UPPAAL - Now, Next, and Future

Tobias Amnell[1] Gerd Behrmann[2] Johan Bengtsson[1] Pedro R. D'Argenio[3]
Alexandre David[1] Ansgar Fehnker[4] Thomas Hune[5] Bertrand Jeannet[2]
Kim G. Larsen[2] M. Oliver Möller[5] Paul Pettersson[1] Carsten Weise[6]
Wang Yi[1]

[1] Department of Information Technology, Uppsala University, Sweden,
[tobiasa,johanb,adavid,paupet,yi]@docs.uu.se.
[2] Basic Research in Computer Science, Aalborg University, Denmark,
[behrmann,bjeannet,kgl]@cs.auc.dk.
[3] Faculty of Computer Science, University of Twente, The Netherlands,
dargenio@cs.utwente.nl.
[4] Computing Science Institute, University of Nijmegen, The Netherlands,
ansgar@cs.kun.nl.
[5] Basic Research in Computer Science, Aarhus University, Denmark,
[baris,omoeller]@brics.dk.
[6] Ericsson Eurolab Deutschland GmbH, Germany,
Carsten.Weise@eed.ericsson.se.

Abstract UPPAAL is a tool for modeling, simulation and verification of real-time systems, developed jointly by BRICS at Aalborg University and the Department of Computer Systems at Uppsala University. The tool is appropriate for systems that can be modeled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels or shared variables. Typical application areas include real-time controllers and communication protocols, in particular those where timing aspects are critical.

This paper reports on the currently available version and summarizes developments during the last two years. We report on new directions that extends UPPAAL with cost-optimal exploration, parametric modeling, stop-watches, probablistic modeling, hierachical modeling, executable timed automata, and a hybrid automata animator. We also report on recent work to improve the efficiency of the tool. In particular, we outline Clock Difference Diagrams (CDDs), new compact representations of states, a distributed version of the tool, and application of dynamic partitioning.

UPPAAL has been applied in a number of academic and industrial case studies. We describe a selection of the recent case studies.

1 Current Version of UPPAAL

In the following, we give a brief overview on UPPAAL's maturing over the years and explain the core functionalities of the current release version.

1.1 Background

UPPAAL [LPY97] is a tool for modeling, simulation and verification of real-time systems, developed jointly by BRICS at Aalborg University and the Department of Computer Systems at Uppsala University. The tool is appropriate for systems that can be modeled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels or shared variables. Typical application areas include real-time controllers and communication protocols.

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 real-valued clock variables and simple data types. It serves as a modeling or design language to describe system behavior as networks of automata extended with clock and data variables. The simulator is a validation tool which enables examination of possible dynamic executions of a system during early design (or modeling) stages. It provides an inexpensive mean of fault detection prior to verification by the model checker which covers the exhaustive dynamic behavior of the system. The simulator also allows visualization of error traces found as result of verification efforts. The model checker is to check invariant and bounded-liveness properties by exploring the symbolic state-space of a system, i.e., reachability analysis in terms of symbolic states represented by constraints.

Since the first release of UPPAAL in 1995, the tool has been further developed by the teams in Aalborg and Uppsala. The run-time and space improvements in the period December 1996 to September 1998 are reported in [Pet99]. Figures 1 and 2 show the variations of time and space consumption in the period from November 1998 until January 2001 in terms of four examples: Fischer's mutual exclusion protocol with seven processes [Lam87], a TDMA start-up algorithm with three nodes [LP97], a CSMA/CD protocol with eight nodes [BDM+98], and the FDDI token-passing protocol with twelve nodes [Yov97]. We notice that the time performance has improved significantly whereas the space improvement is only marginal.

In July 1999 a new version of UPPAAL, called UPPAAL2k, was released. This new version, which required almost two years of development, is designed to improve the graphical interface of the tool, to allow for easier maintenance, and to be portable to the most common operating systems while still preserving UPPAAL's ease-of-use and efficiency. To meet these requirements, it is designed as a client/server application with a verification server providing efficient C++ services to a Java client over a socket based protocol. This design also makes it possible to execute the server and the GUI on two different machines.

The new GUI, shown in Figure 3, integrates the three main tool components of UPPAAL, i.e., the system editor, the simulator, and the verifier. Several new functionalities have been implemented in the tool. For example, the new system editor has been tailored and extended for the new system description language of UPPAAL2k (see below), the simulator can be used to display error traces generated by the verifier, and the verification interface has been enriched with a

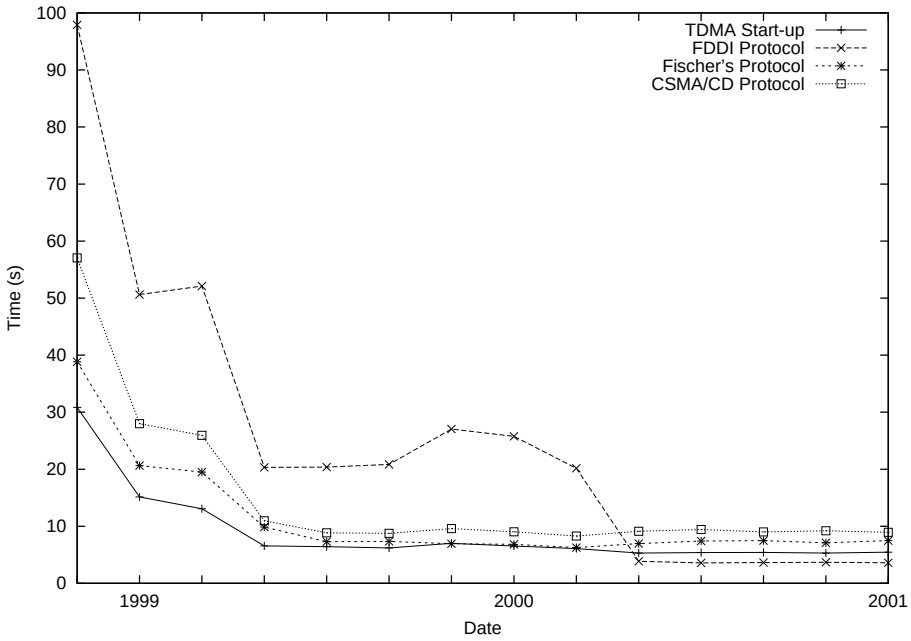

Figure1. Time (in seconds) benchmarks for the UPPAAL version 3.x, from internal version November 1998 to January 2001. Up to the second version in year 2000 the settings '-WA' (i.e. no warnings and convex-hull approximation) were used. For the later versions the settings '-WAa' (where '-a' activates the (in-)active clock reduction) were used. All tool versions were compiled with gcc 2.95.2 and executed on the same Sun UltraSPARC-II, 400 MHz machine.

requirement specification editor which stores the previous verification results of a logical property until the property or the system description is modified.

1.2 The Latest UPPAAL Release Version

The current UPPAAL version has a rich modeling language, that supports process templates and (bounded) data structures, such as data variables, constants, arrays, etc. A process template is a timed automaton extended with a list of formal parameters and a set of locally declared clocks, variables, and constants. Typically, a system description will consist of a set of instances of timed automata declared from the process templates, and of some global data, such as global clocks, variables, synchronization channels, etc. In addition, automata instances may be defined from templates re-used from existing system descriptions. Thus, the adopted notion of process templates (particularly when used in combination with the possibility to declare local process data) allows for convenient re-use of existing models.

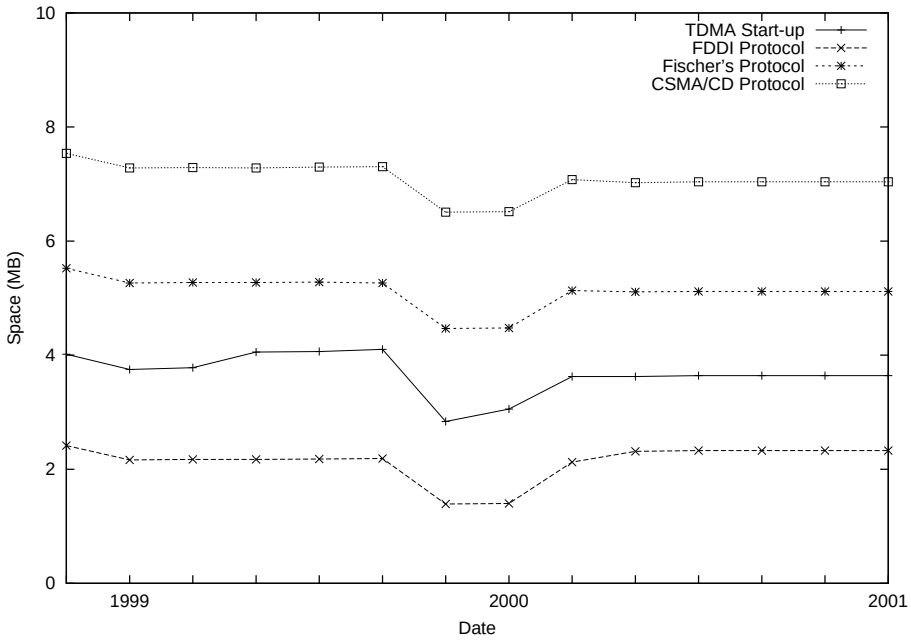

Figure2. Space (in MB) benchmarks for UPPAAL version 3.x, from internal version November 1998 to January 2001. Up to the second version in year 2000 the settings '-WAS' (where '-S' activates control-structure reduction [LLPY97]) were used. For the later versions the settings '-WAaS 2' (where '-S 2' is similar to '-S') were used.

The simulator allows both random and guided tracing through the model. One symbolic state is displayed at a time, where the control locations are visualized with red bullets in the timed automata graphs and data is shown by means of equations and clock constraints. Sub-windows can be scaled or dragged out, and the level of detail can be adjusted for user convenience. In the simulator, the user can steer to any point of an elapsed trace and save/load traces of the model. If the model checking engine detects an error trace, it can be handed over to the simulator for inspection.

The UPPAAL model-checking engine is the working horse of the tool. Therefore it is implemented in C++, whereas the GUI of the tool is implemented in Java. To interface the model-checking server, the GUI uses a socket-based protocol. This means that the GUI and verification server can be executed on two different machines. The verification server can also handle several simultaneous connections to serve several GUI clients running on different machines. By default the GUI automatically spawns a verification server process on the local machine [1].

[1] The command line options `-serverHost` *host* `-serverPort` *port* can be used to instruct the GUI to connect to a server at machine *host* on port *port*.

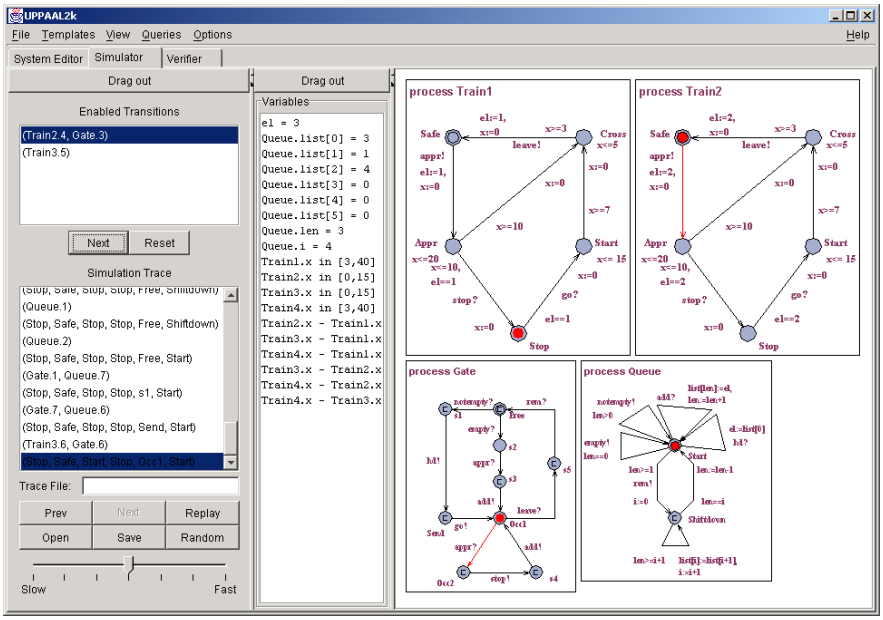

Figure3. UPPAAL2k's simulation tool on screen.

At the core of UPPAAL verification engine we find a forward-style state-space exploration algorithm. In principal, we might think of this as a variation of searching the states (nodes) of a directed graph. For this, two data structures are responsible for the potentially huge memory consumption. The first – the WAITING list – contains the states that have been encountered by the algorithm, but have not yet been explored, i.e., the successors have not been determined. The second – the PASSED list – contains all states that have been explored. The algorithm takes a state from the WAITING list, compares it with the PASSED list, and in case it has not been explored, the state itself is added to the PASSED list while the successors are added to the WAITING list.

The properties, that the model checking engine can check, describe a subset of timed computation tree logic (TCTL). In short, the four (un-nested) temporal quantifiers `E<>`, `A[]`, `E[]`, and `A<>` are supported, which stand for *possibly*, *always*, *inevitably*, and *potentially always*. In addition the operator ϕ `-->` φ is supported, which stands for the leadsto property `A[]` $(\phi \rightarrow$ `A<>` $\varphi)$. An option for deadlock checking is also implemented but it is currently only available in the stand-alone verifier `verifyta`.

This UPPAAL2k verification server has been extended with various optimization options, described in our publications and elsewhere in the literature. The current version supports the *bit-state hashing* under-approximation technique which has been successfully used in the model-checking tool SPIN for several years. A technique for generating an over-approximation of a system's reachable

state-space based on a *convex-hull* representations of constraints is also supported. Finally, an abstraction technique based on *(in-)active clock reductions* is available.

2 New Directions of UPPAAL

Several research activities are conducted within the context of UPPAAL. In this section we report on developments that extend the core functionalities of the tool.

2.1 COUPPAAL: Cost-Optimal Search

UPPAAL was initially intended to prove the correctness of real time systems with respect to their specification. If a system does not meet the specification UPPAAL finds an error state and can produce diagnostic information on how to reach this error state. However, we often prefer to think of these states as desired goal states and not as error states. To give an example. Consider four persons, who have to cross a bridge that can only carry two persons at a time. Then, one would like to know whether they can reach the safe side, given additional constraints and deadlines. This can be expressed with a timed reachability question, and if the goal state is reachable, the trace gives also a feasible schedule. We can use this approach to generally solve timed scheduling problems. In process industry for example, it is often valuable to know whether it is possible to schedule the production steps such that all constraints are met. In [Feh99,HLP00], we derive feasible schedules for a part of a steel plant in Ghent, Belgium, and a LEGO model of this plant.

Even though it is often hard to find a solution, as soon as a feasible solution is found, the question arises, whether this solution is optimal with respect to time or the number of actions. To address this, we included concepts that are well known from branch and bound algorithms to UPPAAL. It is then possible to derive optimal traces for *Uniformly Priced Timed Automata* (UPTA) [BFH$^+$]. In this model the cost increases with a fixed rate as time elapses, or with a certain amount if a transition is taken. The cost is treated as a special clock with extra operations, but such that we can still use the efficient data structures currently used in UPPAAL. First results for the steel plant and several benchmark problems were obtained in [BFH$^+$], and we hope to include an option that allows to find optimal traces to goal states in the next release of UPPAAL.

To be able to find time-optimal traces is very useful, but in many situations we would like to have a more general notion of cost. We proposed the model of *Linearly Priced Timed Automata* (LPTA) to be able to model for example machines that use a different amount of energy per time unit. This model extends timed automata with *prices* on all transitions and locations. In these models, the cost of taking an action transition is the price associated with the transition, and the cost of delaying d time units in a location is $d \cdot p$, where p is the price

associated with the location. The cost of a trace is simply the accumulated sum of costs of its delay and action transitions.

To treat LPTA algorithmically, we introduce *priced zones*, which assign to a zone a linear function that defines the minimal cost of reaching a state in that zone. In [BFH+00] it was shown that given a set of goal states the cost-optimal trace is computable. This result is quite remarkable since several similar extensions of timed automata have been proven to be undecidable. A prototype implementation allows us to perform first experiments [LBB+01].

2.2 PARAMETRIC-UPPAAL: Solving Parameterized Reachability Problems

Timed model checking if frequently applied with the intention to find out, whether the timing constants of the model are correct. A common problem is to adjust timing parameters in a way, that yield a desired behavior. This can be achieved if we given a timed automaton with *parameters* in the guards and if some or all values for the parameters are synthesized to make the model behave correctly, i.e., satisfy a certain TCTL formula. We call this *parametric model checking*. This problem is addressed in [AHV93], where it is shown to be undecidable for systems with three clocks or more. A semi-decision procedures is suggested in [AHV93] which finds the correct values for the parameters when it terminates.

We extend the model of timed automata to *parametric timed automata* by adding a set of parameters. Guards in parametric timed automata can be on the form $x \bowtie e$ or $x - y \bowtie e$ where e is a linear expression over the parameters. Having guards of this type gives a natural way of defining a symbolic state-space including parameters. Instead of having integers in the entries of a DBM we use *parametric DBMs* (PDBMs) where the entries are linear expressions over the parameters.

All the operations on DBMs are based on adding or comparing entries of DBMs. Without knowing anything about the values of the parameters we can in general not compare linear expressions over the parameters to each other or to integers. Comparing a parameter p to the constant 3 has two possible outcomes depending on the value of p. When such comparisons arises we will have to distinguish both possibilities. We will do this by adding a *constraint set* to a PDBM, consisting of constraints of the form $e \bowtie e'$ where e and e' are linear expressions and $\bowtie \in \{<, \leq, >, \geq\}$. In the example from before we will then split into two cases, one where the constraint $p < 3$ is added to the constraint set and one where $p \geq 3$ is added to the constraint set. We can now compare entries of PDMs based on their constraint sets.

Changing DBMs to PDBMs and letting symbolic states consist of the location vector, a PDBM, and a constraint set, the standard algorithm for state-space exploration can be used. When a state satisfying the property is found the constraints in the constraint set of the state gives the constraints on the parameters needed for the state to be reachable. If we want to find all the possible values

for the parameters we need to search the complete state-space to find all the different constraint sets making a goal state reachable.

We have implemented a parametric version of UPPAAL allowing parameters in clock guards and invariants. For deciding minimum between linear expressions we have borrowed a LP solver from the PMC tool [BSdRT01]. Parametric versions of the root-contention protocol and the bounded retransmission protocol have been analyzed using the implementation and minor errors in two published papers on these protocols have been discovered.

Since the problem is undecidable, UPPAAL is not guaranteed to terminate. As a pragmatic remedy, our algorithm outputs an explored state and the corresponding constraint set, as soon as it if found to satisfy the property. This allows the user to get partial results which can be very useful and in many cases are the full results though the search has not terminated. It is also possible to give initial constraints as input which in many case will make the search terminate much faster, or check whether partial results obtained are actually the full results.

2.3 STOPWATCH-UPPAAL: From Timed Automata to Hybrid Systems

For purposes of efficiency, the modeling language of UPPAAL was initially designed to be rather limited in expressive power. In particular, when modeling hybrid systems composed of discrete controller programs and continuous plants the timed automata model underlying UPPAAL is rather restrictive.

One useful extension of timed automata is that of linear hybrid automata [HHWT97]. In this model guards may be general linear constraints and the evolution rate of continuous variables may be given by arbitrary intervals. Consequently, model-checking and reachability checking is known to be undecidable for this model and more importantly the state-space exploration requires manipulation and representation of general polyhedra, which is computationally rather expensive.

In [CL00] an extension of UPPAAL with stopwatches (clocks that may be stopped occasionally) has been given allowing an approximate analysis of the full class of linear hybrid automata to be carried out using the efficient data structures and algorithms of UPPAAL.

In particular, this work investigates the expressive power of stopwatch automata, and shows as a main result that any finite or infinite *timed language* accepted by a *linear hybrid automaton* is also acceptable by a stopwatch automaton. The consequences of this result are two-fold: firstly, it shows that the seemingly minor upgrade from timed automata to stopwatch automata immediately yields the full expressive power of linear hybrid automata. Secondly, reachability analysis of linear hybrid automata may effectively be reduced to reachability analysis of stopwatch automata. This, in turn, may be carried out using an easy (over-approximating) extension of the efficient reachability analysis for timed automata to stopwatch automata. In [CL00] we also report on preliminary experiments on analyzing translations of linear hybrid automata using a stopwatch-extension of UPPAAL.

2.4 PRUPPAAL: Probabilistic Timed Automata

UPPAAL can check whether a network of timed automata satisfies a safety or a liveness (timed) property. Many times, this type of properties are not expressive enough to assert adequately the correctness of a system. Take for instance the well known Alternating Bit Protocol (ABP). Using UPPAAL, we can check whether the ABP satisfies properties like "*every message that is sent will eventually be received*" or "*every message that is sent will be received within Δ μsec.*" In fact we will see that the former is satisfied but not the latter, regardless of the value of Δ. If our interest is to provide quality of service, the latest property becomes as important as the former one. However, the fact that the ABP does not satisfy the second property does not necessarily make it an incorrect protocol. Knowing the probability with which a message is lost or damaged during transmission, we can determine the probability that a message is received within Δ μsec. The correctness of the ABP is now depend on whether we consider that such a probability measure is satisfactory.

Verification of probabilistic timed systems is one of the future directions pursued by UPPAAL. Probabilistic timed automata are a natural extension of timed automata with probabilities. The probabilistic information is attachted to edges. Now, an edge has the form $s \xrightarrow{g,a} p$ where s is a control node, g is a guard, a is an action name, and p is a probability function on pairs of set of clocks to be reset and control nodes. Figure 4 depicts a probabilistic timed automaton, that models a lossy channel. A message that is sent can be lost with probability $\frac{1}{100}$, otherwise it is transmitted within 10 to 20 nanoseconds. You can think of this automaton as model of the medium in the ABP.

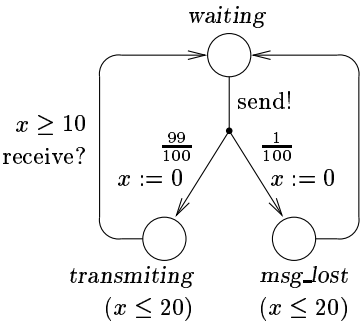

Figure4. A lossy channel.

On the setting of probabilistic timed systems we formally describe properties using PTCTL [HJ94]. PTCTL extends TCTL with modalities to express probabilities. For instance, $\mathsf{P}_{\geq 0.95}(\forall\Box_{\leq 1000} \mathit{received})$ expresses that with probability at least 0.95, every message is received within 1000 nanoseconds in any possible execution.

Solutions to model check probabilistic timed automata have been proposed in [Jen96] and [KNSS99]. Unfortunately these approaches are based on the construction of a region graph [ACD93] and therefore they heavily suffer from the state explosion problem. Another solution proposed in [KNSS99] is to use a modification of the forward reachability technique implemented in UPPAAL [YPD94]. Unfortunately, such a modification cannot decide the validity of simple reachability properties in general.

Our proposal is to use minimization techniques [ACH+92] in order to obtain (probabilistic) zone graphs that are stable and which behave in a similar manner to region graphs. However, this technique is still significantly more expensive

when compared to the usual forward reachability analysis. In order to reduce the state space we plan to explore the use of CDD's [LWYP99] to represent non-convex zones as well as dynamic partition techniques [JHR99].

2.5 HUPPAAL: Hierarchical Structures for Modeling

Hierarchical structures are a popular theme in specification formalisms, such as statecharts [Har87] and UML [BRJ98]. The main idea is that locations not necessarily encode atomic points of control, but can serve as an abbreviation for more complex behavior. If a non-atomic location is entered, this may trigger a cascade of events irrelevant to a higher level of the system. If a more detailed view is required, the explicit description of the sub-component can be found isolated, since dependencies between the different levels of hierarchy are restricted.

The immediate benefit is a concise description, which allows to view a complex system on different levels of abstraction and nevertheless contains all information in detail. Moreover, symmetries can be expressed explicitly: If two sub-components A and B of a super-state S are structurally identical, they may be described as instantiations of the same template (with possibly different parameters). Copies of states may exist for notational convenience, ambiguities are resolved by a unique-name assumption.

We believe that UPPAAL can benefit greatly from these concepts, since they support a cleaner and more structured design of large systems. The model can be constructed top down, starting with a very abstract notion that is refined subsequently. The simulator can then be used to validate the model against the intuition of the designer. Conceptually, it is possible to reason about the model with different stages of granularity. Compositional verification can make use of this, if local information suffices to establish safety- and deadlock-properties. With respect to property-preserving abstractions, the structural information gives a natural refinement relation.

A second—however ambitious—goal is to exploit the structure in shaping more efficient model-checking algorithms. Related work [AW99] indicates, that locality of information can be exploited straightforward in reachability analysis. Also, the work in [LNAB+98] indicate that—at least for un-timed systems—one may exploit the hierarchical structure of a system during analysis. In UPPAAL this is more difficult, since all parallel processes implicitly synchronize on the passage of time. Approaches for local-time semantics [BJLY98] have yet to be shown to improve verification time in reasonable scenarios, i.e., where the dependency between parallel sub-components is low, thus that not all interleavings have to be taken into account.

As a first step towards this, we work on a careful definition of hierarchical timed automata, that support encapsulation and local definitions. In particular, the synchronization of joins raises semantic problems that can be resolved in various ways.

Case-studies are planned to test the naturalness of these definitions in complex examples. We experiment with a prototype translation of hierarchical timed automata into a parallel composition of (flat) timed automata. This flattened

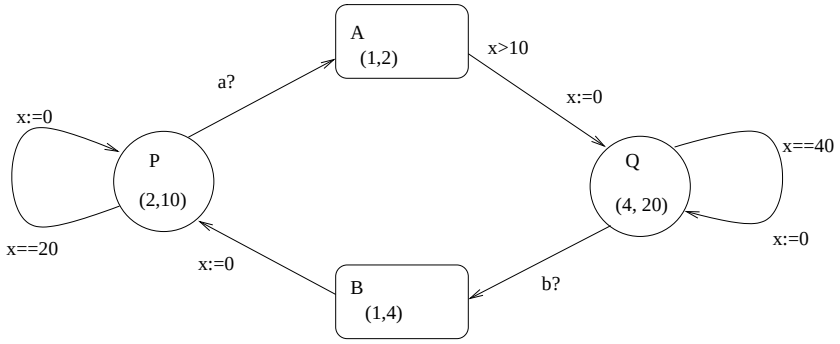

Figure5. Timed Automaton with Periodic and Sporadic Tasks.

system necessarily contains auxiliary constructs to imitate the behavior of the hierarchical ones. We expect the case-studies to give an intuition, whether this translation slack is tolerable.

The design of the hierarchical timed automata is meant to be close to UML statechart diagrams. As for the real-time aspect, one output of this considerations will be a real-time profile, that defines an extension of UML formalisms with clocks and timed invariants in a standard way. This work is carried out in the context of AIT-WOODDES project No IST-1999-10069.

2.6 ExUppaal: Executable Timed Automata

In this work we develop an executable version of timed automata. We view a timed automaton as an abstract model of a running software. The model describes the possible external events (alphabets accepted by the automaton) that may occur during the execution and the occurrence of the events must follow the timing constraints (given by the clock constraints). But the model gives no information on how these events should be handled. We use an extended version of timed automata ([EWY99]) with real time tasks that may be periodic and/or sporadic.

The main idea is to associate each node of an automaton with a task (or several tasks in the general case). A task is assumed to be an executable program with two given parameters: its worst case execution time and deadline. An example is shown in Figure 5. The system shown consists of 4 tasks as annotation on nodes, where P, Q are periodic with periods 20 and 40 respectively (specified by the constraints: x==20 and x==40), and A, B are sporadic or event driven (by event a and b respectively). The pairs in the nodes give the computation times and deadlines for tasks e.g. for P they are 2 and 10 respectively.

Intuitively, a discrete transition in an extended timed automaton denotes an event releasing a task and the guard (clock constraints) on the transition specifies all the possible arrival times of the associated task. Note that in the simple automaton shown in Figure 5, an instance of task A could be released

before the preceeding instance of task P has been computed. This means that the scheduling queue may contains at least P and A. In fact, instances of all four tasks may appear in the queue at the same time.

Semantically, an extended automaton may perform two types of transitions just as an ordinary timed automaton. In addition, an action transition will release a new instance of the task associated with the destination node. Assume that there is a queue (the scheduling queue) holding all the task instances ready to run. It corresponds to the ready queue in an operating systems. Whenever a task is released, it will be put in the scheduling queue for execution. A semantic state of an extended automaton is a triple consisting of a node (the current control node), clock assignment (the current setting of the clocks) and a task queue (the current status of the ready queue). Then a delay transition of the timed automaton corresponds to the execution of the task with earliest deadline and idling for the other waiting tasks, and a sequences of discrete transitions corresponds to a sequence of arrivals of tasks. Naturally a sequence of tasks is *schedulable* if all the tasks can be executed within their deadlines and an automaton is schedulable if all task sequences are schedulable.

In [EWY99], it is shown that the schedulability problem for extended automata can be solved by reachability analysis for non-preemptive tasks. It is equivalent to prove that all schedulable states are schedulable. For preemptive tasks, unfortunately the problem is undecidable. In fact the model will be as expressive as timed automata with stop watches.

Currently we are working on automatic code synthesis for the extended model. Inspired by the design philosophy of synchronous languages e.g. Esterel, we assume that the underlying RT operating system guarantees the *Synchrony Hypothesis*, that is the OS system functions takes little time compared to the worst case execution times and deadlines of tasks. The idea is to use system functions (primitives) provided by the underlying operating system or run-time system, to code the discrete transitions (the control structure) of an automaton, and to compute the tasks on nodes by procedure calls or light weight threads.

If an automaton is schedulable (checked by schedulability analysis that all tasks instances can be computed within their deadlines), and the synchrony hypothesis is guaranteed by the underlying operating system, the generated code in execution will meet the constraints imposed on the tasks.

2.7 Hybrid Automata Animation

In several case-studies with UPPAAL we have identified a need to visualize the execution of the automata. Currently the simulator in UPPAAL's GUI allows an interactive "execution" of the modeled system. The user can manually select one of the enabled transitions and go to the next state of the system. This can be very helpful in understanding the model, but it is still on the difficulty level of the actual automaton. To make good use of the simulator the user needs to understand all the details of the modelling language and all details of the specific system.

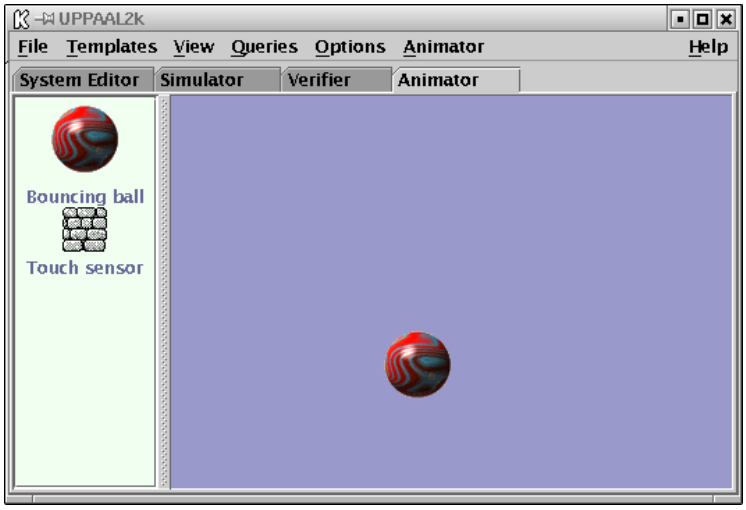

Figure6. A protoype of the hybrid automata animation tool in UPPAAL.

To describe a typical situation, consider one person performing the modeling and verification of a system, whereas another person wants to validate that the model is "correct" in the sense that it is an accurate description of the actual system. Exploring all possible simulation traces is often a very tedious work. With a visualization tool, where the user can interact with the underlying model on a higher level via buttons, sliders, and other objects in a graphical environment this validation task becomes much simpler.

Several other tools have responded to this demand, for example MATLAB/-Simulink and Statemate, where graphical animation of the models are possible. By considering simulation and animation of hybrid automata, we adopt these techniques and aim at taking them one step further. The plan is to generalize the model of timed automata in UPPAAL to the more expressive model of hybrid automata, where changes of a state is defined by ordinary differential equations (ODE). To each location we associate a set of ODE's that describe how real-valued variables change over time. This more expressive model will be used only in the animator to model and visualize the behavior a system's environment. The system itself will still normally be modeled with timed automata.

The animation is based on the values of the variables, the current location, and the signals. The values of the variables are calculated at discrete time points using numerical solution methods. To solve the ODE's we use a free package named CVODE [2]. Around this we have implemented a *Hybrid Automata Interpreter* that handles the automata transitions, synchronizations, etc., and allows the user to define the ODE's using a library of mathematical functions. The

[2] More information about the CVODE package can be found at the web site `http://www.netlib.org`.

values that come out of the Hybrid Automata Interpreter are used to drive the animation.

In the animation tool, the user defines a *view* of the whole system by setting certain parameters. For instance, in a 2-dimensional view two variables x and y could be used to give the position of an image illustrating the modeled component, and the current location of the corresponding automaton could be visualized as color-changes in the image. The user could also decide what actions (e.g. mouse-clicks) should correspond to signals sent to the visualized automata model.

Following the example of UPPAAL's multi-platform user interface (see Section 1), the animator is implemented in Java. In this way it fits seamlessly into the existing tool architecture. Figure 6 shows the animator when used to simulate a bouncing ball.

3 Recent Developments in UPPAAL

In this section we describe the recent developments in UPPAAL, which are primarily aimed at improving the efficiency of the model-checker of the tool. In particular, the development of new internal data-structures, and approximation and partial-order reduction techniques are considered relevant.

3.1 CDD's: Clock Difference Diagrams

Difference Bound Matrices (DBM's) as the standard representation for time zones in analysis of Timed Automata have a well-known shortcoming: they are not closed under set-union. This comes from the fact that a set represented by a DBM is convex, while the union of two convex sets is not necessarily convex.

Within the symbolic computation for the reachability analysis of UPPAAL, set-union however is a crucial operation which occurs in every symbolic step. The shortcoming of DBM's leads to a situation, where symbolic states which could be treated as one in theory have to be handled as a collection of several different symbolic states in practice. This leads to trade-offs in memory and time consumption, as more symbolic states have to be stored and visited during in the algorithm.

DBM's represent a zone as a conjunction of constraints on the differences between each pair of clocks of the timed automata (including a fictitious clock representing the value 0). The major idea of CDD's (Clock Difference Diagrams) is to store a zone as a decision tree of clock differences, generalizing the ideas of BDD's (Binary Decision Diagrams, see [Bry86]) and IDD's (Integer Decision Diagrams, see [ST98])

The nodes of the decision tree represent clock differences. Nodes on the same level of the tree represent the same clock difference. The order of the clock differences is fixed a-priori, all CDD's have to agree on the same ordering. The leaves of the decision tree are two nodes representing true and false, as in the case of BDD's.

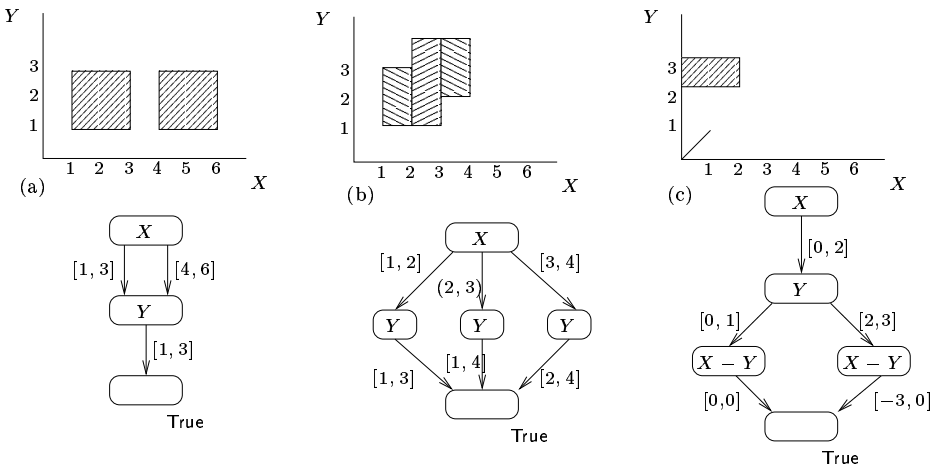

Figure7. Three example CDD's. Intervals not shown lead implicitly to False.

Each node can have several outgoing edges. Edges are labeled with integral intervals: open, half-closed and closed intervals with integer values as the borders. A node representing the clock difference $X - Y$ together with an outgoing edge with interval I represents the constraint "$X - Y$ within I". The leafs represent the global constraints true and false respectively.

A path in a CDD from a node down to a leaf represents the set of clock values with fulfill the conjunction of constraints found along the path. Remember that a constraint is found from the pair node and outgoing edge. Paths going to false thus always represent the empty set, and thus only paths leading to the true node need to be stored in the CDD. A CDD itself represents the set given by the union of all sets represented by the paths going from the root to the true node. From this clearly CDD's are closed under set-union. Figure 7 gives three examples of two-dimensional zones and their representation as CDDs. Note that the same zone can have different CDD representations.

All operations on DBM's can be lifted straightforward to CDD's. Care has to be taken when the canonical form of the DBM is involved in the operation, as there is no direct equivalent to the (unique) canonical form of DBM's for CDD's.

CDD's generalize IDD's, where the nodes represent clock values instead of clock differences. As clock differences, in contrast to clock values, are not independent of each other, operations on CDD's are much more elaborated than the same operations on IDD's. CDD's can be implemented space-efficient by using the standard BDD's technique of sharing common substructure. This sharing can also take place between different CDD's.

Experimental results have shown that using CDD's instead of DBM's can lead to space savings of up to 99%. However, in some cases a moderate increase in run time (up to 20%) has to be paid. This comes from the fact that operations

involving the canonical form are much more complicated in the case of CDD's compared to DBM's. More on CDD's can be found in [LWYP99] and [BLP+99].

3.2 Compact Representation of States

Symbolic states are the core objects of state space search and their representation is one of the key issues in implementing an efficient verifier. In the earlier versions of UPPAAL each entity in a state (i.e., an element in the location vector, the value of an integer variable or a bound in the DBM) is mapped on a machine word. The reason for this is simplicity and speed. However, the number of possible values for each entity is usually small, and using a whole machine word for each of them is often a waste of space.

To solve this problem two additional, more compact, state representations have been implemented. In both of them the discrete part of each state is encoded as a number, using a multiply and add scheme. This encoding is much like looking at the discrete part as a number, where each digit is an entity in the discrete state and the base varies with the number of different digits.

In the first packing scheme, a DBM is encoded using the same technique as the discrete part of the state. This gives a very space efficient but computationally expensive representation, where each state takes a minimum amount of memory but where a number of bignum division operations have to be performed to check inclusion between two DBMs.

In the second packing scheme, some of the space performance is sacrificed to allow a more efficient inclusion check. Here each bound in the DBM is encoded as a bit string long enough to represent all the possible values of this bound plus one *test bit*, i.e., if a bound can have 10 possible values then five bits are used to represent the bound. This allows cheap inclusion checking based on ideas of Paul and Simon [PS80] on comparing vectors using subtraction of bit strings.

In Figure 8 we see that the space performance of these representations are both substantially better than the traditional representation, with space savings of between 25% and 70%. As we expect, the performance of the first packing scheme, with an expensive inclusion check, is somewhat better, space-wise, than the packing scheme with the cheap inclusion check.

Considering the time performance for the packed state representations (see Figure 9), we note that the price for using the encoding with expensive inclusion check is a slowdown of 2 – 12 times, while using the other encoding sometimes is even faster than the traditional representation.

3.3 Partial Order Reduction for Timed Systems

Partial-order reduction is a well developed technique, whose purpose is to reduce the usage of time and memory in state-space exploration by avoiding to explore unnecessary interleavings of independent transitions. It has been successfully applied to finite-state systems. However, for timed systems there has been less progress. The major obstacle to the application of partial order reduction to

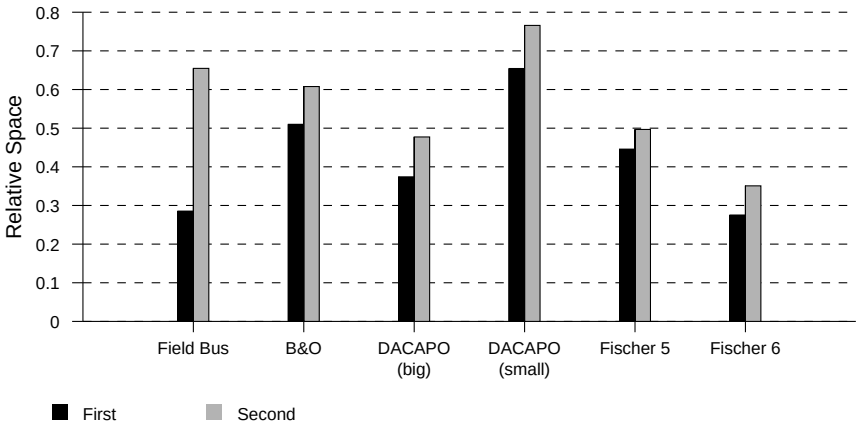

Figure8. Space performance for the two packing schemes (denoted First and Second).

timed systems is the assumption that all clocks advance at the same speed, meaning that all clocks are implicitly synchronized. If each process contains (at least) one local clock, this means that advancement of the local clock of a process is not independent of time advancements in other processes. Therefore, different interleavings of a set of independent transitions will produce different combinations of clock values, even if there is no explicit synchronization between the processes or their clocks.

In [BJLY98], we have presented a partial-order reduction method for timed systems based on a *local-time* semantics for networks of timed automata. The main idea is to remove the implicit clock synchronization between processes in a network by letting local clocks in each process advance independently of clocks in other processes, and by requiring that two processes *resynchronize* their local time scales whenever they communicate. The idea of introducing local time is related to the treatment of local time in the field of parallel simulation. Here, a simulation step involves some local computation of a process together with a corresponding update of its local time. A snapshot of the system state during a simulation will be composed of many local time scales. In our work, we are concerned with verification rather than simulation, and we must therefore represent sets of such system states symbolically.

A symbolic version of the local-time semantics is developed in terms of predicate transformers, which enjoys the desired property that two predicate transformers are independent if they correspond to disjoint transitions in different processes. Thus we can apply standard partial order reduction techniques to the problem of checking reachability for timed systems, which avoid exploration of unnecessary interleavings of independent transitions. The price is that we must introduce extra machinery to perform the resynchronization operations on local

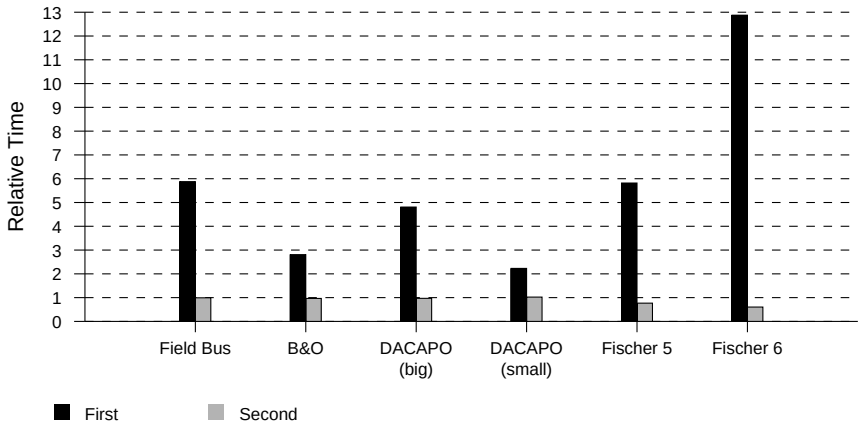

Figure9. Time performance for the two packing schemes (denoted First and Second).

clocks. A variant of DBM representation has been developed for symbolic states in the local time semantics for efficient implementation of our method.

We have developed a prototype implementation based on the technique. Unfortunately, our experimental results are not so satisfactory, which is not so surprising due to the large number of local clocks introduced. We are still struggling for an efficient implementation.

3.4 DUPPAAL: Distributed State Space Exploration

Real time model checking is a time and memory consuming task, quite often reaching the limits of both computers and the patience of users. An increasingly common solution to this situation is to use the combined power of computers connected in a cluster. Good results have recently been achieved for UPPAAL by distributing both the model checking algorithm and the main data structures [BHV00].

Recall the basic state-space exploration described briefly in Section 1.2. The distributed version of this algorithm is similar. Each node (processing unit) in the cluster will hold fragments of both the WAITING list and the PASSED list according to a distribution function mapping states to nodes. In the beginning, the distributed WAITING list will only hold the initial state. What ever node hosts this state will compare it to its still empty PASSED list fragment and consequently explore it. Now, the successors are distributed according to the distribution function and put into the WAITING list fragment on the respective nodes. This process will be repeated, but now several nodes contain states in their fragment of the WAITING list and quickly all nodes become busy exploring their part of the state space. The algorithm terminates when all WAITING list

fragments are empty and no states are in the process of being transfered between nodes.

The distribution function is in fact a hash function. It distributes states uniformly over its range and hence implements what is called *random load balancing*. Since states are equally likely to be mapped to any node, all nodes will receive approximately the same number of states and hence the load will be equally distributed.

This approach is very similar to the one taken by [SD97]. The difference is that UPPAAL uses symbolic states, each covering (infinitely) many concrete states. In order to achieve optimal performance, the lookup performed on the PASSED list is an inclusion check. An unexplored symbolic state taken from the WAITING list is compared with all the explored symbolic states on the PASSED list, and only if non of those states cover (include) the unexplored symbolic state it is explored. For this to work in the distributed case, the distribution function needs to guarantee that potentially overlapping symbolic states are mapped to the same node in the cluster. A symbolic state can be divided into a discrete part and a continuous part. By only basing the distribution on the discrete part, the above is ensured.

Peculiarly, the number of explored states is heavily dependent on the search order. For instance, let s and t be two symbolic states such that s includes t. Thus, if s is encountered before t, t will not be explored because s is already on the PASSED list and hence covers t. On the other hand, if we encounter t first, both states will be explored. Experiments have shown that breadth first order is close to optimal when building the complete reachable state-space. Unfortunately, ensuring strict breadth first order in a distributed setting requires synchronizing the nodes, which is undesirable. Instead, we order the states in each WAITING list fragment according to their distance from the initial state, exploring those with the smallest distance first. This results in an approximation of the breadth first order. Experiments have shown that this order drastically reduces the number of explored states compared to simply using a FIFO order.

This version of UPPAAL has been used on a Sun Enterprise 10000 with 24 CPUs and on a Linux Beowulf cluster with 10 nodes. Good speedups have been observed on both platforms when verifying large systems (around 80% of optimal at 23 CPUs on the Enterprise 10000).

3.5 Dynamic Partitioning: Tackling the State Explosion Problem

This line of work addresses the *state-space explosion* problem that has to be overcomed in the verification of systems described by a parallel composition of several automata.

Recall that basic algorithm implemented in UPPAAL is an *exact* reachability algorithm that computes for each reachable location of the global system a finite union of zones. One promising idea here is to make use of *approximations* in order to reduce the complexity of this algorithm, and nevertheless stay conservative with respect to safety properties. In many cases, this greatly improves performance without sacrificing relevant information.

The current release of UPPAAL already contains options for convex-hull approximation of zones, basically associating one unique zone to each reachable control location. Such a zone represents then an upper-approximation of the exact reachable clock values in the considered location. Another possible approximation would consist in associating the same zone to *several* locations. We will use a combination of these two techniques.

Now, a major difficulty is to adjust the level of approximation used. A tradeoff has to be found between precision and efficiency. Rough approximations make analysis cheaper but may fail in showing non-trivial properties; more precise analyses may be too expensive to be able to deal with big systems.

The solution we propose [JHR99,Jea00,Jea] is defined within the framework of abstract interpretation theory [CC77]. It relies on the use of an abstract lattice combining Boolean and numerical properties (e.g. zones), and exploits the partitioning of the state space of the system in order to adjust the precision of the analysis. Now, given a safety property, it is hardly possible to guess the good partition to check it, i.e., the coarsest partition that is still detailed enough to enable the proof of this property. We propose to start the analysis with a very coarse partition, and to automatically refine it according to the needs of verification, until the obtained precision enables a proof of the property, or until the partition cannot be refined in a reasonable way any more.

This technique has been implemented in the tool NBAC, using convex polyhedra to represent numerical properties, and has been successfully applied to the verification of synchronous programs [Jea00,Jea]. Work is currently done to extend the tool with continuous time semantic, and to connect it to the UPPAAL language for timed automata. We are also considering to replace the convex polyhedra lattice used in the tool by the cheaper lattice of zones, used in UPPAAL, or possibly the new lattice of octagons [Min00], that generalizes zones by allowing constraints of the form $m \leq x_i + x_j \leq M$.

4 Recent Case Studies

UPPAAL2k has been applied in a number of case studies. In this section we briefly describe a selection of the more recent ones. A more complete overview is given on the UPPAAL home page `http://www.uppaal.com/` (see the section "Documentation").

In [DY00], David and Wang report on an industrial application of UPPAAL to model and debug a commercial field bus communication protocol, AF100 (Advant Field-bus 100) developed and implemented by process control industry for safety-critical applications. The protocol has been running in various industrial environments over the world for the past ten years. Due to the complexity of the protocol and various changes made over the years, it shows occasionally unexpected behaviors. During the case study, a number of imperfections in the protocol logic and its implementation are found and the error sources are debugged based on abstract models of the protocol; respective improvements

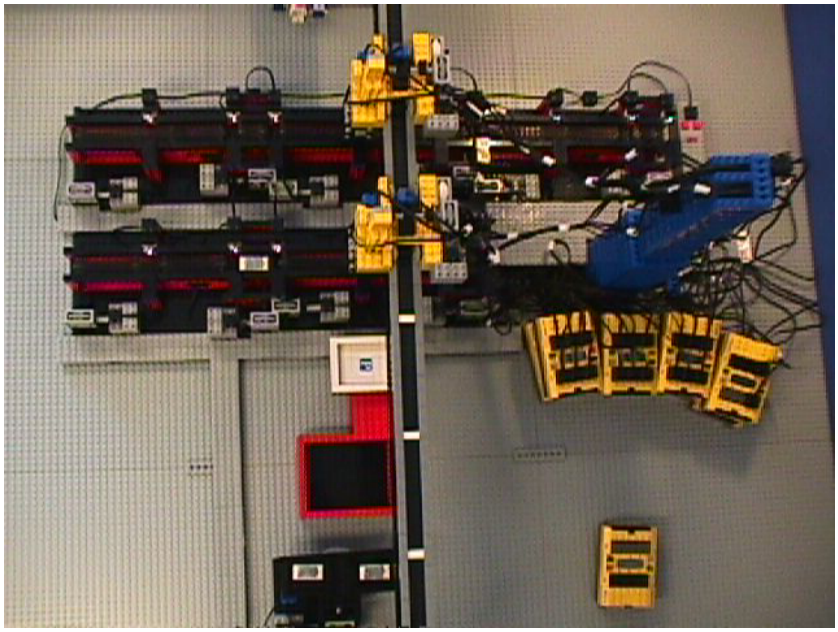

Figure10. An overview of the LEGO® plant.

have been suggested.

In [HLP00], Hune et al. address the problem of synthesizing production schedules and control programs for the batch production plant model built in LEGO® MINDSTORMS™ RCX™ shown in Figures 10. A timed automata model of the plant which faithfully reflects the level of abstraction needed to synthesize control programs is described. This makes the model very detailed and complicated for automatic analysis. To solve this problem a general way of adding guidance to a model by augmenting it with additional guidance variables and transition guards is presented. Applying the technique makes synthesis of control problems feasible for a plant producing as many as 60 batches. In comparison, only two batches could be scheduled without guides. The synthesized control programs have been executed in the plant. Doing this revealed some model errors.

The papers [Hun99,IKL+00] also consider systems controlled by LEGO® RCX™ bricks. Here the studied problem is that of checking properties of the actual programs, rather than abstract models of programs. It is shown how UPPAAL models can be automatically synthesized from RCX™ programs, written in the programming language *Not Quite C*, NQC. Moreover, a protocol to facilitate the distribution of NQC programs over several RCX™ bricks is developed and proved to be correct. The developed translation and protocol

are applied to a distributed LEGO® system with two RCX™ bricks pushing boxes between two conveyer belts moving in opposite directions. The system is modeled and some verification results with UPPAAL2k are reported.

In [KLPW99], Kristoffersen et. al. present an analysis of an experimental batch plant using UPPAAL2k. The plant is modeled as a network of timed automata where automata are used for modeling the physical components of the plant, such as the valves, pumps, tanks etc. To model the actual levels of liquid in the tanks, integer variables are used in combination with real-valued clocks which control the change between the (discrete) levels at instances of time which may be predicted from a more accurate hybrid automata model. An crucial assumption of this discretization is that the interaction between the tanks and the rest of the plant must be such that any plant event affecting the tanks only occurs at these time instances. If this assumption can be guaranteed (which is one of the verification efforts in this framework), the verification results are exact and not only conservative with respect to a more accurate model, where the continuous change of the levels may have been given by some suitable differential equation.

The paper [LAM99] reports on the first time, that a part of the Ada run-time complex has been formally verified. To eliminate most implementation dependencies and constructs with not clearly specified behavior in Ada, the Ravenscar Tasking Profile is used to implement the concurrency part. This significantly advances the possibility to formally verify properties of concurrent programs. The case study uses UPPAAL to prove fourteen properties, where one depends directly on an upper bound on a real-time clock value.

In an ongoing case study [AJ01], UPPAAL is applied to model and analyze a generalized version of a car looking system developed by Saab Automobile. The looking system is distributed over several nodes in the internal communication network that exists in all modern vehicles. The system consists of a central node gathering information and based on this instructing sub nodes attached to the physical hardware to lock or unlock doors, trunk lid, etc. The input sources are different kinds of remote controllers, speed sensors, automatic re-locking time-outs etc. which based on predefined rules may activate the locking mechanism. The model of the system is derived from the actual functional requirements of the looking system used at Saab Automobile. During the currently ongoing work with verifying the functional requirements of the model, some inconsistencies and other problems between requirement have been found and pointed out to the engineers.

5 Online Available Distributions

UPPAAL2k is currently available for Linux, SunOS and MS Windows platforms. It can be downloaded from the UPPAAL home page `http://www.uppaal.com/`.

Since July 1999, the tool has been downloaded by more than 800 different users in 60 countries. On the home page, you also find answers to frequently asked questions, online documentation, tutorials, and related research articles.

An open mailing list at `http://groups.yahoo.com/group/uppaal` serves as a lively discussion forum for both UPPAAL users and developers.

References

[ACD93] Rajeev Alur, Costas Courcoubetis, and David Dill. Model Checking in Dense Real Time. *Information and Computation*, 104:2–34, 1993.

[ACH+92] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, David Dill, and Howard Wong-Toi. Minimization of Timed Transition Systems. In *Proc. of CONCUR '92, Theories of Concurrency: Unification an d Extension*, pages 340–354, 1992.

[AHV93] Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. Parametric Real-time Reasoning. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing*, pages 592–601, 1993.

[AJ01] Tobias Amnell and Pontus Jansson. Report from astec-rt auto project — central locking system case study. In preparation, 2001.

[AW99] Rajeev Alur and Bow-Yaw Wang. "Next" Heuristic for On-the-fly Model Checking. In *Proc. of CONCUR '99: Concurrency Theory*, number 1664 in Lecture Notes in Computer Science, pages 98–113. Springer–Verlag, 1999.

[BDM+98] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos: A model-Checking Tool for Real-Time Systems. In *Proc. of the 10th Int. Conf. on Computer Aided Verification*, number 1427 in Lecture Notes in Computer Science, pages 546–550. Springer–Verlag, 1998.

[BFH+] Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim G. Larsen, Paul Pettersson, and Judi Romijn. Efficient Guiding Towards Cost-Optimality in UPPAAL. Accepted for publication in TACAS'2001.

[BFH+00] Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim G. Larsen, Paul Pettersson, Judi Romijn, and Frits Vaandrager. Minimum-Cost Reachability for Priced Timed Automata. Submitted for publication. Available at `http://www.docs.uu.se/docs/rtmv/papers/bfhlprv-sub00-1.ps.gz`, 2000.

[BHV00] Gerd Behrmann, Thomas Hune, and Frits Vaandrager. Distributing Timed Model Checking – How the Search Order Matters. In *Proc. of the 12th Int. Conf. on Computer Aided Verification*, number 1855 in Lecture Notes in Computer Science, pages 216–231. Springer–Verlag, 2000.

[BJLY98] Johan Bengtsson, Bengt Jonsson, Johan Lilius, and Wang Yi. Partial Order Reductions for Timed Systems. In *Proc. of CONCUR '98: Concurrency Theory*, number 1466 in Lecture Notes in Computer Science. Springer–Verlag, 1998.

[BLP+99] Gerd Behrmann, Kim G. Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Efficient Timed Reachability Analysis Using Clock Difference Diagrams. In *Proc. of the 11th Int. Conf. on Computer Aided Verification*, number 1633 in Lecture Notes in Computer Science. Springer–Verlag, 1999.

[BRJ98] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1998.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean-Function Manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, August 1986.

[BSdRT01] Giosuè Bandini, R. F. Lutje Spelberg, R. C. M. de Rooij, and W. J. Toetenel. Application of Parametric Model Checking - The Root Contention Protocol. In *Proc. of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)*, 2001.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. *Proc. of the 4*th *ACM Symposium on Principles of Programming Languages*, January 1977.

[CL00] Franc Cassez and Kim G. Larsen. The Impressive Power of Stopwatches. In *Proc. of CONCUR '2000: Concurrency Theory*, number 1877 in Lecture Notes in Computer Science, pages 138–152. Springer–Verlag, 2000.

[DY00] Alexandre David and Wang Yi. Modelling and Analysis of a Commercial Field Bus Protocol. In *Proc. of 12*th *Euromicro Conference on Real-Time Systems*, pages 165–172. IEEE Computer Society Press, June 2000.

[EWY99] Christer Ericsson, Anders Wall, and Wang Yi. Timed Automata as Task Models for Eventdriven Systems. In *Proceedings of RTSCA 99*. IEEE Computer Society Press, 1999.

[Feh99] Ansgar Fehnker. Scheduling a Steel Plant with Timed Automata. In *Proc. of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA99)*, pages 280–286. IEEE Computer Society Press, 1999.

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[HHWT97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: A Model Checker for Hybrid Systems. In Orna Grumberg, editor, *Proc. of the 9th Int. Conf. on Computer Aided Verification*, number 1254 in Lecture Notes in Computer Science, pages 460–463. Springer–Verlag, 1997.

[HJ94] Hans A. Hansson and Bengt Jonsson. A Logic for Reasoning about Time and Reliability. *Formal Aspects of Computing*, 6:512–535, 1994.

[HLP00] Thomas Hune, Kim G. Larsen, and Paul Pettersson. Guided Synthesis of Control Programs Using uppaal. In Ten H. Lai, editor, *Proc. of the IEEE ICDCS International Workshop on Distributed Systems Verification and Validation*, pages E15–E22. IEEE Computer Society Press, April 2000.

[Hun99] Thomas Hune. Modelling a Real-time Language. In *Proceedings of FMICS*, 1999.

[IKL+00] Torsten K. Iversen, Kåre J. Kristoffersen, Kim G. Larsen, Morten Laursen, Rune G. Madsen, Steffen K. Mortensen, Paul Pettersson, and Chris B. Thomasen. Model-Checking Real-Time Control Programs — Verifying LEGO Mindstorms Systems Using uppaal. In *Proc. of 12*th *Euromicro Conference on Real-Time Systems*, pages 147–155. IEEE Computer Society Press, June 2000.

[Jea] Bertrand Jeannet. Dynamic Partitioning in Linear Relation Analysis. Application to the Verification of Reactive Systems. to appear in *Formal Methods and System Design*, Kluwer Academic Press.

[Jea00] Bertrand Jeannet. *Partitionnement dynamique dans l'analyse de relations linéaires et application à la vérification de programmes synchrones.* PhD thesis, Institut National Polytechnique de Grenoble, September 2000.

[Jen96] Henrik E. Jensen. Model Checking Probabilistic Real Time Systems. In B. Bjerner, M. Larsson, and B. Nordström, editors, *Proceedings of the 7th Nordic Workshop on Programming Theory,* Göteborg Sweden, Report 86, pages 247–261. Chalmers University of Technolog, 1996.

[JHR99] Bertrand Jeannet, Nicolas Halbwachs, and Pascal Raymond. Dynamic Partitioning in Analyses of Numerical Properties. In *Static Analysis Symposium, SAS'99*, Venezia (Italy), September 1999.

[KLPW99] Kåre Kristoffersen, Kim G. Larsen, Paul Pettersson, and Carsten Weise. Vhs Case Study 1 - experimental Batch Plant using UPPAAL. BRICS, University of Aalborg, Denmark, `http://www.cs.auc.dk/research/FS/-VHS/cs1uppaal.ps.gz`, May 1999.

[KNSS99] Marta Z. Kwiatkowska, Gethin Norman, Roberto Segala, and Jeremy Sproston. Automatic Verification of Real-Time Systems with Probability Distributions. In J.-P. Katoen, editor, *Proceedings of the 5th AMAST Workshop on Real-Time and Probabilistic System,* Bamberg, Germany, number 1601 in Lecture Notes in Computer Science, pages 75–95. Springer–Verlag, 1999. An extended version will appear in Theoretical Computer Science.

[Lam87] Leslie Lamport. A Fast Mutual Exclusion Algorithm. *ACM Trans. on Computer Systems*, 5(1):1–11, February 1987. Also appeared as SRC Research Report 7.

[LAM99] Kristina Lundqvist, Lars Asplund, and Stephen Michell. A Formal Model of the Ada Ravenscar Tasking Profile; Protected Objects. In Springer-Verlag, editor, *Proc. of the Ada Europe Conference*, pages 12–25, 1999.

[LBB+01] Kim G. Larsen, Gerd Behrmann, Ed Brinksma, Ansgar Fehnker, Thomas Hune, Paul Pettersson, and Judi Romijn. As Cheap as Possible: Efficient Cost-Optimal Reachability for Priced Timed Automata. Submitted for publication, 2001.

[LLPY97] Fredrik Larsson, Kim G. Larsen, Paul Pettersson, and Wang Yi. Efficient Verification of Real-Time Systems: Compact Data Structures and State-Space Reduction. In *Proc. of the 18*th *IEEE Real-Time Systems Symposium*, pages 14–24. IEEE Computer Society Press, December 1997.

[LNAB+98] Jørn Lind-Nielsen, Henrik Reif Andersen, Gerd Behrmann, Henrik Hulgaard, Kåre J. Kristoffersen, and Kim G. Larsen. Verification of Large State/Event Systems Using Compositionality and Dependency Analysis. In Bernard Steffen, editor, *Proc. of the 4*th *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1384 in Lecture Notes in Computer Science, pages 201–216. Springer–Verlag, 1998.

[LP97] Henrik Lönn and Paul Pettersson. Formal Verification of a TDMA Protocol Startup Mechanism. In *Proc. of the Pacific Rim Int. Symp. on Fault-Tolerant Systems*, pages 235–242, December 1997.

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

[LWYP99] Kim G. Larsen, Carsten Weise, Wang Yi, and Justin Pearson. Clock Difference Diagrams. *Nordic Journal of Computing*, 6(3):271–298, 1999.

[Min00] Antoine Miné. The Numerical Domain of Octagons and Application to the Automatic Analysis of Programs. Master's thesis, École Normale Supérieure de Paris, 2000.

[Pet99] Paul Pettersson. *Modelling and Analysis of Real-Time Systems Using Timed Automata: Theory and Practice.* PhD thesis, Department of Computer Systems, Uppsala University, February 1999.

[PS80] Wolfgang J. Paul and Janos Simon. Decision Trees and Random Access Machines. In *Logic and Algorithmic*, volume 30 of *Monographie de L'Enseignement Mathématique*, pages 331–340. L'Enseignement Mathématique, Université de Genève, 1980.

[SD97] Ulrich Stern and David L. Dill. Parallelizing the Murφ Verifier. In Orna Grumberg, editor, *Proc. of the 9th Int. Conf. on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 256–267. Springer–Verlag, June 1997. Haifa, Isreal, June 22-25.

[ST98] Karsten Strehl and Lothar Thiele. Symbolic Model Checking of Process Networks Using Interval Diagram Techniques. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD-98)*, pages 686–692, 1998.

[Yov97] Sergio Yovine. Kronos: A verification Tool for Real-Time Systems. *Springer International Journal of Software Tools for Technology Transfer*, 1(1/2), October 1997.

[YPD94] Wang Yi, Paul Pettersson, and Mats Daniels. Automatic Verification of Real-Time Communicating Systems By Constraint-Solving. In Dieter Hogrefe and Stefan Leue, editors, *Proc. of the 7th Int. Conf. on Formal Description Techniques*, pages 223–238. North–Holland, 1994.