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 [3], 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 [4] (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.

Related Work.

Formal verification can be made in parallel in two ways: modeling languages can be designed to be easy to use in...

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

EuroMPI/ASIA ’14, September 9-12 2014, Kyoto, Japan
Copyright 2014 ACM 978-1-4503-2875-3/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642769.2642784.
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 [13]. PKind [14] 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. PRE chore [8] 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 [9] and exploration is made in parallel. More recently, two 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 [13]. PKind [14] 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. PRE chore [8] 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 [9] and exploration is made in parallel.

Outline.
Section 2 recalls the necessary preliminaries and states our main objective. Section 4 proposes distributed algorithms for parameter synthesis. Section 5 gives our practical solution relying on MPI. Section 6 compares our algorithms experimentally. Section 7 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 $\pi$ is a function $\pi : P \rightarrow \mathbb{Q}^+$. We will often identify a valuation $\pi$ with the point $(\pi(p_1), \ldots, \pi(p_M))$. An integer point is a valuation $\pi : P \rightarrow \mathbb{N}^M$. 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 $\pi$ and a constraint $K$, we write $\pi \models K$ if $\pi$ satisfies $K$; geometrically speaking, this corresponds to the fact that the polyhedron $K$ contains the point $\pi$.

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 $A$, we call its discrete behavior the set of all possible alternating sequences of control states and actions allowed in $A$. Parametric Timed Automata (PTA) [1] 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 [6], Roméo [16], SpaceEx [11], or Imitator [1]). (For syntax and semantics of (P)TA, see, e.g., [12]). Given a PTA $A$ and a point $\pi$, we denote by $A[\pi]$ the (non-parametric) TA where each parameter $p_i$ was replaced with $\pi(p_i)$.

2.3 The Inverse Method
In [2], the inverse method (IM) was proposed: given a PTA $A$, this semi-algorithm takes advantage of a reference point $\pi_0$, 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 $\pi'$ satisfying $K$, the discrete behavior of $A[\pi']$ is equal to the discrete behavior of $A[\pi_0]$. As a consequence, any linear-time property (expressed using, e.g., LTL) valid for $A[\pi_0]$ is also valid for $A[\pi']$, 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., [17]) 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., [6] 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 [1]. 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 $A$ and $\pi$, different calls to $IM(A, \pi)$ may yield different constraints incomparable with each other. This situation is graphically depicted in Figure 1b: a first execution of $IM(A, \pi)$ gives $K$, whereas a second execution may give $K'$. 3) Non-confluence implies non-completeness: there may exist points $\pi'$ outside $IM(A, \pi)$ such that $A[\pi]$ and $A[\pi']$ have the same discrete behavior. Again, in Figure 1b, if $IM(A, \pi)$ gives $K$, then any point $\pi'$ in $K' \setminus K$ has the same discrete behavior as $A[\pi]$ 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 [3] into the behavioral cartography algorithm (BC): given a PTA $A$ 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 $A$. 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
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. When all points are enumerated, the algorithm returns the set of tiles.

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.

choosePoint() is a 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 16, where points \( \pi_1, \pi_2, \pi_3 \) may yield the same tile \( C \). In the remainder of this section, we will consider two Master–Workers algorithms, that will instantiate \( \text{choosePoint}() \).

### 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 \( \text{choosePoint}() \) 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 16 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 \( \text{choosePoint}() \) 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 (\( \text{max} \)) 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 \( \text{max} \) 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

\( \text{IMITATOR} \) [4] 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) [7] for polyhedra operations. Until this work, \( \text{IMITATOR} \) was entirely sequential, i.e., single process and single-threaded. In order to implement the algorithms of Section 3 and use \( \text{IMITATOR} \) in a distributed setting, we had to significantly modify its internal structure.

#### Message Passing

We distributed \( \text{IMITATOR} \) using the Message Passing Interface [10] [12], which is the \textit{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 [3], a package featuring OCaml bindings for MPI functions.

MPI send and receive functions use a \textit{tag} to identify messages and \textit{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 \textbf{wildcards}, i.e., receive from \textit{any_source} and/or with \textit{any_tag}. The actual sender and/or the tag are found in a data structure filled by the receive function: its \textit{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 \textit{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 \textit{any_source} wildcard in reception on the master’s side. However, we discovered that OCamlMPI had an unexpected behavior related to this \textit{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.

### 5. EXPERIMENTAL VALIDATION

We measured the performance of the distributed \( \text{IMITATOR} \) 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

\( ^3 \text{Source. binaries, models and results are available at } \text{http://www.lipn.fr/~andre/patator/} \)
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.

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) machines. We proposed a first parallel depth-first search algorithm for LTL in [9], it was then extended to emptiness checking of timed Büchi automata in [15]. The next step will be to extend these algorithms to parameter synthesis.

7. REFERENCES

Figure 3: Experimental results


