Electronic Notes in Theoretical Computer Science 68 No. 4 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume68.html 20 pages

Zeus: A Distributed Timed Model-Checker Based on Kronos

V. Braberman [a,1,4], A. Olivero [b,2,5], F. Schapachnik [a,3]

[a] *Computer Science Department, FCEyN, Universidad de Buenos Aires, Buenos Aires, Argentina*

[b] *Department of Information Technology, FIyCE, Universidad Argentina de la Empresa, Buenos Aires, Argentina*

Abstract

In this work we present Zeus, a Distributed Model-Checker that evolves from the tool Kronos [8] and that currently can handle backwards computation of TCTL-reachability properties [1] over timed-automata [2].

Zeus was developed following a software architecture centric approach. It introduces some interesting features such as *a priori* graph partitioning, a sophisticated machinery to reach optimum performance (communication piggybacking and delayed messaging) and dead-time utilization, where every processor uses time intervals of inactivity to perform auxiliary, time-consuming tasks that will later speed up the rest of the computation.

Although some good results have been obtained, early experiments pinpointed the difficulties of getting speedups using a parallel asynchronous version. We also propose some paths to overcome those obstacles.

1 Introduction

Research in Model-Checking is focused on increasing the size of the problems tools can deal with. The ultimate wave has been the use of Distributed-Computing, where a cluster of computers work together to solve the problem [16,4,20].

The usefulness of a distributed strategy is often measured by the "speedup" gained. Speedup with n processors is computed as $\frac{t_1}{t_n}$ where t_i is the time it

[1] Research supported by UBACyT grants X156 and X094.
[2] Research supported by UADE grant PSI.2.2002.
[3] Email: fschapac@dc.uba.ar
[4] Email: vbraber@dc.uba.ar
[5] Email: aolivero@uade.edu.ar

takes to finish the verification with i processors. The goal is usually to get linear speedups, although verifying cases where the sequential version exhausted its memory is also considered a success.

Much successful work has been done to distribute untimed model checkers [20,16,3,11,18,5,12,14], etc. However, except for the work of Behrmann, Hune and Vaandrager on a distributed version of UPPAAL [4], not much have been done about parallelizing or distributing Timed Model-Checkers.

In this work we present Zeus, a Distributed Model-Checker that evolves from the tool Kronos [8] and that currently can handle backwards computation of TCTL-reachability properties [1] over timed-automata [2].

The rest of the paper is organized as follows: in section 2 we describe the Kronos tools from which ours evolves. Section 3 presents the most interesting points of our Distributed Model Checker. Section 4 shows the performance on a couple of configurations for a version of the Train-Gate-Controller case study and section 5 presents the lessons learned, as well as paths to be taken in research to be.

2 The Kronos tool

The tool Kronos [8] formally checks whether a real-time system meets its requirements. It is founded on the theory of timed automata [2] and timed temporal logics, like TCTL [1].

Although being successfully used in real world applications [17,21,9] it fails to cope with the same state explosion problem that harvest most of the other tools. Main problems are storage exhaustion and inability to use multiprocessor equipment when available.

Even though a deep technical knowledge of Kronos formalisms and algorithms is not required to understand this paper, we will introduce the main concepts and will refer the interested reader to the ample field's literature (e.g., [2,8], etc.).

Timed automata is a real-time 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. When a transition is taken, clocks are allowed to be reset to zero. Transitions are associated with a guard which is a predicate over the clocks. The guard determines when a transition can be taken.

The inclusion of clocks generates an infinite state space (control locations plus clock valuations). Fortunately, this does not imply undecidability of many interesting problems such us reachability. To deal with infinite state manipulation, tools like Kronos 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) [10] is typically used to manipulate such kind information. Non-convex sets are represented as union of convex sets. Kronos performs

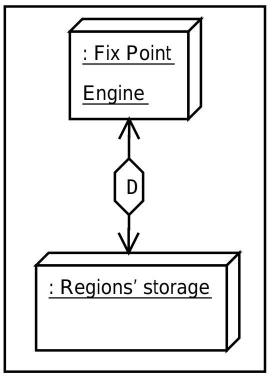


Fig. 1. Kronos architecture.

reachability queries as a backwards propagation of non-convex sets over the graph of control locations. This propagation is a fix point calculation of the appropriate precedence operators that starts from target states. The final answer is whether or not the initial states belong to the computed fix point.

Full TCTL verification is mainly based on the previously explained algorithm [13].

In Kronos jargon, each convex set is called a *zone*, and a union of convex sets is called a *region* (not to be confused with the region graph of [2]).

From a conceptual standpoint, its architecture can be seen as a fix-point engine that reads and writes its non-convex sets into a regions' storage component, as depicted in figure 1.

3 Distributing Kronos: Zeus

To obtain a Distributed Timed Model-Checker a number of issues must be dealt with. As *sine qua non* requirement, the resulting architecture must be provable correct. A sketch of the correctness proof can be found in section 3.1.

On the performance side, a delicate balance should be established. On one hand maximum distribution is desired. On the other, minimum communication between processors should be required. The rationale for requiring minimum communication should not be thought as network latency, as commonly referred in the literature, but as context switching. This is because the processor has to temporarily stop the profitable fix-point calculation to take care of the unavoidable message passing, often involving hardware interrupts and context switching at the operating system level.

While extending Kronos we preserve the explicit representation of control graph and a symbolic representation of clock values. While most work in the area uses an on-the-fly construction of the control graph and a hashing function for its nodes (locations) that tries to balance the load between processing nodes, we introduce an *a priori* control graph partitioning which in the first version is based on the tool METIS [15]. If m is the number of edges, METIS uses an $O(m)$ heuristic approach to deal with the *NP-complete* problem of finding the minimum cut.

Though an on-the-fly construction of control graph is more appealing, particularly when reachable nodes are a small subset of the control locations or when error locations are in fact reachable, we believe that working with the whole control graph may offer advantages over on-the-fly construction whenever feasible[6] :

- It establishes a suitable testbed to experiment different graph partitioning strategies. The current version considerably reduces the number of edges that cross processor boundaries, thus minimizing communication requirements and consequently context switching.
- It establishes a suitable framework for dynamic node redistribution when load balance is compromised due to huge differences on the number of symbolic regions that the processors are handling. Although the current version does not has the ability to migrate locations between processors, it is seen as a very promising research path (more about this on section 5).
- Extensions to full TCTL are easy to achieve following Kronos algorithms.
- Distribution partition and configuration reuse is possible whenever only time constraints of the model are changed.

Two other strategies were put in place to help reduce the message exchange between processing nodes. An important decision that can largely affect performance is the use of either a *push* schema, where nodes send each other regions that they will need, or a *polling* schema, where nodes explicitly request regions as they need them. Our first decision was to use a mixed strategy based on message piggybacking. Processing nodes basically behave as in the *polling* schema, but when a processor A needs a region located in another processor B, it will request all B's regions that are of interest to A –not just the one needed right now– and it will send all its regions that B needs. This mechanisms should reduce communication overhead, because fewer messages are exchanged.

To further reduce communication, a state machine determines if a processing nodes really needs to ask for regions. It has three possible modes:

"Low regions-demand, busy processing node" In this state the processor is busy with the fix-point calculation and can go along without requesting newer regions from another processing node, thus leaving the chance that a future request will bring a larger region, due to the monotonicity of the calculus.

"High regions-demand, idle processing node" In this state the processing node has reached a temporary fix-point, so it is idle and its requests are immediately delivered.

[6] I.e., when the graph of control locations is not huge compared to the involved DBMs' quantity.

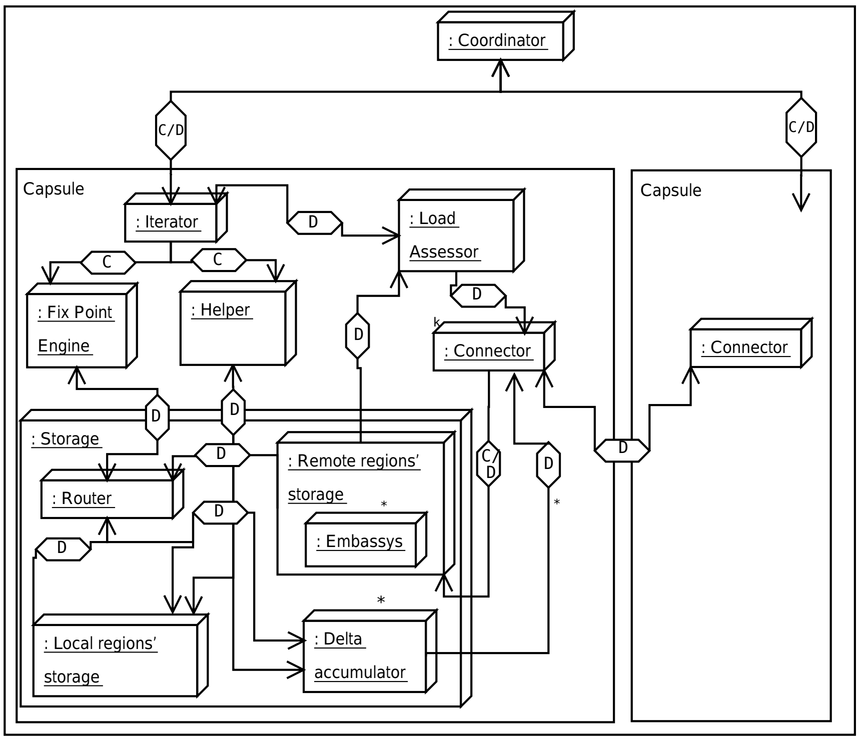


Fig. 2. ZEUS architecture.

"High regions-demand, busy processing node" This state serves the purpose of predicting the previous one. The processing node is not yet idle, but an heuristic[7] predicts that it will be soon, so its requests are send without delay.

To handle all this design considerations, an architecture as shown in figure 2 was built. Its apparent complexity obeys to a key design decision: separation of concerns. The architecture should serve as a testbed for experimenting with a family of design decisions concerning synchronization, region communication and load balance. Thus, we aimed at a loosely coupled solution where issues like fixed point calculation and the use of *polling* or *push* schema remains as independent as possible. This strategy lead us to the identification of some aspects that were mapped into different components.

From an architectural point of view, a *capsule* is a set of components and connectors associated to a processing node. Each partition is assigned to a processing node.

The *Fix-Point Engine* of a *capsule* is in charge of computing, for the associated partition, the set of states that can reach the target states. A key idea behind the architecture is to conceal from the *Fix-Point Engine* the fact that there is distribution. In order to achieve that, a *storage* component is refined into several pieces: a *router* component is in charge of hiding which

[7] The heuristic is the determination that the number of regions changed per iteration is decreasing below certain threshold.

capsule each location is assigned to. During fix-point computation if the *Fix-Point Engine* needs to read a region assigned to the same *capsule*, the *router* will direct the petition to the *local regions' storage* component. If the location resides in another *capsule*, its associated regions will be requested to the corresponding *embassy*.

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[8]. When no new information has arrived, they forward the request to the *connector*, asking it to send a network message to the corresponding *capsule*. When the new regions arrive they became visible to the *Fix-Point Engine* at the next 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, but are also responsible for deciding when petitions are actually sent out to the network, thus implementing the delayed messaging strategy. To make that decision, they query the *load assessor* component and ignore the petition if the *capsule* is in "Low regions-demand" state. In fact, request forwarding happens only when the *capsule* is in "High regions-demand" and there is no answer pending. In this case, piggybacking has to take place, so the *connector* attaches every unsent region that is of interest to the target *capsule* to the request (*delta accumulator* take a role in this, as explained later on). The receiving *connectors* immediately answer requests, regardless of their own *capsule* regions-demand.

The *delta accumulators* serve as a repository of all new regions that are produced by the *Fix-Point Engine* and that must be eventually be visible to other *capsules*. Again, the *Fix-Point 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. Obviously, the *load assessor* component is in charge of determining whether the *capsule* is idle or not and whether its regions-demand is high or low.

The *helper* performs data representation compression (i.e., representing regions with less number of DBMs if possible) whenever the *Fix-Point Engine* reaches a temporary local fix-point. Compression is applied in the local repository and the *delta accumulator*. This leads to a "semantically harmless" dead time utilization that hopefully will make future computations and messaging lighter.

It is the *iterator* responsibility to determine whether the control flow will be at the *Fix-Point Engine* component or at the *helper* component.

The *coordinator* is in charge of starting the verification, including partition management and distribution of work among *capsules*. During the model-checking phase, the *coordinator* sporadically receives from the *capsules*

[8] For technical reasons special care is taken to exhibit the same information to every request belonging to the same *Fix-Point Engine* iteration.

information about the existence of local fix-points and data to estimate the number of messages flowing through the network. Hence, the *coordinator* is the component that detects that fix-point or initial states have been reached.

For the interested reader, a more detailed description of the architecture is offered as appendix 5.

3.1 Sketch of a correctness proof.

To avoid a big step, instead of straightly proving the correctness of Zeus we have incrementally proved correctness of intermediate versions that resemble more to Kronos architecture (see figure 3). Two of them are of special interest: the first one is what we called "naïve distribution", which introduces distribution with very simple synchronous access to remote information, and the second consists in the introduction of sophisticated machinery like connectors, high/low regions-demand, piggybacking, etc. (see figure 3(e)). To prove them correct we follow a common strategy in proofs of distributed algorithm by splitting the demo into two parts: (a) the semialgorithm described converges to the minimum fix-point, and (b) the *coordinator* will eventually detect fix-point when it happens.

To address the first issue we resort to the concept of "asynchronous iterations", due to Cousot [7].

Definition 3.1 Asynchronous Iterations

Let $(P, \sqsubseteq)$ be a complete lattice, n a positive integer and $F : P^n \rightarrow P^n$ a monotonic operator.

Let $\langle c^\delta : \delta \in Ord \rangle$ be a sequence of elements of $\mathbb{N}_n = \{1, \ldots, n\}$ such that:

a) $(\forall \delta \in Ord)(\forall i \in \mathbb{N}_n)((\exists \alpha \geq \delta)(i = c^\alpha))$

Let $\langle \tau^\delta : \delta \in Ord \rangle$ be a sequence of elements of Ord^n such that:

b) $(\forall i \in \mathbb{N}_n)(\forall \delta \in Ord)(\tau_i^\delta < \delta)$

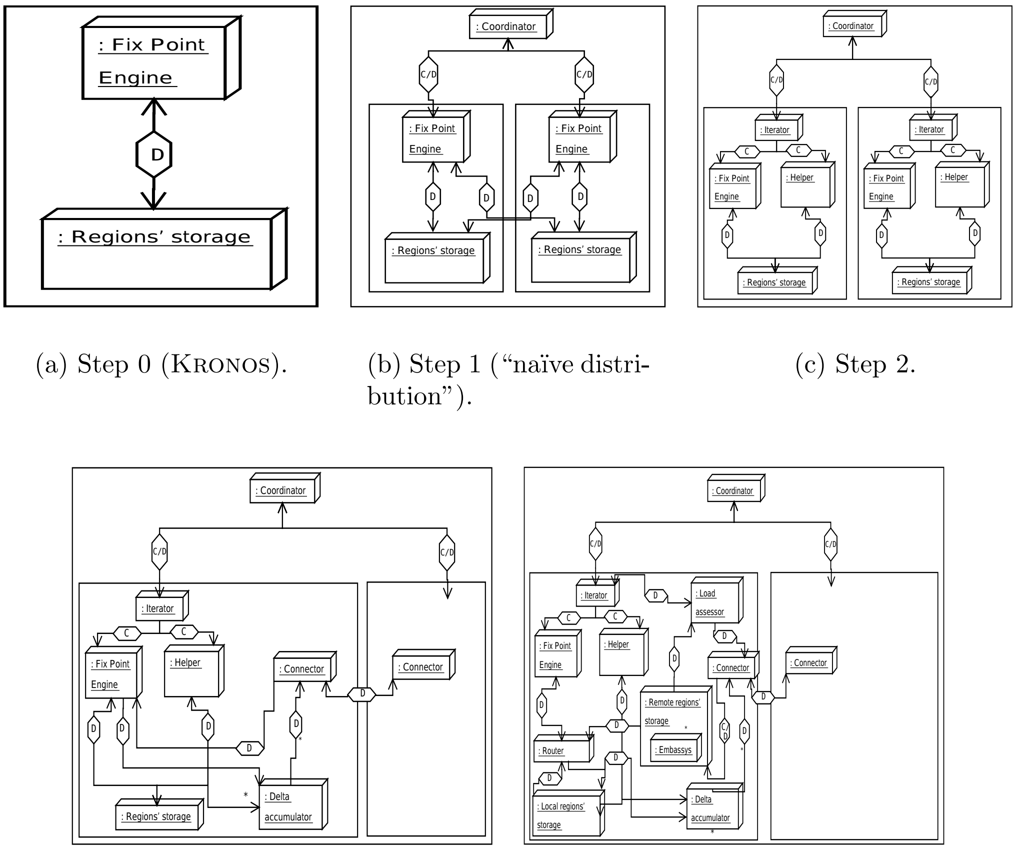
c) $(\forall \delta \in Ord)(\forall i \in \mathbb{N}_n)(\exists \beta \geq \delta)((\forall \alpha \geq \beta)(\delta \leq \tau_i^\alpha)))$

d) $(\forall \beta, \delta \in Ord)((\beta \text{ is a limit ordinal } \wedge \beta < \delta) \implies ((\forall i \in \mathbb{N}_n)(\beta \leq \tau_i^\delta)))$

An *Asynchronous Iteration* of the operator F and the sequences $\langle c^\delta : \delta \in Ord \rangle$ and $\langle \tau^\delta : \delta \in Ord \rangle$, starting from $D \in P^n$, is defined as follows:

$$\begin{aligned}
x^0 &= D \\
x_i^\delta &= x_i^{\delta-1} && \forall \text{ successor ordinal } \delta \wedge \forall i \neq c^\delta \\
x_i^\delta &= F_i(x_1^{\tau_1^\delta}, \ldots, x_n^{\tau_n^\delta}) && \forall \text{ successor ordinal } \delta \wedge \forall i = c^\delta \\
x_i^\delta &= \textstyle\bigsqcup_{\delta' \leq \delta} x_i^{\delta'} && \forall \text{ limit ordinal } \delta
\end{aligned}$$

In [7] it is proven that asynchronous iterations are stationary sequences and their limit is indeed the least fix point of operator F starting from D.



(a) Step 0 (Kronos). (b) Step 1 ("naïve distribution"). (c) Step 2.

(d) Step 3. (e) Step 4.

Fig. 3. Architecture evolution.

That is, asynchronous iterations provide a model of iterative computing that guarantees that if some hypothesis are met, then the minimum fix-point will be reached even if the computation does not follow the standard order of fix-point iterative calculus. In particular, during the computation described by this model, data from different past instants might be used. This is the kind of phenomenon that arises in distributed computation, where remote data used in a given calculus may have changed at the original site. Of course, some other hypothesis should hold, like fairness of computation (3.1.a). Among the non-trivial ones, roughly speaking, is the requirement that information produced in a given point in time at a given processing node will eventually be available to any other remote calculus that relies on it (3.1.c).

For instance, to prove that these requirements hold for Zeus, we showed that (a) local fix points are bound to happen, (b) consequently "high regions-demand" mode is entered, then (c) request will eventually be made, and thus (d) updated info will return as answers. Full proof of correctness can be found in [19].

3.2 Implementation notes

Zeus' is written in about 7000 lines of modular `ANSI C` code. It uses TCP/IP sockets for interprocess communication, wrapped in a communication library.

It relies on Kronos and METIS libraries for DBM computation and for location distribution, respectively.

It runs on most Unix platforms.

4 Preliminary experimental results

The example we take as our guiding case study is a version of the Train Gate Controller common in literature. The controller registers the status of trains and actuates over the gate accordingly. The property to assess (expressed as a reachability problem over an observer) is whether or not a train can access the railway crossing while the gate is up or has just gone down.

Table 1: Size of TGC case study.

#Automaton	#Locations	#Transitions	#Clocks
Each train	3	3	1
Gate	4	10	1
Controller (5 trains)	18	205	1
Controller (6 trains)	21	280	1
Observer	5	22	1
Composition (5 trains)	4764	27850	8
Composition (6 trains)	14388	90262	9

We defined tt_x as the total wall-clock time of a verification run with x processors, tc_x as the time it takes to copy the automata and related info to all the processors, and ti_x as the total idle time (over all the participant *capsules*). We make a distinction because the copying process is suboptimal and can be enhanced. The *Speedup* function S_x is defined as follows:

$$f_x = \frac{(tt_x - tc_x)}{t_1}$$

$$S_x = \frac{1}{f_x}$$

f_x represents the fraction of time the run takes to finish with x processors, compared to just one processor. As can be seen, we consider a linear speedup as ideal.

In our very first experiments, done on an Intel cluster composed by 10

Example	Environment	#procs	tt_x (sec)	tc_x (sec)	$f_x \times 100\%$	S_x
TGC5, reachable target state	Darwin	1	405	0	100%	1
		2	77	24	13.09%	7.64
		3	130	48	20.25%	4.94
	Speedy	1	248	0	100%	1
		2	25	1	9.68%	10.33
		3	129	1	51.61%	1.94
		4	37	3	13.71%	7.29
		5	202	3	80.24%	1.25
		6	28	5	9.27%	10.78
TGC5, unreachable target state	Darwin	1	889	0	100%	1
		2	5155	25	577.07%	0.17
		3	3625	49	402.25%	0.25
TGC6, reachable target state	Darwin	1	∞	N/A	N/A	N/A
		2	∞	N/A	N/A	N/A
		3	320	146	N/A	N/A
	Speedy	1	∞	N/A	N/A	N/A
		2	∞	N/A	N/A	N/A
		3	∞	N/A	N/A	N/A
		4	9199	74	100%	4
		5	1607	77	16.63%	6.01
		6	1659	103	16.91%	5.91

Table 2
Results for TGC.

nodes[9], and on a Silicon Graphics IP27 with four 270 Mhz processors[10] and 1 Gb RAM running IRIX64 6.5, we were indeed able to deal with bigger problems than the single processor version. Moreover, in many cases we also obtained important speedups (see table 2).

However, in several cases the performance dropped when we increased the number of processors. Concerned by these unsatisfactory results we decided to refocus the experimental analysis. The flexibility of Zeus allowed us to quickly code different communication strategies:

- The original *pseudo-polling* schema, where *capsules* request information when they have a "high-regions demand", and that features communication piggybacking.
- A *push* schema, where nodes send the regions they produce to others that

[9] All of them with 64 Mb of memory, Linux 2.2.15-4mdk kernel and 5 with Pentium III 733 Mhz processor and the other 5 with AMD Athlon 940 Mhz connected through an 100 Mbps Ethernet network. Only six of them were available. Due to different processor speeds on the cluster, faster processors where used first, seeking to avoid assigning a speedup to certain configuration when that would have been caused because of the introduction of a faster processor.
[10] Only three of them were available.

need them without delay. This model is similar to the one used in distributed on-the-fly reachability graph exploration [20].

- A *subscription* schema. This is a variation of the *pseudo-polling* schema that eliminates many unneeded messages. When two consecutive requests from a *capsule* fail to bring back information, a subscription between the source of the petition and target *capsule* takes place. This means that no new requests are issued, to avoid flooding the target. When the target generates new regions, it sends them to the subscribed requester, as in the *push* schema.

We also used several partition strategies to improve load balance:

- Minimal Cut: using METIS, as described earlier.
- TL-Weighted Minimal Cut: using METIS features to add weight information to locations. Locations with the property to be reached weight ten times the normal weight[11].
- OpL-Weighted Minimal Cut: using METIS features to add weight information to locations. That information comes from the metrics about the number of convexes that are operated in each location[12] during a previous verification on a monoprocessor (see below). The idea is to see how an supposedly good distribution influences verification times.
- Random Distribution: an uniform distribution of control nodes with no locality preservation.

The goal of the new set of experiments was getting answers about how some identified key issues affect total times: distribution of locations among *capsules*, communication strategy and symbolic state space representation fragmentation.

We recorded several metrics, the most meaningful ones being: idle time of each *capsule*, time spent at I/O at each *capsule*, time spent at the *Fix-Point Engine* (referred as "Time at fpe" in the tables), the number of interruptions that each *capsule* receives (referred as "#req. rcvd." in the tables). As a metric to measure the load balance we took a time-demanding operation such as the difference between symbolic states (see line 10 of algorithm 1). Then we accumulate for each node the product of the number of convexes that represent the symbolic states to be operated (it makes sense because, for a given set of clocks, this product approximately determines the computational complexity of the operation). We refer to this metric as "#ops." in the tables.

Looking at experimental data, there was a particularly bad case, which showed up on two processors and in which 90% of the work seems to fall into one *capsule*. No doubt there is a problem in balance, but one would expect times to be similar to the single processor version, yet they were up to almost seven times worse. Early experiments revealed the main problem was not

[11] TL stands for Target Locations.

[12] OpL stands for Operations per Location.

I/O, as can be seen in table 4. We decided to run the experiments over the Silicon Graphics machine to easily reason and pinpoint what was going on each *capsule*. These experiments are not meant to be conclusive since we are using just a single example and a low number of processors. Nevertheless, some phenomena might warn about challenges that make a good parallelization of backwards computation a non-trivial task.

The following results were obtained over TGC5 with unreachable "ERROR" state; this means that all the state space that reaches the target locations is generated (and the initial state doesn't belong to that set).

We observed that the *subscription* schema always performed a little better than the original *pseudo-polling* one, so we choose to only report on that one and on the *push* strategy. On the other hand, we do not report new results based on pure minimal cut since the weighted versions seems to perform better. Also, as expected, random distribution of locations performed extremely bad and examples did not finish in 4 hours of wall-clock time. Unfortunately, the current version of the tool logs metrics only after finishing, so we can not exhibit data. Let us add that the I/O time for *capsule* 0 also includes the I/O of *coordinator*, since in the current version the *coordinator* resides in that *capsule*.

We structured the data per partition strategy. Along with each table, we provide partial conclusions on how different versions performed.

Table 3: Results for TGC5, unreachable target state, single processor version.

	1 processor				
	tt (s)			*tc* (s)	*ti* (s)
	888			0	0
	Idle time (s)	Time at fpe (s)	I/O (s)	#req. rcvd	#ops.
Cap. 0	0	868	0	0	3273740

TL-Weighted Minimal Cut

The partitioning is clearly unsatisfactory in terms of workload distribution. Though *subscription* performs better than *push*, the drop of performance wrt. the monoprocessor version is remarkable due to a (probable) fragmentation problem. As it was reported by [6] and [4], changing the evaluation order during fix point calculation may exacerbate the fragmentation in the representation of regions (i.e., the number of convexes needed to represent a region). Note that the number of operations increased and I/O time had little impact. See table 4.

All the runs seems to indicate that the *push* strategy has some problem that is not captured by our current metrics, except for the total wall-clock time.

Table 4: Results for TGC5, unreachable target state, TL-Weighted Minimal Cut partitioning.

Comm. strategie	2 processors					3 processors				
Subscription	*tt* (s)			*tc* (s)	*ti* (s)	*tt* (s)			*tc* (s)	*ti*
	2962			24	2914	2041			49	3944
	Idle time (s)	Time at fpe (s)	I/O (s)	#req. rcvd	#ops.	Idle time (s)	Time at fpe (s)	I/O (s)	#req. rcvd	#ops.
Cap. 0	2914	3	75	1	25734	1971	3	55	6	67793
Cap. 1	0	2916	0	2	8596123	0	1994	1	17	7905481
Cap. 2						1973	0	53	3	942
Push	*tt* (s)			*tc* (s)	*ti* (s)	*tt* (s)			*tc* (s)	*ti*
	5271			24	5222	2194			49	4244
	Idle time (s)	Time at fpe (s)	I/O (s)	#req. rcvd[13]	#ops.	Idle time (s)	Time at fpe (s)	I/O (s)	#req. rcvd	#ops.
Cap. 0	5222	3	146	55	25734	2118	3	74	132	69306
Cap. 1	0	5225	0	748	8567574	0	2149	0	384	8303893
Cap. 2						2126	0	74	2	942

OpL-Weighted Minimal Cut

In this case we obtained better results in terms of workload distribution, as we expected. As a consequence, the interchange of information increased. Fragmentation still exists, and explains why time is not reduced wrt. the monoprocessor version. An unpleasant fact is that neither the *subscription* schema, neither the *push* schema finished with 3 processors. A possible explanation is an exacerbation of the fragmentation problem. See table 5.

Table 5: Results for TGC5, unreachable target state, OpL-Weighted Minimal Cut partitioning.

Comm. strategie	2 processors					3 processors				
Subscription	*tt* (s)			*tc* (s)	*ti* (s)	*tt* (s)			*tc* (s)	*ti*
	1650			24	1111	∞			N/A	N/A
	Idle time (s)	Time at fpe (s)	I/O (s)	#req. rcvd	#ops.	Idle time (s)	Time at fpe (s)	I/O (s)	#req. rcvd	#ops.
Cap. 0	1110	494	102	3	2293082					
Cap. 1	1	1558	55	12	2949683					
Cap. 2										
Push	*tt* (s)			*tc* (s)	*ti* (s)	*tt* (s)			*tc* (s)	*ti*
	4252			24	4077	∞			N/A	N/A
	Idle time (s)	Time at fpe (s)	I/O (s)	#req. rcvd	#ops.	Idle time (s)	Time at fpe (s)	I/O (s)	#req. rcvd	#ops.
Cap. 0	4077	116	91	748	704244					
Cap. 1	0	4207	0	666	3962755					
Cap. 2										

[13] Please note that when using a *push* strategy "#req. rcvd" are really messages containing regions received, as there are no requests.

As can be seen in tables 4 and 5, communication strategies did not influence the outcome of the experiments, maybe because the example is small and because hardware interrupts are not involved when communicating on a single machine. It must be noted that there seems to be a direct relationship between the previously mentioned metric (number of operations per location) and the time results.

In order to mitigate the fragmentation problem, we decided to apply a conservative abstraction which consists in applying convex hull when a region is going to be transported to another *capsule*. In what follows we report on a couple of experiments using this idea.

OpL-Weighted Minimal Cut, using convex hull

Transmitting just one convex per region reduces the wall-clock time for the *subscription* schema but there is still too much idle time. The *push* schema performance is far from being satisfactory, and we have no explanation for the phenomenon using the metrics proposed. See table 6.

Table 6: Results for TGC5, unreachable target state, OpL-Weighted Minimal Cut partitioning, using convex-hull.

Comm. strategie	2 processors					3 processors				
Subscription	*tt* (s)			*tc* (s)	*ti* (s)	*tt* (s)			*tc* (s)	*ti*
	932			24	893	992			49	1673
	Idle time (s)	Time at fpe (s)	I/O (s)	#req. rcvd	#ops.	Idle time (s)	Time at fpe (s)	I/O (s)	#req. rcvd	#ops.
Cap. 0	494	394	1	7	1056302	763	162	12	26	313815
Cap. 1	399	488	3	4	815933	910	38	14	6	147286
Cap. 2						0	923	0	45	1204280
Push	*tt* (s)			*tc* (s)	*ti* (s)	*tt* (s)			*tc* (s)	*ti*
	3602			24	3288	730			49	1152
	Idle time (s)	Time at fpe (s)	I/O (s)	#req. rcvd	#ops.	Idle time (s)	Time at fpe (s)	I/O (s)	#req. rcvd	#ops.
Cap. 0	3288	271	67	730	753880	474	189	11	485	246701
Cap. 1	0	3557	0	673	2743298	678	8	13	709	151443
Cap. 2						0	661	0	480	1021703

TL-Weighted Minimal Cut, using convex hull

In this case we have very good results for the case of three processors, and, surprisingly enough, have a poor performance on two. This shows that the fragmentation problem is not solved by the strategy of sending simpler regions between *capsules* or improving workload distribution. The phenomenon seems more complex and might depend on arrival times of regions or characteristics of partition strategies we have not yet identified. See table 7.

[13] Didn't finish in 4 wall-clock hours.

[13] Also didn't finish in 4 wall-clock hours.

Table 7: Results for TGC5, unreachable target state, TL-Weighted Minimal Cut partitioning, using convex hull.

Comm. strategy	2 processors					3 processors				
Subscription	*tt* (s)			*tc* (s)	*ti* (s)	*tt* (s)			*tc* (s)	*ti*
	4861			24	4815	452			49	766
	Idle time (s)	Time at fpe (s)	I/O (s)	#req. rcvd	#ops.	Idle time (s)	Time at fpe (s)	I/O (s)	#req. rcvd	#ops.
Cap. 0	4815	3	120	1	25734	382	2	10	4	21662
Cap. 1	0	4816	0	2	11105518	0	407	0	15	2310121
Cap. 2						384	0	10	3	942
Push	*tt* (s)			*tc* (s)	*ti* (s)	*tt* (s)			*tc* (s)	*ti*
	4833			24	4789	527			49	913
	Idle time (s)	Time at fpe (s)	I/O (s)	#req. rcvd	#ops.	Idle time (s)	Time at fpe (s)	I/O (s)	#req. rcvd	#ops.
Cap. 0	4798	3	136	1	25734	456	2	14	121	22121
Cap. 1	0	4799	0	33	11105518	0	480	0	363	2510605
Cap. 2						457	0	17	2	942

5 Conclusions and future work

Zeus is a Distributed Timed Model Checker based on the Kronos tool that was described and proved correct using software architectures. To the authors knowledge, it is the first Distributed Model Checker for timed systems based on backwards reachability calculation. It was conceived as a testbed for a wide range of design decisions such as communication schema or type of synchronization. Our preliminary results for asynchronous versions, although showing promising speedups in some cases, were surprisingly counterintuitive but justified the need for an open architecture to test alternative concepts.

The irregular behavior seen in some cases might be associated to the sensitivity of the data structures to the ordering of operations. On one hand, it would desirable to do as much local work at a *capsule* as possible, without communicating with the rest. Intuition told us that the pros of this asynchronous strategy are: (a) good use of parallel processing resources, (b) avoidance of an early propagation of regions that will eventually be included into another, thus reducing the number of propagations. On the other hand, when communication takes place this may lead to operations between "mature" regions with a large number of convexes. Also, breadth-first traversal is not honored (which is reported as a good strategy to avoid fragmentation [4]).

More research should be done to gain insight on these phenomena and propose solutions for the parallel setting. This might require the development of *ad hoc* visualization techniques, something we are currently working on. A future direction to explore is improving the representation of regions in the sense of the number of convexes required to represent a given region.

Another line of research we are currently exploring is the development of a synchronous version of Zeus, trying to mimic the order in which Kronos applies the operators in the sequential version. More precisely, this version would apply synchronically every fix point iteration at each *capsule*. Thought it looses potential parallelism, it may be possible to keep the number of zone operations close to what Kronos performs on a monoprocessor. Therefore, if good load balance is achieved, this strategy may redound in important speedups.

We also believe that dynamic balance could eventually be applied for a better usage of distributed resources. Fortunately, Zeus architecture seems to be an appealing starting point for location migration between neighbor *capsules* looking for a better load balance. This can be done as an architecture reconfiguration were new links are established between processing *capsules*.

A natural next step is the extension of Zeus to deal with full TCTL, mainly by adapting Kronos strategies to the distributed setting. Besides, forward calculation can be easily incorporated in this scenario, if required. Obtaining counterexample traces is also a relevant area or research (e.g., [4]).

Acknowledgments

We would like to thank Sergio Yovine for making Kronos libraries available to us.

References

[1] Alur, R., C. Courcoubetis and D. L. Dill, *Model-checking in dense real-time*, Information and Computation **104** (1993), pp. 2–34.
URL `http://citeseer.nj.nec.com/alur93modelchecking.html`

[2] Alur, R. and D. L. Dill, *A theory of timed automata*, Theoretical Computer Science **126** (1994), pp. 183–235.
URL `http://citeseer.nj.nec.com/alur94theory.html`

[3] Barnat, J., L. Brim and J. Stríbřná, *Distributed LTL model-checking in SPIN*, in: *SPIN*, 2001, pp. 200–216.
URL `http://citeseer.nj.nec.com/article/barnat00distributed.html`

[4] 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.
URL
`http://citeseer.nj.nec.com/behrmann00distributing.html`

[5] 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.
URL `http://citeseer.nj.nec.com/326863.html`

[6] 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).
URL `http://www.elsevier.com/locate/entcs/volume65.html`

[7] Cousot, P., "Methodes Iteratives de Construction et D'Aproximation de Points Fixes D'Operateurs Monotones sur un Treillis, Analyse Semantique des Programmes," Ph d. thesis, Université Scientifique et Médicale de Grenoble, Institut National Polytechnique de Grenoble (1978).

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

[9] Daws, C. and S. Yovine, *Two examples of verification of multirate timed automata with* KRONOS, in: *Proceedings of the* 16th *IEEE Real-Time Systems Symposium (RTSS'95)* (1995), pp. 66–75.
URL `http://citeseer.nj.nec.com/daws95two.html`

[10] 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.

[11] Garavel, H., R. Mateescu and I. M. Smarandache, *Parallel state space construction for model-checking*, in: M. B. Dwyer, editor, *Proc. of the 8th International SPIN Workshop*, Toronto, Canada, 2001, pp. 217–234.
URL `http://citeseer.nj.nec.com/474094.html`

[12] Grumberg, O., T. Heyman and A. Schuster, *Distributed symbolic model checking for μ-calculus*, in: *Computer Aided Verification*, 2001, pp. 350–362.
URL `http://citeseer.nj.nec.com/473825.html`

[13] 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.

[14] Heyman, T., D. Geist, O. Grumberg and A. Schuster, *Achieving scalability in parallel reachability analysis of very large circuits*, in: O. Grumberg, editor, *Computer Aided Verification, 12th International Conference*, LNCS **1855** (2000), pp. 20–35.
URL `http://citeseer.nj.nec.com/heyman00achieving.html`

[15] 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).

[16] Lerda, F. and R. Sisto, *Distributed-memory model checking with SPIN*, in: *Proc. of the 5th International SPIN Workshop*, LNCS **1680** (1999).
URL `http://citeseer.nj.nec.com/lerda99distributedmemory.html`

[17] M. Bozga, O. Maler, A. Pnueli and S. Yovine, *Some progress in the symbolic verification of timed automata*, in: O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, LNCS **1254** (1997), pp. 179–190.
URL `http://citeseer.nj.nec.com/bozga97some.html`

[18] 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.
URL `http://citeseer.nj.nec.com/ranjan96binary.html`

[19] 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`

[20] Stern, U. and D. L. Dill, *Parallelizing the Murφ verifier*, in: *Computer Aided Verification*, LNCS **1254** (1997), pp. 256–278.
URL `http://citeseer.nj.nec.com/stern97parallelizing.html`

[21] Tripakis, S. and S. Yovine, *Verification of the fast reservation protocol with delayed transmission using the tool* KRONOS, in: *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium (RTAS'98)* (1998), pp. 165–170.
URL `http://citeseer.nj.nec.com/186425.html`

Appendix A - Architecture description

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

Zeus architecture description can be more easily understood centering at the *Fix-Point Engine*. This is the component that runs the fix-point calculation, much as it does in Kronos (shown as algorithm 1). 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 *Fix-Point Engine* is unaware of such distribution, it makes no difference between local and remote locations, except for the fact that it iterates writing only over the local ones. This is why there is a *router* component inside the *storage*.

Upon reception of a *read* petition from the *Fix-Point Engine*, the *router* delivers regions from the *local regions' storage*, if their location is local, or from the *remote regions' storage*, if they are 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 visible from a remote *capsule*. The rationale for the existence of *delta accumulators* is that regions difference is very expensive, so it makes sense to duplicate what needs to be sent instead of having them in one place and wasting a lot of time for each transmission, actually computing the difference. *Delta accumulators* are emptied when regions are sent to other *capsules*.

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

```
function Fix_Point_Engine.iterate(): nat
 1: $\#changes := 0$
 2: for all $s \in storage.local\_discrete\_states()$ do
 3:    $R_s := storage.get\_region(s)$
 4:    $PredE := \emptyset$
 5:    for all $suc$ discrete successor of $s$ do
 6:       $R_{suc} := storage.get\_dif\_region(suc)$
 7:       $PredE := PredE \cup pred_e(R_{suc}, inv(suc), reset\_clocks(suc))$
 8:    end for
 9:    $PredT := pred_t(PredE)$
10:    $\Delta R_s := PredT - R_s$
11:    if $\neg\emptyset?(\Delta R_s)$ then
12:       $\#changes++$
13:       $storage.set\_dif\_region(s, \Delta R_s)$
14:       $storage.set\_region(s, R_s \cup PredT)$
15:    end if
16: end for
17: $storage.iteration\_ended()$
18: return $\#changes$
```

Algorithm 1: Distributed reachability algorithm (runs inside the *Fix-Point Engine*).

are contained inside the *remote regions' storage*. One can think that there is an *embassy* at the other end of every *delta accumulator*. They not only store the regions that arrive through the network, but they also posses a small state machine that controls if the remote *capsule* needs to be asked for a region –i.e., if the *embassy* is asked for a region but is currently empty– or whether the maximum possible region for a location has already been produced, thus avoiding extra requests. It is also responsible of showing the same content to every request during an iteration.

When an *embassy* needs new regions from the remote *capsule*, it contacts the appropriate *connector*. 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 *embassys*, it has only one *connector* handling bi-directional communication to B. The same happens at B.

In a naïve approach, a *connector* merely proxies request from a *capsule* to the other. However, a few twists were made for performance, i.e., for minimizing interruptions to the fix-point calculation at the other end. First, *connectors* do *piggybacking*: instead of asking for a particular locations' region, they ask for the regions of the full set of locations they represent; also, to "compensate" the other *capsule* for the interruption, they send the regions of the full set of locations that are of interest to the *capsule* receiving the request. These regions come from the corresponding *delta accumulators*. It should be noted that, except for the piggybacking taking place, this is a *polling* schema, thus the name of *pseudo-polling*. *Connectors* perform another function also, that needs the description of the *load assessor* component to be explained.

Load assessor establishes not only if the *capsule* is idle or working, but also how bad it needs new regions. It is a key component for the communication model. Its full details were explained on section 3.

When requested to send a petition, the *connectors* check with the *load assessor* to see if there is a "high regions-demand", in which case they proceed, as explained before. However, if there is a "low regions-demand", they will silently drop the request.

As was noted before, a *capsule* can be idle, meaning that global fix-point is not yet reached because there is still work going on at other *capsules*, but a local fix-point was established anyway. To take advantage of the idle time, a new component, the *helper*, is introduced. It performs auxiliary task, such as representation compaction of the regions at the *delta accumulators* and at the *local regions' storage*.

Finally, an *iterator*, is responsible of deciding to run either the *Fix-Point Engine* or the *helper* based on the readings of the *load assessor*, and also of sending statistics to the *coordinator*.

The *coordinator* starts the process, partitions the graph, distributes the workload and established 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 more detailed description of the architecture including state machines and transducers can be found in [19].