Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
Electronic Notes in Theoretical Computer Science 128 (2005) 3–18
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

On-the-fly Workload Prediction and Redistribution in the Distributed Timed Model Checker Zeus[1]

V. Braberman[a,5], A. Olivero[b,2,6], F. Schapachnik[a,4,3]

[a] *Departamento de Computación, FCEyN, Universidad de Buenos Aires, Buenos Aires, Argentina*

[b] *Centro de Estudios Avanzados, FIyCE, Universidad Argentina de la Empresa, Buenos Aires, Argentina*

Abstract

In this work we present the on-the-fly workload prediction and redistribution techniques used in Zeus [12,13], a Distributed Model Checker that evolves from the tool Kronos [14].
After reviewing why it is so hard to have good speedups in distributed timed model checking, we present the methods used to get promising results when verifying reachability properties over timed automata [3].

Keywords: Distributed Timed Model Checking, Zeus, Kronos, Prediction, Redistribution, Reconfiguration, Load-Balance, Timed Automata, Reachability.

1 Introduction

An obstacle for a wide adoption of model checking technology is scalability. Verifying even medium-size designs can quickly exhaust memory or processing capacity of rather powerful computers. In recent years, there has been an

[1] Research supported by UBACyT 2003 X020 and BID OC/AR PICT 11738 grants.
[2] Partially supported by UADE project TSI04B.
[3] Partially supported by an IDS 2003 grant.
[4] Email: fschapac@dc.uba.ar
[5] Email: vbraber@dc.uba.ar
[6] Email: aolivero@uade.edu.ar

doi:10.1016/j.entcs.2004.10.015

increasing interest in the use of Distributed Computing as a way to increase the size of the models tools can deal with [24,6,29].

Much successful work has been done to distribute *untimed* model checkers [29,24,4,17,26,7,18,21,8,19,23]. However, except for some work on a distributed version of UPPAAL [6,5] and our own [12,13], not much has been previously done about parallelizing or distributing *timed* model checkers. Because of the inherently different data structures involved, the timed and untimed cases lead to inherently different parallelization strategies and challenges. Timed model checkers are usually based on Difference Bound Matrices (DBM) [16]. Though they are symbolic representations of state space, they conceptually differ from Binary Decision Diagrams which are the basic data structure for a large class of untimed model checking tools and their distributed counterparts. Thus, most of the strategies and ideas for distributing BDD-based model checking algorithms (e.g., slicing large BDDs [21]) seem not to be directly applicable to the timed setting.

The rest of the article is organized as follows: in section 2 we recall ZEUS architecture. The problems found on both asynchronous and synchronous versions are described in section 3. Then we present the techniques developed to overcome such issues. Section 5 showcases some case-studies where these methods improved verification times. Finally, we outline future research paths.

2 Recalling Zeus

ZEUS is a distributed model checker that evolves from KRONOS [14]. As such, it works over models expressed in terms of timed automata, by means of a backwards propagation calculus. It was first presented in [12] and [13], being its main features a rigorous correctness proof for its distributed algorithm, and a high degree of flexibility due to its software architecture approach. Also, it runs on any network of Unix-like workstations, not requiring a Beowulf or similar clustering environment.

It was built with "*design for change*" in mind. That was a key requirement since there are many degrees of freedom that are very hard to set in advance when building a distributed tool. This flexibility allowed us to easily construct both synchronous and asynchronous versions with a variety of communication strategies, and also implement various distribution methods.

Although a deep knowledge of timed automata and TCTL logic is not required to understand this article, we provide a quick summary as appendix A. One important detail is that, like KRONOS, ZEUS currently uses DBMs (called *zones*) to represent convex sets, and *regions*, which are unions of zones (not to be confused with the region graph presented in [3]). For every discrete state

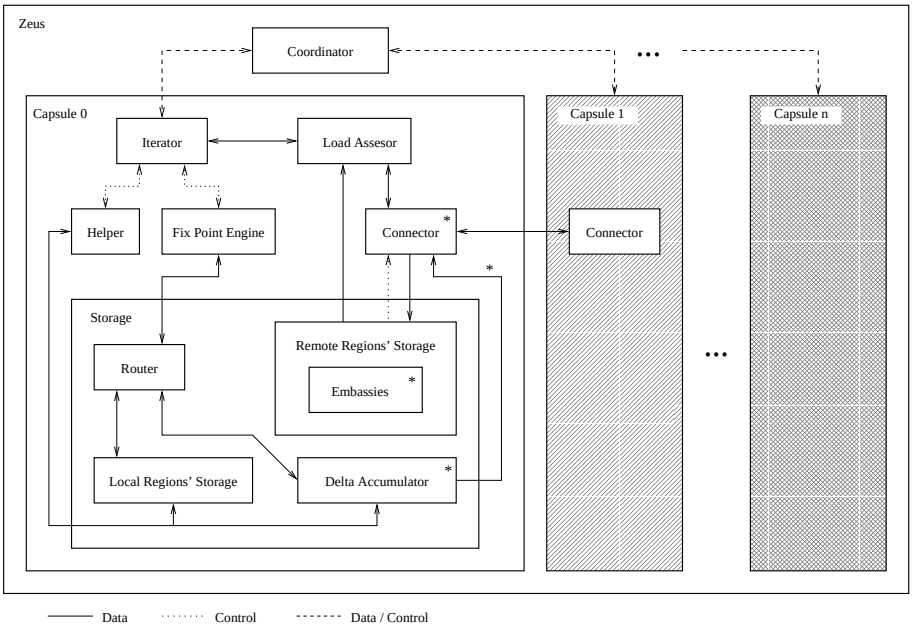


Fig. 1. ZEUS General Conceptual Architectural View (∗ means repetition).

s (also called *location*), we will define I_s as its invariant and $s \xrightarrow{e} s'$ will mean that s' is a discrete successor of s, being $\rho_{s \xrightarrow{e} s'}$ and $g_{s \xrightarrow{e} s'}$ the transition's reset clocks and guard, respectively. Guards are normalized to their intersection with I_s. We will use $pred_e$ and $pred_\tau$ to refer to the discrete and timed predecessor operators. Both of them are applied iteratively to generate R_s, the set of reachable states belonging to location s. ΔR_s stands for the difference between the values of R_s at the current and previous iteration.

2.1 ZEUS *Conceptual Architecture*

To handle the required flexibility, an architecture as shown in fig. 1 was built. Its apparent complexity obeys to a key design decision: separation of concerns. The architecture was built as a testbed for experimenting with a family of design decisions concerning synchronization, region exchange and load balance. Thus, we aimed at a loosely coupled solution where issues like fixed point calculation and the communication schema remain as independent as possible. This strategy led us to the identification of some aspects that were mapped into different components.

Each processor working in a distributed ZEUS computation is called a *capsule*. We also use *capsule* to refer to the processes running inside the processors and their associated data structures and components.

The basic components are a *Fixpoint Engine* that performs the computation (reading and writing regions into the *local regions' storage* and *remote regions' storage*), and *connectors* which are used to communicate with other *capsules*. The *remote regions' storage* is made of *embassies*, which receive the remote regions from the *connectors*.

There is also a global *coordinator*, which starts the process, partitions the control graph, distributes the workload and establishes whether global fix point has been reached or not.

Some more details can be found in appendix B. It should be noted that this architecture corresponds to the asynchronous version, the synchronous one being simpler.

3 The Counterintuitive Wrong Way – Why it's Hard

In this section we will present, in a somehow historical order, the process that led to the current version of Zeus. We will address the motives for the decisions taken, the problems that appeared and what our response to them was.

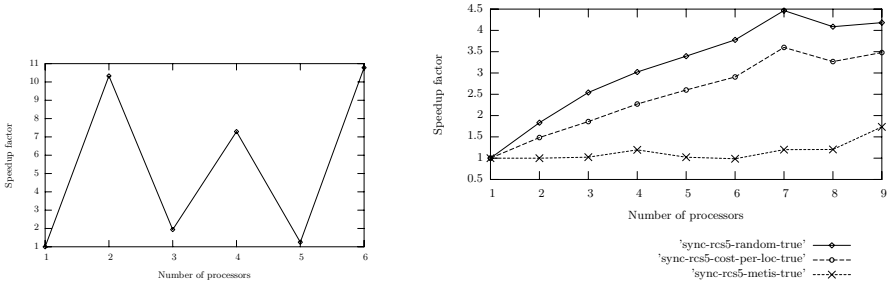
3.1 Asynchronous Version

When building a distributed algorithm the intuition is that an asynchronous one would make better use of computational resources, because waiting times tend to be minimized. With this idea in mind, the first version of Zeus was asynchronous. While it was able to handle cases unfit for the monoprocessor version [7], an unexpected result was observed: in some models adding processors increased verification times. For instance, the *RCS5* [8] case study behaves this way, as can be seen in fig. 2(a). The *Speedup factor* for n processors is defined based on time to completition as $time_1/time_n$.

Digging into this strange behavior showed a very interesting issue. The asynchronicity changed the relative order of operations, and although at the end the result is semantically the same set of regions, its syntactic representation is different, involving more zones. It can be viewed as a *fragmentation* of the data structure. Some authors reported similar findings when breadth-first traversal is not followed to calculate fix points [6,11].

[7] Even showing super-scalar speedups. These happen because of a distribution that gives little work to the processor with the initial location: almost all of its work is to check for inclusion of the timed initial state into the reached set.

[8] *RCS5* is an example inspired in the well-known Railroad Crossing System [1], with 5 trains. In this model, the *error* state can be made reachable or unreachable just by modifying a few constants.



(a) Asynchronous version.

(b) Synchronous version with different initial distribution methods.

Fig. 2. Speedups for the reachable case of *RCS5*.

The most computational expensive operation of the fix point calculus is region differentiation, which takes $O(n\ m)$ being n and m the number of zones in each region (this bound disregards the number of clocks, which is fixed for a given system). So, that different growing order led to an increase in computation time, and thus wall-clock time.

The number of zones grew that much that this problem alone outweighed all the performance features such as communication piggybacking, delaying messaging, minimal cut distribution, dead-time utilization, etc.

To gain some insight in the fragmentation phenomenon, the tool was instrumented to accumulate the cost of each differentiation (i.e., the number of operations actually required). For the unreachable instance of the *RCS5* case-study, in a monoprocessor version the cost was approx. 697 millions while the calculus of the 2-processor asynchronous version produced approximately 1.3 billion. Other examples presented similar behavior.

Considering this problem, two research lines forked: on one hand, a search for a new representation structure more fit for distribution started. On the other, we decided to try a synchronous version, which was relatively easy to construct due to Zeus modular architecture.

3.2 Synchronous Version

This new version was aimed at reproducing the relative order of operations that the monoprocessor would do, knowing that the synchronicity introduced could waste some potential parallel operations.

A sketch of its main cycle is shown as figure 3(a). Every *capsule* notifies the *coordinator* the end of each iteration. When all the *capsules* have finished, the *coordinator* broadcasts the clearance to begin the exchange phase, where *capsules* exchange new regions as needed. Once done, they resume the

Table 1
%waste for example *RCS5* (reachable case).

# cap	Random		Location-cost Weighted		Min Cut	
	total time (secs.)	*%waste*	total time (secs.)	*%waste*	total time (secs.)	*%waste*
1	110.40	0.00	110.40	0.00	110.40	0.00
2	60.22	0.97	74.12	33.95	109.98	49.90
3	43.44	3.00	59.16	39.07	107.34	66.53
4	36.52	8.24	48.44	33.80	91.93	69.99
5	32.52	14.59	42.30	36.14	107.44	79.84
6	29.23	16.43	37.85	33.20	111.44	83.09
7	24.72	8.76	30.56	22.66	91.56	82.49
8	27.01	31.74	33.66	44.47	91.26	84.90
9	26.40	39.86	31.61	51.64	63.37	79.42

iteration process without *coordinator* intervention.

1: **while** ¬*GlobalEnd* **do**
2: #changes = fixpoint_engine.iterate()
3: notify_phase_end(ITERATION, #changes)
4: wait_for_coord_clearance(EXCHANGE)
5: exchange_regions()
6: notify_phase_end(EXCHANGE)
7: **end while**

(a) Zeus Synchronous Iteration Algorithm.

1: **for all** $s \in local\_discrete\_states()$ **do**
2: $PredE = \bigcup_{s \xrightarrow{e} t} pred_e(\Delta R_t, g_{s \xrightarrow{e} t}, \rho_{s \xrightarrow{e} t})$
3: $PredT = pred_\tau(PredE)$
4: $\Delta R_s = PredT - R_s$
5: $R_s = R_s \cup PredT$
6: **end for**

(b) *Fixpoint Engine iterate()* Algorithm.

Fig. 3. Zeus Algorithms

Although times improved –as can be seen in figure 2(b)–, still some cases showed a decrease in verification times while adding processors. Our first attempt was to use different partitioning strategies, including METIS-based [22] minimum cut, random distribution and an oracle-provided partition that balanced the actual weights of a previous exploratory run. There were still dips in the curve.

Researching that problem, we decided to measure the wasted time in each iteration, computed as the accumulation of idle times (line 4 of figure 3(a)) over the complete verification. More precisely, being *#cap* the number of capsules and *#it* the number of iterations, we get:

$$\%waste = \frac{WastedTime}{(TotalTime - I/O\ time) \times \#cap} \times 100, \text{ where } WastedTime = \sum_{i=1}^{\#it} \sum_{c=1}^{\#cap} Waiting_{i,c}$$

This metric, along with the previously mentioned results, pinpointed that the workload varied in each iteration, so an initially fair distribution could become extremely heavy on some processors and light on others in a matter of a few iterations. Table 1 shows the correlation between bad speedups in figure 2(b) and the total wasted time.

Very unbalanced iterations, specially the latter ones where the number of zones is larger, could mean that in practice only one processor is doing the work while the others are idle, undermining the time previously gained and producing a worst result in the overall process.

4 Going Dynamic

The obvious conclusion was going to a dynamic version – dynamic in the sense that locations and their associated computational work, could migrate from a processor to another in each iteration, so the workload could be evenly distributed during the whole verification.

Two problems had to be solved to migrate locations: the workload for the next iteration had to be predicted, and then the data had to be exchanged without breaking the distributed invariant (see appendix B for an overview).

4.1 Workload Prediction

As was previously said, region differentiation (line 4 of figure 3(b)) is the most expensive operation performed by the *Fixpoint Engine*, so in order to predict the workload, the first thing to do is to obtain a stricter bound on its computational complexity.

To get that bound, the code was instrumented to report the function's elapsed time as well as the actual cost for different candidate formulae, bearing in mind that a good formula should be able to predict run times. Early enough it became obvious that the $O(n\ m)$ bound was very thick, and did not distinguish between pairs of operands with very different run times.

A closer inspection of the code revealed a $\Theta(f(dim(r_2), d(r_1, r_2)))$ bound, where:

- r_1 and r_2 are the two regions being differenced,
- *dim* represents the dimension of the parameter region in terms of accumulated number of clock differences over the set of zones (that is, the number of zones times the square of the number of clocks),
- f encodes the details of the implementation (which are not presented because they are meaningless to the general result and very dependent on the particular coding used), and
- d represents the number of zones that the actual difference will have. Including such a value in the computational complexity of a function is a license we are taking, because it might not fit perfectly on the definition, but it certainly influences the run time.

Two tricky parts arose: first, f includes the number of `malloc()` [9] invocations made, as well as integer comparisons. Clearly both kinds of operations do not take the same amount of clock-time, as the first involves requesting a service to the underlying OS, probably at the expense of some contex switches.

[9] `malloc()` is the C-language function call used to request dynamic memory.

We decided to establish a factor k, so we can express that a `malloc()` call takes as much as k integer operations. This factor is determined at the beginning of the verification, because it is platform dependent.

Readers familiar with low-level system coding might have noticed that the real cost of a `malloc()` depends on the fragmentation of the memory at the time of the call. This is indeed the case on most platforms, so k is no more that an estimate. As can be seen further on, other uncertainties also had to be filled with estimates.

The most complicated part was guessing what the value of d will be for a particular pair of parameters. This problem was two-fold, because although the second parameter (R_s) for the next iteration is known at the end of the current one (line 5 of fig. 3(b)), the other ($PredT$) is yet to be determined and expensive enough to compute in advance.

To obtain reasonable values for the two uncertainties ($PredT$ and $d(PredT, R_s)$), the following mechanism was developed, based on the intuition that many calculi are usually repeated and some that are not, have only slightly different inputs: each time the real operation was performed, the size of its parameters was rounded to its most significant digit, and was stored along with the size of the result (for example, if two regions of size 23456 and 337 were subtracted, obtaining a region of size 128, the tuple $\langle 20000, 300, 128\rangle$) would have been stored). If a new operation has the same rounded parameters size, the old values are overwritten, on the assumption that recent values would predict better future results.

So when the real value needs to be estimated, its rounded estimated parameters sizes are looked up in the collected information. If no match is found, a default value is used. This is the worst case of the estimation. It should be noted that this is kind of a simplistic approach to best-fit matching, and more sophisticated algorithms could be used, but it has the advantage of having a small overhead in terms of both space and time, and featured very good results. Experimenting with other alternatives is in our *to-do list* for future research.

The technique showed a surprising predicting power, featuring a 80% to 90% hit ratio after the first few iterations –which don't usually take considerable time– on most examples. Also the combined workload predicting method showed a very good match ratio when compared to the actual metric. It should be noted that this match ratio does not mean that the predicted and actual values are the same. It means that their relative percentual difference is very close, along all the locations of a same iteration. This is what is needed in order to evenly distribute workload.

The prediction process takes place at the *coordinator*, which feeds from

statistical data sent by all of the *capsules* at the end of each iteration. The volume of this data is $O(l)$ where l is the number of locations at each *capsule*, but being packed integers, they are really not very significant compared to a typical region exchange. Also, it should be remembered that in timed model checking the size of the control graph is not usually that big, even for large models.

4.2 Dynamic Location Redistribution

Once the relative weights of each location have been predicted, a new distribution has to be established. On one hand minimum workload difference is desired, but on the other a large number of location migrations could take a long time by themselves. The current version uses the ParMeTiS library [28] which is based on heuristic methods to handle efficiently graph repartitioning problems. ParMeTiS tries not only to balance weight while minimizing number of movements, but also tries to reach minimum cut. Although that would seem like a good idea because it minimizes communication, this is not such a pressing issue in a synchronous environment over a fast local area network. We are still looking for a method that will not aim at minimum cut, so an even better balance could be achieved [10] . It should be pointed out that a number of good methods exist for rebalancing in the untimed scenario (see [21,25] among others). Unfortunately they are not directly applicable as they usually don't have to deal with locations having different (unsplitable) weights.

The *coordinator* runs the redistribution algorithm –which is neglectably fast–, computes the difference with the previous partition and broadcasts it to the *capsules*. They receive it along with their clearance to go into the region exchange phase, so prior to going into the proper exchange, the first part of the migration begins.

For each location migrated from c_i to c_j (note that always $c_i \neq c_j$), there are three possible cases, from the perspective of the c_k *capsule*:

- Both the previous and current *capsules* are remote ($c_i \neq c_k \neq c_j$).
- The location used to be remote, but now is local ($c_i \neq c_k = c_j$).
- The location used to be local, but now is remote ($c_i = c_k \neq c_j$).

In all of the cases *connectors* need to be reconfigured, so they know which locations they are actually serving. Because the topology and boundaries changed, there could be *connectors* to some other *capsules* no longer needed, and new ones might have to be created. In the last two cases, besides recon-

[10] Unluckily ParMETIS does not handle disconnected graphs, so the obvious trick of disregarding the edges can't be used.

figuring *connectors*, new *embassies* might need to be instantiated, or old ones destroyed.

More importantly, in the last two cases the content of the *local regions' storage* for the affected location needs to be exported by a *capsule* and imported by another. The corresponding locations are marked, and said regions are exchanged during the region exchange phase. Once finished, the migrations list is processed again, so clean up activities can take place, including the removal of former local regions.

The results of the process must be that each *capsule* is reconfigured as if the current partition has never changed since the beginning. Although conceptually simple (fig. 4), implementation details make it a quite involved process.

```
1: while ¬GlobalEnd do
2:    #changes = fixpoint_engine.iterate()
3:    notify_phase_end(ITERATION, #changes)
4:    location_migrations = wait_for_coord_clearance(EXCHANGE)
5:    process_migrations(location_migrations)
6:    exchange_regions()
7:    cleanup(location_migrations)
8:    notify_phase_end(EXCHANGE)
9: end while
```

Fig. 4. Zeus Synchronous Iteration Algorithm with Dynamic Migrations.

5 Preliminary Experimental Results

Finding case studies for this particular work was a hard task, because in "toy" size examples, like *RCS5*, which verify completely in a few minutes in a monoprocessor, the migration time itself outweighs the benefits of the gained parallelism. On the other hand, real-size examples usually require many processors to finish, so even exhibiting speedups because of the redistribution, it is hard to know how good they are.

The experiments were run on a cluster consisting of 6 Linux 2.4 workstations connected through a 100 Mbps Ethernet network, each one running on a 1.8 GHz AMD Athlon XP processor, with 256 MB of RAM.

Due to space restrictions, we selected two representative examples:

- *RCS6*: This is a 6-train version of the previously mentioned Railroad Crossing System case-study, with an unreachable *error* state. It was processed by ObsSlice [10], which reduces timed automata preserving the validity of TCTL formulae. The system has 5288 locations, 36072 transitions and 9 clocks.

Table 2
%waste for examples *RCS6* and *MinePump*.

# cap	*RCS6*				*MinePump*			
	Static		Dynamic		Static		Dynamic	
	total time (secs.)	*%waste*	total time (secs.)	*%waste*	total time (secs.)	*%waste*	total time (secs.)	*%waste*
1	Ran Out of Mem.	N/A	ROM	N/A	ROM	N/A	ROM	N/A
2	779.97	10.45	762.83	7.15	2508.31	42.33	2364.03	37.67
3	531.91	10.48	537.73	11.04	2437.15	60.47	1865.56	49.46
4	524.93	31.31	427.32	14.46	2421.92	70.17	1848.12	59.43
5	381.20	24.08	347.21	15.03	2255.17	74.32	1744.81	66.49
6	669.78	64.14	296.08	14.33	2472.72	80.29	1514.58	69.89

- *MinePump*: This model is based on a design for a fault-detection mechanism for a distributed mine-drainage controller as found in [9]. It also has an unreachable *error* state. The system –reduced by ObsSlice and Optikron [11] – has 4452 locations, 21932 transitions and 6 clocks.

Table 2 (first half) shows that although the static run of *RCS6* did exhibit increased running times while adding processors in some steps, the dynamic version didn't. Also, the 6-processor run shows a very stepped decreased of the wasted time.

It can be seen in the second half of table 2 (corresponding to *MinePump*) that although times did improve considerably with the dynamic version, there is still a lot of wasted parallelism. The explanation lies not in the predictions, which were quite accurate, but in the repartitionings computed by ParMeTiS, which could be much better if they didn't consider the edge cut of the discrete graph as an optimization target.

6 Conclusions and Future Work

We introduced the on-the-fly workload prediction and redistribution techniques used in Zeus.

After reviewing why it is so hard to have good speedups in distributed timed model checking, both in the synchronous and asynchronous setting, we presented the algorithms used to get promising results. Although the workload prediction seems to be quite good, more work needs to be done to find a better repartitioning method.

It is worth mentioning that the same principles could be applied to predict memory utilization. Balancing by size would allow verification to succeed with less processors, which is also a goal of the distributed effort.

Future work includes improving the redistribution mechanism, fine tuning the workload prediction algorithm, as well as looking into new data structures. In this last sense, we are working in a CDD-like data structure [30] that doesn't seem to suffer from the fragmentation problem, at least with the same

[11] Optikron [15] is a tool that performs redundant and inactive clock optimizations.

intensity. It is very tempting because it might allow us to use the full potential of the asynchronous version.

Also, we plan to extend these concepts to an on-the-fly model checking algorithm, where the complete discrete graph is not necessary known *a priori*. Among the preliminary ideas is predicting new composite locations, so they can be evenly distributed before they are found.

References

[1] Alur, R., C. Courcoubetis, D. Dill, N. Halbwachs and H. Wong-Toi, *An implementation of three algorithms for timing verification based on automata emptiness*, in: *Proceedings of the 13th IEEE Real-time Systems Symposium*, Phoenix, Arizona, 1992, pp. 157–166.

[2] Alur, R., C. Courcoubetis and D. L. Dill, *Model-checking in dense real-time*, Information and Computation **104** (1993), pp. 2–34.

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

[4] Barnat, J., L. Brim and J. Stríbrná, *Distributed LTL model-checking in SPIN*, in: *SPIN*, 2001, pp. 200–216.

[5] Behrmann, G., *A Performance Study of Distributed Timed Automata Reachability Analysis*, in: *Workshop on Parallel and Distributed Model Checking, afiliated to CONCUR 2002 (*13^{th} *International Conference on Concurrency Theory)*, ENTCS **68** (2002).

[6] Behrmann, G., T. Hune and F. W. Vaandrager, *Distributing timed model checking - how the search order matters*, in: *Computer Aided Verification*, LNCS **1855** (2000), pp. 216–231.

[7] Ben-David, S., T. Heyman, O. Grumberg and A. Schuster, *Scalable distributed on-the-fly symbolic model checking*, in: *Formal Methods in Computer-Aided Design*, 2000, pp. 390–404.

[8] Bollig, B., M. Leucker and M. Weber, *Parallel model checking for the alternation free μ-calculus*, in: 7^{th} *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '01)*, LNCS **2031**, 2001, pp. 543–558.

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

[10] Braberman, V., D. Garbervetsky and A. Olivero, *Improving the verification of timed systems using influence information*, in: *TACAS 2002, held as part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002*, LNCS **2280** (2002), pp. 21–36.

[11] Braberman, V., C. López Pombo and A. Olivero, *On improving backwards verification for timed automata*, in: *TPTS 2002, satellite event for the Joint European Conference on Theory and Practice of Software, ETAPS 2002*, ENTCS **65** (2002).

[12] Braberman, V., A. Olivero and F. Schapachnik, *Zeus: A distributed timed model checker based on kronos*, in: *Workshop on Parallel and Distributed Model Checking, affiliated to CONCUR 2002 (*13^{th} *International Conference on Concurrency Theory)*, ENTCS **68** (2002).

[13] Braberman, V., A. Olivero and F. Schapachnik, *Issues in Distributed Model-Checking of Timed Automata: building* ZEUS, to appear in International Journal of Software Tools for Technology Transfer (2004).

[14] Daws, C., A. Olivero, S. Tripakis and S. Yovine, The Tool KRONOS, in: *Proceedings of Hybrid Systems III*, LNCS **1066** (1996), pp. 208–219.

[15] Daws, C. and S. Yovine, *Reducing the number of clock variables of timed automata*, Proceedings IEEE Real-Time Systems Symposium (RTSS '96) (1996), pp. 73–81.

[16] Dill, D. L., *Timing assumptions and verification of finite-state concurrent systems.*, in: *International Workshop of Automatic Verification Methods for Finite State Systems*, LNCS **407** (1990), pp. 197–212.

[17] Garavel, H., R. Mateescu and I. M. Smarandache, *Parallel state space construction for model-checking*, in: M. B. Dwyer, editor, *Proc. of the 8^{th} International SPIN Workshop*, Toronto, Canada, 2001, pp. 217–234.

[18] Grumberg, O., T. Heyman and A. Schuster, *Distributed symbolic model checking for μ-calculus*, in: *Computer Aided Verification*, 2001, pp. 350–362.

[19] Heljanko, K., V. Khomenko and M. Koutny, *Parallelisation of the petri net unfolding algorithm*, in: *Tools and Algorithms for Construction and Analysis of Systems (TACAS '02)*, 2002, pp. 371–385.

[20] Henzinger, T. A., X. Nicollin, J. Sifakis and S. Yovine, *Symbolic Model Checking for Real-Time Systems*, Information and Computation **111** (1994), pp. 193–244.

[21] Heyman, T., D. Geist, O. Grumberg and A. Schuster, *Achieving scalability in parallel reachability analysis of very large circuits*, Formal Methods in System Design **21** (2002), pp. 317–338.

[22] Karypis, G. and V. Kumar, *Parallel multilevel k-way partitioning scheme for irregular graphs*, Technical report, University of Minnesota, Department of Computer Science / US Army HPC Research Center. Minneapolis, USA. (1998).

[23] Krcal, P., *Distributed explicit bounded LTL model checking*, in: L. Brim and O. Grumberg, editors, *Electronic Notes in Theoretical Computer Science*, ENTCS **89** (2003).

[24] Lerda, F. and R. Sisto, *Distributed-memory model checking with SPIN*, in: *Proc. of the 5^{th} International SPIN Workshop*, LNCS **1680** (1999).

[25] Nicol, D. and G. Ciardo, *Automated parallelization of discrete state-space generation*, Journal of Parallel and Distributed Computing **47** (1997), pp. 153–167.

[26] Ranjan, R., J. Sanghavi, R. Brayton and A. Sangiovanni-Vincentelli, *Binary decision diagrams on network of workstations*, in: *International Conference on Computer Design*, 1996, pp. 358–364.

[27] Schapachnik, F., "Distributed and Parallel Verification of Real-Time Systems," Degree thesis, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires (2002). URL: http://www.cvi.com.ar/zeus/tesis_schapachnik.ps.gz

[28] Schloegel, K., G. Karypis and V. Kumar, *A unified algorithm for load-balancing adaptive scientific simulations*, Technical report, University of Minnesota, Department of Computer Science / US Army HPC Research Center. Minneapolis, USA. (2000).

[29] Stern, U. and D. L. Dill, *Parallelizing the Murφ verifier*, in: *Computer Aided Verification*, LNCS **1254** (1997), pp. 256–278.

[30] Wang, F., *Efficient verification of timed automata with BDD-like data-structures*, in: L. Zuck, P. Attie, A. Cortesi and S. Mukhopadhyay, editors, *Verification, Model Checking, and Abstract Interpretation: 4^{th} International Conference, VMCAI*, LNCS **2575** (2003), pp. 189–205.

Appendix A. Foundations: Timed Automata and TCTL

The tool's theoretical foundations are found in the theory of timed automata [3] and the timed temporal logic TCTL [2]. A timed automaton models the behavior of a single process or component of the system while requirements are expressed by means of TCTL formulae.

Timed automata are a formalism that incorporates positive real valued clocks to automata notation. Clocks record the time elapsed between events. All clocks are synchronized, that is, they all advance at the same pace. Each control location (automata node) has attached an invariant (a clock constraint). When a transition is taken, clocks are allowed to be reset to zero. Each transition has an associated guard (a predicate over the clocks that define its enabling condition), and a label to name the executed event.

At any time, the state is determined by the control location and the values of clocks which must satisfy the location invariant. The system can evolve in two different ways: either an enabled transition is taken, changing the control location and some clocks are reset while the others keep their values unaltered, or it may let some amount of time pass. In the last case, clocks increase according to the elapsed time while still satisfying the location invariant, and the system remains in the same location.

Complex systems can be built by the label-synchronized product of the automata representing each component.

Properties over models can be expressed in terms of TCTL formulae. In practice, most properties can be written in terms of reachability (sometimes adding a virtual observer automaton), that is whether a given set of states is reachable from the initial state by an execution of the model. Those target states are named by labeling locations with propositions.

Note that the existence of real-valued clocks generates an infinite state space (control locations plus clock valuations). Fortunately, this does not imply undecidability of many interesting problems such as reachability. To deal with infinite state manipulation, tools like KRONOS and ZEUS represent convex sets of clock valuations as conjunctions of inequalities involving one clock or the difference between two clocks (e.g., $1 \leq x \leq 5 \wedge x - y > 8$). A data structure called Difference Bound Matrices (DBM) [16] is typically used to manipulate such kind of information. Non-convex sets are represented as union of convex sets. ZEUS performs reachability queries as a backwards propagation of non-convex sets over the graph of control locations. This propagation is a fix point calculation that can be informally described as: starting with the set of target states and adding in each iteration every other state that can reach in a single step some state within the set. In our case, the final answer

is whether or not the initial states belong to the computed fix point (i.e., we want to know if the target states can be reached from the initial states, or not). When the target states are unreachable the complete state space needs to be explored, so this is usually considered a worst-case scenario.

Although currently only reachability is supported in ZEUS, full TCTL verification is a possible extension. For instance, KRONOS' is based on the previously explained algorithm [20].

Appendix B: Zeus Architecture

ZEUS architecture can be more easily understood centering at the *Fixpoint Engine*. This is the component that runs the fix point calculation, much as it does in KRONOS [12]. It reads and writes regions to the *storage* component. Each region belongs to a control location that might be assigned either to the current *capsule* or to some other. Because the *Fixpoint Engine* is unaware of such distribution, it makes no distinction between local and remote locations, except for the fact that it writes only over the local ones. Upon reception of a *read* request from the *Fixpoint Engine*, the *router* delivers regions from the *local regions' storage*, if their location is local, or from the *remote regions' storage*, if it is remote. In case of receiving a *write* request –which only happen for local regions– it stores them in the *local regions' storage* and in a *delta accumulator*. There is a *delta accumulator* for each local control location that should be eventually visible from a remote *capsule*.

Again, the *Fixpoint Engine* is not aware of the existence of the *delta accumulators*; it is the *router*'s job to guarantee that written local regions also get there whenever that information is known to be eventually processed by a neighbor capsule.

The rationale for the existence of *delta accumulators* has to do with technical concerns regarding regions' difference. Basically, they serve the purpose of storing information instead of having to recalculate it when needed. *Delta accumulators* are emptied when regions are sent to other *capsules*.

Within each *capsule* there is an *embassy* for each neighbor *capsule*. When requested for a region, they immediately answer back if they have some "new" regions to provide. For technical reasons special care is taken to exhibit the same information to every request belonging to the same *Fixpoint Engine* iteration..

As well as there are *delta accumulator* to benefit the sending phase of a regions' exchange, there are *embassies* to benefit the receiving phase. They are

[12] Interested readers can find the algorithm in [12].

contained inside the *remote regions' storage*. One can think that there is an *embassy* at the opposite end of every *delta accumulator*. It is also responsible of showing the same content to every request during an iteration.

A *capsule* A has a $connector_{A,B}$ iff there is at least one edge between a location in A and a location in B. *Connectors* handle network communication. That is, given two *capsules*, A and B, there might be many edges in the control graph going from locations assigned to A to locations assigned to B, call it k. Although A can have up to k *delta accumulators* and k *embassies*, it has only one *connector* handling bi-directional communication to B. The same happens at B.

The *helper* performs data representation compression (i.e., representing regions with a smaller number of zones if possible). Compression is applied in the local repository and the *delta accumulator*. This leads to a "semantically innocuous" dead time utilization that hopefully will make future computations and messaging lighter.

The *coordinator* starts the process, partitions the graph, distributes the workload and establishes whether global fix point has been reached or not. It also collects statistics. It receives information from the *iterators* on the *capsules* to make decisions.

A formal description of the architecture including state machines and transducers can be found in [27].