Babylon: An Integrated Toolkit for the Specification and Verification of a Class of Parameterized Systems

Pierluigi Ammirati[1] Giorgio Delzanno[1] Pierre Ganty[2]
Gilles Geeraerts[2] Jean-François Raskin[2] ⋆ ⋆⋆ Laurent Van Begin[2] ⋆⋆⋆

[1] Dipartimento di Informatica e Scienze dell'Informazione,
Università di Genova, via Dodecaneso 35, 16146 Genova, Italy
giorgio@disi.unige.it

[2] Département d'Informatique
Université Libre de Bruxelles, Blvd Du Triomphe, 1050 Bruxelles, Belgium
jraskin@ulb.ac.be

Keywords. Parameterized Verification, Symbolic Data Structures, Benchmarks

1 Background, Motivations and New Contributions

Automated verification of properties of *parameterized* concurrent systems, i.e., systems consisting of an arbitrary number of processes, is an important and challenging research goal. Parameterized verification problems arise in several different contexts like *multiprocessors systems*, *distributed databases*, *client-server protocols*, and *modern multi-threaded programming* (e.g. using Java). During the five last years, a lot of effort has been put in studying the theoretical aspects of the verification of parameterized systems. These verification problems are infinite state in nature and the borderline between decidability and undecidability has been studied carefully [1, 8–10, 13]. Decidability results are known for interesting sub-classes of parameterized systems (see e.g. [1, 9]). In particular, a large class of safety properties are decidable for systems consisting of many copies of finite-state machines without identities [9]. In our framework, we focus on parametrized systems that can be analysed by making a *counting abstraction*. In this case, identities of components are lost in the analysed abstract model. So, our specification language will not be able to represent parametrized systems that have a particular topology for example. For systems with such topologies, there are recent (mainly theoretical) works for their verification using regular model-checking [4].

Similarly to the finite-state case, verification of parameterized systems suffers from the *symbolic state explosion* problem, i.e. during the analysis, the size of the symbolic structures used to represent state space tends to become unmanageable by a machine. To attack this problem, in the last years we aimed our research efforts at (1) finding efficient data structures to support abstract algorithms, e.g. the algorithm of Esparza-Finkel-Mayr [5, 7], (2) extending the algorithms to more general classes of parameterized systems (e.g. with asynchronous rendez-vous and global variables) that can be used in practical applications, e.g., as abstract models for cache coherence protocols and multithreaded Java programs [6, 7].

To make our technology usable, we have designed a specification language dedicated to the description of that restricted class of parameterized systems. This language has been linked with the more general model of *Counter Machines* and to a verification engine implementing our symbolic structures as well as other efficient symbolic structures for exploring the state spaces of infinite state systems. The Babylon [2] project is a collaboration (via student exchanges) between the Computer Science Department of the University of Bruxelles and Genova, involving two senior

⋆ This author was partially supported by a "Crédit aux chercheurs", Belgian National Fund for Scientific Research.

⋆⋆ This author was partially supported by a FRFC grant 2.4530.02.

⋆⋆⋆ This author was supported by a Walloon Region grant "First Europe".

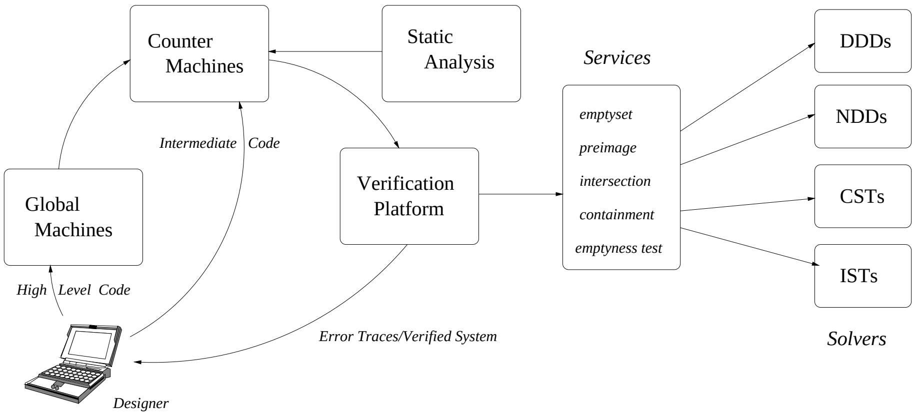

Fig. 1. Tool Architecture

researchers, one PhD student, and several master students and undergraduate students. Our main goal is to transfer the concepts and techniques studied in [5–7] into a new practical tool specialized in parameterized verification and to offer a platform to compare symbolic structure for the verification of parametrized and infinite state systems.

In this paper we will discuss the important aspects of our project and toolkit.

2 The Architecture of BABYLON

In Fig. 1 we depict the current architecture of BABYLON. The main input language of the toolkit is the language of *Global Machines*, this high level language is first translated in the lower level language of *Counter Machines* and then analyzed by a static analyzer and a verification platform. The static analysis of the low level specifications are used to improve the efficiency of fixpoint computations. The verification platform implements Pre and Post fixpoint computations on different symbolic structures able to represent infinite sets of integer tuples. We now give some more details about the different components of this architecture.

Global Machines: a Specification Language for Parameterized Systems. BABYLON provides two specification languages for networks of processes. **Global Machines** form a high-level language to describe *networks* of finite-state machines communicating via *rendez-vous*, *broadcast* and *asynchronous rendez-vous* [7]. The specification language also provides global boolean variables often useful to model global components of parametrized systems, e.g., as *global locks* to guarantee mutual exclusion properties. Individual processes are specified via a generic template, called *local machine*, in which we declare the set of *local states* and the set of possible transitions common to every of its instances [1]. Global Machines (formally defined in [7]) generalize previously introduced formalisms for parametrized systems, e.g. broadcast protocols [8]. In Fig. 2, we give an example of high level specification. Local machines describe the behavior of a *central server* (that controls the access to a shared resource), and of a generic client. The clients and the server communicate via rendez-vous (e.g. actions labeled with $!req$, and $?req$) and broadcast (the invalidation-phase labeled with $!!inv$ and $??inv$). Via a broadcast the state of a collection of clients is synchronously updated (i.e. all those in state *Use* move to *Null*). In the initial global states we declare that our system has a unique server (#Server=**1**), but may have an arbitrary number of clients (#Client=**any**). For this protocol, we would like to check that no configuration in which more than one client is

[1] Note that dynamic creation of processes can be simulated by having *dummy* local states where we have any local machines at the beginning. Dinamic creation of one process consists to move one corresponding local machine from a dummy state to a state corresponding to the beginning of the behaviour of the created process.

Global Machine Client-Server Protocol:
begin
Local Machine Server:
begin
States Idle, Serve, Busy;
Transitions
Idle $\xrightarrow{req?}$ Serve;
Serve $\xrightarrow{inv!!}$ Busy;
Busy $\xrightarrow{ack!}$ Idle;
Initial state Idle;
end;
Local Machine Client:
begin
States Null, Wait, Use;
Transitions
Null $\xrightarrow{req!}$ Wait;
Use $\xrightarrow{inv??}$ Null;
Wait $\xrightarrow{ack?}$ Use;
Initial state Null;
end;
Initial Global State #Server=1; #Client=**any**;
Unsafe States #Use $\geq$ 2;
end.

$\Rightarrow$

Counter Machine
Vars 6
Rules
$x_n \geq 1, x_i \geq 1 \;\rightarrow$
$x_n = x_n - 1 \quad x_w = x_w + 1$
$x_i = x_i - 1 \quad x_s = x_s + 1$
$x_s \geq 1 \;\rightarrow$
$x_s = x_s - 1 \quad x_b = x_b + 1$
$x_n = x_n + x_u \quad x_u = 0$
$x_b \geq 1, x_w \geq 1 \;\rightarrow$
$x_w = x_w - 1 \quad x_u = x_u + 1$
$x_b = x_b - 1 \quad x_i = x_i + 1$
Initial state
$x_i = 1, x_s = 0, x_b = 0,$
$x_n \geq 1, x_w = 0, x_u = 0$
Target state $x_u \geq 2$

Fig. 2. From High-level to Intermediate Code

in state *Use* is reachable (with #Use $\geq$ 2 we express the violations to a *safety property*) and this for any number of clients in the system. Note this is a so-called upward-closed property. For those properties, the iterations of the Pre operator defined on global machines is guaranteed to terminate.

Counter Machines as Low Level Language. **Counter Machines** are a lower level formalism that we use as target of a translation from global machines via a *counting* abstraction. These Counter Machines are composed by some integer counters and transitions wich make some operations on the counters. These operations can be any linear transformations. Intuitively, we use a finite collection of counters (as many as the number of local states and of the Boolean variables) to keep track of the number of individual processes in each local state during the execution of the Global Machine. The counter machine produced by compiling our sample Global Machine is shown in Fig. 2. The counter x_n is associated to the local state *Null*, etc. Counter machines are also interesting as they are a widely used formalism to describe programs that manipulate integer variables. Thus, having chosen Counter Machines as low level input language, allow us to experiment with research ideas, originally introduced for parametrized verification, in the context of the broader class of integer systems.

A Multi-Solver Constraint-based Verification Engine The **Verification Platform** takes in charge the verification of safety properties of models described in one of the input languages. Our platform makes use of the *services* provided by a common interface that lies in between the model checking algorithm and a collection of different symbolic structures for handling *sets of tuples of integer*. The solver we have integrated in our architecture are: the **CSTs** library [7] and its extension **ISTs** to interval constraints, the **DDDs** library of [15], and the automata-based **NDDs** library of [21]. The services provided by the different solvers consist of a set of symbolic operations (*intersection*, *containment* test, *preimage* and *postimage* computation, etc) needed by symbolic backward and forward reachability algorithms. The verification platform considers them as *oracles* that implement the corresponding set operations. The efficiency of the algorithms is improved by heuristics and optimizations that, when possible, are formulated independently from the chosen constraint solvers. For example, we use a static analysis, based on the results the Structural Theory of Petri Nets [20], to compute invariants of Counter Machines and use those invariants in order to cut, often dramatically [6, 7], the search space during backward explorations of the state spaces of those Counter Machines. Formulating the heuristics at this level allows for a *fair* comparison of the efficiency of the different solvers when executed on the same verification problem. The constraint solvers we incorporate in our architecture represent currently the most sophisticated technology that can be used to efficiently manipulate *large collections* of linear arithmetic constraints over integers.

Covering Sharing Trees (CSTs for short) have been introduced by Raskin and Delzanno in [5] as an extension of Sharing Trees [22]. CSTs have been specially developped to represent *upward-closed sets* of tuples of integers. Their efficiency for the verification of concurrent systems has been demonstrated (see [6, 7]). In a CST, the *generators* of the upward-closed sets are stored in a Sharing Tree, hence allowing the *sharing* of the common prefixes and suffixes of these generators. Though most of the operations on CSTs have exponential complexity, they behave well in practice. Moreover, polynomial-time approxmations have been developped for some of them. CSTs are closed under union and intersection but not under complementation.

Interval Sharing Trees (ISTs) are an extension of CSTs, allowing to specify both lower and upper bounds on the variables (where the upper bounds are implicitely always equal to $+\infty$ in a CST). ISTs allow to store finite unions of integer-valued intervals. ISTs are closed under union and complementation.

Difference Decision Diagrams (DDDs) have been developped by Møller and Lichtenberg [14] to store reachable states of timed systems [17]. DDDs allow to represent constraints on differences of real-valued or integer valued clocks. They are slightly more expressive than ISTs. The structure of DDDs is similar the one of Binary Decision Diagrams. Each node simply tests a condition of the form $x - y < c$ or $x - y \leq c$ (where x and y are real-valued variables, and c is a real-valued or integer valued constant). Like BDDs, DDDs can be *ordered* and *reduced* to normal form. All the boolean operations on DDD have polynomial-time complexity and keep the closure.

Number Decision Diagrams (NDDs) have been introduced by Boigelot and Wolper [3]. NDDs allow to store sets of tuples of integers that can be caracterized by a Pressburger arithmetic expression. Simply putting, a NDD representing a set U is a *finite state automaton* that accepts all the tuples (under a given encoding) belonging to U, and rejects the others. The operations on NDDs are thus the well-known operations on automata.

Other Data Structures : The design of BABYLON allow us to easely add new data structures in the verification platform. Such another data structures could be for example the IDDs developped by Strehl and Thiele wich are BDD-like structure where intervals are accociated to edges (see [19]). In [18], it is shown that a data structure representing convex polyhedra with Hilbert's basis seems also interesting for model-checking of parametrized systems. This kind of data structure has been already compared with the library used in HYTECH [12], see [18], and it could be interesting for example to make a comparison with the NDD library.

A Possible Application: Validation of Multithreaded JAVA *programs.* JAVA is nowadays the standard programming language for multithreaded programs. Those programs often offer services that must be used concurrently by any number of clients. Multithreaded JAVA programs are thus natural candidates for parameterized verification. When defining Global Machines we had in mind the verification of that class of programs. The communication mechanisms offered by Global Machines are exactly what is needed to model synchronization skeletons of typical multi-threaded JAVA programs. In the next future, we will design an abstract interpreter able to extract automatically from JAVA source code abstract communication skeletons expressed as Global Machines, and thus analyzable by our BABYLON toolkit. In [7], we have analyzed using CSTs several examples of communication skeletons that we have extracted by hand from JAVA programs.

2.1 Benchmarks, Expected Results and Deadlines

Automatic verification of infinite state systems is not yet as mature as finite state verification. Not so much test cases are currently available to evaluate infinite state system verification engines and new research ideas. So, a secondary goal of our project is to collect as much concrete examples of infinite state or parameterized systems as possible (see the *benchmarks* page [2] for a first series of examples). Those examples will be useful to make a comparison of the usability, flexibility, robustness, and efficiency of the different symbolic structures. We have already tested with success our **CSTs** on a large set of examples, see [6, 7]; those examples are mainly extensions of infinite

Case Study	P	T	H	I	CST	IST	DDD	NDD
Inc/Dec	32	28		10	9.77s	11.86s	43.89s	×
Inc/Dec	32	28	√	10	0.82s	0.73s	↑↑	—
P/C	18	14		25	44.49s	72.62s	↑↑	×
P/C	18	14	√	1	0s	0s	0.34s	—
2P/2C$_1$	44	37		14	38m	1h17m	↑↑	×
2P/2C$_1$	44	37	√	14	2.86s	4.74s	↑↑	—
2P/2C$_2$	44	37		29	55m	1h41m	↑↑	×
2P/2C$_2$	44	37	√	1	0s	0s	0.2s	—

Fig. 3. Execution time results on an AMD Athlon 900Mhz 500Mbytes: **P**=n. places, **T**=n. transitions; **H**=pruning via invariants;**S**=n. iterations before reaching the fixpoint/finding a bug;

Case Study	P	T	H	I	CST	IST	DDD	NDD
Inc/Dec	32	28		10	3520kb	3564kb	76336kb	×
Inc/Dec	32	28	√	10	2572kb	2416kb	↑↑	—
P/C	18	14		25	6444kb	12952kb	↑↑	×
P/C	18	14	√	1	1776kb	1488kb	83504kb	—
2P/2C$_1$	44	37		14	26740kb	71192kb	↑↑	×
2P/2C$_1$	44	37	√	14	3564kb	5136kb	↑↑	—
2P/2C$_2$	44	37		29	31876kb	95824kb	↑↑	×
2P/2C$_2$	44	37	√	1	1500kb	1536kb	83860kb	—

Fig. 4. Memory Usage Results on an AMD Athlon 900Mhz 500Mbytes: **P**=n. places, **T**=n. transitions; **H**=pruning via invariants;**S**=n. iterations before reaching the fixpoint/finding a bug;

state Petri nets and infinite state abstractions of JAVA programs. Those examples often contains much more than 30 integer variables and can not be handled by tools like HYTECH[2]. Fig.3 and Fig. 4 show first experimental results. As the DDD library allocates all the usable memory (75MB) at the beginning, ↑↑ means that the library exhausted all its available nodes. × means that the library run out of 5 hours. These experiments correspond to the computation of the applications of the Pre operation starting from an upward-closed set of *bad* states on abstractions of JAVA programs [3]. We also use structural invariants [20] to prune the search space as described in [6]. As a first result, it seems that NDDs are not a good alternative for solving this kind of problem as the NDD library never permit to reach the fixpoint. We can also see that CSTs and ISTs seems to need less memory than DDDs. For the P/C example without using the heuristic on structural invariants, CSTs and ISTs use less than 13MB of memory but the DDD library stops after some iterations because no nodes are available anymore, wich means that at least 75MB are used to store DDDs.

References

1. P. A. Abdulla, K. Cerāns, B. Jonsson and Y.-K. Tsay. General Decidability Theorems for Infinite-State Systems. In *Proc. LICS'96*, pp. 313–321, 1996.
2. The BABYLON Page: CSTs and Benchmarks for Parameterized Systems. http://www.disi.unige.it/person/DelzannoG/CST/
3. Bernard Boigelot. *Symbolic Methods for Exploring Infinite State Spaces.* PhD thesis, Université de Liège, faculté des sciences appliquées, 1997-1998.
4. A. Bouajjani,B Jonsson,M. Nilssen,T. Touili. Regular Model Checking. In *Proc. CAV 2000*, LNCS 1855, 2000.
5. G. Delzanno, and J.-F. Raskin. Symbolic Representation of Upward Closed Sets. In *Proc. TACAS 2000*, LNCS 1785, pp. 426-440, 2000.

[2] HYTECH is a tool that has been designed to analyze efficiently hybrid systems, but it can also be used to explore the state space of counter machines.

[3] Note that reach a fixpoint is guarantee for this kind of problem.

6. G. Delzanno, J.-F. Raskin, and L. Van Begin. Attacking Symbolic State Explosion. In *Proc. CAV 2001*, LNCS 2102, 2001.
7. G. Delzanno, J.-F. Raskin, and L. Van Begin. Toward the Automated Verification of Multithreaded Java Programs. To appear in *Proc. TACAS 2002*, 2002.
8. E. A. Emerson and K. S. Namjoshi. On Model Checking for Non-deterministic Infinite-state Systems. In *Proc. LICS '98*, pp. 70–80, 1998.
9. J. Esparza, A. Finkel, and R. Mayr. On the Verification of Broadcast Protocols. In *Proc. 14th Annual Symp. on Logic in Computer Science (LICS'99)*, pp. 352–359, 1999.
10. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! TCS 256 (1-2):63–92, 2001.
11. B. Grahlmann. The PEP Tool. In *Proc. CAV'97*, LNCS 1254, pp. 440-443, 1997.
12. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: a Model Checker for Hybrid Systems. In *Proc. CAV'97*, LNCS 1254, pages 460-463, 1997.
13. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, E. Shahar. Symbolic Model Checking with Rich Assertional Languages. In *Proc. CAV '97*, pp. 424–435, 1997.
14. Jesper Møller and Jakob Lichtenberg. Difference decision diagrams. Master's thesis, Department of Information Technology, Technical University of Denmark, Building 344, DK-2800 Lyngby, Denmark, August 1998.
15. J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference Decision Diagrams. In *Proc. CSL '99*, pp. 111-125, 999. The DDDs package is avail. at the URL http://www.itu.dk/research/ddd/
16. J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Fully symbolic model checking of timed systems using difference decision diagrams. In *Proc. FloC Workshop SMC*, 1999.
17. J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. On the symbolic verification of timed systems. Technical Report IT-TR-1999-024, Department of Information Technology, Technical University of Denmark, Building 344, DK-2800 Lyngby, Denmark, February 1999.
18. T. Rybina, A. Voronkov. Using Canonical Representations of Solutions to Speed up Infinite-State Model Checking. *Submitted for publication*, 2002.
19. K. Strehl, L. Thiele. Symbolic Model Checking of Process Networks Using Interval Diagram Techniques. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'98)*, pages 686–692. 1998.
20. M. Silva, E. Teruel, and J. M. Colom. Linear Algebraic and Linear Programming Techniques for Analysis of Place/Transition Net Systems. In W. Reisig and G. Rozenberg, editors. Lectures on Petri Nets I: Basic Models. *Advances in Petri Nets*, LNCS 1491, pages 308-309. Springer, 1998.
21. P. Wolper and B. Boigelot. On the Construction of Automata from Linear Arithmetic Constraints. In *Proc. TACAS 2000*, LNCS 1785, pages 1-19, March 2000. The NDDs package is avail. at the URL: http://www.montefiore.ulg.ac.be/~boigelot/research/lash/ndd/doc.html
22. Denis Zampuniéris. *The Sharing Tree Datastructure*. PhD thesis, Université de Liège, faculté des sciences appliquées, 1997.