SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **00**:1–6 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Developing UPPAAL over 15 years

Gerd Behrmann[1], Alexandre David[2,*], Kim Guldstrand Larsen[2], Paul Pettersson[3], Wang Yi[4]

[1] *NORDUnet A/S, Copenhagen, Denmark*
[2] *Department of Computer Science, Aalborg University, Denmark*
[3] *Mälardalen Real-Time Research Centre, Mälardalen University, Sweden*
[4] *Department of Information Technology, Uppsala University, Sweden*

SUMMARY

Uppaal is a tool suitable for model checking real-time systems described as networks of timed automata communicating by channel synchronisations and extended with integer variables. Its first version was released in 1995 and its development is still very active. It now features an advanced modelling language, a user-friendly graphical interface, and a performant model checker engine. In addition, several flavours of the tool have matured in recent years. In this paper, we present how we managed to maintain the tool during 15 years, its current architecture with its challenges, and we give future directions of the tool.

KEY WORDS: UPPAAL, real-time, model-checker, development

INTRODUCTION

UPPAAL is first of all a research tool born from the collaboration of Uppsala and Aalborg universities [44]. It takes its theoretical roots from Alur and Dill's pioneer work on timed automata [2]. Its performance originally comes from zones [36] as a representation for states and the efficient implementation of operators on its canonical data-structure known as difference bound matrix (DBM) [13]. Since then the development has been fuelled by scientific results on algorithms or new data structures [7, 8, 10, 18, 37, 38], academic case-studies [41, 17, 24, 28, 20], industrial case-studies [39, 21, 12, 1, 4], and also teaching [27].

On the other hand, having such a tool helps to develop and test new theories and algorithms, which has given us synergy during the last decade between tool development and theoretical results.

*Correspondence to: Department of Computer Science, Aalborg University, Denmark

Recently, the tool has blossomed into several domain specific versions, namely, CORA [8, 9] (cost-optimal reachability), TRON [34, 35] (online testing), COVER [29, 30] (offline test generation), TIGA [6] (timed game solver), PORT [25] (component based and partial order), PRO (extension with probabilities, in progress), and TIMES [23, 3] (scheduling and analysis). These extensions are made based on a common code base, re-using basic data structures to represent states, store them, and perform common operations such as delay, intersection, or computing successor states.

CORA is based on linearly priced timed automata [11]. The model extends timed automata with a special cost variable whose rate is specified for every state. The algorithm uses guiding to solve minimum cost reachability problems.

TRON is a testing tool suited for black-box conformance testing [43, 32] of timed systems. It is mainly targeted for embedded software commonly found in various controllers. Testing is done online in the sense that that tests are derived, executed, and checked while maintaining the connection to the system in real-time.

COVER is a tool for creating test suites from UPPAAL models with coverage specified by coverage observers a.k.a. observer automata.

TIGA is an extension for solving reachability and safety problems on timed game automata. Its algorithm [16] is a symbolic extension of the on-the-fly algorithm suggested by Shann et al [40] for linear-time model-checking of finite-state systems. It is used for controller synthesis [31], it has application to testing [19], and it has been extended to synthesis under partial observability [15].

PORT is a version targeted to component-based modelling and verification. Its interface is developed as an Eclipse plug-in. The tool supports graphical modelling of internal component behaviour as timed automata and hierarchical composition of components. It is able to exploit the structure of such systems and apply partial order reduction techniques successfully [26].

PRO is an extension for timed automata with probabilities [5, 33]. The model is extended with branching nodes that allow the user to specify weights for every outgoing edge. The engine can then compute probability bounds to reach specified states. It is work-in-progress.

TIMES is a tool-set for modelling, schedulability analysis, synthesis of (optimal) schedules and executable code. Its modelling language is timed automata extended with tasks. It models systems that can be described as a set of tasks that are triggered periodically or sporadically by time or external events. The release pattern is given by a timed automaton and the tool performs schedulability analysis on it. TIMES works by encoding the problem into timed automata and it uses the UPPAAL engine for the checks. It translates back the answer in terms of Gantt chart to visualise schedules. There are other tools that are using UPPAAL as a back-end verification engine, e.g., REX [22].

In this paper we focus on the "core" tool UPPAAL and present our experience in developing and maintaining it for the last decade. In particular, we present the backbone architecture that has allowed us to expand the tool on different variants of timed automata. The following sections give an overview of the tool architecture, our experience in the process of building the tool, and future development directions.

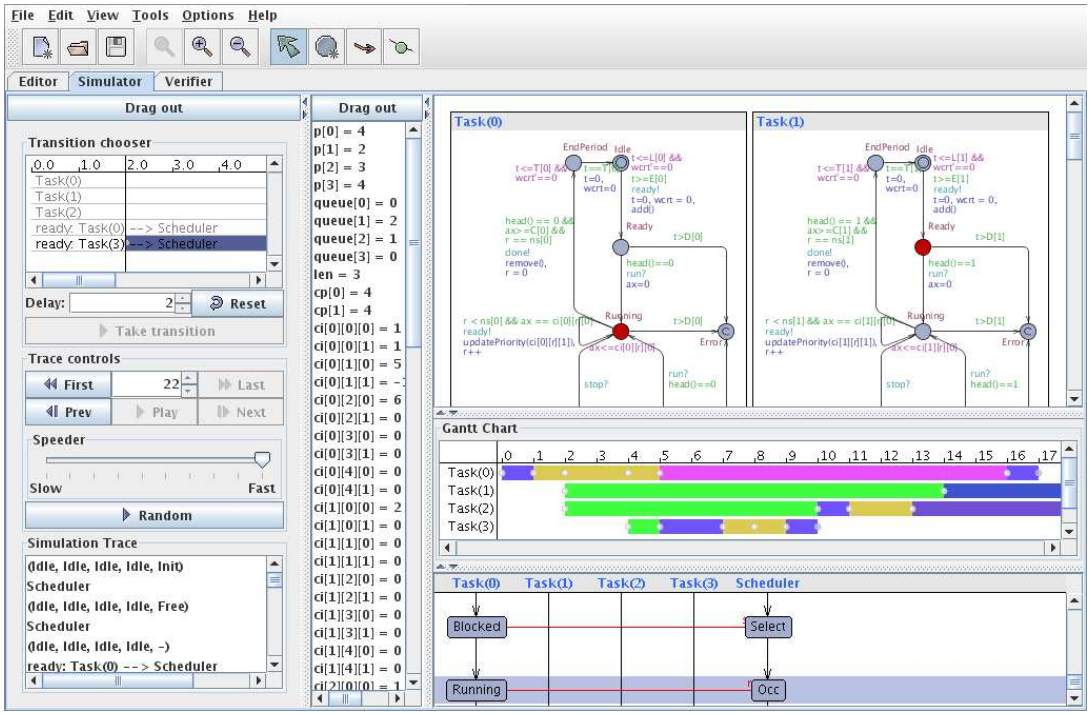

Figure 1. View of the "concrete" simulator of TIGA.

OVERVIEW OF THE TOOL ARCHITECTURE

Client-server architecture. The tool has two main components: a graphical user interface written in Java and a model-checker engine written in C++. The interface runs almost effortlessly on different platforms and we can exploit the rich functionalities available for programming interfaces inherent to the libraries coming with the Java programming language. The C++ language gives us both advanced object oriented programming and performance. These two components form a basic client server architecture with the graphical interface (client) communicating with the model checker (server) via a local pipe[†] or the network[‡]. This separation of concerns makes UPPAAL easier to port and maintain on different platforms.

The graphical interface has three "tabs" that correspond to the main tasks a user needs to do: to edit a model in the editor, to simulate it in the simulator, and submit verification queries

[†]A common inter-process communication mechanism.
[‡]The verification can be done on a remote server, which is a rarely exploited feature.

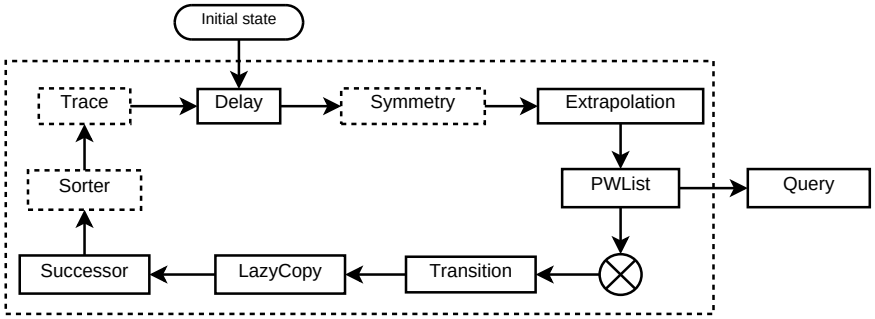

Figure 2. Pipeline architecture for the reachability filter.

to the model-checker. Additionally, the user may come back to the simulator to visualise a trace generated by the verification. Figure 1 gives a view of the simulator of the tool. The different variants of the tool have specialised interfaces and the figure shows the simulator used in TIGA. On the left is the *command* part where the user can select transitions, go back in the trace, play randomly, or navigate through the current trace (history of states). The list in the middle shows the values of the variables and clocks. The timed automata are shown on the right and below them a Gantt chart and a message sequence chart. The simulator of UPPAAL lacks the Gantt chart and has other similar components, although instead of navigating with *concrete* clock valuations, the user sees *symbolic* states. The point here is to stress reuse of components across different tools. It is important to amortise development costs over time on different specialisations of the tool without having to rewrite everything from scratch. This is obvious but the insidious consequence is that it is often difficult in practice to publish on these new additions. This is due to the lack of dedicated conferences where tool developments can be reported on.

Pipeline architecture. The model-checker itself (the *engine*) is designed around a pipeline architecture [7] where each block or *filter* processes states and sends them to the next stage as shown in Fig. 2. The figure shows the configuration for the *reachability* filter. Other algorithms such as the *liveness* and the *leadsto* checkers have their own filters built on the same principle. In this example, the initial state is pushed to the reachability filter in its delay component to start the exploration. Then it runs its main loop that takes states from our (unified passed and) waiting list structure, explores them, and puts the successors in that structure. This structure (also called the *PWList*) implements one (coloured) state set with states marked waiting and passed. It is unified in the sense that we have one structure instead of the traditional waiting and passed lists that need two lookups in hash tables per loop iteration of the reachability algorithm. Only states coloured as waiting are explored and inclusion check between symbolic states is done against all states. The main chain for the exploration is *Transition* (which transitions can be taken) - *Successor* (execution of the transitions) - *Delay* (let time pass)

- *Extrapolation* (apply an appropriate extrapolation to ensure finite exploration) - *PWList* (inclusion check and mark the state to be explored) - *Query* (evaluate the formula if the state was not included).

In fact we inserted a *LazyCopy* filter to reduce copies of states between the transition and successor filters. This filter really copies states only when necessary, e.g., computing one successor only does not require a copy and two successors require one copy only. It acts as a one place buffer. When priorities are used in the model, this filter is swapped by another filter that is going to buffer transitions and sort them by priority, without changing the rest of the pipeline. Some filters are optional, such as *Sorter* that can sort transitions, *Trace* that is used to store traces, or *Symmetry* that is projecting the states to a representant of its equivalence class (orbit) when symmetry is used in the model. In addition, different kinds of extrapolations can be used depending on the model, which results in different kinds of instances for the *Extrapolation* filter. We note that it is simpler to have the logic (in terms of if statements) to instantiate the right type of a component once and use the generic design to connect the components and use them transparently. The reader understands that the combination of these features gives rise to a lot of configurations. The point here is to keep orthogonal features separated.

The overall pipeline architecture allows us to reason about the algorithm in terms of blocks that we can change if we need another semantics. Implementing another checker, e.g. a timed game solver, is relatively easy and consists in adding components that will do the backward propagation, changing the first filter to either explore forward or backward, add a post-processing filter to detect what is winning or losing in the game after *Extrapolation*, and changing the graph representation. The new pipeline still has the same structure and follows the same design. To change the semantics of the game, e.g. to implement simulation checking [14], we change *Transition* that implements the transition relation and *Delay* to allow turn-based delays.

There are two important points that this architecture illustrates: object-oriented programming and reuse of components. The filters are in practice abstract classes so these components are managed at a high level. Second, we can reuse these filters for different pipeline configurations, i.e., for different checkers. We note that the architecture is also fit for functional languages.

Additional components. In addition to these components, UPPAAL contains a virtual machine to execute the compiled byte-code of our C-like input language supporting user defined functions and types. This allows the user to write complex and compact models while still limiting the state-space explosion – complexity can be concentrated in functions to avoid using intermediate states.

We currently distribute some open source components, such as the parser and the difference bound matrix (DBM) library. The DBM library has a Ruby binding, which allows for quick prototyping. The parser understands the XML format we use in UPPAAL, which allows other researchers to use the same format. The DBM library handles DBMs and federations (unions of DBMs) used to represent symbolic states. The DBM library supports a wide range of operations including subtractions and merging of DBMs.

TOOL BUILDING PROCESS

Tools are not prototypes. It is relatively easy to produce prototypes as proofs of concept of some theory or algorithm to strengthen a paper but it is notoriously more difficult to develop a tool that is going to survive the test of time. Unfortunately, prototypes are more common in practice. Building and maintaining tools takes a lot of time and is generally given less academic credits compared to more theoretical work, which explains the limited number of maintained tools. In the domain of formal methods, tools are crucial and they also serve as a dissemination means for theoretical results. Tools do have a positive impact through the case-studies they allow other users to do, often in collaboration, which is important to amortise the development cost (in terms of time) and earn publications (otherwise we perish)[§].

Who develops? The first question for developing tools is who is going to do it? Most of the time it is done by master or Ph.D. students, which makes sense economically since professors cannot afford writing C++ code. However, when temporary developers who have their own agenda (own thesis to write) work on the tools, there is the obvious issue that someone needs to take over otherwise the tool will disappear. In addition, temporary developers do not have a long term vision and are interested (rightfully so) in their own thesis. Over the years, changing teams without a common interest or focus means that the code will degrade if there is no control. What happened for us was that there were some Ph.D. students who stayed in the team for a long time, long enough to lay down a solid architecture and durable design. As a first rule of survival, *one should have a solid design* and encourage people to stick to it even if they do not like it. At some point in time old design decisions will not make sense any more but this is a different issue.

Code size. When the code grows (see Fig. 3) it is increasingly difficult for new people to use the code so it becomes important to have some permanent staff to take care of it and revise it so it can offer a limited and more useful interface. This is a considerable effort that is essential to the survival of the tool. In the past we had a few such revisions: the *pipeline* architecture, the *virtual machine*, and handling of *federations* in the model-checker. The size and complexity of the code has now become a barrier for new internal people but it is also a serious problem for external collaboration. We need a new revision to update the interfaces of the different components and add more abstraction to the code. For long term development, it is important to have some permanent staff to take care of such revisions and keep a long term vision. However, it is a trade-off between academic and development work.

Code aging. Curiously code ages. This is due to the fact that developers forget old code and new methods or libraries appear over the years, which makes the code become older or deprecated. In addition, progress in compilers also means that special efforts in the past to make some algorithms efficient are now obsolete, e.g., we can commonly address iteratively elements

[§]The well-known motto *publish or perish* emphasised by a system holding that name is a testament of the tool development dilemma in academia.

in matrices by expressions like $dbm[i * dim + j]$ with confidence that the compiler will *not* use relatively expensive multiplications for element accesses. To counter code aging, documentation is vital. Our experience has been to "document" the code using `doxygen` formatted comments. There is no real documentation apart from these comments, although some efforts have been made to describe overall design decisions and the communication protocol. We have crash courses to inform new programmers, which is a limited solution. As for the comments, they are extensive and they keep the memory of former developers. It is a weakness in the development process to lack stand-alone *white papers* that give technical details on the code but this has not been our priority.

Life cycles. The tool has gone over different life cycles over the years. A life cycle can be defined by major changes in architecture that are needed to accommodate new developments. It happens when old designs become too obsolete for new additions that were not foreseen in the past. The first cycle was with the original `ATG` graph editor[¶] and an early custom simulator. The second cycle introduced an integrated graphical editor, the client-server architecture still in use today, and an improved engine. The third cycle is the current one with a modular pipeline architecture. This pipeline architecture is probably the determinant factor for keeping additions of new features without breaking the tool. In terms of features it is interesting to note that early developments efforts were focused on performance improvements and then later on interface and language features. The later developments of the tool introduced new algorithms to handle different problems rather than improvements of the current algorithms.

During a cycle, the development is incremental, following the current design and making changes until the amount of desired new features and algorithms conflicts too much with the design. At this point there is a major effort to redesign (or re-factor) the code. The current architecture has lived up to its expectations for the approximately 8 years, during which we could re-use existing components and create new ones that we could litteraly plug together. However, the plethora of new variants of the tool hides current internal issues with the architecture and now is the time for a major update.

Distributed development. We use a centralised version management system (`CVS` and later subversion), which allows distributed teams to work on the same code. This is common for distributed projects. A given checkout of the repository contains all variants of the tool but each of them is located in its own separated module. Developers are responsible for few modules (their own) and modify other modules occasionally only. The key here is to have *responsibility* for the different parts for maintenance. In addition, we have the simple rules *committed code must compile* and *any distributed code must pass the regression test*. Since breaking these rules produces heated reactions, they tend to be observed. The goal here is to keep discipline.

Testing. For a tool in the field of formal methods we would expect to apply formal verification techniques to it to ensure its correctness. Let's say research is not there yet. The code base has currently 200+ KLoC in C++ that implement algorithms that are themselves notoriously

[¶]This is an editor tool used by UPPAAL from 1995 to 1999.

difficult to prove. There are tools we have used to assist us, such as `gcov`, `purify`, and `valgrind`. However, what we routinely do is to test. We use regression testing on a battery of known examples and results. When a bug is discovered, we insert that new example in the test suite and make sure new versions pass the new tests. This is an automatic process handled by a script.

Bug management. Another well known tool we are using is the bug management system *bugzilla*. Bugs are not only program errors but also requested features. They are sorted by priority that developers can set. Errors usually come with examples to reproduce them. They are added to the regression tests when the errors are corrected. Sometimes a change in the code triggers a new error that was not present in the past. We use binary search on the revision number (in our subversion repository) to find which revision introduced the changes that triggered that error. This is a simple and very effective technique.

Cross-platform. An integral part of the development process is to take care of cross-platform development. Early on we decided to stick with one compiler, `gcc/g++`. We can use the same code and change a few headers only and compile for Windows, Linux, and MacOS. By doing so we can also take advantage of some useful gnu extensions. We dropped support for SunOS due to absence of users and also machines installed with SunOS. All three supported platforms are actively used with an increase for MacOS in recent years. What we do to manage this is to keep third party libraries at a minimum. Currently we need `libxml2` compiled for all platforms and we use `boost` headers only. The rest is generated code by tools like `bison` and `flex`. Compilation is done under Linux for Linux and Windows, MacOS binaries are compiled on a Mac. We foresee problems in the future when supporting multi-threads since we will have to use additional libraries such as `Win32-pthread` to support POSIX threads (to begin with, the library needs to be patched for Win64).

Community. Finally, to survive, a tool needs its community. We have a discussion forum[‖] that our user community uses to ask or answer questions and maintain an active discussion on the tool. In fact, this helps us tremendously because we cannot handle all newcomers to the tool individually and we are grateful to users who help each other. The community also provides us with new problems and case-studies, which in turn instill progress in algorithms and theory.

CHALLENGES

The first challenge is to manage complexity and size of the project. Implementing advanced algorithms is tricky, specially when it is in a formal tool that is used for verification. As shown in Fig. 3 the code (in kilo lines of code of C/C++) has been steadily growing. This growth comes from new variants and algorithms that are added to the repository. The count

[‖] http://tech.groups.yahoo.com/group/uppaal/

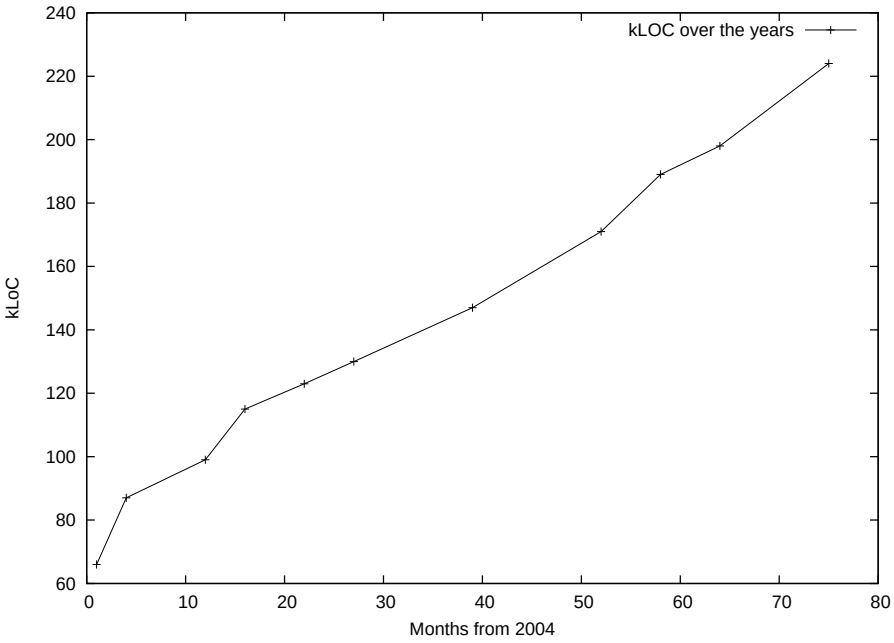

Figure 3. Evolution of the code base.

includes all code (used or not) for all variants of UPPAAL for the model-checker engine only. The graphical interface adds 40+ kLoC in Java.

The second challenge is to keep improving performance and features of the tool despite growing algorithm complexity. Table I shows the evolution of the performance of the tool. Experiments have been performed on a Pentium D 2.80GHz with 1GB RAM. We use `memtime` that measures time and polls memory (not reliable below 0.1s). Entries marked '-' denote verifications that were stopped after 2h or 900M. The models are available on *www.uppaal.org* under *Examples/benchmarks*. Apart from the performance improvements, recent versions support user-defined functions and symmetry reduction. These features are not used in the experiments but they would further improve performance.

The third challenge is to cope with new extensions of the tool to explore different theoretical paths. The current architecture has been pushed to implement the different known flavours of UPPAAL but also to extend every checker. Recent extensions to UPPAAL include priorities and stop-watches. TIGA was recently extended with a simulation checker. It is being extended with a new timed interface checker. Although the overall pipeline architecture accommodates these extensions, we have reached the limit of some "implementation details". These are: 1) there can be only one global system, 2) long wished features, such as clock constraints on receiving edges of broadcast synchronisations, are now needed, 3) the engine is designed for

Table I. Evolution of performance in terms of time (seconds) and memory consumption (MB).

Version	CSMA5	CSMA7	CSMA12	Fischer5	Fischer7	Fischer12	HDDI7	HDDI12
3.0.39	*8.4s*	-	-	*4.2s*	-	-	*36.3s*	-
	7.2MB	-	-	10.6MB	-	-	20.1MB	-
3.2.12	*0.3s*	*417s*	-	*1.6s*	-	-	*7.2s*	-
	3.8MB	145MB	-	6.8MB	-	-	11.9MB	-
3.3.25	*0.2s*	*198s*	-	*1.1s*	-	-	*3.2s*	-
	3.4MB	113MB	-	6MB	-	-	8.4MB	-
3.4.6	*<0.1s*	*40.7s*	-	*0.3s*	*4706s*	-	*0.1s*	*5.3s*
	3.1MB	34.5MB	-	4.9MB	267MB	-	1.6MB	14.1MB
4.0.11	*<0.1s*	*0.2s*	*33.8s*	*<0.1s*	*0.4s*	*418s*	*<0.1s*	*0.4s*
	1.6MB	38MB	115MB	1.6MB	38.1MB	300MB	1.6MB	38MB
4.1.2	*<0.1s*	*0.2s*	*41.9s*	*<0.1s*	*0.3s*	*341s*	*0.05s*	*0.2s*
	1.6MB	21.6MB	99MB	21MB	21.6MB	248MB	1.6MB	22.9MB

32-bit architectures, 4) there is no multi-core support, 5) there is only one kind of symbolic state, and the list goes on. These are obstacles for doing compositional model-checking where we would need to handle several systems and combine results. In addition, it is difficult to adapt the engine to different kinds of systems without changing core structures such as the states. Currently, when compiling CORA, one C macro is changed to swap to a different type of DBM supporting costs. That works because we made sure the commonly needed interface was exactly the same. This is a very limited solution.

Another challenge is to use modern technology to its full potential. Updating to 64-bit is mainly technical. Taking real advantage of 64-bit is challenging. Modern compilers have the ability to *vectorize* code** but this is still limited to simple algorithms and not to critical $O(n^3)$ algorithms that we have. Going for multi-core support (multi-threaded UPPAAL) is more difficult. There have been experiments in the past in this direction and we know that the current architecture could be adapted by having one thread per pipeline copy. This fits memory locality but we also know that it did not work so well because blocking data-structures (access protected by mutex) were major bottlenecks. It is crucial to have non-blocking structures such as [42] if we want to use multi-cores efficiently, even though this is a temporary solution that will last at most 10 years††. In addition, we want to make the components extendable more easily in particular to allow more people to work on UPPAAL without having to know what most of the code is doing. The bottom line is that there are research opportunities but not all issues are research related.

**This in essence allows the use of SIMD instructions (single instruction, multiple data) on streams of data.
††Shared memory architectures do not scale and message passing based architectures will take over.

FUTURE

We have shown in this paper the main challenges that we have faced in building UPPAAL over the years along with our own solutions. The conclusion is to get the synergy theory–implementation–case-studies that in turn provides with *publications*. There is no bullet-proof solution and we consider ourselves to have been lucky to have started at the right time and got such a good response from the community to get this synergy.

UPPAAL has already spawned one company, UP4ALL[‡‡], that sells a version of the tool for commercial uses. Another market we intend to target is testing. Research tools really have a future if they can be applied and used outside academia, as witnessed by Lustre/SCADE. However, their future as free academic tool is uncertain as discussed in this paper in relation with the dilemmas. The situation is that tool paper tracks exist and show the interest in academic tool development but they are often on the side of main conferences and they usually accept short papers with short talks. This could be improved to stimulate tool development in the community.

To continue the development on the academic path, we are exploring different domains as the different flavours of UPPAAL show. That also means that a new life-cycle with another architectural revision is now needed to cope with more extensions of UPPAAL. That will enable us to let other researchers experiment with the internals of UPPAAL and still maintain our core engine.

ACKNOWLEDGEMENTS

It is important to remember that UPPAAL is the result of the cumulative efforts of many collaborators. Among them we would like to thank early pioneers Johan Bengtsson and Fredrik Larsen, former contributor of the graphical interface Carsten Weise, and active contributor and maintainer Marius Mikučionis (UPPAAL and TRON). We also thank contributors of different extensions of UPPAAL, among them Didier Lime (TIGA), John Håkansson (PORT), Anders Hessel (COVER), Leonid Mokrushin (TIMES), Jakob Illum (CORA), Arild Haugstad (PRO). Last but not least we thank our supporting user community.

REFERENCES

1. I. AlAttili, F. Houben, G. Igna, S. Michels, F. Zhu, and F.W. Vaandrager. Adaptive scheduling of data paths using uppaal tiga. In S. Andova et. al., editor, *Proceedings of the First Workshop on Quantitative Formal Methods: Theory and Applications (QFM'09)*, number 13 in Electronic Proceedings in Theoretical Computer Science, pages 1–12. arXiv:0912.1897v1, 2009.
2. Rajeev Alur and David L. Dill. Automata for Modeling Real-Time Systems. In *Proc. of ICALP*, volume 443 of *LNCS*, pages 322–335, 1990.
3. Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times: A tool for modelling and implementation of embedded systems. In J.-P. Katoen and P. Stevens, editors, *Proc. of*

[‡‡]To contact UP4ALL email *sales@uppaal.com*.

*the 8*th *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 2280 in Lecture Notes in Computer Science, pages 460–464. Springer–Verlag, 2002.
4. Yoann Arnaud, Jean-Louis Boimond, José E.R. Cury, Jean Jacques Loiseau, and Claude Martinez. Using uppaal for the secure and optimal control of agv fleets. In *7th Workshop on Advanced Control and Diagnosis ACD 2009*, 2009. http://hal.archives-ouvertes.fr/hal-00463480.
5. Danièle Beauquier. On probabilistic timed automata. *Theoretical Computer Science*, 292:65–84, 2003.
6. Gerd Behrmann, Agnes Cougnard, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. UPPAAL-TIGA: Time for playing games! In *CAV'07*, number 4590 in LNCS, pages 121–125. Springer, 2007.
7. Gerd Behrmann, Alexandre David, Kim G. Larsen, and Wang Yi. Unification & Sharing in Timed Automata Verification. In *SPIN Workshop 03*, volume 2648 of *LNCS*, pages 225–229, 2003.
8. Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim G. Larsen, Paul Pettersson, and Judi Romijn. Efficient Guiding Towards Cost-Optimality in UPPAAL. In T. Margaria and W. Yi, editors, *Proc. of the 7th Int. Conf. on TACAS*, number 2031 in LNCS, pages 174–188. Springer-Verlag, 2001.
9. Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim Guldstrand Larsen, Paul Pettersson, Judi Romijn, and Frits W. Vaandrager. Minimum-cost reachability for priced timed automata. In *HSCC*, pages 147–161, 2001.
10. Gerd Behrmann, Kim G. Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Efficient Timed Reachability Analysis Using Clock Difference Diagrams. In *Proc. of the 12th Int. Conf. on CAV*, volume 1633 of *LNCS*, pages 341–353. Springer, 1999.
11. Gerd Behrmann, Kim Guldstrand Larsen, and Jacob Illum Rasmussen. Priced timed automata: Algorithms and applications. In *FMCO*, pages 162–182, 2004.
12. Johan Bengtsson, W.O. David Griffioen, Kåre J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Automated verification of an audio-control protocol using UPPAAL. *Journal of Logic and Algebraic Programming*, 52–53:163–181, 2002.
13. Johan Bengtsson and Wang Yi. *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, chapter Timed Automata: Semantics, Algorithms and Tools, pages 87–124. Springer-Verlag, 2003.
14. Peter Bulychev, Thomas Chatain, Alexandre David, and Kim G. Larsen. Efficient on-the-fly Algorithm for Checking Alternating Timed Simulation. In *FORMATS'09*, number 5813 in LNCS, pages 73–87. Springer, 2009.
15. F. Cassez, A. David, K. G. Larsen, D. Lime, and J.-F. Raskin. Timed control with observation based and stuttering invariant strategies. In *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis*, volume 4762 of *LNCS*, pages 192–206. Springer, 2007.
16. Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR'05*, volume 3653 of *LNCS*, pages 66–80. Springer–Verlag, August 2005.
17. P.R. D'Argenio, J.-P. Katoen, T.C. Ruys, and J. Tretmans. The bounded retransmission protocol must be on time! In E. Brinksma, editor, *Proceedings of the Third Workshop on Tools and Algorithms for the Construction and Analysis of Systems,* Enschede, The Netherlands, volume 1217 of *Lecture Notes in Computer Science*, pages 416–431. Springer, 1997.
18. Alexandre David, John Håkansson, Kim G. Larsen, and Paul Pettersson. Model Checking Timed Automata with Priorities using DBM Subtraction. In *Proc. of the 4th Int. Conf. on FORMATS*, volume 4202 of *LNCS*, pages 128–142, 2006.
19. Alexandre David, Kim G. Larsen, Shuhao Li, and Brian Nielsen. Cooperative testing of uncontrollable real-time systems. In *4th workshop of Model-Based Testing (MBT'08)*, 2008.
20. Alexandre David, M. Oliver Möller, and Wang Yi. Formal Verification of UML Statecharts with Real-Time Extensions. In Ralf-Detlef Kutsche and Herbert Weber, editors, *FASE, 5th Int. Conf. 2002*, volume 2306 of *LNCS*, pages 218–232. Springer, 2002.
21. Alexandre David and Wang Yi. Modelling and Analysis of a Commercial Field Bus Protocol. In *Proc. of Euromicro-RTS'00*, pages 165–172. IEEE Computer Society, 2000.
22. AnneMarie Ericsson, Mikael Berndtsson, Paul Pettersson, and Lena Pettersson. Verification of an industrial rule-based manufacturing system using rex. In *1st International Workshop on Complex Event Processing for Future Internet*. ISSN 1613-0073 ceur-ws.org, September 2008.
23. Elena Fersman, Paul Pettersson, and Wang Yi. Timed automata with asynchronous processes: Schedulability and decidability. In J.-P. Katoen and P. Stevens, editors, *Proc. of the 8*th *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 2280 in Lecture Notes in Computer Science, pages 67–82. Springer–Verlag, 2002.
24. Biniam Gebremichael, Frits Vaandrager, and Miaomiao Zhang. Analysis of the zeroconf protocol using uppaal. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages

242–251. ACM, 2006.
25. John Håkansson, Jan Carlson, Aurelien Monot, Paul Pettersson, and Davor Slutej. Component-Based Design and Analysis of Embedded Systems with UPPAAL PORT. In Sung Deok Cha, Jin-Young Choi, Moonzoo Kim, Insup Lee, and Mahesh Viswanathan, editors, *ATVA*, volume 5311 of *LNCS*, pages 252–257. Springer, 2008.
26. John Håkansson and Paul Pettersson. Partial Order Reduction for Verification of Real-Time Components. In *Proc. of the 5th Int. Conf. on FORMATS*, LNCS, pages 211–226. Springer-Verlag, 2007.
27. R. Hamberg and F.W. Vaandrager. Using model checkers in an introductory course on operating systems. *Operating Systems Review*, 42(6):101–111, 2008.
28. F. Heidarian, J. Schmaltz, and F.W. Vaandrager. Analysis of a clock synchronization protocol for wireless sensor networks. In A. Cavalcanti and D. Dams, editors, *Proceedings of FM 2009: Formal Methods*, volume 5850 of *LNCS*, pages 516–531. Springer-Verlag, 2009.
29. Anders Hessel and Paul Pettersson. A Test Case Generation Algorithm for Real-Time Systems. In Hans-Dieter Ehrich and Klaus-Dieter Schewe, editors, *Proc. of the Fourth ICQS*, pages 268–273. IEEE Computer Society, 2004.
30. Anders Hessel and Paul Pettersson. Cover — A Test-Case Generation Tool for Timed Systems. In Alexandre Petrenko, Margus Veanes, Jan Tretmans, and Wolfgang Grieskamp, editors, *Testing of Software and Communicating Systems: Work-in-Progress and Position Papers, Tool Demonstrations, and Tutorial Abstracts of TestCom/FATES*, pages 31–34. Institute of Cybernetics at Tallinn University of Technology, 2007.
31. Jan Jakob Jessen, Jacob Illum Rasmussen, Kim G. Larsen, and Alexandre David. Guided controller synthesis for climate controller using UPPAAL-TIGA. In *Proceedings of the 19th International Conference on Formal Modeling and Analysis of Timed Systems*, number 4763 in LNCS, pages 227–240. Springer, 2007.
32. Moez Krichen and Stavros Tripakis. *Model Checking Software*, volume 2989 of *LNCS*, chapter Black-Box Conformance Testing for Real-Time Systems, pages 109–126. Springer-Verlag, 2004.
33. Marta Kwiatkowska, Gethin Norman, Jeremy Sproston, and Fuzhi Wang. Symbolic model checking for probabilistic timed automata. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, volume 3253 of *LNCS*, pages 293–308. Springer-Verlag, 2004.
34. K.G. Larsen, M. Mikučionis, and B. Nielsen. Online Testing of Real-time Systems Using UPPAAL. In *FATES'04*, LNCS, pages 79–94, Linz, Austria, September 2004.
35. Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. Testing real-time embedded software using uppaal-tron: an industrial case study. In *the 5th ACM international conference on Embedded software*, pages 299 – 306. ACM Press New York, NY, USA, September 18–22 2005.
36. Kim G. Larsen, Paul Pettersson, and Wang Yi. Model-Checking for Real-Time Systems. In *Proc. of Fundamentals of Computation Theory*, number 965 in LNCS, pages 62–88, August 1995.
37. 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 18th IEEE RTSS*, pages 14–24. IEEE Computer Society Press, December 1997.
38. Fredrik Larsson, Paul Pettersson, and Wang Yi. On Memory-Block Traversal Problems in Model Checking Timed Systems. In Susanne Graf and Michael Schwartzbach, editors, *Proc. of the 6th Conf. on TACAS*, number 1785 in LNCS, pages 127–141. Springer-Verlag, 2000.
39. Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal Design and Analysis of a Gear-Box Controller. In *Proc. of the 4th Workshop on TACAS*, number 1384 in LNCS, pages 281–297. Springer-Verlag, March 1998.
40. X. Liu and S. Smolka. Simple Linear-Time Algorithm for Minimal Fixed Points. In *Proc. 26^{th} Conf. on Automata, Languages and Programming (ICALP'98)*, volume 1443 of *LNCS*, pages 53–66. Springer, 1998.
41. 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. IEEE Computer Society Press, December 1997.
42. Chien-Hua Shann, Ting-Lu Huang, and Cheng Chen. A practical nonblocking queue algorithm using compare-and-swap. In *Seventh International Conference on Parallel and Distributed Systems*, pages 470–475, 2000.
43. Jan Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, 1992.
44. 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 FORTE'94*, pages 223–238. Chapman & Hall, 1994.