tol95] R. M. Tol. *Formal Design of a Real-Time Operating System Kernel*. PhD thesis, Rijksuniversiteit Groningen, 1995.

[Yi91] W. Yi. *A Calculus of Real Time Systems*. PhD thesis, Department of Computer Scienece, Chalmers University of Technology, 1991.

[Yov97] S. Yovine. Kronos: A verification tool for real-time systems. *Springer International Journal of Software Tools for Technology Transfer*, 1(1/2), Oct. 1997.

[YPD94] W. Yi, P. Pettersson, and M. Daniels. Automatic Verification of Real-Time Communicating Systems by Constraint-Solving. In *7th International Conference on Formal Description Techniques*, pages 223–238, 1994.