Distributed Behavioral Cartography of Timed Automata* †

Étienne André, Camille Coti, Sami Evangelista
Université Paris 13, Sorbonne Paris Cité, LIPN, CNRS, UMR 7030, F-93430, Villetaneuse, France
{first.last}@univ-paris13.fr

ABSTRACT

Real-time systems, characterized by a set of timings constants (internal delays, timers, clock speeds), need to be perfectly reliable. Formal methods can prove their correctness but, if one of the timing constants changes, verification needs to be restarted from scratch. Also, variations of some delays (even infinitesimal) may lead to the specification violation. It is thus interesting to reason parametrically, and synthesize constraints on the timing constants seen as parameters to formally guarantee the specification. We propose here an attempt to distribute a synthesis algorithm, the behavioral cartography, and we evaluate two work distribution algorithms. The parallelization gives promising results and opens perspectives toward verification of larger models.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*

Keywords

Parameter synthesis, formal verification, message passing, master–worker parallelization, work scheduling heuristics

1. INTRODUCTION

Real-time systems need to be perfectly reliable, in terms of both functional (discrete behavior) and timed correctness (e.g., all deadlines met). Formal methods can prove that a system, characterized by a set of timings constants (internal delays, timers, clock speeds), satisfies both a functional and a timed specification. However, this is not entirely satisfactory for several reasons. First, if one of the timing constants changes (e.g., if a component is replaced with another one that has a different delay), then the specification may not hold anymore, and the design process shall be restarted from the beginning. Second, once the system is implemented, infinitesimal variations of some delays may lead to dramatic changes in the global behavior, and lead to a violation of the specification. It is thus interesting to reason parametrically, and synthesize constraints on the timing constants seen as parameters to formally guarantee the specification.

Contribution.

In [AF10], we proposed the behavioral cartography algorithm (BC), that covers a bounded parameter domain with tiles, i.e., parametric zones of uniform discrete behavior. BC relies on iterative calls to another algorithm (the "inverse method" IM), that builds a tile from a reference valuation (or point). The advantage of BC is that, in any tile, choosing another point within the tile will have no impact on the global, discrete system behavior. This parameter domain exploration is very time-consuming. In order to tackle larger case studies, we aim at taking advantage of the power of distributed systems such as clusters. Parallel execution on several cores is a way to speed-up the execution of a program: the computation is shared between the cores that participate to the computation. In contrast to other synthesis algorithms, BC is a priori well-suited for distributed executions due to its intrinsically decomposed nature. However, although the data handled by each instance of IM are independent from each other, it is not easy to efficiently distribute BC since finding the right points on which to call IM so as to avoid overlapping is not trivial. Overlapping two tiles leads to redundant computations and therefore, slows down the overall performance. In this work, we evaluate different exploration heuristics, aiming at minimizing this tile overlapping. To the best of our knowledge, this is the first distributed algorithm for the synthesis of timing parameters. We implemented a master-worker parallel implementation of IMITATOR [AFKS12] (that implements BC) and we present an evaluation of two work distribution algorithms. We show that the speed-up obtained by the parallelization of the code depends on some properties of the model. Overall, the parallelization of the parameter domain exploration gives promising results and opens perspectives toward verification of larger models at larger scale.

*(This work is partially supported by Université Paris 13's Projet BQR SynPaTiC ("Synthèse de paramètres distribuée et multi-cœurs").

†This is the author version of the paper published with the same title in the proceedings of the EuroMPI/ASIA 2014 conference; the final version is available at www.acm.org.

EuroMPI/ASIA '14, September 9-12 2014, Kyoto, Japan

http://dx.doi.org/10.1145/2642769.2642784.

Related Work.

Formal verification can be made in parallel in two ways: modeling languages can be designed to be easy to use in a distributed fashion, or the verification algorithms themselves can be parallelized. Our approach fits in the second category. There were many works on distributed verification algorithms, so we name only a few. APMC is a probabilistic model-checker relying on random path generation and exploration. This exploration can be made in parallel on different processes [HBE+10]. PKind [KT11] is another parallel model checker, using an event-based parallel execution model: processes compute their data set, and send invariants they find to all the other processes. Eventually, a synchronizing process (sort of "master process") decides when the computation needs to end. PReach [DPBB+10] was implemented in the functional language for distributed systems Erlang. It uses a depth-first search algorithm across the parallel processes: the tree exploration is made in parallel. More recently, two algorithms were proposed to address multi-core LTL verification [ELPP12] and emptiness checking of timed Büchi automata [LOD+13]. Finally, the master-worker parallelization scheme has been widely used to compute tasks that are independent from each other. In [SWB98] it was also shown how it can give good load balancing between the parallel processes.

Outline.

Section 2 recalls the necessary preliminaries and states our main objective. Section 3 proposes distributed algorithms for parameter synthesis. Section 4 gives our practical solution relying on MPI. Section 5 compares our algorithms experimentally. Section 6 gives future perspectives.

2. PRELIMINARIES

2.1 Parameter Constraints

Throughout this paper, we assume a set $P = \{p_1, \ldots, p_M\}$ of M *parameters*, i.e., unknown constants. A (parameter) *valuation* π is a function $\pi : P \to \mathbb{Q}_+^M$. We will often identify a valuation π with the *point* $(\pi(p_1), \ldots, \pi(p_M))$. An *integer* point is a valuation $\pi : P \to \mathbb{N}^M$. A (linear) constraint on the parameters is a set of linear inequalities on P, and can be seen as a polyhedron in M dimensions. Given a point π and a constraint K, we write $\pi \models K$ if π satisfies K; geometrically speaking, this corresponds to the fact that the polyhedron K contains the point π.

2.2 Parametric Timed Automata

Timed Automata (TA) are finite state automata (made of control states and transitions labeled with actions), extended with clocks, i.e., real-valued variables evolving at the same constant rate. Clocks can be compared with constants in guards (condition to be verified when taking a transition) and invariants (condition to be verified to remain in a control state), and can be reset along transitions. In this work, given a TA $\mathcal{A}$, we call its discrete behavior the set of all possible alternating sequences of control states and actions allowed in $\mathcal{A}$. Parametric Timed Automata (PTA) [AHV93] extend (TA) by allowing parameters in place of constants. The synthesis problem consists in deriving parameter valuations (usually in the form of a constraint) such that a given property is satisfied, e.g., the non-reachability of some control state. PTA and their extensions have been extensively used in the past two decades to model and verify real-time systems, typically hardware components, scheduling problems or communication protocols. Several tools were developed (e.g., TReX [ABS01], Roméo [LRST09], SpaceEx [FGD+11], or Imitator [AFKS12]). (For syntax and semantics of (P)TA, see, e.g., [AHV93, ACEF09].) Given a PTA $\mathcal{A}$ and a point π, we denote by $\mathcal{A}[\pi]$ the (non-parametric) TA where each parameter p_i was replaced with $\pi(p_i)$.

2.3 The Inverse Method

In [ACEF09], the *inverse method* (*IM*) was proposed: given a PTA $\mathcal{A}$, this semi-algorithm takes advantage of a reference point π, and generalizes it in the form of a *tile*, i.e., a linear constraint K on the parameters where the discrete behavior is uniform (see Figure 1a). That is, for any point π' satisfying K, the discrete behavior of $\mathcal{A}[\pi']$ is equal to the discrete behavior of $\mathcal{A}[\pi]$. As a consequence, any linear-time property (expressed using, e.g., LTL) valid for $\mathcal{A}[\pi]$ is also valid for $\mathcal{A}[\pi']$, and vice-versa.

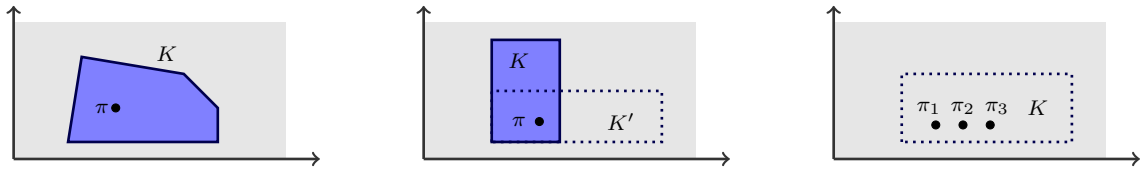
An application of *IM* is to derive robustness conditions for the system. The study of the robustness in real-time systems (see, e.g., [Mar11]) aims at deciding whether infinitesimal variations of time (due to, e.g., slightly extended or shrunk deadlines, or clock drifts) may impact the overall, discrete system behavior. The inverse method was shown to give a measure of the system robustness, and conditions can be derived to render robust a non-robust system (see, e.g., [APP13] in the setting of parametric time Petri nets). However, *IM* suffers from several drawbacks: *1*) It may not terminate. Indeed, parameter synthesis for PTA is known to be undecidable [AHV93]. Nevertheless, it behaves "well" in the sense that it terminates for all case studies we considered, except for small examples designed on purpose to show non-termination. *2*) It is non-confluent: given $\mathcal{A}$ and π, different calls to $IM(\mathcal{A}, \pi)$ may yield different constraints incomparable with each other. This situation is graphically depicted in Figure 1b: a first execution of $IM(\mathcal{A}, \pi)$ gives K, whereas a second execution may give K'. *3*) Non-confluence implies non-completeness: there may exist points π' outside $IM(\mathcal{A}, \pi)$ such that $\mathcal{A}[\pi]$ and $\mathcal{A}[\pi']$ have the same discrete behavior. Again, in Figure 1b, if $IM(\mathcal{A}, \pi)$ gives K, then any point π' in $K' \setminus K$ has the same discrete behavior as $\mathcal{A}[\pi]$ but does not belong to K.

Finally note that, in general, tiles have no predefined "shape": they are general polyhedra in M dimensions that can have arbitrary size, number of vertices, and edge angles.

2.4 The Behavioral Cartography

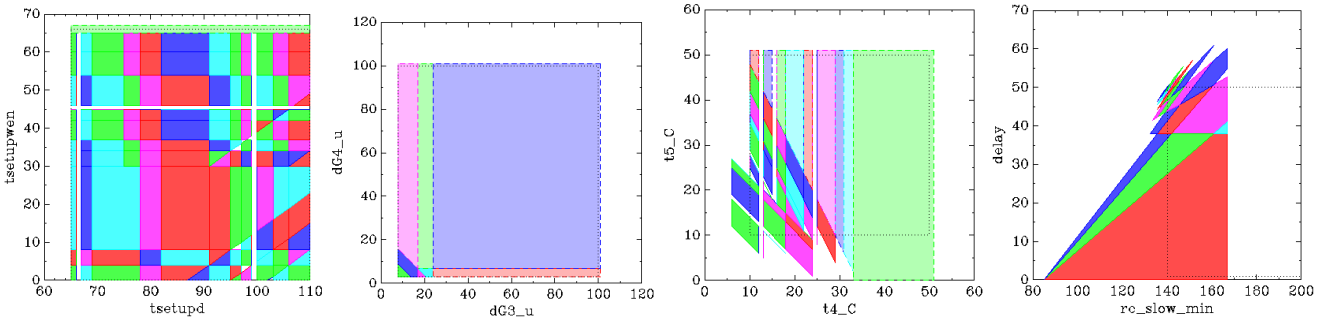
IM was extended in [AF10] into the behavioral cartography algorithm (*BC*): given a PTA $\mathcal{A}$ and a bounded parameter domain D (usually a rectangle in M dimensions), by repeatedly calling *IM* on (some of the) integer points of D (of which there is a finite number), one is able to cover D with tiles. Hence, the result is a cartography of the tiles in which the discrete behavior is uniform in $\mathcal{A}$. Unfortunately, the general "shape" of the cartography is entirely arbitrary, since tiles can have any shape themselves. Figure 2 gives examples of cartographies in 2 dimensions. Whereas Figure 2a is rather homogeneous in terms of size and positions of the tiles, this is not at all the cases for the other ones.

2.5 Objective



(a) Graphical example of IM (b) Non-confluence of IM (c) Choosing points

Figure 1: The inverse method: graphical representation of examples and problems



(a) SPSMALL memory (b) Flip-flop circuit (c) Scheduling problem (d) Root contention protocol

Figure 2: Examples of graphical behavioral cartographies in 2 dimensions

In order to tackle larger case studies, our objective is to take advantage of the iterative nature of the cartography (in contrast to most, if not all, other known synthesis algorithms), and to distribute it on N processes. There is no theoretical obstacle in doing so, since all calls to IM are independent from each other. The challenge is rather to optimize the points on which IM is called, so that as few redundant constraints as possible are computed.

3. DISTRIBUTING THE CARTOGRAPHY

3.1 The Non-distributed Cartography

Unfortunately, finding the set of integer points not covered by a list of polyhedra is a problem that has no known efficient practical solution. Hence, there is no other choice than enumerating all points. In the non-distributed cartography, an empty set of tiles S is first created. Then, each point in D is sequentially checked to see if it is covered by any tile in S. If not, IM is called on this point, and the result is added to S.[1] When all points are enumerated, the algorithm returns the set of tiles. (See Algorithm 1 in Appendix A for a formal algorithm.)

3.2 Master-Worker Parallelization Scheme

A simple solution could have been to split the rectangle D into N subparts, and then ask each process to handle its own subpart in an independent manner. This domain decomposition method is often used for regular data distributions, where all subparts require the same processing time, and preferably on domain shapes such as rectangles or hypercubes, that can easily be mapped on a grid of processes. Unfortunately, this solution is not satisfactory due to the unknown "shape" of the cartography. Indeed, it often happens that some tiles use most of the space in D, whereas other tiles can be concentrated in a very small area (see, e.g., Figures 2b and 2d); this cannot be known before computing the cartography. We thus propose here master–workers algorithms. Workers ask the master for a point on which to call IM, then execute IM from that point, and finally send the corresponding result to the master. This is a *pull-based* algorithm, since workers pull work from the master.

The master will not call IM itself, but will instead distribute points to the workers. Whereas this may be a loss of efficiency for few processes (since the master is not calling IM), this shall be compensated for a large number of processes. Moreover, since workers pull work when they are done with their previous computation, this parallel computation scheme balances the load between workers automatically. If a worker takes a long time to compute its part of work, the other workers can compute several parts of work in the meantime. As a consequence, pull-based master-worker parallelization schemes are particularly interesting for BC, regarding the specificities of the computation made by IM.

Note that the master does not itself take part to the cartography; although this could be achieved using non-blocking communication, the risk is that its own computations be too frequently interrupted to be useful.

3.3 An Abstract Algorithm for the Master

We describe here an "abstract" algorithm for the Master. The master starts by creating an empty set of tiles. The workers send a work request to mean that they are ready to receive some work. The master then sends an initial point to each process that requested some work. Then, while D is not entirely covered, every time a process sends a result, the master stores it, and sends a new point to the process, computed using function *choosePoint*(). Finally, once all integer points are covered, the master receives the remaining processes and sends stop signals. (A formal algorithm is given in Algorithm 2 in Appendix A). *choosePoint*() is a

[1]Note that, due to the non-complete nature of IM, one could call IM on a point already covered by another tile; however, experiments showed that it is always more costly to do so.

generic function that picks a new uncovered point; it will be instantiated in Sections 3.4 and 3.5, leading to concrete algorithms for the master.

The way points are picked by the master to be distributed to the workers is therefore a highly critical question. Choosing points in a wrong manner can lead to a dramatic loss of efficiency. For example, choosing points very close to each other may most probably lead to the redundant computation of the same tile. This situation is graphically depicted in Figure 1c, where points π_1, π_2, π_3 may yield the same tile C. In the remainder of this section, we will consider two Master–Workers algorithms, that will instantiate *choosePoint*().

3.4 Sequential Enumeration

Our first concrete algorithm is a direct extension of the monolithic algorithm: as in the non-distributed BC, we enumerate all points one after the other. The *choosePoint*() function thus returns the next point (considered using a sequential order, e.g., dimension after dimension) that is not yet covered by any tile. Its main advantage is that it is cheap on the master side. Its main drawback is the risk of redundant computation by the workers, due to the situation graphically depicted in Figure 1c: for example, at the beginning, the N processes will ask for work, and the master will give them the first sequential N points, all very close to each other, with a high risk of redundant computation. Nevertheless, if the tiles are "small", i.e., if they contain few integer points, or if the computation time for each tile is large, then this algorithm may turn out to be not too inefficient (though not too good either), as we will see in Section 5.

3.5 Random Selection and Sequential

In order to prevent the problem of redundancy coming from the sequential algorithm, we design a random-based algorithm. In this second algorithm, the *choosePoint*() function randomly computes a point, and then checks whether it is covered by any tile; if not, it is returned. Otherwise, a second try is made, and so on, until a given maximum number (*max*) of attempts is reached. In that latter case, we switch to the sequential algorithm until all points are covered. This step is necessary to guarantee the full coverage of the integer points. Indeed, stopping after *max* tries could give a probabilistic coverage (e.g., 99 %) of integer points, but cannot guarantee the full coverage. Once more, since finding the points not covered by a list of tiles has no practical solution, this sequential check is the only option. The need for running a sequential check so that all points are covered could lead to the same problem as in the first algorithm. We will see in Section 5 that this algorithm behaves much better than the purely sequential one.

4. DISTRIBUTING IMITATOR WITH MPI

IMITATOR [AFKS12] is a tool implementing *IM* and *BC* (among other synthesis algorithms), entirely written in the functional, object-oriented language OCaml, and relying on the Parma Polyhedra Library (PPL) [BHZ08] for polyhedra operations. Until this work, IMITATOR was entirely sequential, i.e., single process and single-threaded. In order to implement the algorithms of Section 3 and use IMITATOR in a distributed setting, we had to significantly modify its internal structure.

Message Passing.

We distributed IMITATOR using the Message Passing Interface [For94, GGHL+96], which is the *de facto* standard library for programming parallel applications on a distributed memory model. MPI official bindings are available in C, C++, Fortran 77 and Fortran 90. We used OCamlMPI[2], a package featuring OCaml bindings for MPI functions.

MPI send and receive functions use a *tag* to identify messages and *ranks* to identify processes in a unique way. For example, a message sent from a process with a given tag can only be matched by a reception that specifies that the message must carry this tag and come from this sender. Receptions can use *wildcards*, i.e., receive from *any_source* and/or with *any_tag*. The actual sender and/or the tag are found in a data structure filled by the receive function: its *status*. As a consequence, tags can be used for signaling purpose. When a worker asks the master for a point, it can either send a result (the result of the previous computation) or simply ask for some work (at the beginning of the computation). We use two different tags to tell the master whether the current communication is a simple work request or a result is coming in the next communication. We also use wildcards for tags on workers in order to differentiate communications that come from the master. Three kinds of messages can be sent by the master. Most messages are work inputs. When the master has distributed all the points, it has no work to distribute anymore: instead, it sends a message carrying a termination tag. In some cases (e.g., when a time limit is reached), the master can tell a worker to stop working. This communication carries the third type of tag.

MPI functions send buffers of datatypes. Hence, the data sent between the master and the workers must be linearized in order to be sent by MPI functions. We implemented *ad hoc* serialization functions that represent points and constraints as strings, which are then converted by OCamlMPI into arrays of characters. However, the data sent by MPI communications have varying sizes. Hence, each data communication is made of two steps: the first communication sends an integer that provides the size of the buffer coming with the second communication. The master must be able to receive results from the workers without any predefined ordering. Hence, we use the *any_source* wildcard in reception on the master's side. However, we discovered that OCamlMPI had an unexpected behavior related to this *any_source* wildcard: the source rank obtained in the status of the reception does not always give the correct rank for the sender. The workaround we use here consists in sending the rank of the sender in a first communication when a worker sends a message to the master. For pull-only communications, the rank is simply sent in the payload of the message. On the other hand, communications that send a result already use this payload to carry the size of the result buffer. As a consequence, we had to use an extra message, which is sent before the size of the result buffer. This extra message can be avoided by sending both the size of the buffer and the worker's rank in a single message. However, such small MPI messages are sent in eager mode and can be easily pipelined by the MPI library and the underlying network, so the latency introduced by this extra message should not harm the performance critically.

[2] https://forge.ocamlcore.org/projects/ocamlmpi/

5. EXPERIMENTAL VALIDATION

We measured the performance of the distributed IMITATOR[3] and compared the two point selection algorithms described in Section 3.3 on a Linux-based cluster. The nodes of this cluster feature two 6-core Intel Xeon X5670 running at 2.93 GHz CPUs (therefore, 12 cores in a NUMA fashion). Each node has 24 GiB of memory and runs a 64-bit Linux 3.2 kernel. The code was compiled using OCaml 3.12.1. The message-passing library we used is Bull's OpenMPI variant for Bullx, and the nodes are interconnected by a 40 Gb/s InfiniBand network.

We focused the performance evaluation on the speed-up yielded by the two parallelization algorithms. Algorithm *choosePoint* affects the performance of the parallel program mainly on how the areas computed by the workers overlap, as described in Section 3.2. Hence, we also measured the number of constraints computed by the program in order to have an idea of the rate of redundant computations. This measure only gives an idea, due to the non-determinism in BC: since IM is non-deterministic, BC can yield a different number of tiles. Furthermore, in the distributed setting, the order in which the points are called may also lead to non-determinism. Note that redundant constraints are harmful in terms of efficiency (since this means a same computation has been performed several times), but not in terms of result: either the redundant constraints can be kept (which is no problem for drawing graphics such as in Figure 2), or they can be eliminated using equality (or inclusion) checks, which adds only a small overhead.

We evaluated the scalability of the parallel implementation of IMITATOR on several use-cases: we summarize here results on two case studies, i.e., Simop, a model of a networked automation system, and Sched3, a model of a schedulability problem. We measured their scalability on up to 36 processes. This can seem as a not very large scale when compared to many scientific parallel applications. However, this explains by several reasons: first, this is a first tentative distribution of a parameter synthesis algorithm, and hence our algorithms are exploratory. Second, the number of processes is intrinsically limited by the number of tiles (a few dozens for these two use-cases).

The speed-up obtained with these use-cases is presented in Figures 3a and 3c. Both use-cases benefit from the parallelization: their executions are sped-up when processes are added. The rate of redundant constraints computed by IMITATOR is given in Figure 3b and 3d. This rate is computed w.r.t. the sequential execution of IMITATOR: it represents the constraints that are computed several times by several workers because they are located in overlapping (or even identical) tiles. Recall that two different points can lead to the same tile, especially when they are close to one another. We can see that the sequential point selection algorithm generates about twice as many redundant constraints than the random algorithm. Hence, the good (and somehow surprising) news is that the random algorithm is much more efficient than its sequential counterpart, even though the random algorithm has to enumerate all points in a second phase just as the sequential one. This is due to two reasons: first, finding points randomly is likely to avoid redundancy (in the first phase). Then, in the second (sequential) phase, if very few points are not covered (i.e., if max is large enough), then the risk of having two processes running IM on close points is low. Finally note that the speed-up of the random algorithm, although positive, is not linear; for Sched3, the speed-up for 36 processes is around 12. Although this gives space for improvement (see Section 6), we believe this is a promising result, considering the difficulties inherent to the problem of parameter synthesis.

We also compare, for the sole random algorithm, the efficiency when varying the max constant. With no surprise, for Simop, the larger max, the more efficient the computation, both in terms of speed-up (Figure 3a) and redundancy (Figure 3b). For Sched3, however, there is no difference between the two values of max.

Let us finally compare the case studies: Sched3 scales better than Simop and induces less redundant constraints. Profiling the processes of the parallel execution on 36 processes showed that with Simop, the workers are busy about 93% of the execution time on average, whereas with Sched3 they are busy 98% of the time. Besides, the average relative time spent by the master to compute the next point to be sent to a worker is 2.3% with Sched3 and 22.3% with Simop. Hence, with a less busy master, workers have also less idle time (waiting for the next point), and this gives a better hope for scalability. This is due to the structure of the case studies: Simop contains many points, and each call to IM is fast; conversely, Sched3 contains less points, and each call is longer, thus leading to a small occupation of the master. More detailed profiling results are presented in Tables 2 and 3 in Appendix B.

Overall, we can see that IMITATOR benefits from this parallelization, which allows it to run (much) faster than on a single process. Besides, because of the unknown shape of the tiles computed by IM, a random point selection algorithm performs better than a sequential one, even though the random algorithm needs to investigate all integer points in a second phase.

6. CONCLUSION AND PERSPECTIVES

We proposed here a first attempt of distributing an algorithm for the synthesis of timing parameters, leading to two parallelization algorithms. Our goal here was to speed-up the execution and make sure it will be able to scale on larger models. The promising results showed by the performance evaluation validate this approach. In the near future, we are going to explore other point selection algorithms and heuristics, so as to verify larger models at larger scale.

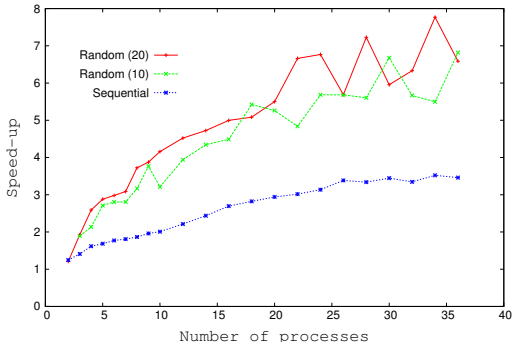
Perspectives.

The master-worker parallelization scheme is one way to distribute computation. Another open perspective consists in looking into other domain decomposition and parallelization schemes, since some of them are more intrinsically scalable than the aforementioned scheme.

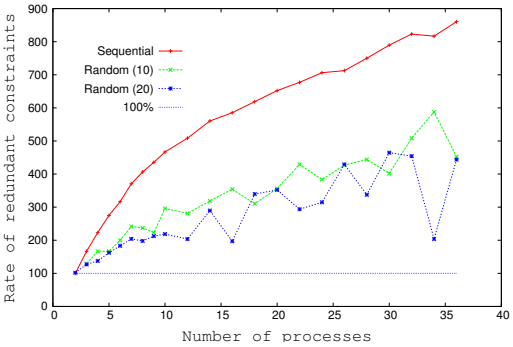
Another future work is the design of heuristics to improve efficiency. An issue is whether to stop or not an execution of IM when its reference point π has been covered by another tile. Although it might give a different tile (due to non-completeness), it is also likely to yield the same tile. Finding cases in which it is generally more efficient to leave the execution running would be an important heuristic.

Orthogonal to this work is the question of running parameter synthesis algorithms on multi-core (shared memory)

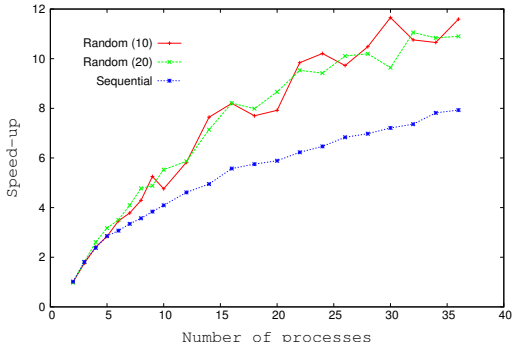
[3]Source, binaries, models and results are available at http://www.lipn.fr/~andre/patator/



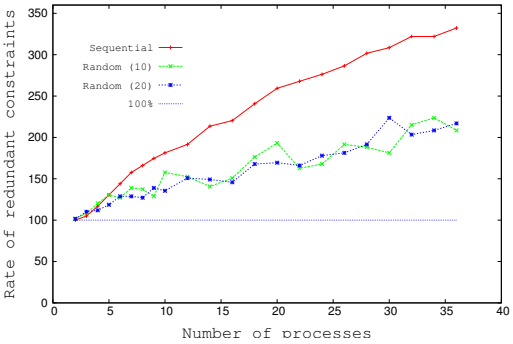
(a) Scalability of the parallel IMITATOR on Simop



(b) Rate of redundant constraint computations on Simop



(c) Scalability of the parallel IMITATOR on Sched3



(d) Rate of redundant constraint computations on Sched3

Figure 3: Experimental results

machines. We proposed a first parallel depth-first search algorithm for LTL in [ELPP12]; it was then extended to emptiness checking of timed Büchi automata in [LOD+13]. The next step will be to extend these algorithms to parameter synthesis.

7. REFERENCES

[ABS01] Aurore Annichini, Ahmed Bouajjani, and Mihaela Sighireanu. TReX: A tool for reachability analysis of complex systems. In *CAV*, LNCS, pages 368–372. Springer, 2001.

[ACEF09] Étienne André, Thomas Chatain, Emmanuelle Encrenaz, and Laurent Fribourg. An inverse method for parametric timed automata. *IJFCS*, 20(5):819–836, 2009.

[AF10] Étienne André and Laurent Fribourg. Behavioral cartography of timed automata. In *RP*, volume 6227 of *LNCS*, pages 76–90. Springer, 2010.

[AFKS12] Étienne André, Laurent Fribourg, Ulrich Kühne, and Romain Soulat. IMITATOR 2.5: A tool for analyzing robustness in scheduling problems. In *FM*, volume 7436 of *LNCS*, pages 33–36. Springer, 2012.

[AHV93] Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. Parametric real-time reasoning. In *STOC*, pages 592–601. ACM, 1993.

[APP13] Étienne André, Laure Petrucci, and Giuseppe Pellegrino. Precise robustness analysis of time Petri nets with inhibitor arcs. In *FORMATS*, volume 8053 of *LNCS*, pages 1–15. Springer, 2013.

[BHZ08] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.

[DPBB+10] Flavio M De Paula, Brad Bingham, Jesse Bingham, John Erickson, Mark Reitblatt, and Gaurav Singh. PREACH: A distributed explicit state model checker. Technical Report TR-2010-05, University of British Columbia, 2010.

[ELPP12] Sami Evangelista, Alfons Laarman, Laure Petrucci, and Jaco Van De Pol. Improved multi-core nested depth-first search. In *ATVA*, volume 7561 of *LNCS*, pages 269–283, 2012.

[FGD+11] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. SpaceEx: Scalable verification of hybrid systems. In *CAV*, LNCS. Springer, 2011.

[For94] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[GGHL+96] Al Geist, William D. Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing L. Lusk, William Saphir, Anthony Skjellum, and Marc Snir. MPI-2: Extending the

message-passing interface. In *EuroPar*, volume 1123 of *LNCS*, pages 128–135. Springer, 1996.

[HBE+10] Khaled Hamidouche, Alexandre Borghi, Pierre Esterie, Joel Falcou, and Sylvain Peyronnet. Three high performance architectures in the parallel APMC boat. In *PMDC*, pages 20–27. IEEE, 2010.

[KT11] Temesghen Kahsai and Cesare Tinelli. PKind: A parallel k-induction based model checker. In *PDMC*, volume 72 of *EPTCS*, pages 55–62, 2011.

[LOD+13] Alfons Laarman, Mads Chr. Olesen, Andreas Engelbredt Dalsgaard, Kim Guldstrand Larsen, and Jaco Van De Pol. Multi-core emptiness checking of timed Büchi automata using inclusion abstraction. In *CAV*, volume 8044 of *LNCS*, 2013.

[LRST09] Didier Lime, Olivier H. Roux, Charlotte Seidner, and Louis-Marie Traonouez. Romeo: A parametric model-checker for Petri nets with stopwatches. volume 5505 of *LNCS*, pages 54–57, York, United Kingdom, 2009. Springer.

[Mar11] Nicolas Markey. Robustness in real-time systems. In *SIES*, pages 28–34. IEEE Computer Society Press, 2011.

[SWB98] Gary Shao, Rich Wolski, and Fran Berman. Performance effects of scheduling strategies for master/slave distributed applications. In *PDPTA*, volume 99, 1998.

APPENDIX

A. ALGORITHMS

A.1 The Non-Distributed Cartography

We recall the original cartography algorithm from [AF10] in Algorithm 1. We extend the $\models$ notation as follows:- given a set S of constraints, we write $\pi \models S$ if there exists some K in S such that $\pi \models K$.

Algorithm 1: Behavioral Cartography $BC(\mathcal{A}, D)$

input : PTA $\mathcal{A}$, point π
output: Set of tiles S

$S \leftarrow \{\}$
foreach *integer point* $\pi \in D$ **do**
  if $\pi \not\models S$ **then** $S \leftarrow S \cup \{IM(\mathcal{A}, \pi)\}$;
return S

A.2 The Master Abstract Algorithm

We give the algorithm in Algorithm 2. We assume several functions: *receiveResult*() is an asynchronous, blocking function that waits until a result or an empty work request is received, and returns the rank n of the sending process and the resulting constraint K. *sendPoint*(n, π) sends point π to process n. *sendStop*(n) sends a stop signal to process n that signifies termination.

Algorithm 2: Distributed version of $BC(\mathcal{A}, D)$: Master

input : PTA $\mathcal{A}$, domain D
output: Set of tiles S

$S \leftarrow \{\}$
`// Normal mode`
while *there are uncovered integer points in* D **do**
  $n, K \leftarrow$ *receiveResult*()
  $S \leftarrow S \cup \{K\}$
  sendPoint$(n,$ *choosePoint*$())$
`// Finalization`
foreach *process* $n \in \{1, \ldots, N\}$ **do**
  $n, K \leftarrow$ *receiveResult*()
  $S \leftarrow S \cup \{K\}$
  sendStop(n)
return S

B. ADDITIONAL EXPERIMENTAL DATA

We give additional details on the 2 use-cases in Table 1. The columns give (from left to right) the use-case name, the number of parameters, of clocks, the number of integer points within the bounded domain, and the (expected) number of tiles.

Tables 2 and 3 present the summary of our experiments, for our use-cases and 2 to 24 processes. The columns of each table present (from left to right) the number of processes, the point selection algorithm, the total number of constraints that are computed, the total execution time, the percentage of time spent waiting by the master, the average percentage of occupation time of the workers and the standard deviation (a high standard deviation can evoke some load unbalance) and the time spent by the master selecting the next point it will send.

Case study	$\lvert P\rvert$	$\lvert X\rvert$	$\lvert D\rvert$	Tiles
Simop	2	8	10,201	48
Sched3	2	13	286	59

Table 1: Information on the use-cases

NP	Selection	Constraints	Total time	Master wait.	W. occ. (mean/std dev)	Next point time
2	seq	48	97.75 s	96.68 %	96.68 %/0	1.38 s
3	seq	80	86.51 s	96.68 %	96.89 %/0.21	2.24 s
4	seq	107	75.38 s	97.28 %	96.72 %/0.45	3.08 s
5	seq	132	72.30 s	97.37 %	96.86 %/0.31	3.70 s
6	seq	152	68.79 s	97.11 %	96.80 %/0.35	4.29 s
7	seq	178	67.39 s	97.27 %	96.86 %/0.58	5.06 s
8	seq	195	65.35 s	97.10 %	96.84 %/0.49	5.68 s
9	seq	209	62.16 s	97.29 %	96.86 %/0.51	6.04 s
10	seq	224	60.70 s	96.50 %	96.96 %/0.52	6.28 s
12	seq	244	55.09 s	96.99 %	96.90 %/0.62	6.99 s
14	seq	269	50.01 s	97.01 %	96.28 %/1.91	7.81 s
16	seq	281	45.24 s	96.99 %	96.30 %/1.16	7.51 s
18	seq	297	43.20 s	97.33 %	95.99 %/1.64	7.77 s
20	seq	313	41.45 s	97.14 %	95.40 %/1.99	8.35 s
22	seq	325	40.39 s	96.84 %	94.98 %/2.39	8.19 s
24	seq	342	40.43 s	97.26 %	93.45 %/4.42	9.01 s
2	random10				97.01 %/0	
3	random10	62	64.40 s	96.12 %	96.97 %/0.86	1.46 s
4	random10	80	57.12 s	97.74 %	97.18 %/0.40	1.56 s
5	random10	80	44.94 s	97.97 %	97.31 %/0.68	1.59 s
6	random10	96	43.50 s	97.79 %	97.33 %/0.82	1.74 s
7	random10	116	43.45 s	97.72 %	97.50 %/0.24	1.64 s
8	random10	114	38.47 s	97.75 %	97.62 %/0.30	1.59 s
9	random10	107	32.35 s	97.79 %	97.40 %/1.34	1.93 s
10	random10	142	37.94 s	97.72 %	97.53 %/0.23	1.97 s
12	random10	135	30.92 s	97.72 %	97.47 %/0.36	2.17 s
14	random10	153	28.06 s	97.71 %	97.06 %/0.95	2.38 s
16	random10	170	27.15 s	97.77 %	96.55 %/1.85	2.41 s
18	random10	149	22.48 s	97.63 %	95.75 %/2.57	2.25 s
20	random10	171	23.17 s	97.76 %	96.66 %/1.63	2.40 s
22	random10	206	25.21 s	97.23 %	94.89 %/3.50	2.63 s
24	random10	213	23.98 s	97.72 %	96.57 %/1.31	2.49 s
2	random20	49	100.73 s	96.93 %	96.93 %/0	1.29 s
3	random20	61	63.30 s	96.97 %	97.14 %/0.17	1.21 s
4	random20	66	47.04 s	96.90 %	97.20 %/0.49	1.35 s
5	random20	78	42.33 s	97.70 %	97.34 %/0.40	1.28 s
6	random20	88	40.89 s	96.50 %	97.33 %/0.71	1.68 s
7	random20	98	39.50 s	96.72 %	97.38 %/0.46	1.76 s
8	random20	95	32.74 s	97.81 %	97.51 %/0.37	1.54 s
9	random20	102	31.46 s	97.77 %	97.55 %/0.32	1.61 s
10	random20	105	29.31 s	97.47 %	97.51 %/0.30	1.63 s
18	random20	163	23.98 s	97.74 %	97.15 %/0.86	2.29 s
20	random20	169	22.16 s	97.87 %	96.20 %/2.16	2.31 s
22	random20	141	18.30 s	97.66 %	94.12 %/4.44	2.40 s
24	random20	185	21.40 s	97.96 %	93.97 %/4.04	2.58 s

Table 2: Detailed profiling data of parallel executions of IMITATOR on the Simop use-case

NP	Selection	Constraints	Total time	Master wait.	W. occ. (mean/std dev)	Next point time
2	seq	59	39.51 s	98.47 %	98.5 %/0	0.05 s
3	seq	62	22.26 s	98.47 %	98.5 %/0.04	0.05 s
4	seq	69	16.96 s	98.47 %	98.5 %/0.05	0.05 s
5	seq	77	14.12 s	98.38 %	98.4 %/ 0.08	0.06 s
6	seq	85	13.12 s	98.48 %	98.4 %/0.09	0.07 s
7	seq	93	12.04 s	98.45 %	98.4 %/0.07	0.08 s
8	seq	98	11.29 s	98.46 %	98.5 %/0.09	0.08 s
9	seq	103	10.49 s	98.53 %	98.4 %/0.08	0.09 s
10	seq	107	9.85 s	98.45 %	98.3 %/0.08	0.10 s
12	seq	113	8.74 s	98.29 %	98.3 %/0.17	0.10 s
14	seq	126	8.14 s	98.23 %	98.2 %/0.20	0.12 s
16	seq	130	7.23 s	98.08 %	98.1 %/0.19	0.13 s
18	seq	142	7.00 s	97.73 %	97.7 %/0.37	0.13 s
20	seq	153	6.84 s	97.41 %	97.7 %/0.40	0.15 s
22	seq	158	6.47 s	98.02 %	98.1 %/0.25	0.14 s
24	seq	163	6.26 s	98.47 %	98.1 %/0.36	0.15 s
2	random10	60	40.47 s	98.54 %	98.54 %/0	0.03 s
3	random10	64	22.93 s	98.63 %	98.60 %/0.03	0.03 s
4	random10	71	16.70 s	98.41 %	98.44 %/0.03	0.04 s
5	random10	77	14.19 s	98.50 %	98.51 %/0.02	0.04 s
6	random10	75	11.62 s	98.52 %	98.52 %/0.03	0.05 s
7	random10	82	10.65 s	98.59 %	98.51 %/0.09	0.05 s
8	random10	81	9.38 s	98.54 %	98.53 %/0.06	0.06 s
9	random10	76	7.67 s	98.30 %	98.42 %/0.11	0.04 s
10	random10	93	8.47 s	98.45 %	98.42 %/0.09	0.07 s
12	random10	90	6.92 s	98.53 %	98.48 %/0.08	0.06 s
14	random10	83	5.27 s	97.76 %	97.91 %/0.20	0.05 s
16	random10	89	4.91 s	96.80 %	97.48 %/0.58	0.07 s
18	random10	104	5.23 s	98.01 %	97.81 %/0.30	0.08 s
20	random10	114	5.09 s	97.47 %	98.23 %/0.39	0.06 s
22	random10	96	4.09 s	98.15 %	97.79 %/0.52	0.09 s
24	random10	104	4.25 s	97.23 %	98.03 %/0.55	0.08 s
2	random20	60	41.04 s	98.56 %	98.56 %/0	0.04 s
3	random20	65	22.18 s	98.47 %	98.45 %/0.02	0.04 s
4	random20	66	15.45 s	98.45 %	98.40 %/0.04	0.04 s
5	random20	70	12.68 s	98.32 %	98.47 %/0.09	0.05 s
6	random20	76	11.49 s	98.52 %	98.47 %/0.05	0.05 s
7	random20	76	9.83 s	98.53 %	98.46 %/0.11	0.07 s
8	random20	75	8.44 s	98.52 %	98.44 %/0.08	0.06 s
9	random20	82	8.25 s	98.43 %	98.48 %/0.08	0.08 s
10	random20	80	7.29 s	98.42 %	98.48 %/0.08	0.06 s
12	random20	89	6.87 s	98.53 %	98.39 %/0.19	0.08 s
14	random20	88	5.64 s	98.43 %	98.22 %/0.17	0.07 s
16	random20	86	4.91 s	98.14 %	97.98 %/0.21	0.07 s
18	random20	99	5.04 s	96.87 %	97.77 %/0.42	0.11 s
20	random20	100	4.65 s	97.67 %	97.57 %/0.64	0.09 s
22	random20	98	4.23 s	98.64 %	97.87 %/0.63	0.07 s
24	random20	106	4.24 s	98.2 %7	97.72 %/0.69	0.08 s

Table 3: Detailed profiling data of parallel executions of IMITATOR on the Sched 3 use-case.