UNIVERSITÉ PIERRE ET MARIE CURIE
LABORATOIRE D'INFORMATIQUE DE PARIS 6

From Symbolic Verification To Domain Specific Languages

YANN THIERRY-MIEG

HABILITATION À DIRIGER DES RECHERCHES

Présentée le 7 Décembre 2016 devant le jury composé de:

Professeur Claude JARD	Univ. Nantes,	*rapporteur*
Professeur Richard PAIGE	Univ. York, UK	*rapporteur*
Professeur Jaco VAN DE POL	Univ. Twente, NL,	*rapporteur*
Professeur Ahmed BOUAJJANI	Univ. Paris Diderot	*examinateur*
Professeur Jean-Michel COUVREUR	Univ. Orléans	*examinateur*
Professeur Serge HADDAD	ENS Cachan	*examinateur*
Professeur Fabrice KORDON	Univ. Paris 6	*examinateur*
Professeur François VERNADAT	INSA Toulouse	*examinateur*

Contents

Introduction

Verification and Validation of Concurrent Systems

The complexity of large software grows faster than engineer's abilities to deal with it. In parallel software grows more common in playing critical roles in business, security, and life critical embedded systems such as cars. It is thus necessary to develop sound development methodologies that can *guarantee* compliance of an implementation with respect to a specification.

Current hardware evolution means concurrent systems are now standard, but their correct design and implementation are particularly difficult. The problem stems from the multitude of possible interleaved executions a designer must consider, leading to the dreaded *state space explosion* problem.

Verification and validation of concurrent software is thus an active and growing field, where users expect ultimately to have automatic quality assurance tools that could guarantee the *good behavior* of a given system.

Testing

The most common industrially applied approach to verification is testing, which consists in a set of runs of the system that seek to reveal a failure. The actual system under test is then executed and its outputs are analyzed by an oracle function that determines whether the test passes or fails. While tests increase the confidence in a running system, and are definitely a necessary component of any software development cycle, by nature they cannot *guarantee* correctness : failing tests do reveal errors, but even if all tests pass the system could be incorrect. Tests are also only applicable relatively late in the development cycle, since you need an implementation. For concurrent systems, testing typically has a very low coverage of possible interleavings, added to test reproducibility issues if the default system scheduler is used.

Theorem Proving

To prove a system correct, theorem proving proposes to start from a set of basic axioms to prove the properties of the system seen as a particular theorem stating correctness. In practice the user expresses what typically amounts to pre and post conditions on behavior and uses an automated theorem prover to help establish a formal proof of correctness. While the approach is very powerful and can prove properties even for large systems [Beh+99; Ler09] it requires expert users, able to formulate intermediate lemmas helpful to the proof and to understand what is

blocking the automated engine from completing the reasoning in complex cases. Such tools thus cannot be completely automated.

Exhaustive Exploration

The last broad category of approaches starts from a model of a system, in some compact form that allows to generate and explore all its possible runs (such as a program, or a behavioral model like a Petri net), then proposes to exhaustively explore this search space to prove that all possible runs of the system satisfy the provided properties.

The positive side is that this approach *can be* fully automated, and offers a very wide scope in terms of possible logics used to define *sought* or *forbidden* behaviors. Since the problems are dual of one another, the approach usually searches for behaviors of the system that exhibit a forbidden behavior. For the end-user, such counter examples can be represented as familiar execution traces or failing tests. If no counter examples are found the system is *proved* correct. The drawback is that due to large data domains and concurrent or asynchronous behaviors, the search space can be huge, even for relatively simple systems.

Abstraction and Static Analysis

Abstraction is usually needed to reduce the search space to a finite representation. An abstraction overapproximates the system behavior, with the guarantee that all runs of the system can be executed by the abstraction. We can then try to prove that the abstraction does not contain any target *bad* behavior. If the abstraction does contain bad behavior, either it is a true counter-example of the system and we can exhibit it, or it is only a possible run of the abstraction but not of the system. This means the abstraction is too coarse, but it possibly can be refined iteratively [Cla+03].

Static analysis and abstract interpretation try to avoid actually exploring all behaviors by reasoning on the structure of the input model [STC98; CC79]. With recent improvements in the use of shapes as a convex envelope of all possible behaviors [Min06], many important properties such as absence of memory errors can be proved even for large scale code used industrially.

Model-checking

In model-checking the *reachability graph* is actually explored. The assumption is that the set of potential states of the system can be reduced to a finite abstraction thereof. States then constitute nodes in the graph, and edges represent discrete event occurrences. Given such a finite representation of all behaviors, many model-checking problems are decidable.

Our contributions focus on improving the model-checking of concurrent systems. This manuscript is composed of three parts.

Outline

In the first part we develop a **symbolic kernel** composed of efficient data structures to represent state spaces. We present in chapter I hierarchical set decision diagrams an original compact data structure to represent sets of states. We then develop the theory of homomorphisms to symbolically represent edges of the state graph in chapter II.

Part B presents innovative **symbolic model-checking algorithms** built on top of this kernel and designed to take advantage of it. We first present the SLAP algorithm for more efficient hybrid LTL model-checking in chapter III. We then present a symbolic-symbolic approach to stack quotient graph reductions with decision diagrams in chapter IV.

In the third part we focus on **leveraging model-driven engineering** techniques to help bridge the gap between industrial software development process and formal methods. To this end we designed a language based front-end to formal verification tools presented in chapter V. The Guarded Action Language (GAL) is defined as a domain specific language catering to the expression of concurrent semantics. Using GAL as a target in model to model transformations, we were able to offer an efficient symbolic model-checker for a wide variety of input formalisms, as described in chapter VI.

We present some mid and long term perspectives to conclude this manuscript.

Each chapter is relatively self-contained, first introducing the specific problem and existing solutions, then giving useful definitions and notations, before presenting the meat of the contribution, and an experimental evaluation. We conclude each chapter with a discussion of the impact or significance of the contribution.

My approach to verification is a pragmatic one, hence these contributions have all been implemented, and are freely distributed as part of the *ITS-tools*.

Part A

Symbolic Kernel

Symbolic approaches reason with *sets* of objects instead of individual elements. In model-checking, the challenge is to compute and represent large state graphs, where nodes are configurations or *states* of the system and edges are events or *transitions*. Symbolic model-checking thus consists in using efficient representations of large sets of states and transitions, such as shared decision diagrams.

We develop in this part our contributions to two essential elements that form the *kernel of any symbolic model-checker* : efficient representation of states and transitions.

Chapter I presents Hierarchical Set Decision Diagrams (SDD) as an efficient data structure for symbolic representation of large sets of states.

Chapter II presents how homomorphisms can encode a high level expression of a transition relation, and how to efficiently apply homomorphisms to large sets of states.

Chapter I

Hierarchical Set Decision Diagrams

I.1 Decision Diagrams for Symbolic Model-Checking

I.1.1 Introduction

Reduced Ordered Binary Decision Diagrams (ROBDD or just BDD) are a data structure introduced in Bryant's thesis [Bry86], and first applied to model-checking in the seminal paper "10^{20} states and beyond" [Bur+92] challenging the whole community with astonishing performances on real examples of hardware design : DD can offer an extremely compact representation for very large sets of data, provided that appropriate symbolic operations are used.

Shared Decision Diagrams (DD) are a data structure to compactly represent and manipulate sets rather than individual elements, hence the term *symbolic*. There are many variants of decision diagrams used for symbolic model-checking, but they all rely on the same underlying principles: nodes of the decision tree are unique in memory thanks to a canonical representation; the number of paths through the diagram (states) can be exponential in the representation size (nodes in the DD); equality of two sets can be tested in constant time; using caches most operations manipulating a DD are polynomial in the representation size; the effectiveness of the encoding strongly depends on the chosen variable ordering [CGP99].

BDD extensions ROBDD are designed to handle binary functions of binary variables, but variants on the principle (shared decision tree, dynamic programming) lead to numerous extensions. For stochastic and quantitative model-checking, multi-terminal DD allow to represent functions mapping to a discrete domain [Her+03]. To handle variables which are integers each node can be allowed more than two children, leading to Algebraic DD [Bah+93] (ADD), Multi-way DD [MC99] (MDD), Data DD [Cou+02] (DDD), or List DD [BP08] (LDD). While the definitions and implementations of these DD differ slightly, they all introduce support for variables with a discrete domain. A multitude of over twenty DD variants have been defined in the literature (see chapter 3 of [Lin09] for an overview).

In this chapter, we will first present classical BDD and one of their extensions to

the integer domain : Data Decision Diagrams (DDD). Then we present our contribution, Hierarchical Set Decision Diagrams (SDD), an original data structure that combines decision diagrams within a hierarchical structure leading more efficient solutions.

I.1.2 Binary Decision Diagram

A reduced ordered binary decision diagram or ROBDD defined in [Bry86] offers a compact representation for boolean functions, based on a particular canonical decomposition of the function through a process called "currying". Basically, given a function $f\ \mathbb{B}^k \mapsto \mathbb{B}$ over k boolean variables $v_1 \dots v_k$, and an order over these variables $v_i < v_j \iff i < j$, we let a node n_k represent f. This node has two outgoing (decision) edges, labeled true and false leading respectively to nodes encoding the functions $f_{v_k} = f \wedge v_k$ and $f_{\bar{v_k}} = f \wedge \bar{v_k}$. Now nodes that would correspond to v_0 are either true or false, encoded by two terminal nodes. This structure is a decision diagram used in many kinds of engineering and risk management related tasks. Such a decomposition of a boolean function is canonical given the variable ordering : any boolean function with the same truth table will produce the same decision diagram.

The reduced aspect of ROBDD is introduced by the observation that there only needs to be two terminal nodes 0 and 1, if nodes of level v_1 share their representation. Hence, there need be no more than at most four nodes at level v_1, respectively encoding functions the possible functions of a single boolean variable $true, v_1, \bar{v_1}$, and $false$. Now suppose that nodes up to level i are unique and indexed (hashed), representing functions over i variables, we have a simple way to build unique nodes representing functions of $i+1$ variables, by indexing the children of a node into a unique table.

This canonical representation gives us complexity bounds based on the representation size, if we use dynamic programming (caches). In particular, union (Boolean or) and intersection (Boolean and) are quadratic in the representation size, and testing equality of two nodes (or functions) is constant time if we have built their BDD representation.

To use BDD for model-checking, we consider a system of k boolean variables, and we encode the characteristic function of sets of states i.e. a function that returns true for states in the set. We then typically try to compute the *reachable* states using a fixpoint.

I.1.3 Data Decision Diagrams

Data Decision Diagrams (DDD, defined in [Cou+02]) extend classical BDD in two respects:
1) variables are considered to have a domain D rather than being restricted to $\mathbb{B}$. This provides a natural representation for systems whose states can be described by a set of integers.
2) operations over DDD are encoded using homomorphisms instead of the usual

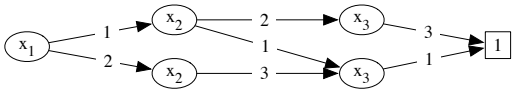


Figure I.1: A DDD with domain $D = \mathbb{N}$ that represents the set of sequences of assignments: $\{(x_1 \leftarrow 2; x_2 \leftarrow 3; x_3 \leftarrow 1;), (x_1 \leftarrow 1; x_2 \leftarrow 1; x_3 \leftarrow 1;), (x_1 \leftarrow 1; x_2 \leftarrow 2; x_3 \leftarrow 3;)\}$.

fashion where another decision diagram with two variables per variable of the state signature is used. Homomorphisms will be discussed in chapter II.

More precisely a DDD is a data structure for representing a set of sequences of assignments of the form $x_1 \leftarrow v_1; x_2 \leftarrow v_2; \ldots; x_n \leftarrow v_n$, where x_i are variables and v_i are values in D. We assume a total order on variables such that all variables are always encountered in the same order in an assignment sequence. The usual DDD definition makes weaker assumptions on variable ordering, but these are out of the scope of this presentation (see [Cou+02]).

We define the terminal **1** to represent the empty assignment sequence, that terminates any valid sequence, and **0** to represent the empty set of assignment sequences.

DDD Let X be a set of variables ranging over domain D. $\delta \in \mathbb{D}$, the set of DDD, is inductively defined by: $\delta \in \mathbb{D}$ if
$\delta \in \{\mathbf{0}, \mathbf{1}\}$ or
$\delta = \langle x, \alpha \rangle$ with $x \in X$, and $\alpha : D \to \mathbb{D}$ is a mapping where only a finite subset of D maps to other DDD than **0**.

By convention, edges that map to the DDD **0** are not represented.

For instance, consider the DDD shown in figure I.1. Each path in the DDD corresponds to a sequence of assignments. Note that contrary to BDD, DDD are not defined as representing a given function, but inductively as a tree like object; the definition of SDD will be constructed similarly. We have consistently used DDD in our work to represent valuations of the memory. We now assume that each assignment sequence in a DDD represents a state of a system.

Consider for instance the dining philosophers problem as modeled by the Petri net Fig.I.2. On Fig. I.3 (left) the set of reachable states of the system for 4 philosophers is represented as a DDD; each path gives the values of the marking for each place of the net in a specific state. Because paths can share both prefix and suffix the representation of the 322 reachable states is very compact.

I.2 Hierarchical Set Decision Diagrams

I.2.1 Intuition

While DDD are defined at a relatively high level of abstraction with respect to other DD, they still lack structure. When attempting to encode states that do have a

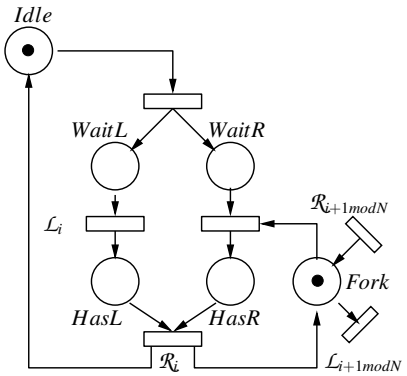


Figure I.2: **Labeled P/T net model of the philosophers**

complex structure with DDD (arrays or matrices to represent part of the state, data structures such as records, subcomponents...) we have to *flatten* this structure away to obtain a plain sequence of integers.

Looking closely at the actual DDD for concrete examples such as Fig. I.3(left) repeated substructures in the decision tree can be seen (humans are very good at pattern recognition). The repeated parts of the graph cannot however share their representation : the canonical form is computed from the terminals up to the root of the DD, so having different children nodes means the nodes are different despite having similar structure.

The solution we proposed was to add hierarchy to the data structure, yielding representations such as Fig. I.3(right). The overall structure of the DDD solution is still present, but the repeated substructure corresponding to the state of a philosopher component can be shared. The DDD that are referenced by the SDD encoding can share their representation, because the SDD part contains the information on the target children node. This phenomenon has also been referred to as implicit terminal. The resulting symbolic encoding is much smaller than pure DDD, and also scales much better when increasing the number of components.

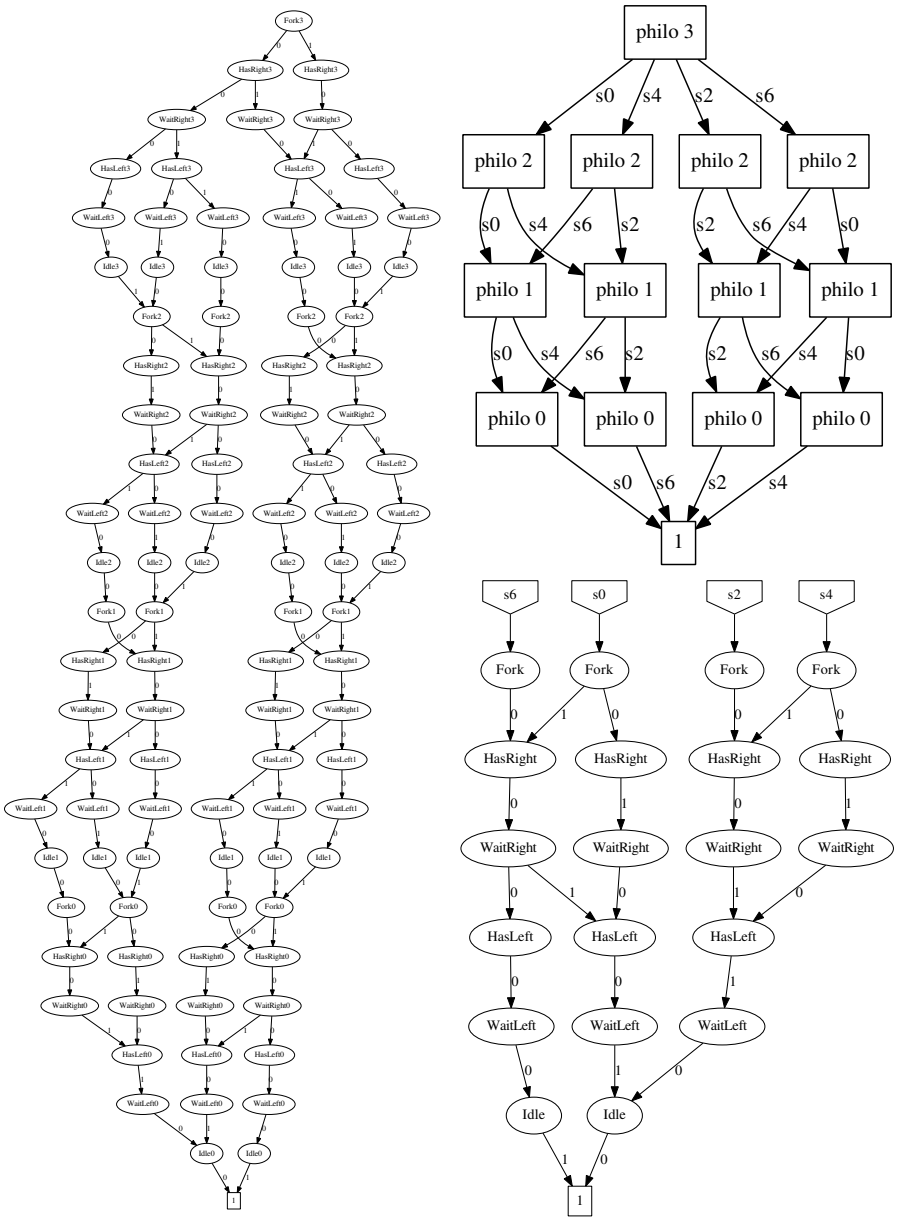
I.2.2 SDD Definition

Hierarchical Set Decision Diagrams (SDD) were first defined in [CTM05], we present a compact definition of them in this section.

SDD are shared decision diagrams in which arcs are labeled by a *set* of values, instead of a single value. This set may itself be represented by an SDD, thus when labels are SDD, we think of them as hierarchical decision diagrams.

SDD are data structures for representing sets of sequences of assignments of the form $\omega_1 \in s_1; \omega_2 \in s_2; \cdots; \omega_n \in s_n$ where ω_i are variables and s_i are sets of values.

We assume no variable ordering, and the same variable can occur several times in an assignment sequence. We define the terminal 1 to represent the empty assign-



(a) Using only DDD

(b) Using SDD (top) with labels referring to DDD (bottom)

Figure I.3: State space for 4 philosophers, encoded using DDD (left) or SDD labeled with DDD (right)

ment sequence, that terminates any valid sequence. The terminal 0 represents the empty set of assignment sequences.

Set Decision Diagram Let *Var* denotes a set of variables, and for any ω in *Var*, $\mathrm{Dom}(\omega)$ represent the domain of ω *which may be infinite*.

$\delta \in \mathbb{S}$, the set of SDD, is inductively defined by:

- $\delta \in \{0,1\}$ or
- $\delta = \langle \omega, \pi, \alpha \rangle$ with:
  - $\omega \in Var$.
  - $\pi = s_0 \cup \cdots \cup s_n$ is a finite partition of $\mathrm{Dom}(\omega)$, i.e. $\forall i \neq j, s_i \cap s_j = \emptyset, s_i \neq \emptyset, n$ finite.
  - $\alpha : \pi \to \mathbb{S}$, such that $\forall i \neq j, \alpha(s_i) \neq \alpha(s_j)$.

By convention, when it exists, the element of the partition π that maps to the SDD 0 is not represented.

Despite its simplicity, this definition supports storage of rich and complex data in a canonical form. The finite partition rule ensures that sets s_i, s_j on outgoing edges from a given node have empty intersection, otherwise we must build three edges labeled $s_i \setminus s_j, s_i \cap s_j$, and $s_j \setminus s_i$. The differing child node constraint $\alpha(s_i) \neq \alpha(s_j)$ forces to fuse two edges with the same target into a single edge labeled $s_i \cup s_j$.

SDD support domains of infinite size (e.g. $\mathrm{Dom}(\omega) = \mathbb{R}$), provided that the partition size remains finite (e.g. $]0..3],]3..+\infty]$). This feature could be used to model clocks for instance (as in [Wan04]). It also places the expressive power of SDD above most variants of DD.

The main strength of SDD is the hierarchy it allows in the data structure : any variant of decision diagram can be used to label the edges of the SDD, representing the domain of variables. We can for instance use DDD to represent components with their local integer variables and SDD to represent a composition of such subsystems. SDD can label SDD, to induce more than one level of depth in the hierarchy.

The definition of SDD also allows to handle paths of variable lengths, if care is taken when choosing the state encoding to avoid creating so-called incompatible sequences (see [CTM05]). This feature is useful when representing dynamic structures such as queues, lists or variable size arrays.

I.3 Evaluation

I.3.1 Hierarchy Helps

Experimental evaluations of SDD are numerous, since the data structure has been implemented several times and used in different tools since its inception.

Implementations

In Genève, Didier Buchs was an early adopter of the technology, he directed Steve Hosttetler's thesis [Hos11] that defines Σ-DD as a special case of SDD, to build a theory of terms and their manipulation, with heavy use of both hierarchy and variable length aspects. Continuing this work, Edmundo Lopez's thesis [LB15] builds a framework to express systems as term rewriting systems and successfully translates this input to Σ-DD for analysis. Their Java implementations of SDD are used in the tools AlPina [Buc+10] and Stratagem [LBCB14].

Jean-Michel Couvreur also pursued this subject [BCN11] with a different definition based on term rewriting systems and favorable performance results with respect to Maude.

Alexandre Hamez developed an optimized C++ SDD engine libSDD, and a model-checker for Petri nets *pnmc* well ranked in the model-checking contest.

Our own C++ libddd implementation is used in the ITS-tools which cover CTL and LTL model-checking of a variety of formalisms. It was also used in PNXDD [Cho+10b], Crocodile [Col+11] for symmetric nets with bags (SNB), Haddock [BET10] for Promela specifications.

Efficiency

Overall, SDD can be very effective when compared to other decision diagram variants, thanks to the hierarchy in the representation. Gains in representation size of one order of magnitude are the norm, not the exception. Because SDD are more compact than plain DDD, and since time complexity of operations is related to representation size in symbolic methods, performance is notably improved both in both time and memory.

To support this claim, tools using SDD or their variants are consistently well ranked in the Model-checking contest at Petri nets (silver or better in state space generation category since 2011). In recent years, Marcie that uses Interval Decision Diagrams IDD [HST09], is the only real competitor to SDD for this category.

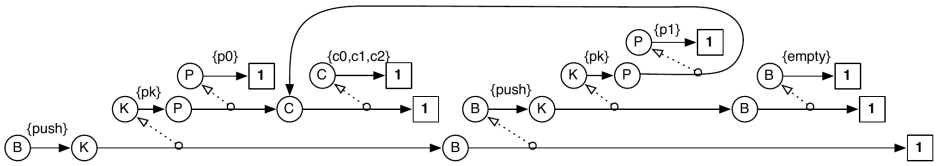
Defining Hierarchy

The issue of finding an appropriate variable order common to all DD is however now compounded by the search for a good decomposition or definition of hierarchy in the representation.

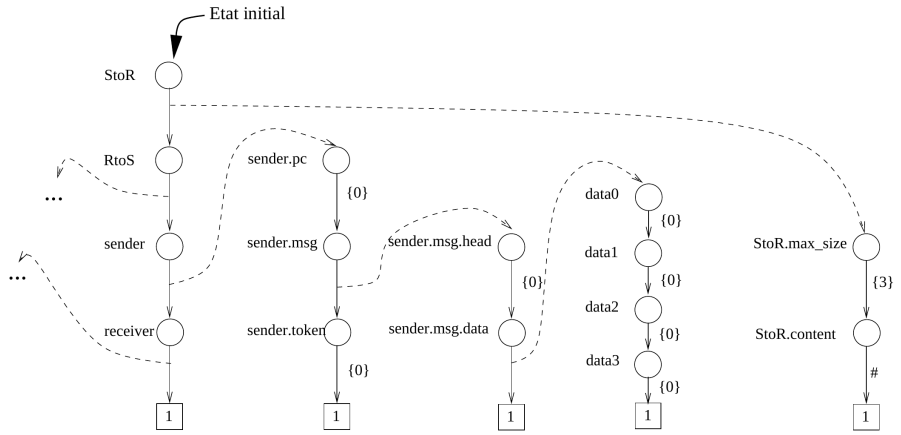
A manual definition of hierarchy can be very effective, but it requires an expert. If the model is obtained from a high level specification language, the structure of the model can be used to define hierarchy; this is the approach we currently use to infer hierarchy from colored Petri nets. Other structural information not explicitly designed for hierarchy can also give good decompositions; for instance we use the NUPN information that comes with some models of the model-checking contest to infer hierarchy. The various automaton of a network of timed automata such as Uppaal uses suggests a natural decomposition that we exploit. In [Buc+10] the algebraic structure of states is used to define their hierarchical encoding, see Fig. I.4 (top). In [BET10] for Promela models natural decompositions were obtained by matching the structure of the SDD to the data structures of the Promela code, see Fig. I.4 (middle). In [Col+11] the data structures representing states were matched

to hierarchy levels, see Fig. I.4 (bottom).

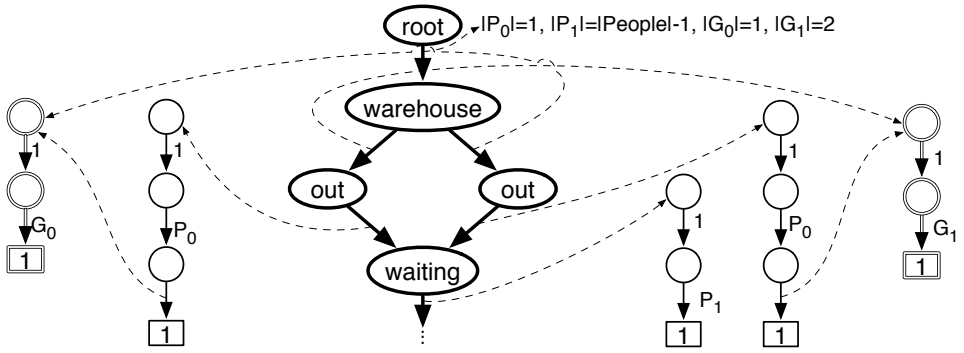
In [Hon+12] performance with and without hierarchy are compared, and use of hierarchy comes out as winning strategy even when using random hierarchical decompositions (provided they are not too deep). The heuristics developed by Silien Hong during this work are available to propose hierarchy levels.



(a) Encoding a set of algebraic terms with $\Sigma - DD$, from [Buc+10]



(b) Encoding states of Promela programs, from [BET10]



(c) Encoding symbolic states of symmetric nets with bags, from [Col+11]

Figure I.4: SDD structure can be used to encode a variety of systems

I.3.2 Exponential Examples

SDD allow to represent the states of a given system in a number of equivalent ways, depending on the hierarchy that is defined. One way of seeing this is that SDD offer to *parenthesize* a parallel composition of n variables, parenthesis being used to express hierarchy levels. Flattening the representation (removing parenthesis) is always possible, yielding a simple DDD with only integer variables. However, hierarchy allows to factorize description of similar structures and behaviors. This can be exploited to provide a more efficient SDD solution for model-checking.

We consider again the dining philosophers problem as modeled by the Petri net Fig.I.2. Let P represent the state of a single philosopher, i.e. a DDD with one variable for each place of the elementary net. We can build variables $M2 = (P /\!\!/ P)$ to represent two adjacent philosophers, then build upon that to define $M4 = (M2 /\!\!/ M2) = ((P /\!\!/ P)(P /\!\!/ P)), M8 = (M4 /\!\!/ M4) = \ldots$ We can thus model 2^n philosophers using log_2 SDD variables only.

Figure I.6 graphically shows the SDD representing the initial and final state space for 2^2 and 2^3 philosophers using such a recursive folding of the state structure.

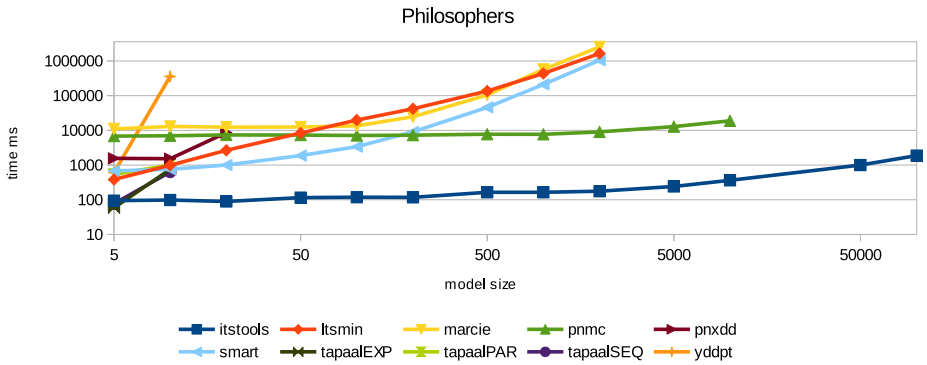


Figure I.5: **Performance charts taken from the model checking contest *mcc2016@ICATPN*. log-log plot of time for state space generation depending on number of philosophers. For** 100000 **philosophers the full state space of size** 1.33×10^{47712} **is explored in roughly one second. The curve for SDD based *its-tools* and *pnmc* is linear despite the logarithmic scale.**

This recursive encoding of the state space yields exceptionally good performance for very regular models, **exponentially more compact** than what can be obtained with other decision diagram variants. The plot of Fig.I.5 is built using data from the 2016 edition of the MCC, note the log-log scale. *Certes*, this kind of regular models correspond to best case scenarios for SDD, but also highlight the power of hierarchy and the differences of SDD with respect to other DD of the literature.

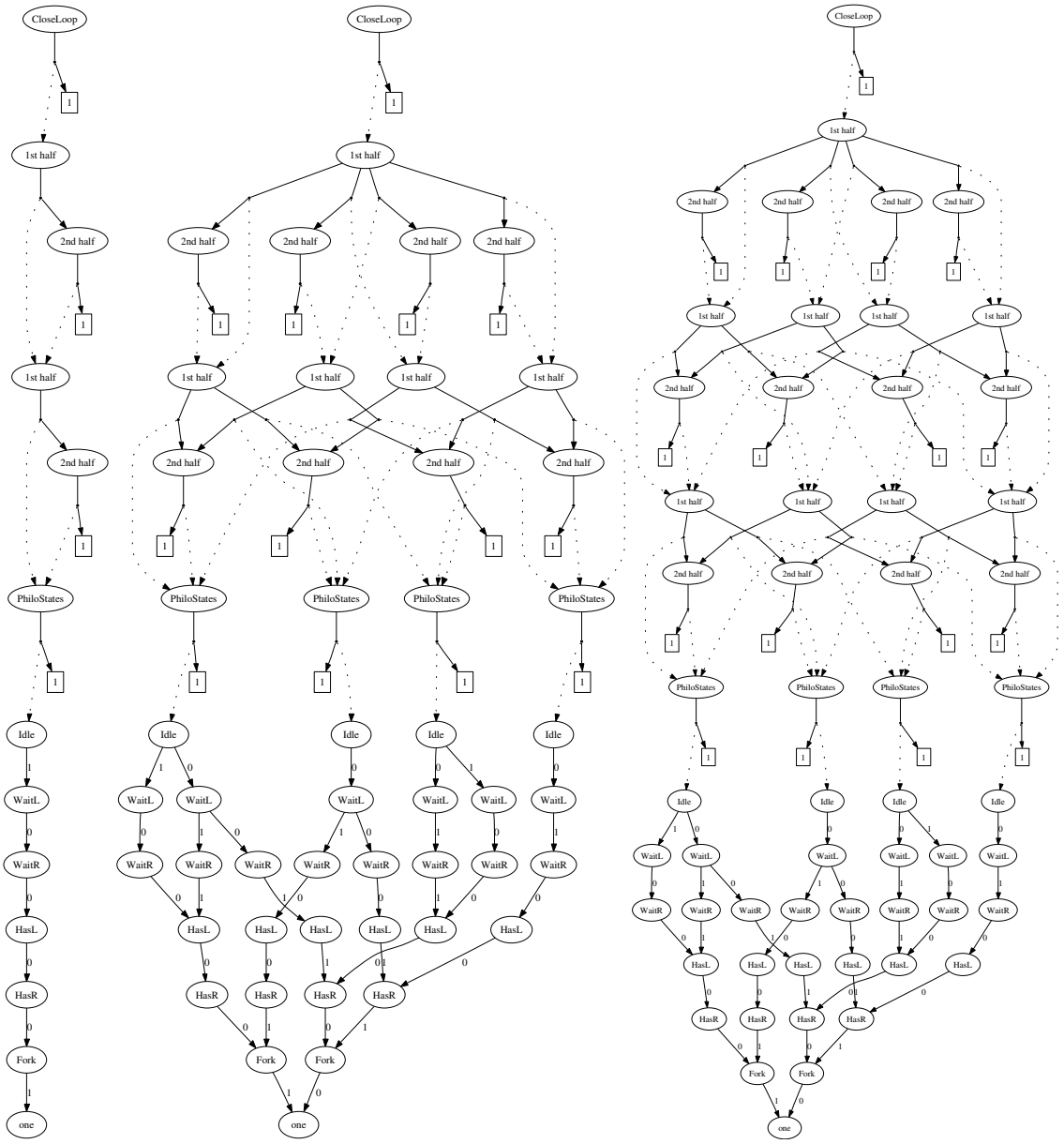


Figure I.6: Initial (left) and final states (center) of the 2^n philosophers problem for $n = 2$. Final state-space for 2^3 philosophers (right). The path that is the most to the right corresponds to a deadlock state : all philosophers have picked up their right fork and are waiting for the left one. Doubling the number of philosophers induces an additional representation cost of 8 nodes and 12 edges.

I.4 Conclusion

SDD are a fundamental part of my work on decision diagrams, the hierarchy of the encoding mostly stacks with or even synergizes very well with the other elements necessary for symbolic model-checking. It provides very efficient encodings in terms of number of nodes in the representation (memory), and since time complexity in DD is also tightly linked to representation size, very fast tools.

The adoption of SDD both within and outside my own team also show the relevance of this contribution, our seminal paper with Jean-Michel Couvreur defining SDD has accumulated 76 cites according to Google scholar. SDD are an important component of at least 8 PhD thesis (myself, V. Beaudenon, A. Hamez, A. Linard, and M. Colange in Paris 6, S. Hostettler and E. Lopez in Genève, D-T. Nguyen in Orléans) to my knowledge.

Chapter II

Homomorphisms as Symbolic Transitions

II.1 Symbolic Transition Relation

II.1.1 Introduction

This chapter presents the second critical ingredient to enable efficient symbolic model-checking : encoding and application of a transition relation to a set of states, usually until some kind of fixpoint is reached. In a symbolic setting, a transition relation takes a set of states and produces the set of states that are successors of the input states. Transition relations thus encode symbolically the edges of the reachability graph.

For the definition of symbolic operations the initial approach of [Bur+92] assumes a finite set of k (boolean) variables, giving 2^k potential states designated by S, and that transition relations are subsets of $S \times S$. The transition relation of a system of k boolean variables, can thus be seen as a function $\mathbb{B}^k \mapsto 2^{\mathbb{B}^k}$ and is usually built and stored as a second decision diagram N, with two variables "before" and "after" for each variable of the system (or unprimed v and primed variables v' in the literature). A specific operation between any subset of the state space S encoded as a decision diagram and the transition relation N yields a decision diagram $S' = N(S)$ representing immediate successors of S.

The global transition relation is then the monolithic union (logical or of behaviors) of all possible transitions. This monolithic approach matches the synchronous semantics of hardware systems, but yields intractable representations in many cases. Simply computing the DD representing N has been shown in some cases to be intractable. This forced to introduce new strategies [Ran+95], where an explicitly managed set of DD store conjuncts of the transition relation. This process, called transition clustering, allows to overcome some of the limits of the monolithic approach. This produces smaller DD, that represent the transition relation $N = \Sigma_i t_i$ in parts or clusters.

A very different approach than using a $2k-level$ DD to represent N is to use homomorphisms as introduced in [Cou+02]. They describe symbolic operations at a high level of abstraction, using a basic set of elementary transformations and an algebraic definition of more complex operations. Additional elementary transformations can be easily integrated as inductive homomorphisms. This approach is not standard, but it has many strengths that stem from the high level of abstraction at which the transition relation is described. We adopted homomorphisms with DDD, and extended their definition to SDD and in several other directions in our work.

II.1.2 Homomorphisms as Transition Relations

Homomorphisms were introduced in [Cou+02] to define operations over DDD. The definitions presented here are refined versions that correspond to our current state of the art, and cover homomorphisms for both DDD and SDD.

Both DDD and SDD support standard set theoretic operations $(\cup, \cap, \setminus)$. They also offer a concatenation operation $\delta_1 \cdot \delta_2$ which replaces 1 terminal of δ_1 by δ_2. This corresponds to a cartesian product. In addition, basic and inductive homomorphisms are introduced as a powerful and flexible mechanism to define application specific operations. The following definitions are formulated with domain $\mathbb{S}$ but they are common to DDD and SDD. To consider these definition for homomorphisms over DDD, simply assume that the set labeling the edge of the SDD is a singleton integer value.

A basic homomorphism is a mapping $\Phi : \mathbb{S} \mapsto \mathbb{S}$ satisfying $\Phi(0) = 0$ and $\forall \delta, \delta' \in \mathbb{S}, \Phi(\delta \cup \delta') = \Phi(\delta) \cup \Phi(\delta')$. The sum $+$ and the composition $\circ$ of two homomorphisms are homomorphisms.

For instance, the homomorphism $\delta * Id$ where $\delta \in \mathbb{S}$, $*$ stands for the intersection and Id for the identity, allows to select the sequences belonging to δ : it is a homomorphism that can be applied to any δ' yielding $\delta * Id(\delta') = \delta \cap \delta'$.

As another example, the homomorphism $\delta \cdot Id$, where $\delta \in \mathbb{S}$, permits to left concatenate sequences. We widely use the left concatenation of a single assignment $(\omega \in s)$, noted $\omega \xrightarrow{s} Id$ (or for $x \leftarrow d$, we note $x \xrightarrow{d} Id$ in DDD context).

Application-specific mappings can be defined by *inductive* homomorphisms. An inductive homomorphism ϕ is defined by its evaluation on the 1 terminal $\phi(1) \in \mathbb{S}$, and its evaluation $\Phi' = \phi(\omega, s)$ for any $\omega \in Var$ and any $s \subseteq \mathrm{Dom}(\omega)$. The expression $\phi(\omega, s)$ is itself a (possibly inductive) homomorphism, that will be applied on the successor node $\alpha(s)$. The result of $\phi(\langle \omega, \pi, \alpha \rangle)$ is then defined as $\sum_{s \in \pi} \phi(\omega, s)(\alpha(s))$, where $\sum$ represents a union.

Inductive Homomorphism Example Let us consider the two following inductive homomorphisms *leq* and *inc*. Both assume that their target variable x has an integer domain $\mathrm{Dom}(x) = \mathbb{N}$, so they function as DDD homomorphisms.

$$
\begin{array}{rcl}
leq(x,k)(\omega,s) & = & \begin{cases} \omega \xrightarrow{\{n \mid n \in s \wedge n \leq k\}} Id & if\, \omega = x \\ \omega \xrightarrow{s} leq(x,k) & else \end{cases} \\
leq(x,k)(1) & = & 1
\end{array}
$$

$$
\begin{array}{rcl}
inc(x)(\omega,s) & = & \begin{cases} \omega \xrightarrow{\{n+1 \mid n \in s\}} Id & if\, \omega = x \\ \omega \xrightarrow{s} inc(x) & else \end{cases} \\
inc(x)(1) & = & 1
\end{array}
$$

leq returns all assignments sequences in which all values of the variable x are less or equal than k, while $inc(x)$ increments all values of x. We can further combine these effects using the algebra of $+$ and $\circ$, for instance in Fig.II.1 we apply a composition $f_d = leq(d,2) \circ inc(d)$ that increments variable d if and only if it is strictly lesser than 2 (the *leq* test is performed *after* the increment). The SDD labeled $S_2 = f_d(S_1)$ is the result of applying f_d once on the initial state S_1.

II.1.3 Available Homomorphisms

Many basic homomorphisms are hard-coded, therefore available off-the-shelf to develop model-checking applications.

The following essential morphisms (defined in [Cou+02]) give us an algebra to work with operations, with $+$ and $\circ$ as main operators. For any $\delta \in \mathbb{S}$:

- Id the identity, $Id(\delta) = \delta$. Neutral element for $\circ$.
- (δ') the constant morphism, $(\delta')(\delta) = \delta'$. The (O) zero morphism is the absorbing element for $\circ$ and the neutral element for $+$.
- $\Phi + \Phi'$, more generally $\Sigma_i \Phi_i$ the sum, $(\Sigma_i \Phi_i)(\delta) = \bigcup_i \Phi_i(\delta)$
- $\Phi \circ \Phi'$ the composition, $(\Phi \circ \Phi')(\delta) = \Phi(\Phi'(\delta))$
- $\Phi * \delta'$ that intersects with a constant, $(\Phi * \delta')(\delta) = \Phi(\delta) \cap \delta'$
- $\Phi - \delta'$ minus operator, $\Phi - \delta'(\delta) = \Phi(\delta) \setminus \delta'$
- $\delta' \cdot \Phi$ left concatenation, $(\delta' \cdot \Phi)(\delta) = \delta' \cdot \Phi(\delta)$
- $\Phi \cdot \delta'$ right concatenation, $(\Phi \cdot \delta')(\delta) = \Phi(\delta) \cdot \delta'$

We have added the notion of **selector homomorphisms**, that respect the property $\forall \delta \in \mathbb{S}, \Phi(\delta) \subseteq \delta$. Since homomorphisms are linear this property induces that the behavior of the operation is limited to pruning some states. This property allows to define additional operators :

- $\Phi * \Phi'$ intersection of morphisms when at least one of Φ or Φ' is a selector, $(\Phi * \Phi')(\delta) = \Phi(\delta) \cap \Phi'(\delta)$.

- $!\Phi$ negation of a selector $\Phi, (!\Phi)(\delta) = \delta \setminus \Phi(\delta)$
- $ite(s,t,f)$ if-then-else control structure, where s is a selector and t and f are morphisms, $ite(s,t,f) = f \circ !s + t \circ s$

We also offer an **invert** unary operator for any morphism, that allows to compute a predecessor relation. Because we consider variables which are a priori unbounded, and that operations can destroy information, the invert can only be computed within a certain context δ_P representing potential states. However, the invert of complex morphisms is induced from the invert of the parts, so this very useful brick for model-checking is available for all transition relations.

- $\Phi^{-1}_{\delta_P}$ invert of a morphism with context δ_P, $\Phi^{-1}_{\delta_P}(\delta) = \{s \in \delta_P \mid \Phi(\{s\}) \cap \delta \neq \emptyset\}$

The **transitive closure** * unary operator allows to perform a fixpoint computation. For any homomorphism h and any node $\delta \in \mathbb{S}$, $h^\star(\delta)$ is evaluated by repeating $\delta \leftarrow h(\delta)$ until a fixpoint is reached.

- $\Phi^\star$ fixpoint or transitive closure operator, such that if $\exists n \in \mathbb{N}$, such that $\Phi^n(\delta) = \Phi^{n+1}(\delta)$, then $\Phi^\star(\delta) = \Phi^n(\delta)$

In other words, $h^\star(\delta) = h^n(\delta)$ where n is the smallest integer such that $h^n(\delta) = h^{n+1}(\delta)$. This operator is often applied to $(Id + h)$ instead of just h, allowing to accumulate newly computed assignment sequences in the result (a.k.a. least fixpoint). Besides the ease of use it offers the user, since it adds easy support for μ-calculus, it helps the user specify that a transitive closure is desired in a high level way. We can in such a context enable sophisticated rewriting optimizations (presented in section II.2) that considerably improve performance.

Specific only to SDD, we have the local construction :

- $\mathcal{L}(\Phi, \omega)$ allows to "carry" a homomorphism Φ to a certain variable ω, and apply Φ to the current state of ω. Thus, it implements an operation local to the variable ω. Any $(\omega \in s)$ in any sequence of δ becomes $(\omega \in \Phi(s))$.

Specific only to DDD, we have the following inductive homomorphisms provided off the shelf. The last two extremely general transformations use a reasoning based on equivalence classes that will be described in section II.3.

- $(x \diamond k)$ comparison of variable to a constant $k \in \mathbb{Z}$, with $\diamond \in \{=, \neq, <, >, \geq, \leq\}$, this selector inductive homomorphism selects only sequences of assignments that respect the provided predicate, i.e. $x \leftarrow k'$ in any sequence of δ must respect constraint $k' \diamond k$
- $(x+ = k)$ increment of variable by a constant $k \in \mathbb{Z}$, updates any $x \leftarrow k'$ in any sequence of δ to be $x \leftarrow k + k'$

- $Pred(b)$ where b is an arbitrary boolean predicate on state variables, this general selector homomorphism keeps only sequences that satisfy b
- $Assign(a,b)$ where a is an arbitrary expression resolving to a variable index, and b is an arbitrary expression on variables. This operation updates the value of a to reflect the value of b.

These modeling bricks are used to build up the semantics of systems in a compositional manner. With *Assign* and *Pred* offered with the library, most users do not need to develop their own homomorphisms anymore, they can easily assemble existing bricks to build practically any intended behavior.

II.2 Automatic Saturation

II.2.1 Computing a least fixpoint

A core element of symbolic model-checking is the computation of a least fixpoint or transitive closure, necessary for most queries such as reachable state space. Building upon the idea of clusters for transitions, *chaining* [RCP95] consists in relaxing the strict BFS proposed in [Bur+92] to allow to reach states that are not direct successors in a single iteration of applying the clusters. The approach applies the clusters of transitions in sequence, so that each cluster may discover successors of states reached by the previous clusters.

For Globally Asynchronous Locally Synchronous (GALS) systems, the *saturation* algorithm [CMS03] is empirically an order of magnitude better than chaining. The semantics of GALS is given as an asynchronous interleaving of locally synchronous actions (e.g. Petri nets). *Saturation* consists in constructing clusters based the top-most variable in transition supports, then based on the interleaving semantics of the conjuncts, the fixpoint is first computed on lower parts of the DD.

More precisely any time a DD node of the state space representation is modified by a transition it is (re)saturated, that is the cluster that corresponds to this variable is applied to the node until a fixpoint is reached. When saturating a node, if lower nodes in the data structure are modified they will themselves be (re)saturated. This recursive algorithm can be seen as particular application order of the transition clusters that is adapted to the DD representation of state space. Saturation is very effective but it is difficult to implement correctly, and it is not available in standard DD packages such as *CUDD* [Som05].

This section presents how using simple rewriting rules we automatically create a saturation effect. This allows to embed the complex logic of this algorithm in the library, offering the power of this technique at no additional cost to users. At the heart of this optimization is the property of *local invariance*.

II.2.2 Intuition

The key idea behind exploiting local invariance is the *propagation* of operations. Indeed, often operations representing transitions do not affect all variables of the state signature. Thus the homomorphism representing the transition can be propagated, skipping the variables which are not relevant for the transition. This allows to limit the number of (useless) intermediate nodes created during an application of a transition relation.

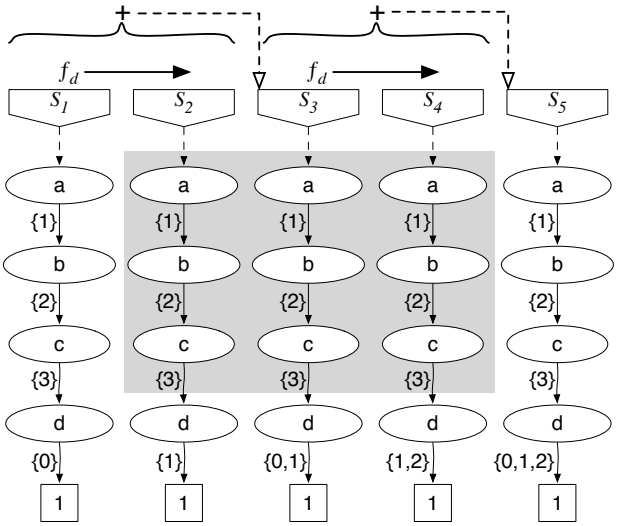


Figure II.1: Effects of propagation

Suppose we have the transition $f_d = leq(d,2) \circ inc(d)$ to apply on S_1 of the figure II.1. We want the full state space, i.e. $(f_d + Id)^\star(S_1)$. A basic BFS application would produce intermediate SDD S_2 to S_4. However, if the evaluation mechanism could know that variables a, b and c are not relevant for the operation, we could propagate $(f_d + Id)^\star$ down to the d node of S_1, work on that node until the * fixpoint is reached, then reconstruct the top of the SDD of S_5. This avoids creation of all the intermediate nodes outlined in grey.

The next subsections formalizes this intuition, allowing to embed this logic in the SDD library. The heart of the contribution was developed during the thesis of Alexandre Hamez [Ham09; HTMK08; HTMK09].

II.2.3 Rewriting Least Fixpoint to Saturation

a Local Invariance Definition

A minimal structural information is needed for saturation to be possible: the variables in the support of operations must be known. To this end we define :

Locally invariant homomorphism An homomorphism h is locally invariant on variable ω iff

$\forall \delta = \langle \omega, \pi, \alpha \rangle \in \mathbb{S}$, $h(\delta) = \sum_{\langle s, \delta' \rangle \in \alpha} \omega \xrightarrow{s} h(\delta')$

Concretely, this means that the application of h doesn't modify the structure of nodes of variable ω, and h is not modified by traversing these nodes. The variable ω is a "don't care" w.r.t. operation h, it is neither written nor read by h. A standard DD encoding [CMS03] of h applied to this variable would produce the identity. The identity homomorphism *Id* is locally invariant on all variables.

b Local Invariance and Elementary Homomorphisms

For an inductive homomorphism h locally invariant on ω, it means that $h(\omega, s) = \omega \xrightarrow{s} h$. A user defining an inductive homomorphism h should provide a predicate $Skip(\omega)$ that returns *true* if h is locally invariant on variable ω. This minimal information allows to dynamically deduce the support of inductive homomorphisms. This information can then be used to dynamically reorder the application of homomorphisms to produce a saturation effect. It is not difficult when writing a homomorphism to define this *Skip* predicate since the useful variables are known, it actually reduces the number of tests that need to be written.

For example, the *inc* and *leq* homomorphisms of section II.1.2 can exhibit the locality of their effect on the state signature by defining *Skip*, which removes the test $\omega = x$ w.r.t. the previous definition since x is the only variable that is not *skipped*:

$$\begin{array}{lcl} leq(x,k)(\omega,s) & = & \omega \xrightarrow{\{n \mid n \in s \wedge n \leq k\}} Id \\ leq(x,k).Skip(\omega) & = & (\omega \neq x) \\ leq(x,k)(1) & = & 1 \end{array} \qquad \begin{array}{lcl} inc(x)(\omega,s) & = & \omega \xrightarrow{\{n+1 \mid n \in s\}} Id \\ inc(x).Skip(\omega) & = & (\omega \neq x) \\ inc(x)(1) & = & 1 \end{array}$$

c Local Invariance and Composite Homomorphisms

For composite homomorphisms the value of the *Skip* predicate can be computed by querying their operands: homomorphisms constructed using union, composition and fixpoint of other homomorphisms, are locally invariant on variable ω if their operands are themselves invariant on ω.

It allows homomorphisms nested in a more complex operation to share traversal of the nodes at the top of the structure as long as they are all locally invariant. When they no longer *Skip* variables, the usual evaluation definition $h + h'(\delta) = h(\delta) \cup h'(\delta)$ is used to affect the current node.

More generally, when we consider a sum of morphisms H we partition its terms into those that skip the current level F and the others G.

Let $H(\delta) = \sum_i h_i(\delta)$ be a sum of morphisms, for any given variable ω, we define $F = \{h_i | h_i.Skip(\omega)\}$ and $G = \{h_i | \neg h_i.Skip(\omega)\}$.

This decomposition $H = F + G$ gives us a union F that is locally invariant on ω and will continue evaluation as a block (propagated). The G part is evaluated using the standard definition $G(\delta) = \sum_{h \in G} h(\delta)$

Thus the minimal *Skip* predicate allows to automatically create clusters of operations by adapting to the structure of the SDD it is applied to. We make very few assumptions on the structure of the DD and the operations, as both the support and the clusters are computed dynamically as the computation progresses.

d Rewriting Least Fixpoint to Saturation

With the rewriting rule of a union $H = F + G$ we have defined, we can now examine the rewriting of an expression $(H + Id)^\star(\delta)$ where H is an arbitrary union of transitions :

$$\begin{aligned}(H + Id)^\star(\delta) &= (F + G + Id)^\star(\delta) \\ &= (G + Id + (F + Id)^\star)^\star(\delta)\end{aligned}$$

The $(F + Id)^\star$ block by definition is locally invariant on the current variable. Thus it is directly propagated to the successor nodes, where it will recursively be evaluated using the same definition as $(H + Id)^\star$.

The remaining fixpoint over G homomorphisms can be evaluated using the *chaining* operation order, which is reported empirically more effective than other approaches [CMS03], a result also confirmed in our experiments.

The *chaining* application order algorithm [RCP95] can be written compactly in SDD as :

$$reach = (\bigcirc_{t \in T}(t + Id))^\star(s_0)$$

We thus finally rewrite:

$$(H + Id)^\star(\delta) = (\bigcirc_{g \in G}(g + Id) \circ (F + Id)^\star)^\star(\delta)$$

e Extension to Hierarchy

We have additional rewriting rules specific to SDD homomorphisms and the $\mathcal{L}$ local construction (see section II.1.2):

$$\begin{aligned}&\mathcal{L}(h, var)(\omega, s) = \omega \xrightarrow{h(s)} Id \\ &\mathcal{L}(h, var).Skip(\omega) = (\omega \neq var) \\ &\mathcal{L}(h, var)(1) = 0\end{aligned}$$

Note that h is a homomorphism, and its application is thus linear to the values in s. Further a $\mathcal{L}$ operation can only affect a single level of the structure (defined by *var*). We can thus define the following rewriting rules, exploiting the locality of the operation :

$$
\begin{array}{lll}
(1) & & \mathcal{L}(h,v) \circ \mathcal{L}(h',v) = \mathcal{L}(h \circ h', v) \\
(2) & & \mathcal{L}(h,v) + \mathcal{L}(h',v) = \mathcal{L}(h + h', v) \\
(3) & v \neq v' \implies & \mathcal{L}(h,v) \circ \mathcal{L}(h',v') = \mathcal{L}(h',v') \circ \mathcal{L}(h,v) \\
(4) & & (\mathcal{L}(h,v) + Id)^\star = \mathcal{L}((h + Id)^\star, v)
\end{array}
$$

Expressions (1) and (2) come from the fact that a local operation is locally invariant on all variables except v. Expression (3) asserts commutativity of composition of local operations, when they do not concern the same variable. Indeed, the effect of applying $\mathcal{L}(h,v)$ is only to modify the state of variable v, so modifying v then v' or modifying v' then v has the same overall effect. Thus two local applications that do not concern the same variable are independent. We exploit this rewriting rule when considering a composition of *local* to maximize applications of the rule (1), by sorting the composition by application variable. A final rewriting rule (4) is used to allow nested propagation of the fixpoint. It derives directly from rules (1) and (2).

With these additional rewriting rules defined, we slightly change the rewriting of $(H + Id)^\star(\delta)$ for node $\delta = \langle \omega, \pi, \alpha \rangle$: we consider $H(\delta) = F(\delta) + L(\delta) + G(\delta)$ where F contains the locally invariant part, $L = \mathcal{L}(l, \omega)$ represents the operations purely local to the current variable ω (if any), and G contains operations which affect the value of ω (and possibly also other variables below). Thanks to rule (4) above, we can write :

$$
\begin{array}{rcl}
(H + Id)^\star(\delta) & = & (F + L + G + Id)^\star(\delta) \\
& = & (G + Id + (L + Id)^\star + (F + Id)^\star)^\star(\delta) \\
& = & (\bigcirc_{g \in G}(g + Id) \circ \mathcal{L}((l + Id)^\star, \omega) \circ (F + Id)^\star)^\star(\delta)
\end{array}
$$

These additional rewriting rules are an extension of the saturation algorithm to decision diagrams featuring structure or hierarchy. Performance evaluation confirms the huge practical benefits of *automatic saturation* induced by this small set of homomorphism rewriting rules and the predicate *Skip*.

II.3 Symbolic Evaluation of Expressions

II.3.1 Arrays and Arithmetic in a Symbolic Setting

We now consider the problem of symbolically encoding transition relations that include concepts such as basic arithmetic or arrays. These features are essential to handle analysis of concurrent programs, we will provide here a general solution.

Classical Approach

Let us define statements as (sequences of) assignments of expressions to variables. The support of a statement is the set of variables it reads or writes to. This notion of locality is heavily exploited, to limit the representation of transitions to the effect they have on variables of their support. For each transition with k'

Boolean support variables, worst case representation size is $2^{k'}$. The symbolic approach was successfully applied to Boolean gate logic where encoding these $\mathbb{B}^{k'} \mapsto 2^{\mathbb{B}^{k'}}$ transition matrices is feasible.

But because classical approaches compute potential to potential $\mathbb{B}^{k'} \mapsto 2^{\mathbb{B}^{k'}}$ transition matrices, a larger support for transitions means exponential growth of the worst case complexity in representation size. It also severely limits the possibilities of saturation-based techniques as their efficiency relies in clusters based on the support of transitions. Hence, a worst case for classical symbolic approaches is when the support of transitions includes all variables.

Transition Support

Moreover, when the input specification includes array or pointer manipulation, any static analysis of statements will necessarily yield pessimistic support assumptions. For instance, a non-constant array access such as `t[i]` *may* depend on the variable `t[0]`. In classical approaches, pessimistic assumptions must include all elements of the array t in the support. Such expressions are commonly encountered in modeling languages such as Promela or Divine [Hol97; Bar+10].

The current state of the art to tackle such complex expressions symbolically in a general way is the approach proposed in LTSmin [BPW10]. A system is defined as consisting of k state variables with a discrete domain D and of transitions described primarily by their support composed of $k' \leq k$ variables. To compute the state space, LTSmin relies on third-party existing explicit model-checkers that provide a computation procedure called for each encountered value of the support in the global state space. Thanks to this projection, the number of these calls is bounded by $D^{k'}$ and in practice is limited to actually encountered states. This tool also implements state-of-the-art symbolic techniques, such as saturation, using classical encoding with two "before" and "after" variables per system state variable.

This approach is however severely challenged when the support grows. If some transitions update a lot of variables, even independently (e.g. increment a set of variables), the support of the transition may be very large. If the high-level model features array manipulation, pessimistic assumptions on the supports end up with supports including most (if not all) state variables. In such an extreme case, the explicit engine is invoked at least once for each state, negating any possible gain from the use of DD. Additionally, such individual insertion of paths in a DD is liable to produce exponential memory peak effects. Large supports also severely limit the possibilities of saturation as clusters are based on the support of transitions.

II.3.2 Intuition

Large supports are often the result of array manipulation or composition of local effects induced by sequences of assignments. We propose to perform a dynamic analysis of statements as they are being resolved, allowing to discover more locality in the remaining effects as expressions are partially evaluated.

In the dynamic case, when evaluating `t[i]`, as soon as the value of the index expression `i` has been reduced to a constant, pessimistic assumptions can be for-

gotten and the support is reduced to the effective cell of the array that is the target of the assignment. We thus can exploit locality to optimize the evaluation, updating the support of expressions as the evaluation progresses.

Compositions of effects are managed as explicit composition of homomorphisms, each of which has a support defined by its underlying expressions. This avoids the problems LTSMin encounters for transitions with a large syntactic support (e.g. increment a large set of variables).

To have efficient symbolic computations of statements, we define an equivalence relation over states with respect to the value of an expression; this induces equivalence classes that can be built dynamically and manipulated symbolically. Intuitively, if efficient manipulation of equivalence classes is possible, then the computation complexity can be proportional to the number of such equivalence classes rather than to the number of actual states.

But our encoding of the expressions needs to avoid any explicit step where states are individually considered in the model-checking algorithm. During the thesis of Maximilien Colange [Col+12; Col13] we defined a general algorithm to compute and manipulate these equivalence classes symbolically on DD. This key ingredient delivers the expressivity of arrays and linear integer arithmetic to the homomorphism framework.

II.3.3 Expressions : Definition

Let Σ be a signature, that is a set of symbols of finite arity. We inductively define the set `Expr` of Σ-*expressions* as $\phi \in$ `Expr` if and only if:

- $\phi \in \Sigma$ of arity 0,
- or $\phi = s(\phi_1, \ldots, \phi_k)$ where $s \in \Sigma$ is of arity k and $\phi_1, \ldots, \phi_k \in Expr$ (ϕ_i is called a sub-expression).

Let D be a domain for expressions. We assume that D is embedded in Σ, so that every element of the domain can be referred to syntactically.

Definition An *interpretation* I is a function that associates to every symbol $s \in \Sigma$ of arity $k > 0$ a (possibly partial) function $I(s) : D^k \mapsto D$, and that maps each symbol of arity 0 to its corresponding element of D.

Intuitively, this formalism captures most programming languages, with pointers and pointer arithmetic. From now on, we assume that there is a finite subset X in D, called *addresses*. The set of addresses X being finite, we note $X = \{x_1, \ldots, x_{|X|}\}$. We assume Σ contains a special symbol δ of arity 1, that allows to access a memory slot given its address. Note that a variable is just a symbolic name for an address. Thus, $I(\delta)$ represents the content of the memory that varies as the program runs. Since we focus on the evolution of the content of the memory, all the interpretations considered from now on are equal for the other symbols (i.e. the operational

semantics for the symbols of the language is known and fixed). Let $\mu = I(\delta)$ designate a *valuation*, i.e. the state of the memory. μ is seen as a (partial, when not all memory contents are known) function from X into D. Since all other symbols have a fixed interpretation, an interpretation I can be described by simply providing μ. Furthermore, all symbols interpretations must be complete functions (only the valuation is allowed to be a partial function). Partial interpretations can be completed by adding a special element to D and mapping the undefined domain onto this special element. This special element corresponds to an error or an undefined behavior. Note that the interpretations of all symbols must take into account this new special element.

Definition Given an interpretation I, an expression $\phi = s(\phi_1, \ldots, \phi_k)$ $(k \geq 0)$ *evaluates* or *reduces* to another expression $eval(I, \phi)$ as follows:

$$eval(I,\phi) = \begin{cases} I(s) \in D & \text{if } s \text{ is a symbol of arity } 0 \\ I(s)(eval(I,\phi_1), \ldots, eval(I,\phi_k)) \in D & \text{if } eval(I,\phi_i) \in D \text{ for all } i \text{ and } I(s) \text{ is defined at this point} \\ s(eval(I,\phi_1), \ldots, eval(I,\phi_k)) & \text{otherwise.} \end{cases}$$

If $eval(I, \phi) \in D$, the evaluation is *complete*.

Notation. We will now abusively denote the evaluation $eval(I, \phi)$ where $I(\delta) = \mu$ by $eval(\mu, \phi)$. If ψ is a (possibly nested) sub-expression of ϕ, $\phi[\psi \leftarrow \theta]$ denotes the expression obtained by substituting the expression θ to ψ in ϕ. Given a valuation μ and a subset of addresses $Y \subseteq X$, $\mu_{|Y}$ denotes the restriction of μ to Y. With these notations, we have, for any variable x, any valuation μ where x is defined, and any expression ϕ: $\phi[\delta(x) \leftarrow \mu(x)] = eval(\mu_{|\{x\}}, \phi)$

Example.

To help in visualizing these definitions, let us use as an example a language supporting a C-like syntax. We give concrete examples here for each element defined abstractly above. We consider a language supporting integers and their manipulation operators (arithmetic +, –, * ... as well as bitwise operations <<,>>,...). The set of considered operators are part of the signature Σ. The domain D is thus integers. The Σ-expressions are built by syntactic combinations of operators, and the literals 0 or 1 are also (terminal) expressions (as D is embedded in Σ).

Then, by definition II.3.3, we must provide an interpretation function I that gives the semantics of all the operators which are used in expressions. The interpretation function works with constants; for our example we should provide the integer output value for each of the binary operators given two integers.

Consider now variables of the program "a,b,c". They are seen as symbolic names and mapped to integers (memory addresses), for instance $0, 1, 2$. The special operator δ allows to read the value of such a variable, hence the expression a is interpreted as $\delta(0)$. We add the notion of array of fixed size *tab*, and access to a cell of an array using $tab[]$. Again *tab* is a symbolic name for a variable mapped to an integer, for instance 3 that is the first memory slot occupied by the array. Then $tab[e]$ where e is an arbitrary expression is a syntactic sugar for $\delta(3 + e)$.

All operators should have complete interpretations: a/b must also be defined when $b = 0$. For this purpose, one or more special constants can be introduced. For a given language manipulating finite types, the definition of the interpretation of most symbols is usually straightforward. We consider that the interpretation of all symbols except δ is fixed throughout the computations. In other words we distinguish the code (all other symbols from the signature) from the data, represented by $I(\delta)$, that may vary as the computation progresses.

Definition II.3.3 formalizes partial evaluation of expressions given an interpretation function. For instance, suppose μ only gives the content of memory slot 0, say $\mu(0) = 12$. Let $\phi = add(\delta(0), \delta(1))$ (usually noted $a + b$). Then $eval(\mu, \phi) = add(eval(\mu, \delta(0)), eval(\mu, \delta(1)))$. We have $eval(\mu, \delta(0)) = I(\delta)(0) = \mu(0) = 12$. However, because μ is not defined for address 1, $eval(\mu, \delta(1)) = \delta(1)$. Hence, $eval(\mu, \phi) = add(12, \delta(1))$ (noted $12 + b$).

II.3.4 Expressions : Equivalence Relation

In practice, a system's state is a valuation of the state variables, and the behavior of the system is described with expressions. When treating such a system using DDD arises the need to evaluate an expression over a *set* of valuations.

More precisely, given an expression ϕ and a set of valuations V, one needs to compute all the evaluations of ϕ by the valuations in V. To achieve this goal efficiently, we rely on an equivalence relation on valuations with respect to the evaluation of a given expression $\sim_\phi^X$.

This equivalence relation is a key notion, allowing efficient evaluation of expressions on sets of valuations.

Definition Given a subset Y of X and an expression ϕ, for all valuations μ, μ' we define the equivalence relation $\sim_\phi^Y$ as follows:

$$\mu \sim_\phi^Y \mu' \Leftrightarrow eval(\mu_{|Y}, \phi) = eval(\mu'_{|Y}, \phi)$$

A trivial case of this equivalence is valuations $\mu \neq \mu'$, that are equal on Y.

Example. As an example for Definition II.3.4, any two μ, μ' such that $eval(\mu, a + b) = eval(\mu', a + b)$ are equivalent. For instance, if both a and b are in Y, $\mu = (a \leftarrow 0, b \leftarrow 1), \mu' = (a \leftarrow 1, b \leftarrow 0)$ are equivalent. If only a is in Y, μ and μ' are not equivalent, since one yields expression $0 + b$ while the other yields $1 + b$.

II.3.5 Evaluating expressions on DDD

Recall that the size of a DDD is often logarithmic in the size of the represented set. The naive approach considers each valuation separately, ending up with a complexity linear in the size of the input set. An efficient solution to this problem should use functions that manipulate the nodes of the data structure representation, so that thanks to caches, the complexity remains proportional to the encoding size.

We propose an algorithm, *EquivSplit*, that partitions a set of valuations (given as a DDD) into equivalence classes with respect to $\sim_\phi^X$. It visits variables in the order given by the DDD, and progressively evaluates the expression. Hence it must work with partial valuations and partially evaluated expressions.

The algorithm *EquivSplit* builds equivalence classes for $\sim_\phi^X$ dynamically based on successive substitution, refinement and merge steps on a partition of the input set. At step i, the goal becomes to remove any dependencies on x_i from the expression ϕ, allowing recursion over $x_{i+1}, \ldots, x_{|X|}$:

- the substitution step uses the partition according to all possible contents of current address x_i (directly provided by the DDD encoding of valuations), to evaluate ϕ with each of these values;
- the look-ahead or refinement step refines the partition by recursively evaluating the reduced expressions over addresses $x_{i+1}, \ldots, x_{|X|}$. This step is only necessary if the expression still depends on x_i after the substitution step : this indicates presence of nested dereference operators δ such as an array access index value, an assignment to x_i of an unresolved expression, or some other complex expression that *may* depend on x_i. This step requires to (recursively) build a partition of children nodes with respect to a target (sub)expression.
- the merge step merges cells of the partition that lead to the same reduced expression over addresses $x_i, \ldots, x_{|X|}$. The merge is performed as soon the expression is resolved, ensuring that the computed partition is not finer than necessary.

The full algorithm is detailed in [Col+13].

Complexity of *EquivSplit*. The refinement step of the algorithm is builds new decision diagrams partitioning sets of states (child nodes) into equivalence classes that may be arbitrarily fine, and depends on the input expression and the input set of valuations. The overall complexity of *EquivSplit* is thus hard to predict and depends on the number of equivalence classes built.

A worst case for our technique would be an expression computing a hash value based on the values in all the memory slots. A perfect hash function would yield equivalence classes limited to singletons, hence encountering exponential worst case complexity (linear over states contained). Conversely, expressions with a small codomain (such as boolean expressions) give a small bound on the maximum number of equivalence classes manipulated by the algorithm.

II.4 Evaluation

Homomorphism Rewriting

We do not present a separate evaluation of automatic saturation in this manuscript. The thesis of Alexandre Hamez [Ham09] reports one to three orders of magnitude improvements in performance. The extension of saturation to the hierarchy of SDD

is very successful, even across several levels of depth in the hierarchy. All the tools we have built on top of the symbolic kernel benefit hugely from it.

The principle of using rewriting strategies for operations to obtain more efficient solutions in a symbolic setting is thoroughly explored in [LB15]. Rewriting strategies are reified, allowing a user to customize them, possibly in a model-specific way. The tool Stratagem [LBCB14] outperformed all the other symbolic tools in the 2014 edition of the MCC@PN for some models where appropriate strategies could be found.

The actual rewriting which is performed is not configurable in ITS-tools, we chose compromises with good overall performance after extensive benchmarks. Since computing a "normal form" would be worst case exponential, application of rewriting rules in the engine is heuristic and relies on pattern matching of at most linear complexity. The current engine includes many more rewriting rules not detailed in this manuscript, mostly dealing with commutativity of operations and their implications on fixpoint computation. These additional rules target scenarios frequently found in temporal logic model-checking such as a least fixpoint under some constraint.

Expression Handling

To give a performance assessment of both our rewriting strategies and our new expression framework we now present some benchmark experiments reported in more detail in [Col+13].

These experiments compare the performance of our ITS-tools [1] to classical state-of-the-art approaches, represented by the tools LTSmin [BPW10] for DD based tools and `super_prove` [BM10] for SAT based tools. We use the models of the BEEM database [Pel07], expressed in Divine a language for communicating process close to Promela.

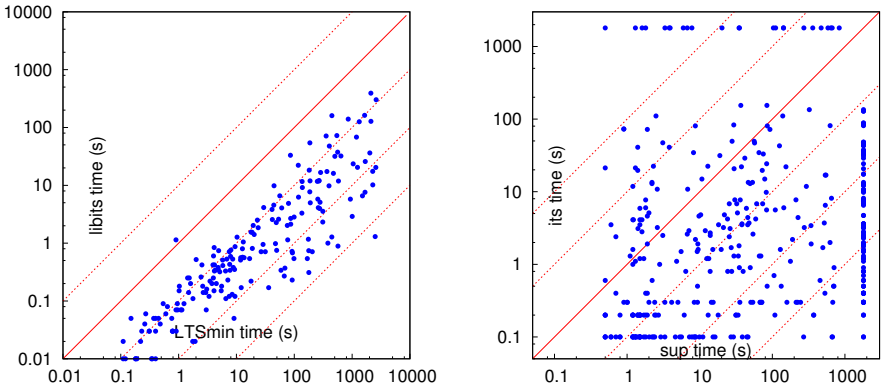
These models have been used before as benchmarks by LTSmin, and they were also part of the hardware model checking contest (HWMCC'12[2]) as SAT instances. LTSmin in these experiments is configured to use the same DDD as ITS-tools, and ITS-tools does not use hierarchy to make algorithmic evaluation of the benefits of our expression framework as fair as possible. `super_prove` was the winner of the HWMCC, and is provided to allow raw performance comparisons to what SAT solvers can do on this type of problem.

These experiments thus exploit both automatic saturation and the expression framework but not hierarchy, they were performed using only DDD. SDD stack very nicely with these other features, gaining an order of magnitude over the performances reported here when the hierarchy levels are appropriately defined. On the other hand, we still currently lack a good heuristic to generate appropriate hierarchy for Divine models, so this is what our engine can do with no manual tuning.

Detailed results of experiments are presented as scatter plots comparing two tools over the whole benchmark. Each point represents a (model,formula) pair

[1] http://ddd.lip6.fr

[2] http://fmv.jku.at/hwmcc12



(a) ITS-tools vs. LTSmin (b) ITS-tools vs. super_prove

Figure II.2: Comparing runtime performance on the BEEM benchmark.

that was tested for reachability with both tools. A point below the diagonal means that `libits` is more efficient than the other tool. Our plots use a logarithmic scale. Lines parallel to the diagonal represent performance ratios of 10, 100 ... (resp. 0.1, 0.01 ...).

The experiments confirm that our *EquivSplit* algorithm performs better than the classical symbolic approach. With the same implementation of DD and the same variable ordering, our implementation is up to 1000 times faster and 100 times less memory consuming than LTSmin.

The comparison to super_prove is more contrasted, since the underlying techniques are so different. DD techniques fail from memory overflow whereas SAT based techniques run into time overflow. Still we treat about 35% more models than `super_prove`, and `libits` is quicker than `super_prove` for 80% of the models treated by both tools, with a speed-up factor up to 1000. On the other models, `super_prove`'s speed-up factor ranges up to 100.

More detailed analysis shows that `libits` runs on average 5 times faster on satisfied properties and 10 times faster on unsatisfied properties than `super_prove`. Unsatisfied instances require both tools to explore the whole reachability graph: these are the hardest problems.

On this benchmark, we show that state-of-the-art symbolic manipulation of decision diagrams can still outperform the best SAT-based techniques.

II.5 Conclusion

Homomorphism rewriting leads to an elegant and flexible expression of saturation like algorithms. Our strategies for symbolic encoding of transition relations give our tool world class performance for state space generation, even when considering complex statements and linear arithmetic.

The theoretical basis of homomorphisms is due to Jean-Michel Couvreur et al.

as introduced in [Cou+02]. As an early adopter and through our later work, we extended this solid foundation with pragmatic extensions such as selector homomorphisms and the invert operator. The rewriting rules for homomorphisms which were introduced during the thesis of Alexandre Hamez [Ham09] have proven their efficiency and been adopted by other homomorphism based implementations [Buc+10; LB15]. The set of rewriting rules can and has been extended to cover more model-checking use case and further improve performance.

The evaluation of complex expressions using *EquivSplit* developed and formalized during the thesis of Maximilien Colange [Col13] generalizes previous solutions using homomorphisms of [Cou+02; BET10] that used swap of adjacent variables as a building block. We also improve more classical DD encoding of transition relations [BPW10] by dynamically exploiting knowledge of the semantics of expressions. Our assumptions are stronger than [BPW10] since we need to know more about the system, but we retain enough expressivity to cover a wide variety of modeling formalisms.

To offer our symbolic kernel to the widest possible user base, we designed a pivot language GAL presented in chapter V that is expressive and general enough to express the semantics of many modeling formalisms. Expressing systems in GAL thus gives end-users access to the efficiency of this *symbolic kernel* described in Part A, without the burden of directly manipulating homomorphisms.

Part B

Symbolic Model-Checking Algorithms

Symbolic model-checking algorithms use a DD based representation of the state graph. Their particularity is that they must reason at each step upon *sets* of states and avoid ever considering states individually. This part presents two symbolic model-checking algorithms that we designed.

LTL is a fragment of temporal logic used to specify properties of systems, and that allows to reason about fairness and infinite runs of a system. Chapter III presents SLAP, a hybrid algorithm mixing symbolic and explicit representations to perform LTL model-checking.

Chapter IV presents a *symbolic-symbolic* approach, or how to compute quotient graphs where nodes represent equivalence classes of states on top of symbolic DD representations.

Both of these contributions build upon the symbolic kernel defined in Part A.

Chapter III

Self-Loop Aggregation Product : SLAP

III.1 LTL model-checking

The symbolic kernel developed in part A can compute reachable states of a system in an efficient manner. This is enough to answer *safety* properties, i.e. properties that all states must verify. Safety properties cover reachability of particular states, invariants and properties such as deadlock freedom.

To express more complex properties, that reason on how states are reachable from one another, variants of temporal logic are used [CGP99]. The most studied fragments of temporal logic in the literature are the Computation Tree Logic (CTL) and the Linear-time Temporal Logic (LTL).

Model checking for Linear-time Temporal Logic (LTL) is usually based on converting the property into a Büchi automaton, composing the automaton and the model (given as a Kripke structure), and finally checking the language emptiness of the composed system or product auotmaton [Var96]. This verification process suffers from a well known state explosion problem. Among the various techniques that have been suggested as improvement, we can distinguish two large families: explicit and symbolic approaches.

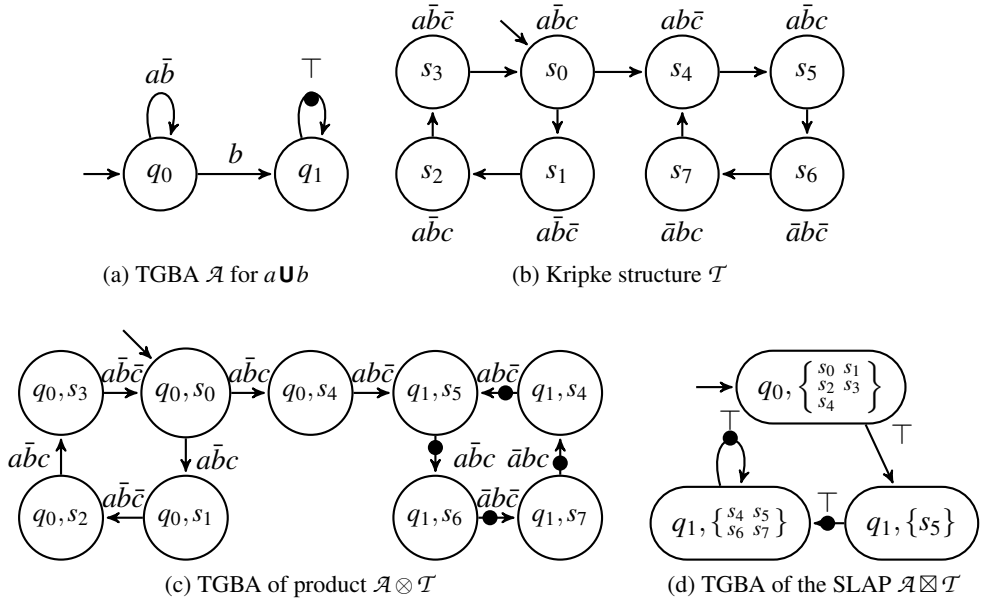
Explicit model checking approaches explore an explicit representation of the product graph. A common optimization builds the graph on-the-fly as required by the emptiness check algorithm: the construction stops as soon as a counterexample is found [Cou+91].

Another source of optimization is to take advantage of stuttering equivalence between paths in the Kripke structure when verifying a stuttering-invariant property [Ete99]: this has been done either by ignoring some paths in the Kripke structure [KV97], or by representing the property using a *testing automaton* [HPV02]. To our knowledge, all these solutions require dedicated algorithms to check the emptiness of the product graph.

Symbolic model checking tackles the state-explosion problem by representing the product automaton symbolically, usually by means of decision diagrams (a concise way to represent large sets or relations). Various symbolic algorithms exist

to verify LTL using fix-point computations (see [Fis+01; SRB02] for comparisons and [KPR98] for the clarity of the presentation). As-is, these approaches do not mix well with stuttering-invariant reductions or on-the-fly emptiness checks.

However explicit and symbolic approaches are not exclusive, some combinations have already been studied [BCZ99; HIK04; STV05; KP08] to get the best of both worlds. They are referred to as **hybrid approaches**. Most of these approaches consist in replacing the Kripke structure by an explicit graph where each node contains sets of states (called aggregates), that is an abstraction preserving properties of the original structure. For instance in Biere et al.'s approach [BCZ99], each aggregate contains states that share their atomic proposition values, and the successor aggregates contain direct successors of the previous aggregate, thus preserving LTL but not branching temporal properties. The Symbolic Observation Graph [HIK04] takes this idea one step further in the context of stuttering invariant properties: each aggregate contains sets of consecutive states that share their atomic proposition values. In both of these approaches, an explicit product with the formula automaton is built and checked for emptiness, allowing to stop early (on-the-fly) if a witness trace is found.



(a) TGBA $\mathcal{A}$ for $a \mathbf{U} b$ (b) Kripke structure $\mathcal{T}$

(c) TGBA of product $\mathcal{A} \otimes \mathcal{T}$ (d) TGBA of the SLAP $\mathcal{A} \boxtimes \mathcal{T}$

Figure III.1: a) a TGBA representing an LTL formula, b) a KS modeling the state graph of a system, c) the classical product of these automata on which the *emptiness-check* is performed, d) SLAP replacement to the classical product. c) and d) agree that accepting runs exist on this example.

Sebastiani et al.'s approach [STV05] is a bit different, as it builds one aggregate for each state of the Büchi automata (usually few in number), and uses a partitioned symbolic transition relation to check for emptiness of the product, thus resorting to

a symbolic emptiness-check (based on a symbolic SCC hull computation).

We present in this chapter a new hybrid algorithm : the *Self-Loop Aggregation Product* (SLAP). SLAP replaces the synchronized product used in the automata-theoretic approach for LTL model checking. The proposed product is an explicit graph of aggregates (symbolic sets of states) that can be interpreted as a Büchi automaton. The criterion used by SLAP to aggregate states from the Kripke structure is based on the analysis of self-loops that occur in the Büchi automaton expressing the property to verify. Our hybrid approach allows on the one hand to use classical emptiness-check algorithms and build the graph on-the-fly, and on the other hand, to have a compact encoding of the state space thanks to the symbolic representation of the aggregates. Our experiments show that this technique often outperforms other existing (hybrid or fully symbolic) approaches.

III.2 Context and Definitions

III.2.1 Boolean Formulas

Let AP be a set of (atomic) propositions, and let $\mathbb{B} = \{\bot, \top\}$ represent Boolean values. We denote $\mathbb{B}(\mathrm{AP})$ the set of all Boolean formulas over AP, i.e., formulas built inductively from the propositions AP, $\mathbb{B}$, and the connectives $\wedge$, $\vee$, and $\neg$.

An assignment is a function $\rho : \mathrm{AP} \to \mathbb{B}$ that assigns a truth value to each proposition. We denote $\mathbb{B}^{\mathrm{AP}}$ the set of all assignments of AP. Given a formula $f \in \mathbb{B}(\mathrm{AP})$ and an assignment $\rho \in \mathbb{B}^{\mathrm{AP}}$, we denote $\rho(f)$ the evaluation of f under ρ. This is simply defined as $\rho(f \wedge g) = \rho(f) \wedge \rho(g)$, $\rho(\neg f) = \neg\rho(f)$, etc. In particular, we will write $\rho \models f$ iff ρ is a satisfying assignment for f, i.e., $\rho \models f \iff \rho(f) = \top$.

We will use assignments to label the states of the model we want to verify, and the propositional functions will be used as labels in the automaton representing the property to check. The intuition is that a behavior of the model (a sequence of assignments) will match the property if we can find a sequence of formulas in the automaton that are satisfied by the sequence of assignments.

III.2.2 Kripke Structure

For the sake of generality, we use *Kripke Structures* (KS for short) to describe the state graph of the model. In a KS, each node of the state graph is labeled by the atomic propositions it satisfies (i.e. an assignment). This formalism thus describes state-based semantics (one could also consider event-based logic, where the *edges* of the state graph bear a label).

Kripke structure A *Kripke structure* is a 4-tuple $\mathcal{T} = \langle \mathrm{AP}, \Gamma, \lambda, \Delta, s_0 \rangle$ where:

- AP is a finite set of atomic propositions,
- Γ is a finite set of *states*,

- $\lambda : \Gamma \to \mathbb{B}^{AP}$ is a state labeling function,
- $\Delta \subseteq \Gamma \times \Gamma$ is a *transition relation*. We will commonly denote $s_1 \to s_2$ the element $(s_1, s_2) \in \Delta$.
- $s_0 \in \Gamma$ is the *initial state*.

Fig. III.1b represents a Kripke structure over $AP = \{a, b, c\}$. The state graph of a system is typically represented by a KS, where state labels in the KS give the atomic proposition truth values in a given state of the system.

It is sometimes convenient to interpret an assignment ρ as a formula that is only true for this assignment. For instance the assignment $\{a \mapsto \top, b \mapsto \top, c \mapsto \bot\}$ can be interpreted as the formula $a \wedge b \wedge \neg c$. So we may use an assignment where a formula is expected, as if we were abusively assuming that $\mathbb{B}^{AP} \subset \mathbb{B}(AP)$.

III.2.3 TGBA

A *Transition-based Generalized Büchi Automaton* (TGBA) is a Büchi automaton in which generalized acceptance conditions are expressed in term of transitions that must be visited infinitely often. The reason we use these automata is that they allow a more compact representation of properties than traditional Büchi automata (even generalized Büchi automata) [DLP04] without making the emptiness check harder [CDLP05].

TGBA A *Transition-based Generalized Büchi Automata* is a tuple $A = \langle AP, Q, \mathcal{F}, \delta, q^0 \rangle$ where

- AP is a finite set of atomic propositions,
- Q is a finite set of states,
- $\mathcal{F} \neq \emptyset$ is a finite and non-empty set of acceptance conditions,
- $\delta \subseteq Q \times \mathbb{B}(AP) \times 2^{\mathcal{F}} \times Q$ is a transition relation. We will commonly denote $s \xrightarrow{f,ac} d$ an element $(s, f, ac, d) \in \delta$,
- $q^0 \in Q$ is the initial state.

An execution (or a run) of A is an infinite sequence of transitions $\pi = (s_1, f_1, ac_1, d_1) \cdots (s_i, f_i, ac_i, d_i) \cdots \in \delta^\omega$ with $s_1 = q^0$ and $\forall i, d_i = s_{i+1}$. We shall simply denote it as $\pi = s_1 \xrightarrow{f_1,ac_1} s_2 \xrightarrow{f_2,ac_2} s_3 \cdots$. Such an execution is *accepting* iff it visits each acceptance condition infinitely often, i.e., if $\forall a \in \mathcal{F}, \forall i > 0, \exists j \geq i, a \in ac_j$. We denote $\mathrm{Acc}(A) \subseteq \delta^\omega$ the set of accepting executions of A.

A behavior of the model is an infinite sequence of assignments: $\rho_1\rho_2\rho_3 \cdots \in (\mathbb{B}^{AP})^\omega$, while an execution of the automaton A is an infinite sequence of transitions labeled by Boolean formulas. The language of A, denoted $\mathcal{L}(A)$, is the set

of behaviors compatible with an accepting execution of A: $\mathcal{L}(A) = \{\rho_1\rho_2\cdots \in (\mathbb{B}^{\mathrm{AP}})^\omega \mid \exists s_1 \xrightarrow{f_1,ac_1} s_2 \xrightarrow{f_2,ac_2} \cdots \in \mathrm{Acc}(A) \text{ and } \forall i \geq 1, \rho_i \models f_i\}$

Fig. III.1a represents a TGBA for the LTL formula $a\mathbf{U}b$ (a is true, Until b becomes true). The black dot on the self-loop $q_1 \xrightarrow{\top,\{\bullet\}} q_1$ denotes an acceptance conditions from $\mathcal{F} = \{\bullet\}$. The labels on edges ($a\bar{b}$,$b$ and $\top$) represent the Boolean expressions over $\mathrm{AP} = \{a,b\}$. There are two other TGBA in Fig. 1, that represent respectively the classical product construction and the SLAP of this TGBA and the Kripke Structure of Fig. III.1b.

We now define a synchronized product for a TGBA and a KS, such that the language of the resulting TGBA is the intersection of the languages of the two automata.

Synchronized product of a TGBA and a Kripke structure Let $\mathcal{A} = \langle \mathrm{AP}', Q, \mathcal{F}, \delta, q^0\rangle$ be a TGBA and $\mathcal{T} = \langle \mathrm{AP}, \Gamma, \lambda, \Delta, s_0\rangle$ be a Kripke structure over $\mathrm{AP} \supseteq \mathrm{AP}'$.

The *synchronized product* of $\mathcal{A}$ and $\mathcal{T}$ is the TGBA denoted by $\mathcal{A} \otimes \mathcal{T} = \langle \mathrm{AP}, Q_\times, \mathcal{F}, \delta_\times, q^0_\times\rangle$ defined as:

- $Q_\times = Q \times \Gamma$,
- $\delta_\times \subseteq Q_\times \times \mathbb{B}^\star(\mathrm{AP}) \times 2^{\mathcal{F}} \times Q_\times$ where

$$\delta_\times = \left\{ (q_1,s_1) \xrightarrow{f,ac} (q_2,s_2) \;\middle|\; \begin{array}{l} s_1 \to s_2 \in \Delta,\ \lambda(s_1) = f \text{ and} \\ \exists g \in \mathbb{B}^\star(\mathrm{AP}) \text{ s.t. } q_1 \xrightarrow{g,ac} q_2 \in \delta \text{ and } \lambda(s_1) \models g \end{array} \right\}$$

- $q^0_\times = (q^0, s_0)$.

Fig. III.1c represents such a product of the TGBA $a\mathbf{U}b$ of Fig.III.1a and the Kripke structure of Fig. III.1b. State (s_0,q_0) is the initial state of the product. Since $\lambda(s_0) = a\bar{b}c$ we have $\lambda(s_0) \models a\bar{b}$, successors $\{s_1,s_4\}$ of s_0 in the KS will be synchronized through the edge $q_0 \xrightarrow{a\bar{b},\emptyset} q_0$ of the TGBA with q_0. In state (q_0,s_4) the product can progress through the $q_0 \xrightarrow{b,\emptyset} q_1$ edge of the TGBA, since $\lambda(s_4) = ab\bar{c} \models b$. Successor s_5 of s_4 in the KS is thus synchronized with q_1. The TGBA state q_1 now only requires states to verify $\top$ to validate the acceptance condition $\bullet$, so any cycle in the KS from s_5 will be accepted by the product. The resulting edge of the product bears the acceptance conditions contributed by the TGBA edge, and the atomic proposition Boolean formula label that comes from the KS. The size of the product in both nodes and edges is bounded by the product of the sizes of the TGBA and the KS.

III.3 Self-Loop Aggregation Product (SLAP)

III.3.1 Intuition

The definition of SLAP combines elements found in other strategies in a novel manner.

From the work on SOG [HIK04; KP08] we took the idea of building an explicitly managed set of aggregates, where the states belonging to each aggregate are stored (and computed) using DD. Building on the results of part A, we also wanted to have aggregates defined using a least fixpoint, so that (automatic) saturation can be used to build aggregates efficiently.

This places us in the hybrid category of algorithms, so an explicit emptiness check will be performed, allowing to interrupt the computation as soon as an accepting run is found.

From the work on the Symbolic Synchronized Product [BHI04] we took the idea that defining a new product rather than an abstraction of the KS allows to adapt the abstractions used during the emptiness check, in a property specific way. The SOG for instance is defined as a KS, that abstracts the original KS of the model with respect to a fixed alphabet (all atomic propositions in the formula automaton). If the formula to verify is something like $a \mathbf{U} b \mathbf{U} c \ldots$, once the product has reached a state of the KS that validates b, proposition a basically becomes irrelevant.

Working at the product level also means that instead of defining an equivalence between the KS of the system and the abstraction, we can be content with a much weaker property, namely that our product construction has an empty language iff. the standard synchronized product is empty.

So our general idea was to use the structure of the TGBA to guide the aggregation procedure on the fly. Our goal was to aggregate as much as possible, yielding a small explicit graph where each aggregate holds many states, trusting the DD engine to scale to large aggregates efficiently.

The following section presents SLAP, a specialized synchronized product that aggregates states of the KS as long as the TGBA state does not change, and no *new* acceptance conditions are visited. More precisely, we study the self-loops of the current state of the formula automaton, and aggregate consecutive states of the system as long a s they are compatible with the labels of self-loops. The *Self-Loop Aggregation Product* (SLAP) preserves full Büchi expressible properties.

III.3.2 Definition

The notion of self-loop aggregation is captured by $\mathrm{SF}(q, ac)$, the **S**elf-loop **F**ormulas (labeling edges $q \to q$) that are weaker in terms of visited acceptance conditions than ac.

When synchronizing with an edge of the property TGBA bearing ac leading to q, successive states of the Kripke will be aggregated as long as they model $\mathrm{SF}(q, ac)$. More formally, for a TGBA state q and a set of accepting condition $ac \subseteq \mathcal{F}$, let us define

$$\mathrm{SF}(q, ac) = \bigvee_{q \xrightarrow{f, ac'} q \in \delta \text{ s.t. } ac' \subseteq ac} f$$

Moreover, for $a \subseteq \Gamma$ and $f \in \mathbb{B}(\mathrm{AP})$, we define $\mathrm{FSucc}(a, f) = \{s' \in \Gamma \mid \exists s \in a, s \to s' \in \Delta \wedge \lambda(s) \models f\}$. That is, first **F**ilter a to only keep states satisfying f, then pro-

duce their **Succ**essors. We denote by $\mathrm{FReach}(a,f)$ the least subset of Γ satisfying both $a \subseteq \mathrm{FReach}(a,f)$ and $\mathrm{FSucc}(\mathrm{FReach}(a,f),f) \subseteq \mathrm{FReach}(a,f)$.

SLAP of a TGBA and a KS Given a TGBA $\mathcal{A} = \langle \mathrm{AP}', Q, \mathcal{F}, \delta, q^0 \rangle$ and a Kripke structure $\mathcal{T} = \langle \mathrm{AP}, \Gamma, \lambda, \Delta, s_0 \rangle$ over $\mathrm{AP} \supseteq \mathrm{AP}'$, the *Self-Loop Aggregation Product* of $\mathcal{A}$ and $\mathcal{T}$ is the TGBA denoted $\mathcal{A} \boxtimes \mathcal{T} = \langle \emptyset, Q_\boxtimes, \mathcal{F}, \delta_\boxtimes, q^0_\boxtimes \rangle$ where:

- $Q_\boxtimes = Q \times (2^\Gamma \setminus \{\emptyset\})$
- $\delta_\boxtimes = \left\{ (q_1,a_1) \xrightarrow{\top, ac} (q_2,a_2) \;\middle|\; \begin{array}{l} \exists f \in \mathbb{B}(\mathrm{AP}') \text{ s.t. } q_1 \xrightarrow{f, ac} q_2 \in \delta, \\ \qquad q_1 = q_2 \Rightarrow ac \neq \emptyset, \text{ and} \\ \qquad a_2 = \mathrm{FReach}(\mathrm{FSucc}(a_1,f), \mathrm{SF}(q_2,ac)) \end{array} \right\}$
- $q^0_\boxtimes = (q^0, \mathrm{FReach}(\{s_0\}, \mathrm{SF}(q^0,\emptyset)))$

Note that because of the way the product is built, it is not obvious what Boolean formula should label the edges of the SLAP product. Since in fact this label is irrelevant when checking language emptiness, we label all arcs of the SLAP with $\top$ and simply denote $(q_1,a_1) \xrightarrow{ac} (q_2,a_2)$ any transition $(q_1,a_1) \xrightarrow{\top, ac} (q_2,a_2)$.

$Q \times 2^\Gamma$ might seem very large but in practice the reachable states of the SLAP is a much smaller set than that of the product $Q \times \Gamma$. Furthermore the FReach operation can be efficiently implemented as a symbolic least fix point using saturation.

Fig. III.1d represents the SLAP built from our example KS, and the TGBA of $a \mathbf{U} b$. The initial state of the SLAP iteratively aggregates successors of states verifying $\mathrm{SF}(q^0,\emptyset) = a\bar{b}$. Then following the edge $q^0 \xrightarrow{b,\emptyset} q_1$, states are aggregated with condition $\mathrm{SF}(q_1,\emptyset) = \bot$. Hence q_1 is synchronized with successors of states in $\{s_0,s_1,s_2,s_3,s_4\}$ satisfying b (i.e., successors of $\{s_4\}$). Because $\mathrm{SF}(q_1,\emptyset) = \bot$ the successors of $\{s_5\}$ are not gathered when building $(q_1,\{s_5\})$. Finally, when synchronizing with edge $q_1 \xrightarrow{\top,\bullet} q_1$, we have $\mathrm{SF}(q_1,\{\bullet\}) = \top$, hence all states of the cycle $\{s_4,s_5,s_6,s_7\}$ are added.

III.4 Evaluation

We implemented within ITS-tools several hybrid or fully symbolic algorithm. We built upon Spot [DLP04; DL14; DL+16] for the translation of formulas to TGBA and the explicit emptiness check. Spot is a library that provides bricks to build your own model checker based on the automata-theoretic approach using TGBAs.

We used as benchmark models classic scalable Petri net examples [CMS03]. The model occurrences we used had from a few million to 10^{66} reachable states. The idea is that we want methods that scale to very large KS, while maintaining a manageable size for the explicit graph. More details on the experimental setup are available in [DL+11] and the companion technical report.

We implemented two fully symbolic algorithms from the literature, OWCTY (One-Way Catch Them Young) and EL (Emerson-Lei) algorithms [Fis+01; SRB02]

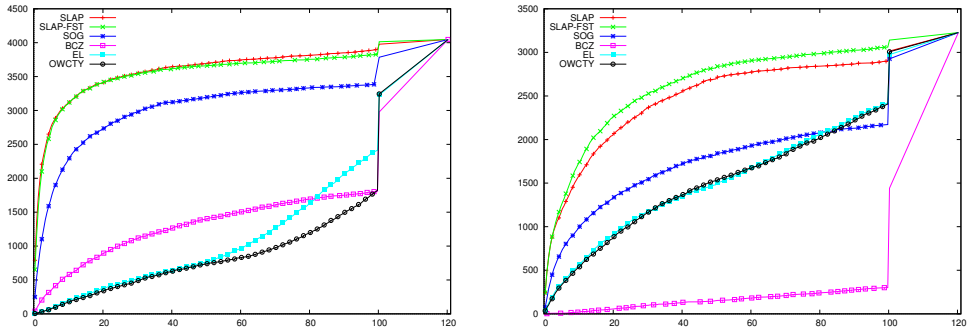


Figure III.2: Cumulative plots comparing the time of all methods. Non-empty products are shown on the left, and empty products on the right.

using the forward transition relation. We also compared to hybrid algorithms SOG [HIK04] (Symbolic Observation Graph) and BCZ [BCZ99] (Biere-Clarke-Zhu).

We also built two versions of SLAP, the one formally described above, and SLAP-FST (Fully Symbolic for Terminal components), a variant that does a symbolic SCC detection on aggregates iff. such an SCC might be accepting. SLAP-FST produces smaller explicit graphs, but performs more symbolic operations. In practice the FST variant is better when the product is empty, and worse when there are accepting runs. In such cases interrupting the computation on-the-fly thanks to the explicit emptiness check yields better overall results.

Fig. III.2 allows to compare the various methods. For each experiment (model/formula pair) we first collect the maximum time reached by a technique that did not fail, then compute for the other approaches what percentage of this maximum was used. The vertical segments visible at 100% thus show the number of runs for which this technique was the worst of those that did not fail. Any failures are plotted arbitrarily at 120%. This gives us a set of values between 0% and 120% for which we plot the cumulative distribution function. For instance, if a curve goes through the (20%,2000) point, it means that for this technique, 2000 experiments took at most 20% of the time taken by the worst technique for the same experiments.

The left plot for the non-empty cases shows that the on-the-fly mechanism allows all hybrid algorithms (SLAP, SLAP-FST, SOG, BCZ) to outperform the symbolic ones (OWCTY, EL). The SLAP and SLAP-FST method take less than 10% of the time of the slowest method in 80% of the cases.

The right plot for the empty cases shows that fully symbolic algorithm behave relatively far better (all methods have to explore the full product anyway).

SLAP-FST and SLAP have similar performance, with a slight edge for SLAP-FST when the product is empty, hence the harder problems. For this reason we compare SLAP-FST to the other methods from the literature in the scatter plots of

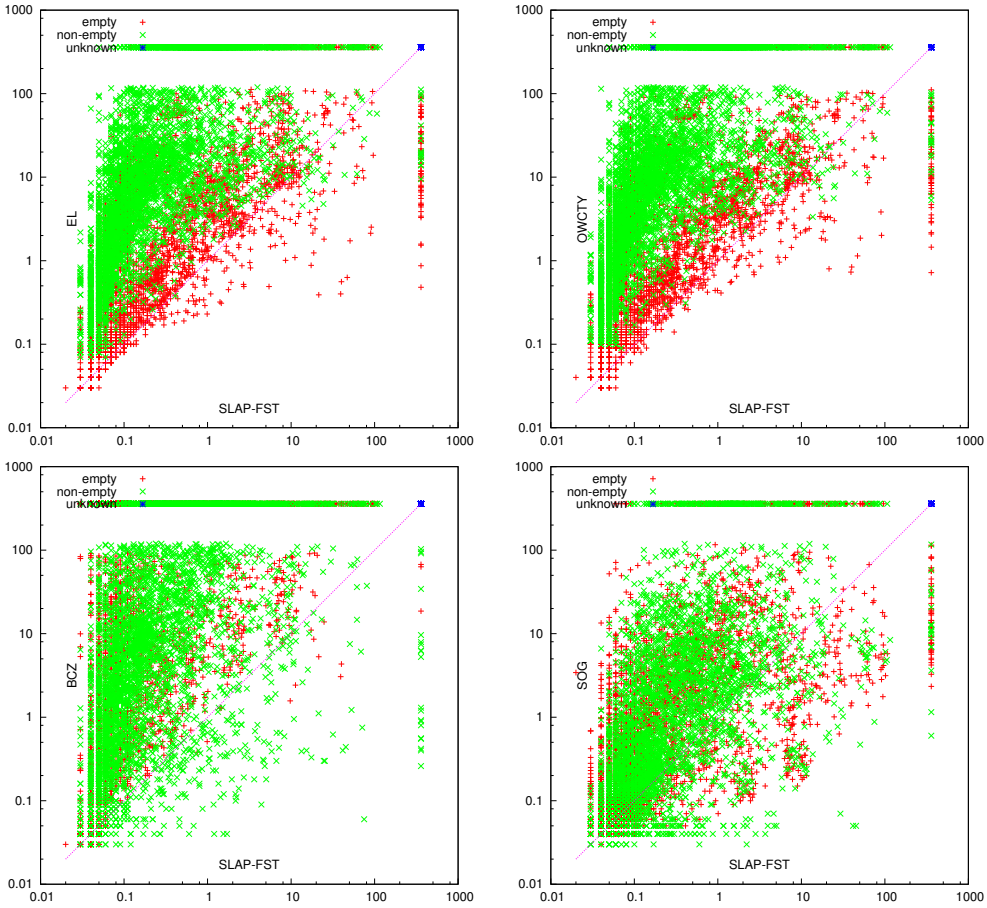


Figure III.3: Comparison of SLAP-FST against four other methods : fully symbolic approaches EL and OWCTY (top) and hybrid algorithms BCZ and SOG (bottom). Green means a counter example trace exists; hybrid methods thus might exit early. Red means the whole product is explored which is typically a harder problem.

Fig. III.3. The performances are presented using a logarithmic scale. Each point represents an experiment, i.e., a model and formula pair. We plot experiments that failed (due to timeout) as if they had taken 360 seconds, so they are clearly separated from experiments that didn't fail (by the wide white band).

Because SLAP can interrupt on the fly, it dominates fully symbolic methods EL and OWCTY when a trace exists (green points). The general trend is still largely in favor of SLAP even when the full product must be explored (red points). Hybrid BCZ can sometimes turn lucky and find a counter example before SLAP (thus some green below the diagonal), but is otherwise dominated by SLAP. The hybrid SOG is the closest competitor to SLAP on this benchmark, but to our advantage, SOG is not able to handle non stuttering-invariant properties.

III.5 Conclusion

The main originality of SLAP is that we only use a very weak property instead of a strong equivalence relation constraint on states belonging to a given aggregate as is the case in other hybrid approaches. We proved in [DL+11] that the SLAP of a given KS and a TGBA accepts a run if and only if the synchronized product of these two structures accepts a run. This nice theorem is all we need to ensure that using the SLAP abstraction to perform LTL model-checking is sound : we will not miss nor create bogus witness runs during the emptiness check procedure.

We have thus developed an original hybrid algorithm, the Self-Loop Aggregation Product (SLAP), that produces very small explicit graphs (good for very large transition systems) and maximizes use of the least fixpoint (thus is efficiently implemented using automatic saturation). SLAP proved to be overall the most effective of the algorithms we had implemented on a large benchmark of Petri net examples. I'm particularly proud of SLAP as it's very effective and it is not a just a variant of existing algorithms, despite the fact that symbolic LTL model-checking has been heavily studied since the 90's.

This work on LTL was complemented by [BS+14] during the thesis of Ala-Eddine Ben Salem, with a fully symbolic algorithm that uses testing automata to represent the formula (rather than Büchi automata) and again seeks to maximize the use of saturation. While it is limited to the stuttering invariant fragment of the logic, for many formulas the approach is very effective.

Chapter IV

Symbolic Symbolic Model-Checking

IV.1 Quotient Graph

Quotient graph approaches use an equivalence relation to build a small abstraction of the state graph that still preserves many properties of interest. Quotient graphs are often called *symbolic* in the literature e.g. the Symbolic Reachability Graph of [Chi+93], because they work with sets of states. Hence we coined the term *symbolic-symbolic* to refer to construction of quotient graph using DD based techniques.

The first model-checking algorithms that build quotient graphs in were developed in the 90's, in [Chi+93] for Well-Formed Petri Nets (now in the ISO standard called Symmetric Nets SN) and in [NID96] for the formalism Murphy.

These formalisms allow to express symmetries on the data domains (symmetric color domains, scalar set...), that induce symmetries on the way events can be executed. Hence we consider as input a symmetry group G over states and the transition relation, and we can build a quotient graph of equivalence classes (also called orbits) of states, that may be exponentially smaller than the full state graph [Cla+98]. This *quotient graph* preserves many properties of interest such as reachability and linear temporal logic provided the property is itself symmetric with respect to G.

To build a quotient graph, the approach most commonly used [NID96; Jun03] consists in using a canonical representative of each orbit. However, an orbit may be of exponential size with respect to the number of elements in the state vector. Thus, the computation of a canonical representative of an orbit has exponential worst case complexity in time and/or memory (if the orbit is actually built).

Junttila [Jun03] proposes a general definition of this approach for systems whose states are integer vectors and symmetry groups are arbitrary permutation groups. Using the Schreier-Sims representation [Sim71] of permutation groups, he proposes an algorithm effective in practice to compute a representative of an equivalence class.

However, the proposed algorithm only deals with explicit encoding of the state

space. Thus, the symbolic-symbolic problem remains hard since the algorithm must be applied on each individual state. Unfortunately, algorithms that manipulate classical explicit data structures must be redesigned to take advantage of DD. This is not always possible, particularly if the algorithm involves separate treatments for every state.

Our goal in this work was to try to combine the quotient graph approach with DD technology, yielding a general approach to building a symbolic-symbolic engine.

IV.2 Context and Definitions

We recall here the theory of symmetry reduction for state space analysis. These definitions are adapted from [Jun03].

IV.2.1 Symmetry Groups of a Transition System

Definition Transition system
A transition system is a tuple (S, Δ, S_0) such that:

- S is a finite set of states,
- $\Delta \subseteq S \times S$ is the transition relation,
- $S_0 \subseteq S$ is the set of initial states.

Transitions for $(s_1, s_2) \in \Delta$ are noted $s_1 \rightarrow s_2$. Symmetries of transition systems are defined using a bisimilarity relation between states.

Definition Symmetry
Let $\mathcal{K} = (S, \Delta, S_0)$ be a transition system. A symmetry of $\mathcal{K}$ is a permutation g over S such that:

- $g.S_0 = S_0$
- g is congruent with respect to the transition relation:
  $\forall s_1, s_2 \in S, s_1 \rightarrow s_2 \Leftrightarrow g.s_1 \rightarrow g.s_2$

G, the set of all symmetries of $\mathcal{K}$, is a group because the composition is associative, and the composition of two symmetries and the inverse of a symmetry are still symmetries.

IV.2.2 Quotient Graph

Definition Equivalence relation $\equiv_G$
Two states $s_1, s_2 \in S$ are said to be symmetric, denoted $s_1 \equiv_G s_2$, if there is a $g \in G$ such that $g.s_1 = s_2$. $\equiv_G$ is an equivalence relation over S. $[x]_G$ denotes the equivalence class (also called orbit) of x under $\equiv_G$.

We may now define the abstraction of a transition system using $\equiv_G$.

Definition Reduced transition system
$\tilde{\mathcal{K}} = (\tilde{S}, \tilde{\Delta}, \tilde{S}_0)$ is a reduction of $\mathcal{K}$ w.r.t. G if and only if:

- $\tilde{S} \subseteq S, \forall s \in S, \exists \tilde{s} \in \tilde{S} : s \equiv_G \tilde{s}$,
- $\tilde{S}_0 \subseteq \tilde{S}$ and $\forall g \in G, g.\tilde{S}_0 \subseteq S_0$,
- $\tilde{\Delta} \subseteq \tilde{S} \times \tilde{S}$,
- if $\tilde{s}_1 \in \tilde{S}$ and $(\tilde{s}_1, s_2) \in \Delta$, then there exists $\tilde{s}_2 \in \tilde{S}$ such that $\tilde{s}_2 \equiv_G s_2$ and $(\tilde{s}_1, \tilde{s}_2) \in \tilde{\Delta}$,
- if $(\tilde{s}_1, \tilde{s}_2) \in \tilde{\Delta}$ then there exists $s_2 \in S$ such that $s_2 \equiv_G \tilde{s}_2$ and $(\tilde{s}_1, s_2) \in \Delta$.

A reduction, $\tilde{\mathcal{K}}$ of $\mathcal{K}$ w.r.t. G preserves the reachability property and, under appropriate conditions, linear temporal formulae [AHI98; Cla+98]. Hence, the verification can be done on $\tilde{\mathcal{K}}$. Note that this definition allows to use several representatives per orbit, generalizing the notion of quotient graph. This approach using several representatives yields a larger reduced structure but may be faster to build [Jun03].

IV.2.3 Explicit Quotient Graph Algorithm

$\tilde{S} \leftarrow \texttt{repr}(S_0)$
$\tilde{\Delta} \leftarrow \emptyset$
repeat
  for $s \in \tilde{S}$ **do**
    $\tilde{S}' \leftarrow \texttt{repr}(\texttt{succ}(s))$
    $\tilde{\Delta} \leftarrow \tilde{\Delta} \cup \{(s, \tilde{s}') | \tilde{s}' \in \tilde{S}'\}$
    $\tilde{S} \leftarrow \tilde{S} \cup \tilde{S}'$
  end for
until a fixpoint is reached

Figure IV.1: The algorithm to generate $\tilde{\mathcal{K}}$

An abstract algorithm using explicit data structures to compute $\tilde{\mathcal{K}}$ is presented on figure IV.1. It uses a function `succ` that maps any state s to its successors: $\texttt{succ}(s) = \{s' | (s, s') \in \Delta\}$.

The critical ingredient is the function `repr` that maps an element $s \in S$ onto its representative $\tilde{s} \in [s]_G$. The size of $\tilde{S}$ depends on the function `repr`, as $\tilde{S} = \texttt{repr}(S)$, with two extreme cases:

- if `repr` is the identity, then $\tilde{S} = S$ and $\tilde{\mathcal{K}} = \mathcal{K}$, there is no reduction.

- if `repr` maps all elements of an orbit onto a *unique representative element*, then $\tilde{S}$ is in bijection with $S_{/G}$, and the size of $\tilde{S}$ is minimal.

Building a perfect `repr` that computes such a *unique* representative is unfortunately as hard as graph isomorphism, a class of complexity is not known to have a polynomial solution [Cla+96; Jun03]. Still, for many groups of symmetry commonly encountered in practice when modeling system the quotient graph approach can be very successful [NID96; Chi+90; Hen+04; BDH02].

IV.3 Symbolic Symbolic State Space

IV.3.1 Intuition

Our goal in this work was to combine use of symmetries with decision diagrams.

Of course, use of SDD rather than plain decision diagrams is one very effective way of exploiting symmetry in a symbolic setting. Hierarchy in SDD can and will exploit some symmetries due to repeated subsystems, provided an appropriate structure and variable ordering can be determined. The recursive folding approach as presented in chapter I can even scale to an exponential number of repeated instances in favorable cases. However, using SDD hierarchy does not build a quotient state space, it just deals with huge state graphs very well in very symmetric cases.

We focus here instead on an approach that tries to directly build the quotient state space using decision diagrams and dedicated operations to perform the canonization procedure.

a Combining Symmetries and Decision Diagrams.

Initial attempts to combine a symbolic representation of sets of states with a computation of a quotient graph met mitigated success. The problem, identified in [Cla+96], is that the orbit relation –allowing to map states to their representative– has exponential size when represented as a BDD, whichever the variable order chosen. Variations such as using several representatives of an orbit, can be more effective but do not fully exploit the symmetry group.

A slightly different approach to build a quotient graph [Chi+93], is to use an abstract representation of orbits rather than explicit states as representatives. This also allows to better exploit symmetries of the transition relation (symmetric events). However this approach can only deal with specific groups of symmetries, and thus cannot easily be generalized to arbitrary state encodings or permutation groups.

Our first successful attempt [TMIP04a] was a tool able to build the *symbolic* reachability graph [Chi+90] of a well-formed net using a *symbolic* representation of states with DDD. While the approach proved effective, able to outperform both pure DD based methods and the explicit quotient graph construction on some examples, it was very specialized and heavily dedicated to both SRG and DDD since

we had to build dedicated homomorphisms for the canonization step. The tool we built thus remained mostly a proof of concept.

A few years later [Col+11], we revisited the approach at the very beginning of the thesis of Maximilien Colange. We built a tool Crocodile supporting construction of a quotient graph for Symmetric Nets with Bags (SNB, an extension of symmetric nets) using SDD. We used the structure of the color domains to infer structure and again had to develop dedicated homomorphisms for the canonization step. The tool again outperforms both techniques in isolation, but requires careful specification using SNB of the input models to fully unlock its potential, lowering its overall applicability.

Both of those tools [TMIP04a; Col+11] did function quite well, somehow contradicting the negative result of Clarke et al. [Cla+96]. We identified as key difference between our approach and [Cla+96] that in both cases we had used a fixpoint algorithm to compute representatives, rather than a single step of an "orbit relation" as was attempted in previous works.

To generalize our solution, in [Col+12] we formalized the hypothesis that made computation of a representative using a fixpoint possible. We obtained a very small set of assumptions on the nature of the system and the symmetries, and simpler decision diagram operations than homomorphisms, that are available for all DD.

We now present these assumptions, then the symbolic-symbolic algorithm allowing to work with arbitrary symmetry groups, and that can be efficiently implemented on top of symbolic data structures.

IV.3.2 Assumptions

a States

We make the assumption that the system's states S are vectors of integers, of fixed size n: $S \subseteq \mathbb{N}^n$.

States being elements of $\mathbb{N}^n$ are naturally represented as a DDD of n variables. Labels of states, if we consider a Kripke structure instead of a transition system, can be encoded as additional state variables.

b Symmetries

Group of Permutations.

We consider symmetries that permute the indexes: $\forall g \in G, \forall v = (v_1, v_2, \ldots, v_n) \in S, g.v = (v_{g.1}, v_{g.2}, \ldots, v_{g.n})$. The group of all permutations over a set of size n is denoted by $\mathcal{S}_n$.

We then manipulate symmetry groups as sets of permutations. Conversely, given a set of permutations H, let $\langle H \rangle$ denote the group generated by H.

To encode the action of a permutation on a set of states, we define for any permutation $g \in \mathcal{S}_n$ a homomorphism $\texttt{apply}(g)$ i.e. $\texttt{apply}(g)(S) = \{g.s | s \in S\}$.

`apply` is built as a composition of transpositions of adjacent elements noted $\tau_{i,i+1}$. We compute a path with the minimal number of these transpositions necessary to achieve the desired effect and compose them to build `apply`. For instance, with $g = (2,3,1,4)$ of $\mathcal{S}_4$, we could use $g = \tau_{1,2} \circ \tau_{2,3}$.

Lexicographic Ordering

States are totally ordered. We use lexicographic ordering, noted $<$. The **canonical representative** $\hat{s}$ of an orbit $[s]_G$ is defined as its smallest element (with respect to $<$). Thus, $\forall s \in S, \hat{s} = min[s]_G$.

We assume we can define $\texttt{reduces}(g)$, a selector homomorphism to retain states that are reduced by g, i.e. $\texttt{reduces}(g)(S) = \{s | s \in S, g.s < s\}$.

It can be expressed as a composition of variable comparisons. For instance, consider the permutation $g = (2,3,1,4)$ of $\mathcal{S}_4$. We have $g^{-1} = (3,1,2,4)$. Hence g reduces $s = (s_1, s_2, s_3, s_4)$ iff
$s_{g^{-1}.1} < s_1 \vee (s_{g^{-1}.1} = s_1 \wedge (s_{g^{-1}.2} < s_2 \vee (s_{g^{-1}.2} = s_2 \wedge (\ldots))))$

This general formula is instantiated for this specific g in the following way:
$s_3 < s_1 \vee (s_3 = s_1 \wedge (s_1 < s_2 \vee (s_1 = s_2 \wedge s_2 < s_3)))$

Let us note that since position 4 is invariant by g, there are only three nested variable comparisons. Subsequent conditions are trivially simplified away. This condition is expressed using a selector homomorphism allowing comparison (by $<$ and $=$) of the value of two variables of a state. The full condition homomorphism is expressed using composition $\circ$ for $\wedge$ and the sum $+$ for $\vee$.

The assumptions on the DD framework are thus that it supports integer variables and variable comparison (and their Boolean combinations) which is pretty standard, but also transposition of variables (swap of values) which is less common in public API of DD packages, but not very hard to implement.

IV.3.3 Symbolic Symbolic algorithm

Given these premises, we use the algorithm of figure IV.2 to canonize a set of states.

$\texttt{set\_canonize}(H \subseteq \mathcal{S}_n, S \subseteq \mathbb{N}^n)$:
repeat
  for $g \in H$ **do**
    $S' \leftarrow \{s | s \in S, g.s < s\}$
    $S \leftarrow S \cup g.S'$
    $S \leftarrow S \setminus S'$
  end for
until S no longer evolves
return S

Figure IV.2: Symbolic algorithm to canonize a set of states.

This algorithm is more compactly expressed using homomorphisms as

$$\texttt{set\_canonize}(H) = (\bigcirc_{g \in H} \texttt{IfThenElse}(\texttt{reduces}(g), \texttt{apply}(g), Id))^\star$$

This algorithm iterates over the permutations of H, applying each one only to the states that it reduces. If the permutations in H are permutations of the symmetry group G of the system, we are ensured that at each step of the algorithm, each state is either left as is, or mapped to a strictly smaller state belonging to its orbit. Since each orbit has a minimum (its canonical representative) this algorithm is guaranteed to converge.

Admittedly, the algorithm might visit each state of an orbit (in decreasing order, one by one), yielding worst case exponential complexity. Since the problem is equivalent to graph isomorphism, this is not surprising. In practice however, with an appropriate choice of a small set of permutations in H, this algorithm can be quite effective.

Let us note that the order in which the permutations of H are considered in the "for" loop (or equivalently, in the composition $\bigcirc_{g\in H}$) does not impact correctness, but may impede performance.

Actually, the choice of H is critical to overall performance of this algorithm. If $H = G$, then this algorithm converges after a single iteration of the outer loop ("repeat"). In other words, for each state, H contains the permutation that maps it to its representative. However, this means that, on the worst case, the size of H is exponential in n. This is congruent with the observations of [Cla+96] in which the orbit relation is shown to be exponential in representation size.

A contrario, when H is small, many iterations may be necessary for the algorithm to converge, but each element of H is likely to reduce larger subsets S'. Since the complexity of applying a permutation to a set of states is related to the representation size (in DDD nodes) and not to the number of states in the set, manipulating larger sets lowers the overall complexity.

a Monotonic$_<$ Property.

To obtain minimality, we would like to choose H such that $\texttt{set\_canonize}(H,S) = \{min[s]_G | s \in S\}$. In essence this means we require that any state s that is not the minimum of its orbit $[s]_G$ can be reduced (according to $<$) by applying a permutation of H.

monotonic$_<$ Let G be a subgroup of $\mathcal{S}_n$.
$H \subseteq G$ is *monotonic*$_<$ w.r.t. G if and only if:

- $\forall s \in S, (\exists g \in G, g.s < s \implies \exists h \in H, h.s < s)$.

In algorithm IV.2, when states can no longer be reduced by any permutation of H, by definition of the *monotonic*$_<$ property, the states in S are the canonical representatives of the input states. When H is *monotonic*$_<$ w.r.t. G, the algorithm returns the set of canonical representatives of the input states.The homomorphism $\texttt{set\_canonize}(H)$ can thus be applied to any set of states, yielding their canonical representatives when H is *monotonic*$_<$.

If H is not $monotonic_<$ w.r.t. G, the algorithm behaves like the one of figure IV.1 when several representatives are used.

Ideally H should be $monotonic_<$ w.r.t. G to obtain maximal reduction, and heuristically for decision diagram based implementations, H should be as small as possible.

Tools such as Bliss [JK07] can efficiently compute symmetries automatically and build a compact representation of arbitrary groups of permutations using the Schreier-Sims representation [Sim71] that consists of a small (guaranteed polynomial) generating set of permutations. However, the generating set they provide is not $monotonic_<$ in general, so it provides poor candidates as H set. Even when it is $monotonic_<$, its size can be much larger than necessary. For instance, the Schreier-Sims representation of the full group of permutations $\mathcal{S}_n$ is quadratic in n, whereas a $monotonic_<$ set of size n exists.

Unfortunately in the general case, the computation of a set H $monotonic_<$ w.r.t. G that is of minimal size, is in $O(n^n)$ with a brute force algorithm. But fortunately, we can exhibit pragmatic $O(n)$ choices of H for the most frequently encountered groups of symmetries in the literature : subsets of $\mathcal{S}_n$ for symmetric objects such as scalar sets, and rotations typical of ring topologies.

Indeed the set of transpositions of adjacent elements $\tau_{i,i+1}$ is $monotonic_<$ for $\mathcal{S}_n$, since computing the smallest lexicographical representative amounts to sorting. So by permuting two adjacent variables if they are in the wrong order we eventually reach a minimum.

For rotations, if r is the rotation $(2,3,\ldots,n,1)$, we have $H = \langle r \rangle = \{id, r, r^2 \ldots r^{n-1}\}$ which is the only $monotonic_<$ set, but fortunately remains of size n. This gives us $monotonic_<$ sets of size n for these two groups.

When the symmetries of the system arise from several symmetry groups (e.g. symmetries of subsystems), we can choose to use the union of their respective $monotonic_<$ sets. This set is not in general guaranteed to be $monotonic_<$ itself (unless the two symmetry groups act on disjoint sets of variables) but can still be used as a good candidate set for the algorithm.

IV.3.4 Illustrative example.

Let us detail the run of the algorithm on a small illustrative example. Figure IV.3 shows the intermediate DDD produced by the application of `set_canonize`(H) to a system of three variables. With $G = \mathcal{S}_3$ as symmetry group, we choose $H = \{\tau_{1,2}, \tau_{2,3}\}$ the set of transpositions of adjacent variables, which is $monotonic_<$ w.r.t. G. We focus on the inner loop in algorithm IV.2. Each step corresponds to the application of an element g of H to the states reduced by g in the current DD. At the end of the algorithm, another iteration is necessary to check for convergence.

As we can see through this toy example, each step of the algorithm simultaneously reduces several states. In a single step, each permutation reduces all the states it can, even if they belong to different orbits.

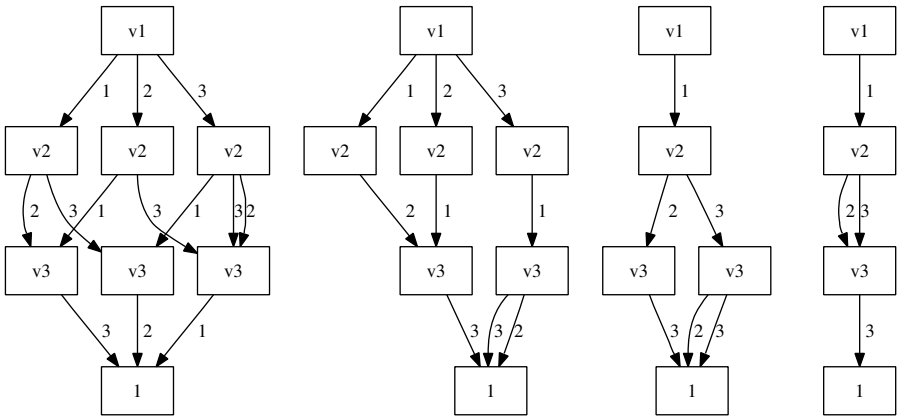


Figure IV.3: Using $G = \mathcal{S}_3$, we obtain $H = \{\tau_{1,2}, \tau_{2,3}\}$. `set_canonize`$(H)$ is applied to (a) containing the orbit of $(1,2,3)$ and the state $(3,3,1)$. This successively gives (b), (c), (d). (d) is the set of canonical representatives $\{(1,2,3),(1,3,3)\}$.

States that belong to the same orbit are progressively collapsed onto their representative. Because of sharing of sub-structures, notice that states $(2,1,3)$ and $(2,3,1)$ in a) are collapsed onto $(2,1,3)$ in b). $(2,1,3)$ is not a canonical representative, but it is smaller than $(2,3,1)$. At this step, the two states are merged, allowing to share any subsequent canonization step. In general, each step –with complexity polynomial in the DD size– might merge exponentially many states. This contrasts with explicit approaches that canonize all these states individually.

IV.4 Evaluation

Experimental evaluation reported in [Col+12; Col13] confirms that the symbolic-symbolic approach can outperform both explicit tools that build a quotient graph such as Lola, and symbolic tools such as ITS-tools without the symmetry reduction. The approach does have a hefty cost in both time and memory since it makes the transition relation more complex, so that the quotient reduction has to be significant before the cost is amortized. For complex groups involving cross products of symmetries, the quotient reduction can build very small DD representation whereas the normal DD representation does not scale well.

While the experimental results are promising, our set of benchmark models was relatively limited, since we lack a tool chain that provides good *monotonic*$_<$ generators in general. More work is needed at the language level to provide this information, possibly through a static analysis of statements [TMDM03] run on GAL models (see Chapter V).

IV.5 Conclusion

We have presented an algorithm to combine symmetry reduction with decision diagrams. Instead of directly representing the orbit relation which is worst case exponential [Cla+96], we introduced a "monotonic" function that converges to the same result. Because this function operates over sets of states, it avoids individual representative computations for each state, thus leading to a general and efficient algorithm to combine the use of symmetries with symbolic data structures.

While we only considered permutations, other types of symmetries on data values, such as $v = (obj_1, obj_2 \ldots ref_1, ref_2)$, and $g.v = (obj_{g.1}, obj_{g.2} \ldots g.ref_1, g.ref_2)$ can be integrated into our algorithm seamlessly. This symmetry is of interest as it corresponds to the case where obj_1 and obj_2 contain similar objects and ref_1, ref_2 are references to these objects, that need to be reindexed if we exchange the positions of the two objects. This case is encountered when canonizing the memory (heap in particular) of a concurrent system.

Part C

A Domain Specific Language for Concurrent Semantics

Domain specific languages (DSL) are small dialects dedicated to a particular domain. They help users to model domain artifacts at a high level of abstraction, then are used to generate more technical artifacts, such as tests, executable programs, or to perform various validations of the models. This part presents GAL a DSL to describe concurrent semantics, and the transformations to GAL we have developed for popular formalisms.

Chapter V presents ITS and GAL that together constitute a language based front-end for model-checking. ITS are focused on hierarchy, and GAL on description of arrays and integer arithmetic.

Chapter VI presents our multi-formalism symbolic model-checker ITS-Tools, and details the transformation process from existing formalisms or DSL to GAL.

Both of these contributions are designed to provide end users with a high-level way of accessing the technology and algorithms developed in parts A and B.

Chapter V

Instantiable Transition Systems and Guarded Action Language

V.1 A Language Based Front-end

Many languages have been proposed in the literature to express the behavior of concurrent systems. If we restrict ourselves to formalisms that support a model-checking approach, we can still identify four major families of interest.

1. Synchronous Languages describe a semantics where at each step, all variables are *synchronously* updated using a function of the previous values. The main concepts are input and output signals, transfer functions and latches to represent memory.

SMV and its evolutions NuSMV [Cim+02] and NuXMV [Cav+14] use such a language. VIS [Bra+96] uses a similar formalism, but has notions of components that can be assembled to build complex specifications.

Modeling concurrent systems with a synchronous language is difficult, it requires adding artificial idle behaviors, making the models more complex than needed. The notion of locality of an event is not present.

2. Petri nets (PN) are one of the most studied formal models for expressing concurrent semantics. Each transition takes tokens and puts tokens into places, emphasizing the notions of resource, locality and concurrency.

Many model-checkers for PN exist such as Lola [Sch03], Tina [BV06] or Marcie [HST09] (10 participants in *MCC*2016@*PN*).

Unfortunately, modeling of data and arithmetic is not supported in PN. Extensions that introduce data include Colored Petri Nets annotated with ML code [Rat+03], Renew nets annotated with Java [CHM16], or Tina's TTS extension to handle Fiacre [Ber+09] that uses compiled code, among many others.

In all of these examples, the data manipulation code added to transitions of the model is opaque, hiding arbitrary code fragments. This essentially restricts analysis to explicit model-checking, as the only way to know the behavior of the annotation is to execute it.

3. Communicating process models are both expressive and very relevant in the context of concurrent systems modeling. Each task executes a behavior described by a control flow graph or an automaton. Tasks can communicate using shared variables and channels that convey messages. Data manipulations are explicit, with data stored in discrete machine types (byte, int...) and fixed size arrays, and support for arithmetic and bitwise operators.

Promela, the language provided by SPIN [Hol97] is the best known example in this category. Divine [Bar+10] differs in syntax, but conceptually offers the same bricks as Promela. Uppaal [Beh+01] is designed for modeling networks of Timed automata, but also supports data manipulation.

These languages are well matched in expressivity by what our symbolic kernel offers, and are thus our primary target. They adequately capture locality of actions, data manipulation, and concurrency.

4. Code is the ultimate implementation of a design. Code analysis is very useful, since many programming errors are the source of critical failures. Tools that work on code search for typical implementation issues such as buffer overflow, heap related memory errors such as double free... Support for expressing more complex properties specific to a problem is mostly limited to safety assertions.

The software verification competition SVCOMP [Bey15] has a fast growing number of participants (from 10 in 2010 to 35 in 2016, though many submissions are small variants of each other). But it has only a single category for concurrency with programs that use the pthread API. Most tools don't support this category yet, with only 15 participants of which only 2 seem to apply sound methods (no negative score) in 2016. There is no benchmark for distributed memory concurrency since there is no standard model of communications in C.

Code includes many implementation and platform specific preoccupations, such as reliance on libraries, call stack, and heap allocations which are not necessarily relevant when designing a concurrent system but make verification much more complex.

Code can also only built in the later stages of development, which delays the validation and increases the cost of correcting potential design errors. Code level descriptions of designs are relevant for the software engineer and the compiler, but bear tenuous links to the business domain that the software actually addresses. In the software engineering process that drives the development, code of the software is only one artifact among many. This is particularly true of process used in the development of critical applications such as avionics. This type of distributed real-time embedded software is our main target, since the requirements on correctness are so high, the use of formal verification tools is appropriate.

Domain Specific Languages

Model-driven engineering (MDE) proposes to define domain specific languages (DSL), which contain a limited set of domain concepts [Voe+13]. This input is then transformed using model transformation technology to produce executable artifacts, tests, documentation or to perform specific validations. In this setting, languages such SMV, Promela or Petri nets have served as target of such model

transformations, since they are the input formalisms of model checkers. Transformations involving complex mechanisms (such as data manipulation or time management) that cannot be easily mapped to the underlying formal model typically create *gadget* (i.e. complex templates to translate a concept), thus increasing both the complexity of the model checking algorithms and the interpretation of results.

In this MDE context, an ideal target language for model checking should be expressive enough to support time, concurrency, compositions and data structures. It should also be simple and flexible enough to describe arbitrary transition relations. Finally, it should support efficient verification techniques such as decision diagrams, symmetries, etc.

GAL is designed as a convenient target formally expressing model semantics. Its syntax respects standards in programming languages (close to C or Java syntax), but has features expressing concurrency, synchronizations and fine control over atomicity of operations. With its symbolic back-end, it helps to bridge the gap between industrial specifications expressed using a DSL and symbolic model-checking tools.

V.2 Instantiable Transition Systems

V.2.1 Context

This section defines Instantiable Transition Systems (ITS), a framework designed to exploit the hierarchical characteristics of SDD for the description of component based systems.

This definition sets a contract that describes a labeled Kripke structure (LKS), i.e. a graph where states satisfy some atomic propositions, and edges bear a label (see chapters III and IV for formal definitions).

To model communications, these behaviors can be composed using label synchronization : two events must occur simultaneously or not at all. Our semantic definitions are aligned with standard labeled synchronized product definitions (e.g [Arn02; LL95]), with some extensions to deal with sequences of simple behaviors executed in a single semantic step. We consider a finite and fixed set of labels.

The down side is that pure event-based synchronization prevents direct communication using shared variables, which is another commonly used communication model. More precisely, the semantics of shared variables can only be expressed if they have a finite a priori known domain. For instance, to write into a shared buffer or channel, a different label can be used to represent each possible message. Since we expect models to be generated this is not a big issue, unless the shared variables have very large (or a priori unbounded) domains.

The benefits of event based synchronization is that we can make much weaker assumptions on how the LKS of components are produced. Event based synchronization allows for instance to use different formalisms to describe each component [Vit+04]. This also closely matches the possibilities of SDD, where edge

values can have arbitrary domains and composition of homomorphisms naturally expresses synchronization of actions.

It enables a very compact expression of cross-products of behaviors : if label a corresponds to n possible actions and b to m actions, the synchronization $a.b$ represents $n \times m$ possible behaviors. When generalizing to long sequences of k synchronization with n choices for each label, the representation using synchronizations is exponentially smaller ($n \times k$) than explicit modeling of each alternative (n^k).

Concerning the definition of actual components, we introduce a notion of type and instance, similar to the notion of class and object in object oriented programming. We separate specification of a) the definition of the behavior (component type), b) the definition of connectors, in our case a finite set of labels, and c) a particular assembly of component *instances* in a given configuration. This meets the essential requirements for an architecture description language (ADL) [MT00]. In this regard, we were particularly inspired by the Fractal component model [BCS09], though many other languages include the idea of instanciation.

V.2.2 Intuition

We wish to benefit from the abilities of our symbolic kernel, and leverage its powerful hierarchical representation with SDD, as well as the definition of complex behaviors built using an algebra $(\circ, +)$ of simpler behaviors.

The structure of a model will closely match the symbolic encoding using SDD. So some subcomponents could themselves be hierarchically decomposed, and their states are described using SDD. Some others components are elementary types such as automata, Petri nets, or GAL models whose state space is encoded using DDD.

Structurally, this tree of compositions can be addressed using a kind of Composite design pattern [Gam+95] : we have a) an abstract type contract that minimally defines an LKS, b) elementary types that do not have subcomponents, and whose semantics is described by LKS, c) composite types that aggregate instances of any type and define synchronizations amongst them.

One specific design goal of ITS was to be able to specify systems that are compact in the SDD solution with a compact model. The extreme example of 2^n philosophers can be solved in $O(n)$ with SDD, so the input describing the model should be of size at most $O(n)$. Instanciation coupled with composite type definitions are the key to solving this issue.

The definition of ITS has evolved quite a bit over time. The first version was called IPN, and considered only hierarchical compositions of Petri nets. We developped this version to serve as transformation target from UML behavioral models [TH08], and used it as an example in [HTMK09]. The first definition using the name ITS in [TM+09] allows arbitrary types for elementary components. However, probably because we came from Petri nets, we defined synchronizations over bags of labels (multiset) which induce needlessly complex notations.

Instead of bags, the definition in [TM+11] uses words over the alphabet of labels, which are both more expressive (sort the letters to obtain a bag) and more closely match standard definitions of labeled synchronized product [Arn02]. However, this definition sometimes forced to artificially add levels in the structure to describe some compositions of unions of behavior.

The current definition is aligned with the more recent definition of GAL and uses *statements*. This version supports arbitrary combinations using $\circ$ and $+$ of behaviors within a single composite. The definition of abstract ITS type has been simplified with now a single successor relation (rather than distinguished *local* actions) that computes successors by a single label (rather than a word in the alphabet of labels). The complexity of the previous definitions is now isolated in the concrete *composite* type definition.

V.2.3 ITS Type and Instance

ITS describe a minimal Labeled Transition System (LTS) style formalism using notions of *type* and *instance* to emphasize locality of actions and to exploit the similarity of copies of a given type. The composition mechanism is based solely on transition *synchronizations* (no explicit shared memory or channel).

Abstract Syntax

Notation: For a tuple $z = \langle X, Y, \cdots \rangle$, we denote by $z.X, z.Y \ldots$ the elements $X, Y, \ldots$.

The following definition sets an abstract contract or interface that must be implemented by concrete ITS types.

Definition An **ITS type** is a tuple $\tau = \langle S, A, Succ \rangle$ where:

- S is a set of states; A is a finite set of action labels that contains a *local* label noted τ;
- $Succ : S \times A \mapsto 2^S$ is a transition function.

An ITS type can be instantiated, possibly several times. With an instance i is associated its ITS type $type(i)$.

Semantics

Reachability Definition: Let i be an ITS instance and s, s' be two states in $type(i).S$. State s' is reachable from s if there exist states $s_0, \ldots s_n \in type(i).S$ such that $s = s_0$, $s' = s_n$ and for all j, $1 \leq j \leq n, s_j \in type(i).Succ(s_{j-1}, \top)$.

The function *Succ* produces successors of a given state reachable by an edge bearing a particular label from the alphabet. Note that *Succ* is the only way to control the behavior of a (sub)system from outside. We support non determinism, a state can have several successors by a given label.

The transition relation of a full system can only be defined in terms of subsystem synchronizations using *Succ* and of independent local behaviors. Hence, a full system is defined by a single instance of a particular type in a specific initial state:

the system is self-contained and thus reachability only depends on the definition of *Succ* using the distinguished local label $\top$.

Remarks. This definition is enriched in practice by specifying an initial state, as well as state-based predicates giving a Kripke state labeling for model-checking purposes.

Symbolic Encoding

To fit in the framework of SDD, we require that the set of states S be represented by an SDD, and that the transition relation *Succ* be represented by a set of homomorphisms, one per label $Succ(\lambda)$.

V.2.4 Composite ITS

We now define a *composite ITS type*, designed to offer support for the hierarchical composition of ITS instances. This new version is enhanced with respect to [TM+11].

An example of composition is given in Fig. VI.4 using our concrete syntax.

Abstract Syntax

Notations: For a tuple $I = (i_1, \ldots, i_n)$ of ITS instances, $|I|$ denotes the size n of I, S_I is the set $type(i_1).S \times \ldots \times type(i_n).S$. For $s \in S_I$ and i an instance, we denote $s[i]$ the component of s that corresponds to i.

Given a tuple I of ITS instances and a set *Lab* of labels, we inductively define the set $Stat_C$ of **composite statements** from :

- $\langle call(i,\lambda)\rangle$ a call statement to a label λ of $type(i)$, that invokes a transition of instance i labelled by λ,
- $\langle \sigma_0; \ldots; \sigma_k\rangle$ a sequence of statements in $Stat_C$ executed in order. The empty sequence has no effect.
- $\langle call_{self}(\lambda)\rangle$ a call statement to a label λ of *Lab*, that invokes a synchronization of the current composite type, labelled by λ. This allows to structure the transition relation and chain behaviors. We syntactically forbid cycles of self-calls.

Definition A **composite** over alphabet *Lab* is a tuple $C = \langle I, Sync\rangle$ where:

- I is a tuple of ITS instances, said to be *contained* by C. We further require that the type of each ITS instance already exists when defining I, in order to prevent circular or recursive type definitions.
- $Sync \subseteq Lab \times Stat_C$ is the finite set of synchronizations, where for $t = \langle \lambda, \sigma\rangle \in Sync$, λ is the label of t and σ its body.

Semantics

Definition Let $C = \langle I, Sync \rangle$ be a composite over alphabet Lab.

Let the **next state function by a statement** $Next_I : S_I \times Stat_C \mapsto 2^{S_I}$, be defined for $s, s' \in S_I$ and $\sigma \in Stat_C$ by : $s' \in Next_I(s,\sigma)$ *iff*

$$\begin{cases} \exists s_0 \dots s_{k+1}, s_0 = s, s_{k+1} = s',\ \forall i \in [1 \dots k], s_{i+1} \in Next_I(s_i, \sigma_i) & \text{if } \sigma = \langle \sigma_0; \dots; \sigma_k \rangle \\ s'[i] \in type(i).Succ(s[i], \lambda) \wedge \forall j \in I, j \neq i, s'[j] = s[j] & \text{if } \sigma = \langle call(i,\lambda) \rangle \\ \exists t = \langle \lambda, \sigma' \rangle \in Sync,\ such\ that\ s' \in Next_I(s, \sigma') & \text{if } \sigma = \langle call_{self}(\lambda) \rangle \end{cases}$$

The ITS type $\tau_C = \langle S, Lab, Succ \rangle$ corresponding to C, is defined by:

- $S = S_I$
- $Succ : S \times A \mapsto 2^S$ is defined for $s, s' \in S, \lambda \in A$ by $s' \in Succ(s,\lambda)$ iff.
  $\lambda = \top$ and $\exists i \in I, s'[i] \in type(i).Succ(s[i], \top) \wedge \forall j \in I, j \neq i, s'[j] = s[j]$
  or $\exists t = \langle \lambda, \sigma \rangle \in Sync$, such that $s' \in Next_I(s, \sigma)$

This definition thus describes an implementation of the generic ITS type contract. It contains either elementary instances (such as LTS, Petri nets or GAL models), or inductively other instances of composite nature.

The set of successors $Succ(s,\lambda)$ is obtained as the union of the successors obtained by executing the body statement of any synchronization with label λ. This models non determinism. For the local label $\top$, the definition adds to the effect of synchronizations labelled $\top$ the states resulting from the action of $\top$ on an arbitrary nested instance, without modifying the other instances. This corresponds to the standard interleaving semantics governing local evolutions of subcomponents.

When the statement is a sequence, the whole sequence is executed in a single atomic step. So the granularity of concurrency, i.e. what constitutes a semantic step, is defined by synchronizations.

When the statement is a nested instance call, only the state of the specific target instance evolves. A classical vector of synchronization [LL95] can be seen as a sequence of calls to labels of nested instances, where each instance is called exactly once.

When the statement is a self call, we (inductively) produce a union of the behaviors of synchronizations bearing the target label. Loops of self-calls are syntactically forbidden, so the behavior must be finite.

Symbolic Encoding

Each instance $i \in I$ of the composite is modeled as an SDD variable v_i. The domain of the variable is given by $\mathrm{Dom}(v_i) = type(i).S$.

The local evolution of an instance uses the *local* homomorphism (see II.1.3) to carry the homomorphism $Succ(\lambda)$ to the appropriate variable, i.e. $\langle call(i,\lambda) \rangle$ becomes $\mathcal{L}(type(i).Succ(\lambda), v_i)$.

The sequence of statements finds an immediate analogy in homomorphism composition with $\circ$.

The union of behaviors that comes from using the same label for several synchronizations is translated to the sum $+$ of homomorphisms. The self call combined with the sequence allows to compose such sums, so that the composite type

can specify arbitrary combinations using $\circ$ and $+$ of behaviors, with elementary behaviors being evolutions of subsystems.

The heavy use of the local homomorphism helps trigger the automatic saturation for SDD (see II.2.3 *e*)).

V.2.5 Scalar ITS

While the definition of an ITS composite permits hierarchical modeling, the notion of *Scalar Set* ITS type, where the synchronizations are defined in a parametric way, deals with "regular" or symmetrically composed systems. This definition is not more expressive than the one for a composite but it allows us to build several equivalent composite representations of a system (see philosophers example in chapter I), with a possible impact on performances. We thus offer a way of describing symmetric models, so that the manually built recursive encodings presented in [TM+09] can be easily applied to symmetric problems. The scalar set captures a frequent symmetric synchronization pattern when using a set of identical instances and its definition is tvery close to those proposed in symmetric Uppaal [Hen+04], Murphi [NID96] or in symmetric nets [Chi+90].

Abstract Syntax

A scalar is a kind of composite ITS that contains a set of n instances all of the same type τ. Given an ITS type τ for instances and a set *Lab* of labels, we inductively define the set $Stat_S$ of scalar statements from :

- $\langle \sigma_0; \ldots; \sigma_k \rangle$ a sequence of statements in $Stat_S$.
- $\langle call_{any}(\lambda) \rangle$ a call statement to a label λ of *any* instance, such that a single instance (chosen arbitrarily) progresses by λ.
- $\langle call_{all}(\lambda) \rangle$ a call statement to a label λ of *all* instances, such that all n instances simultaneously progress by λ.
- $\langle call_{next}(\lambda, \lambda') \rangle$ a circular synchronization where instance i (chosen arbitrarily) progresses by λ and its successor $i+1$ (computed modulo n) progresses by λ'.
- $\langle call_{self}(\lambda) \rangle$ a self call identical in nature and purpose to the composite

Definition A **scalar** over alphabet *Lab* is a tuple $C = \langle \tau, n, Sync \rangle$ where:

- τ is the type of the contained instances (no circular definitions allowed)
- $n \in \mathbb{N}$ is the size of the set
- $Sync \subseteq Lab \times Stat_S$ is the finite set of synchronizations, where for $t = \langle \lambda, \sigma \rangle \in Sync$, λ is the label of t and σ its body.

Semantics

The ITS semantics of a scalar can be deduced from its transformation to a normal composite type.

A scalar $C = \langle \tau, n, Sync \rangle$ can be interpreted as a composite tuple $C = \langle I, Sync' \rangle$ where $|I| = n$ and $\forall i \in I, type(i) = \tau$. Each synchronizations in $Sync$ has an image in $Sync'$ with the same label and transformed statements. Statements common to composite and scalar are left unchanged. The statements specific to the scalar are transformed to composite statements :

- The statement $\langle call_{any}(\lambda) \rangle$ introduces a new label l, and n synchronizations in $Sync'$ such that $\forall i \in I, \exists \langle l, \langle call(i, \lambda) \rangle \rangle \in Sync'$. The statement is then replaced by a self call $\langle call_{self}(l) \rangle$.
- The statement $\langle call_{all}(\lambda) \rangle$ is replaced by a sequence of n statements $\langle call(i, \lambda) \rangle$ for all instances $i \in I$.
- The $\langle call_{next}(\lambda, \lambda') \rangle$ circular synchronization is handled similarly to the *any* case except that the body of the newly introduced synchronizations is a sequence of two statements $\langle \langle call(i, \lambda) \rangle; \langle call(i+1[n], \lambda) \rangle \rangle$. We use $i+1[n]$ to designate the next instance modulo n.

A scalar that uses at least one circular synchronization statement implicitly declares an successor relationship between the instances, following a ring topology. These particular types of symmetric synchronization patterns are well studied in the litterature. We were particularly influenced by Well-formed nets [Chi+90] when defining the scalar type.

A scalar set represents a regular model pattern and produces a homogeneous representation of parametric models. Furthermore, because this pattern is very constrained, different semantically equivalent encodings can be considered at the SDD level.

Symbolic Encoding

In [TM+09], several strategies were manually experimented to encode such regular models, the most basic one building a composite containing n instances of the embedded type as described above. This can be generalized by building a composite of n/k instances of a composite containing k instances (or $k+1$ to capture the remainder of the division n/k) of the basic type. More subtle are recursive encoding strategies, where the type of a (sub-)composite containing k instances is itself defined as a group of groups of instances. This recursive strategy leads in some cases (like for the dining philosophers) to logarithmic overall complexity in time and memory.

With these additional definitions of scalar set, the encoding strategy can be configured by the user at run time, by simply setting an option. Two parameters guide the encoding: The width gives the number of variables at any given level of composite, and the depth gives the number of levels of hierarchy or nesting introduced. The user can choose to bound one or the other and select the more

efficient. For instance the flat encoding of Fig. I.3 *b*) has width 4 and depth 1, while the encoding of Fig. I.6 (center) has width 2 and depth 2 (not counting the "CloseLoop" artifact).

We thus offer a language based way to possibly benefit from the very favorable recursive foldings of SDD, which thanks to hierarchy can be exponentially more efficient than what is available with other decision diagram variants.

V.3 Guarded Action Language

V.3.1 Context

To use a homomorphism based symbolic engine, the approach initially proposed in [Cou+02] is direct encoding of the semantic bricks with user-defined inductive homomorphisms, then combining them to produce the desired effect.

This approach was applied successfully, during the thesis of V. Beaudenon for a subset of Promela [BET10], by F. Breant for a DSL called LfP [Gil+04], by ourself for Petri net variants [HTMK09], then for time Petri nets with discrete time assumptions [TM+11]. We also built a lot of other custom homomorphisms, e.g. for set canonization [TMIP04b] or to build a behavioral model of an automatic highway [Bér+08]...

However, with the breakthrough of homomorphism rewriting [HTMK08], custom homomorphisms became much less useful. Before this, writing a single homomorphism that both tests and updates a variable (as originally proposed in [Cou+02]) was more efficient than writing a composition of test and update. With rewriting, the reverse is usually true, as the dynamic grouping of operations allows to share traversals of the DD, and the composition of smaller bricks potentially leads to reuse of cache entries. We instead started building up a library of general purpose homomorphisms, described in section II.1.3. The essential brick providing efficient and general evaluation of expressions and array [Col+12] enabled definition of a new elementary type for ITS : the Guarded Action Language GAL.

V.3.2 Intuition

GAL is intended to be the target of model to model transformations. We thus develop an efficient symbolic encoding of GAL semantics only once, and let users define transformations to GAL, rather than defining and implementing homomorphisms themselves.

The language must serve as intermediate or pivot for various formalisms and domain-specific languages. It should therefore make very few hypothesis on the nature of the model, and keep high level semantic concepts to a bare minimum. For instance, we shouldn't introduce concepts such as channels or process, because these inevitably come with semantic variations such as bounds, urgency or priority. One should be able to describe such semantics however, so we want high expressivity with a low number of concepts.

This approach is similar to the evolution seen in compilers, where the definition of a machine independent assembly code such as LLVM [LA04] allows to define the semantics of several programming languages separately by compiling to LLVM. Then LLVM facilities such as optimization and the support for execution on diverse hardware are shared amongst languages.

Transforming (compiling) a formalism or DSL to GAL provides the same benefits : besides enabling model-checking with our symbolic engine (or hardware in the analogy), transformations to GAL will allow to profit from static analysis and rewriting defined at GAL level, as well as support for analysis of GAL with other technologies such as partial order reductions, SAT/SMT... We thus designed GAL as an *assembly language to describe concurrent semantics*.

V.3.3 GAL definition

We define GAL as a pivot language that essentially describes a generator for a labelled finite Kripke structure using a C like syntax. This simple yet expressive language makes no assumptions on the existence of high-level concepts such as processes or channels. While direct modeling in GAL is possible (and a rich eclipse based editor is provided), the language is mainly intended to be the target of a model transformation from a (high-level) language closer to the end-users.

A **GAL** model contains a set of integer variables and fixed size integer arrays defining its state, and a set of guarded transitions bearing a label chosen from a finite set. We use C 32 bit signed integer semantics, with overflow effects; this ensures all variables have a finite (if large 2^{32}) domain. GAL offers a rich signature consisting of all C operators for manipulation of the `int` and `boolean` data type and of arrays (including nested array expressions). There is no explicit support for pointers, though they can be simulated with an array *heap* and indexes into it. In any state (i.e. an assignment of values to the variables and array cells of the GAL) a transition whose boolean guard predicate is true can fire executing the statements of its body in a single atomic step. The body of the transition is a sequence of statements, assigning new values to variables using an arithmetic expression on current variable values. A special $call(\lambda)$ statement allows to execute the body of any transition bearing label λ, modeling non-determinism as a label based synchronization of behaviors. A special fixpoint instruction is provided allowing to express modal μ-calculus least and greatest fixpoints thus giving the language a potent expressive power.

Abstract Syntax

An example of a GAL system that uses its concrete textual syntax is given in Fig. VI.3.

We omit the semantic definition details related to arithmetic and Boolean expressions, since they have the same syntax and semantics as in the C language. We use C 32 bit signed integer semantics, with overflow effects; this ensures all variables have a finite (if large 2^{32}) domain. We assume $\mathbb{Z} = [-2^{31}, \ldots, 2^{31} - 1]$ in the following.

Given a set of variables and of arrays, the set Int_e of integer expressions is the smallest set containing integers (constants), variables, array access expressions (an array name and an expression for the target index), the combinations of expressions with binary operators $+$ (add), $*$ (multiply), $/$ (divide), $\%$ (modulo), $-$ (minus), $\ll$ (left-shift), $\gg$ (right-shift), $\hat{}$ (bitwise xor), $|$ (bitwise or), $\&$ (bitwise and), unary minus $-$ and bitwise complement $\tilde{}$ of an integer expression.

Boolean expressions $Bool_e$ are inductively defined using constants true, false, comparisons using $<$ (strictly less than), $<=$ (less than), $==$ equals, $!=$ (differs), $>=$ (greater than), $>$ (strictly greater) between two integer expressions, as well as Boolean combinations using $\&\&$ (and), $||$ (or), $=>$ (implies) of two Boolean expressions, or the ! (negation) of a Boolean expression. They can also be embedded into integer expressions, true being interpreted as 1 and false as 0.

GAL transition effects are described by statements in $Stat$, a set inductively built from:

- $\langle lhs = rhs \rangle$ an assignment of an integer expression rhs to a variable or to the cell of an array designated by lhs,
- $\langle \sigma_1; \ldots; \sigma_k \rangle$ a sequence of k semi-colon separated statements. We note nop (no-operation) the empty sequence of statements,
- $\langle call(\lambda) \rangle$ a call statement to a label λ, that non-deterministically invokes one of the transitions labelled λ.

  Note that circular $call$ expressions are forbidden, disallowing recursion.

Assignments compute the value of the right hand side and update the value of the left hand side. We thus have sequential update semantics, similar to most programming languages, but different from the synchronous assumptions of SMV [Cim+02]. The sequence allows to chain the effect of several statement. The *call(λ)* statement models non-determinism; the transition chosen is any transition bearing label λ, allowing a state to have several successors. It functions like the self-call of the composite type of section V.2.4.

More formally:

Definition A GAL system over a set Lab of labels containing $\top$ is a tuple $\mathcal{G} = \langle Vars, Arrays, Trans \rangle$ where:

- $Vars$ is a set of integer variables,
- $Arrays$ is a set of integer arrays; for $a \in Arrays$, we let $|a|$ designate its fixed size,
- $Trans \subseteq Lab \times Bool_e \times Stat$ is a set of transitions; for $t = \langle \lambda, g, \sigma \rangle \in Trans$, $\lambda \in Lab$ is the label of t, $g \in Bool_e$ is the $guard$ of t, and $\sigma \in Stat$ is the $body$ of t.

Semantics

Let $\mathcal{G} = \langle Vars, Arrays, Trans \rangle$ be a GAL over alphabet Lab.

The semantics of $\mathcal{G}$ is defined as a transition system whose set of states is $\mathcal{S} = \mathbb{Z}^{|Vars|} \times \Pi_{a \in Arrays} \mathbb{Z}^{|a|}$, giving a value to each variable and array cell. Because $\mathbb{Z}$ is assumed to be finite, $\mathcal{S}$ is finite (resolving most decidability issues). Interpretation $e(s)$ of an expression e in a state s yields a constant value belonging to the range of e (integer or boolean). When lhs designates the left-hand side of an assignment, it must be either a variable (x) or an array access expression ($tab[i]$). In either case we let $lhs(s)$ denote the fully resolved variable, when interpreting the array index expression in state s. Conversely for any variable or array cell v, we note $s[v]$ the value of v in state s.

Let the next state function by a statement $Next_I : S \times Stat \mapsto 2^S$, be defined for $s, s' \in S_I$ and $\sigma \in Stat$ by : $s' \in Next_I(s, \sigma)$ *iff*

$$\begin{cases} s'[lhs(s)] = rhs(s) \wedge \forall v \neq lhs(s), s'[v] = s[v] & \text{if } \sigma = \langle lhs = rhs \rangle \\ \exists s_0 \dots s_{k+1}, s_0 = s, s_{k+1} = s',\ \forall i \in [1 \dots k], s_{i+1} \in Next_I(s_i, \sigma_i) & \text{if } \sigma = \langle \sigma_0; \dots; \sigma_k \rangle \\ \exists t = \langle \lambda, g, \sigma' \rangle \in Trans, g(s) \wedge s' \in Next_I(s, \sigma') & \text{if } \sigma = \langle call(\lambda) \rangle \end{cases}$$

The ITS type $\tau_G = \langle S, Lab, Succ \rangle$ corresponding to $\mathcal{G}$, is defined by:

- $S = \mathbb{Z}^{|Vars|} \times \Pi_{a \in Arrays} \mathbb{Z}^{|a|}$
- $Succ : S \times A \mapsto 2^S$ is defined for $s, s' \in S, \lambda \in A$ by $s' \in Succ(s, \lambda)$ iff. $\exists t = \langle \lambda, g, \sigma \rangle \in Sync$, such that $g(s)$ holds and $s' \in Next_I(s, \sigma)$

The non-deterministic *call* construct combined with the sequence is particularly important to allow expression of transition relations that are a composition of sum of effects (e.g action a or a' followed by action b or b'). Making all the alternatives explicit ($ab, ab', a'b, a'b'$) could lead to an exponential blowup of the model size.

Symbolic Encoding

States are sequences of integers, which can be represented by DDD. Assignment is implemented using the algorithms described in section II.3. Guards are implemented as selector homomorphisms. The sequence is described using composition $\circ$ and the call introduces a sum $+$ of behaviors.

V.3.4 Parametric GAL

GAL also features parametric constructs to comfortably express common patterns. They can be degeneralized, and amount to syntactic sugar.

If-Then-Else. This classic control structure can be rewritten to a pair of transitions and a call, but is more familiar to users than the latter. The **abort** statement returns the empty set of successors, and can be used to discard some branches of the alternative.

Range. We let a GAL definition contain the definition of named subsets of $\mathbb{Z}$ called *ranges*. A transition t can bear an arbitrary number of formal parameters, each having a name and range. The parameters can be used like ordinary variables

within the definitions of the guard and body of t, though they cannot be assigned new values. They can also be used within the definition of the label of t, and in the definition of labels used in any call statements of the body of t.

Parametric transition. Defining such a transition t is equivalent to defining a set of transitions, containing one transition for each element in the cartesian product of the parameter ranges (i.e. each possible assignment of values to parameters). In each of these transitions which have no parameters, the guard, body and label of t are replaced by a version where each parameter reference is replaced by a constant (its assigned value). Occurrence of a parameter in a label (that of t itself or occurring in a call of the body of t) builds a new label where the parameter is substituted by a string representing its numeric value.

This mechanism is similar in many ways to the way colored Petri net transitions are defined with respect to their unfolded P/T net version. This construct makes specifications much more compact and readable in many cases. It also eases traceability when the GAL model is obtained by a model transformation. It also helps exhibit nice symmetry properties of the transition relation, depending on how the parameters are actually used in the guard and body of t. Lastly, reasoning on parameters before discarding them through instantiation can allow to significantly reduce the transition relation representation size.

Sequential iteration. Given these ranges, we also introduced a limited iteration $\langle for(p : r)\{b\}\rangle$ statement (for each p in r, do b), where p is a parameter with range r and b is a body statement. It is equivalent to a sequence of $|r|$ statements $\langle b_1; \ldots; b_{|r|}\rangle$, where each b_i is the statement b where the parameter p is replaced by its value in r. This construct mostly eases modeling when manipulating arrays. It can be seen as a dual for the use of parameters in transitions (that builds a sum or union of $|r|$ effects), since it builds a composition of $|r|$ effects.

Instantiation. GAL models are structurally analyzed before model-checking, allowing to simplify away the parametric features. This analysis simplifies expressions that can be statically evaluated, removes structurally unreachable behaviors (e.g. transitions with false guards), and instantiates parameters with on the fly simplifications. Other simplifications and rewritings (described on the webpage) are also available, some of which are more involved such as attempting to rewrite transitions with several parameters as a sequence of calls to transitions with a single parameter each. When parameters are in fact independent (no statement uses them both), having a sequence of choices (rather than the explosion due to choosing all parameter values at once) leads to a transition relation in desirable composition of sums of effects form, with possibly an exponentially more compact GAL specification than plain instantiation of parameters.

One-hot. Any variable or array can be tagged with the "hotbit(r)" keyword, indicating the user wants a one-hot state encoding, where a variable with domain $0..n-1$ is encoded as n Boolean variables with only one "hot" bit set to 1. Apart from this keyword at declaration, the variable is manipulated normally in the GAL syntax. We then use a GAL to GAL transformation to instantiate such variables, translating accesses and assignments to the variable to reflect the one-hot encoding.

The translation involves adding a parameter to represent the current value and testing that its corresponding bit is hot in transition guards. We further automatically identify and tag variables that could benefit from one-hot encoding: any variable that is only assigned constants (this allows to statically compute the range) and whose domain size is greater than a threshold (we use 8) is set by default to one-hot encoding. By increasing locality, one-hot encodings can be favorable to DD techniques, for instance this feature is often used to encode locations of automata in symbolic model-checkers.

At model-checking time, every statement is encoded as a symbolic operation. ITS-tools then fully exploits commutativity and on-the-fly simplifications at every level of the evaluation to adaptively exploit the structure of the decision diagram encoding the states (see [HTMK08; Col+13]).

Textual GAL files can of course be built directly, but we also offer an EMF [Ste+09] compliant meta-model of GAL that can be used in conjunction with MDE tools and manipulated programmatically in Java. Several rewriting rules then perform simplification and optimization of GAL models, that benefit all input formalisms. The reduced GAL model can then be used for model-checking using *ITS-tools*.

V.4 Evaluation

The ITS and GAL language provide a high level framework to express the semantics of concurrent systems and their composition. Composition of behaviors is based on classical products of labelled transition systems. We use GAL to model the elementary components, with a rich syntax and expressivity.

The most similar approach is LTSMin [BPW10], that also offers support for multi-formalism symbolic model-checking. The essential difference is that we require that all the semantics be described (using a GAL model), while LTSMin has opaque functions of k variables, which must be provided as code implementations. Opaque functions are better to integrate existing code (explicit checker) or data structures using a unique index for complex objects (such as DBM in Opaal adapter [Dal+12]). However, not having full knowledge of the semantics induces some inefficiency in symbolic representation of transition relations (especially when support of transition grows, see discussion in section II.3), and precludes analysis by methods such as SAT/SMT.

We think providing a language based front-end rather than a "contribute code" front-end is easier for end-users and better aligned with MDE trends.

V.5 Conclusion

This chapter presented the Instantiable Transition Systems (ITS) framework and the Guarded Action Language (GAL). ITS has been designed for the description of component based systems, while GAL is a C-like description of the components.

Both are connected with our verification library where the states of the resulting systems are encoded with various kinds of decision diagrams. More precisely, the hierarchical characteristics of systems use Hierarchical Set Decision Diagrams SDD, while the data content is encoded with Data Decision Diagrams DDD, using the recent efficient algorithms of [Col+13] to encode GAL semantics.

We claim that the association GAL/ITS, with the underlying symbolic engine based on Decision Diagrams, provides a flexible and efficient solution to model checking-based analysis.

Chapter VI

Applications and Case Studies

VI.1 A Multi-Formalism Model-Checker

We have invested a significant effort in the implementation of tools for model-checking. Tools are by themselves a significant scientific contribution to our eyes. Particularly if they are distributed as FOSS (Free Open Source Software), providing a platform for sharing development between tools at source code level, and for the development of new algorithms.

We present in this chapter the verification toolset ITS-tools, featuring a symbolic model-checking back-end engine based on hierarchical set decision diagrams (SDD) that supports reachability, CTL and LTL model-checking and a user-friendly eclipse based front-end. We support as primary input the general purpose formalism GAL.

ITS-tools offers symbolic model-checking of large concurrent specifications expressed in a variety of formalisms: communicating process (Promela, DVE), timed specifications (Uppaal timed automata, time Petri nets) and high-level Petri nets. We are focused on verification of (large) globally asynchronous locally synchronous specifications, an area where DD naturally excel due to independent variations of (small) parts of the state signature.

We leverage model transformation technology to support model-checking of formalisms or domain specific languages (DSL). Models are transformed to GAL, a simple yet expressive language with finite Kripke structure semantics.

Most of these elements are visible in Fig. VI.1.

VI.1.1 Symbolic Kernel

ITS-tools use symbolic representations of sets of states using decision diagrams to face the combinatorial state space explosion of finite concurrent systems. Its kernel is **libDDD**, a C++ decision diagram library supporting Data Decision Diagrams DDD and hierarchical Set Decision Diagrams SDD and was described in part A. Operations on these decision diagrams are encoded using homomorphisms

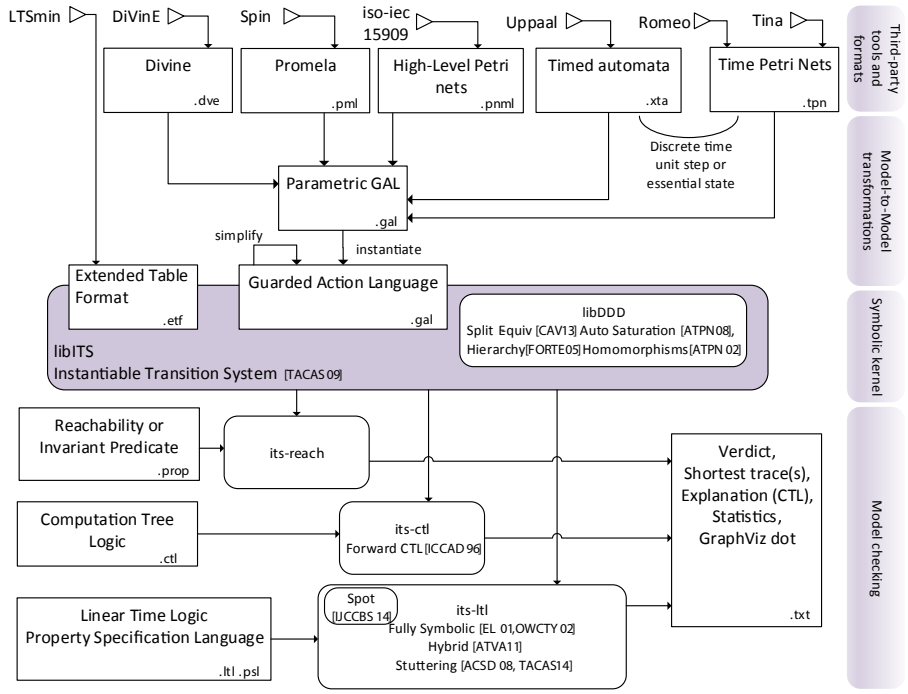


Figure VI.1: Architecture of ITS-tools. Square boxes are files, rounded boxes are tools.

(see chapter II), giving a user great flexibility and expressive power. The library can automatically and dynamically rewrite these operations to produce saturation effects in least fixpoint computations (see section II.2). The Split-equiv algorithm (see section II.3) enables efficient evaluation of complex expressions including array subscripts and arithmetic, a feature heavily used to symbolically encode the semantics of GAL.

libITS is a C++ library built on top of libDDD, offering a simple and uniform API to write symbolic model checking algorithms for any system that can be described as an Instantiable Transition System (ITS). An ITS is essentially a labeled transition system with successor and predecessor functions described as operating on sets of states, and a boolean predicate function enabling state based logic reasoning. The tool supports compositions of labeled transition systems by directly using hierarchy in the state representation reflecting the composition (see chapter V) libITS has native adapters for several formalisms (not represented on the figure), we focus in this chapter on GAL.

ETF support A native ETF to ITS adapter is provided with libITS, supporting this output format of LTSmin. ETF files [BPW10] represent the semantics of a finite Kripke structure in a format adapted to symbolic manipulation. This allows to analyze (CTL, LTL) models expressed in the many formalisms that LTSmin supports, provided generation of ETF succeeds (essentially if LTSmin can compute

all reachable states).

VI.1.2 Model-checking

Using the ITS API we have built several model-checking tools. The tool **its-reach** can compute reachable states, and shortest witness paths (one or more if so desired) to target states designated by a boolean predicate. In a discrete time setting, this can be used to compute best or worst case time bounds on runs. It can also perform bounded depth exploration of a state space (a.k.a. bounded model-checking). It implements several heuristics to compute a static variable order for the input model, the default is based on FORCE [AMS03].

The tool **its-ctl** performs verification of CTL properties (though fairness constraints are currently not supported). It reuses a component of VIS [Bra+96], a model-checking tool for verification and synthesis of gate level specifications, to transform input formulae into forward CTL form [INH96]. Forward CTL often allows (but not always) to use the forward transition relation alone, which is easier to compute than the backward (predecessor) transition relation. When a statement destroys information (i.e. is not reversible), backward exploration requires to compute potential domains for variables, based on the set of reachable states. This leads to an over-approximation, whose refinement may be costly if it requires to intersect decision diagrams (see [Col13] for more details). Hence forward CTL verification is more efficient in general, and furthermore many subproblems can be solved using least fixpoints (e.g. Forward Until) that benefit from automatic saturation at DD level.

The tool **its-ltl** performs hybrid (i.e. that build an explicit graph in which each node stores a set of states as a decision diagram) or fully symbolic verification of LTL and PSL properties. The transformation of the formula into a (variant of) Büchi automaton and the emptiness checks of the product for hybrid approaches rely on Spot [DL+16; DL14], a library for LTL and PSL model-checking. Fully symbolic model-checking uses forward variants of Emerson-Lei [EL87] or One-Way Catch Them Young [SRB02]. The hybrid approaches efficiently exploit saturation and often outperform fully symbolic ones [DL+11]. When the property is stuttering invariant (e.g. $LTL \setminus X$) we also offer optimized hybrid [KP08] and fully symbolic [BS+14] algorithms that exploit saturation.

Other prototypes for solving games [Zha+10] and to exploit symmetries [Col+12] on top of decision diagrams have been built, showing the versatility of the ITS API, but these tools are not part of the current release.

VI.1.3 Model transformations

Model-driven engineering (MDE) proposes to define domain specific languages (DSL), which contain a limited set of domain concepts [Voe+13]. This input is then transformed using model transformation technology to produce executable artifacts, tests, documentation or to perform specific validations. In this context

GAL is designed as a convenient target formally expressing model semantics. We thus provide an EMF [Ste+09] compliant meta-model of GAL that can be used to leverage standard meta-modeling tools to write model to model transformations. This reduces the adoption cost of using formal validation as a step of the software engineering process.

We have implemented translations to GAL for several popular formalisms used by third party tools. We rely on XText [EB10] for several of these: with this tool we define the grammar and meta-model of an existing formalisms, and it generates a rich code editor (context sensitive code completion, on the fly error detection,...) for the target language. The editor obtained after some customization is then often superior to that of the original tool. We applied this approach for the DVE language of DiVinE [Bar+13], the Promela language of Spin [Hol97] and the Timed Automata of Uppaal [Beh+01] (in Uppaal's native XTA syntax).

The translation for DVE (succinctly presented in [Col+13]) is quite direct, since the language has few syntactic constructs, and they are almost all covered by GAL. Channels are modeled as arrays, process give rise to a variable that reflects the state they are in. Similarly, the translation for Promela presents no real technical difficulty, although a first analysis of Promela code is necessary to build the underlying control flow graph (giving an automaton for each process). We currently do not support functions and the C fragment of Promela.

Discrete time. The support for TA and TPN uses discrete time assumptions. Note that analysis in the discrete setting has been shown to be equivalent to analysis in a dense time setting provided all constraints in the automata are of the form $x \leq k$ but not $x < k$ [HMP92; Bey01]. This is due to the fact that if bounds are open, timings in zones with non integer values may equivalently be represented (in terms of future behavior) by an integer timing touching one of its border. For both of these formalisms, we build a transition that represents a one time unit delay and updates clocks appropriately. This transition is in fact a sequence of tests for each clock, checking if an urgent time constraint is reached (time cannot elapse), if the clock is active (increment its counter) or if it is inactive either because it will be reset before being read again, or because it has reached a value greater than any it could be tested against before a reset (do nothing). A transition is thus either a discrete change of automaton state or a delay of one time unit.

A translation from high-level Petri nets (HLPN) conforming with the recent iso standard (thus produced by a variety of tools) is also available. HLPN are roughly to Place/Transition nets what parametric GAL are to GAL: they are not more expressive (if all data types are finite) but they are much more compact and readable. Interestingly, the instantiation of GAL parameters is often much less explosive than the translation from HLPN to P/T nets: synchronizations of independent behaviors (e.g. interaction between a server S and a client C) can be represented using a sequence of $call(\lambda)$ in GAL, where the P/T net must explicitly have a transition for each possible synchronization choice.

A high-level transition with n formal parameters with domain D produces $|D|^n$ basic transitions when instantiated. If the parameters are independent (e.g. put

x from place $p1$ to $p2$, and y from $p3$ to $p4$) the behavior can be expressed as a sequence of two calls to GAL labels, representing the $|D|$ choices of x followed by the $|D|$ choices of y. In favorable cases, GAL instantiation thus produces only $n * |D|$ transitions. This rewriting is fully automated, thanks to a fine grain analysis of statement semantics. It has allowed to analyze HLPN models that other tools relying on first instantiating the HLPN to a basic PN could not treat due to explosion of the PN size.

VI.2 Modeling Discrete Time

We now show in more detail how GAL can be used to model discrete time applications.

VI.2.1 Time Petri nets

We consider the formalism of Time Petri Nets (TPN) with discrete time semantics, and show how to map it to GAL. TPN are used to compactly model concurrent timed behaviors. Their semantics is usually expressed by Discrete Time Transition Systems (DTTS), which are transition systems equipped with two types of (atomic) transitions: action transitions as usual and an *elapse* transition, corresponding to a global elapsing of 1 time unit. Regarding the problem of marking reachability, it has been proved [Pop06; MLR08] that discrete time semantics capture all possible behaviors, even those obtained with dense time semantics. This makes it possible to compare experimentation results in both cases.

Concerning the syntax, we choose an extended definition of TPN because this leads to easier and more compact modeling abilities. There is a quite strong community of extended TPN users and our definition below captures a wide superset of what is understood as TPN in the literature: we consider an enabling predicate and a firing function as syntactic requirements, instead of defining various sorts of arcs. This homogeneously subsumes extensions such as reset arcs, read (or test) arcs, inhibitor arcs, and even non-deterministic extensions like hyper-arcs (because *fire* maps to $2^{\mathbb{N}^{Pl}}$), which are offered for instance by the Roméo tool [Gar+05]. The rich formalism demonstrates the flexibility of our tool, which supports arbitrary models with (finite) discrete time transition system semantics.

Definition A Time Petri Net is a tuple $\mathcal{N} = \langle Pl, Tr, A, enabled, fire, \ell, m_0, \alpha, \beta \rangle$ where:

- Pl is a finite set of *places*, Tr is a finite set of *transitions* (with $Pl \cap Tr = \emptyset$),
- A is a finite set (alphabet) of *action labels* which contains a distinguished *local* label $\top$,
- $enabled : \mathbb{N}^{Pl} \times Tr \mapsto \{true, \ false\}$ is an *enabling predicate*, $fire : \mathbb{N}^{Pl} \times Tr \mapsto 2^{\mathbb{N}^{Pl}}$ is a *transition firing function*, $\ell : Tr \mapsto A$ is a *labeling function*,

- $m_0 \in \mathbb{N}^{Pl}$ is the *initial marking* of the net,
- $\alpha : Tr \mapsto \mathbb{N}$ and $\beta : Tr \mapsto \mathbb{N} \cup \{\infty\}$ are functions satisfying $\forall t \in Tr, \alpha(t) \leq \beta(t)$ called respectively *earliest (*α*)* and *latest (*β*)* transition firing times.

Transitions t for which $\alpha(t) = \beta(t) = 0$ are called *urgent*. For instance, standard Place/Transition nets are usually defined using pre (noted **Pre**) and post (noted **Post**) functions : $Pl \times Tr \mapsto \mathbb{N}$. Then, for a marking $m \in \mathbb{N}^{Pl}$ and a transition $t \in Tr$, enabling is defined by $enabled(m,t)$ iff $\forall p \in Pl,\ m(p) \geq \mathbf{Pre}(p,t)$ and transition firing by $fire(m,t) = \{m'\}$ with $\forall p \in Pl,\ m'(p) = m(p) - \mathbf{Pre}(p,t) + \mathbf{Post}(p,t)$. Inhibitor arcs **Inh** and test arcs **Test** are defined similarly and add enabling conditions to a transition: $enabled(m,t)$ iff $\forall p \in Pl,\ m(p) < \mathbf{Inh}(p,t) \wedge m(p) \geq \mathbf{Test}(p,t)$. Note that the *enabling* predicate only considers markings while timing conditions are defined separately. In definition VI.2.1, transitions are equipped with labels for further composition of nets (see Section V.2.4). With slight notation abuse, we write $enabled(t)$ for the predicate $enabled(m,t)$ over markings.

VI.2.2 Encoding TPN into GAL

To encode a $\mathcal{N} = \langle Pl, Tr, A, enabled, fire, \ell, m_0, \alpha, \beta \rangle$, we define the corresponding GAL as follows:

- Assume that $Pl = \{p_1, \ldots, p_n\}$. Each place p in Pl gives rise to a variable p, with initial value the initial marking of p;
- Each transition t in Tr, such that $\alpha(t) \neq 0 \vee (\beta(t) \neq 0 \wedge \beta(t) \neq \infty)$ gives rise to a clock variable t_c with initial value 0. Transition clocks with the interval $[0,0]$ are taken into account in the semantics but not given a variable (since their value is trivially 0 in all states). Transition clocks with interval $[0,\infty[$ are entirely suppressed as they are not useful semantically.
- Each transition t in Tr produces a GAL transition $t = \langle \ell(t), g, \sigma \rangle$. The guard g is defined by $g = enabled(t) \wedge (\alpha(t) = 0 \vee t_c \geq \alpha(t))$. Similarly, σ is defined from the firing function *fire* in a straightforward manner by assignments to the variables that correspond to the places. If t is associated with a clock variable t_c, we add a statement $\langle t_c = 0 \rangle$ to the transition effects, to reset its clock. The last statement of every transition is a call to the label "reset" with $call(reset)$, to reset clocks of transitions newly disabled by the firing of t.
- The reset transition is $r = \langle reset, True, \sigma \rangle$, where σ is a sequence of $|Tr|$ statements $ite(enabled(t), nop, t_c = 0)$, for $t \in Tr$. Statements concerning clock variables that have been suppressed are simply *nop*.
- Additionally, the GAL model contains a “time elapse” transition $e_{\mathbb{1}} = \langle \mathbb{1}, True, \sigma \rangle$, where σ is a sequence of $|Tr|$ statements $\sigma_0; \ldots; \sigma_k$, where σ_i is the statement: $ite(enabled(t),\ ite(t_c < \beta(t), t_c = t_c + 1, abort), nop)$. This statement is simplified to $ite(enabled(t), abort, nop)$ for transitions with interval

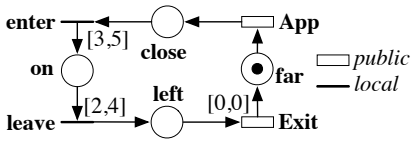


Figure VI.2: Behavior of a train in [BV03].

$[0,0]$. For transitions with $\alpha(t) \neq 0$ and with $\beta(t) = \infty$ as latest firing time, the statement is written $ite(enabled(t), ite(t_c < \alpha(t), t_c = t_c + 1, nop), nop)$ thus allowing the clock variable to progress up to $\alpha(t)$ for firing, but not keeping track of its value above $\alpha(t)$. The statement is simply *nop* for transitions with no time constraint $\alpha(t) = 0$ and $\beta(t) = \infty$.

VI.2.3 Examples

Example of TPN.

The example of Fig. VI.2, describing the behavior of a train in the train-crossing example [BV03], is used to illustrate the encoding of a TPN into GAL (see Fig. VI.3). Each place is encoded as a variable, as well as each transition clock. We note that the clock of **Exit** is not encoded because it is an urgent transition with a 0 delay. Then, each transition has its corresponding code in GAL, defining its firing condition. Finally, `elapse` lets time elapse by one unit when no enabled time transition has reached its upper bound, and `reset` sets the clocks of disabled transitions to 0.

Example of Composition.

As an example of composition, let us consider a gate modeled as a TPN with labels "Close" and "Open" that respectively opens and close a barrier. The model is not provided here (see [TM+11]), but for the purpose of composition its implementation is not important, only its interface or exported labels are. The barrier takes 1 to 2 time units to move from open to close position and vice-versa. The `traingate` composite (see Fig. VI.4) allows to close the gate when a train is approaching and open it when it leaves. However, when several trains share the same gate, the gate could be opened by a leaving train while another one is in place "close" or "on". Additionally, the composite bears an `elapse` synchronization that forces time to elapse at the same speed in enclosed components.

To solve this issue, we introduce a smarter composite having the same interface as a `gate`: `controlledgate` (see Fig. VI.4). It carries a counter that is used to trigger the gate only when the first (respectively last) train arrives (respectively leaves) the section protected by the gate. This example shows a composition of a TPN with a GAL as well as a sequence of synchronizations such as `leavelast`: this sequence can only be fired if, after decrementing `c`, its value is 0.

```
GAL train {
 int far=1, on=0, left=0, close=0;
 int leave.clock=0, enter.clock=0;
 transition App [far >= 1]
            label "App" {
  far = far - 1;
  on = on + 1;
  self.reset;}
 transition enter
     [close>=1 && enter.clock>=3] {
  close = close - 1;
  on = on + 1;
  enter.clock = 0;
  self.reset;}
 transition leave
     [on >= 1 && leave.clock >= 2] {
  on = on - 1;
  left = left + 1;
  leave.clock = 0;
  self.reset;}
 transition Exit [left >= 1]
            label "Exit" {
  left = left - 1;
  far = far + 1;
  self.reset;}
 transition elapse [True]
            label "elapse" {
  if (close >= 1) {
   if (enter.clock < 5) {
    enter.clock = enter.clock + 1;
   } else {abort;}
  }
  if (on >= 1) {
   if (leave.clock < 4) {
    leave.clock = leave.clock + 1;
   } else {abort;}
  }
  if (left >= 1) {abort;}
 }
 transition reset [True]
            label "reset" {
  if (! close >= 1) {enter.clock = 0;}
  if (! on >= 1) {leave.clock = 0;}
 }
}
```

Figure VI.3: The GAL system encoding of the TPN shown in Fig. VI.2

```
composite traingate {
 train t;
 gate g;
 synchronization arrive
  {t.App ; g.Close;}
 synchronization leave
  {t.Exit ; g.Open;}
 synchronization elapse label "elapse"
  {t.elapse; g.elapse; }
}

GAL counter {
 int cpt = 0;
 transition inc label "inc"
  {cpt = cpt + 1;}
 transition dec [cpt > 0] label "dec"
  {cpt = cpt - 1;}
 transition z [cpt==0] label "iszero"
  {}
 transition nz [cpt!=0] label "notzero"
  {}
}
composite controlledgate {
 counter c ;
 gate g;
 synchronization enterfirst label "Close"
  {c.iszero; c.inc; g.close;}
 synchronization enterother label "Close"
  {c.notzero; c.inc;}
 synchronization leavelast label "Open"
  {c.dec; c.iszero; g.open;}
 synchronization leaveother label "Open"
  {c.dec; c.notzero;}
 synchronization elapse label "elapse"
  {g.elapse;}
}
```

Figure VI.4: Composite description of variants on a gate and its controller.

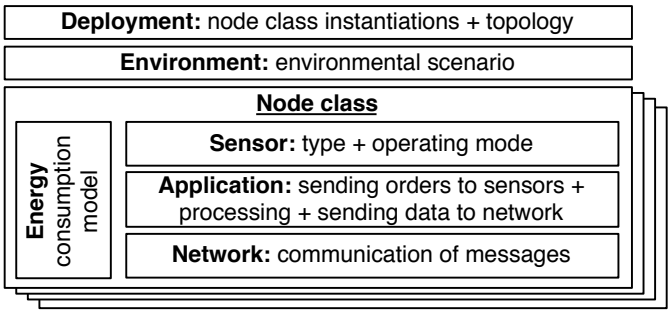


Figure VI.5: Structure of a Verisensor specification.

VI.3 Case studies

We present here some significant applications of the ITS-tools that we are aware of.

The Behavioral Consistency Checker (BCC) developed in the ModelPlex IP project [TH08] transformed UML activity diagrams to an early version of ITS to verify a set of predefined consistency rules (absence of deadlocks, all activities can eventually terminate...).

In [BM+13] ITS-tools were used to analyze compositions of time Petri nets produced from a DSL VeriSensor dedicated to wireless sensor network modeling. Verisensor separates specification of different aspects of a WSN through a notion of *views*, represented in figure VI.5.

These views are combined to form the complete model, with heavy use of ITS and label synchronization (figure VI.6). Depending on the target property, simplified variants can be used for parts of the system, giving some property specific abstraction. The specifications analyzed contained more than 50 clocks, many of which are concurrently enabled, preventing analysis by explicit tools such as Tina.

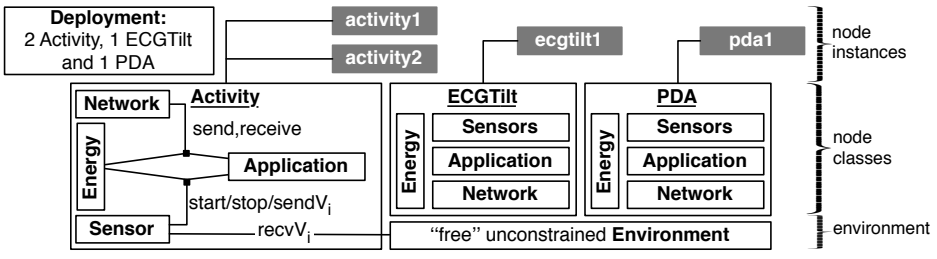


Figure VI.6: Translation to ITS of Verisensor uses label synchronizations (arcs) and instantiation (each box is an instance). This figure depicts a Body Area Network.

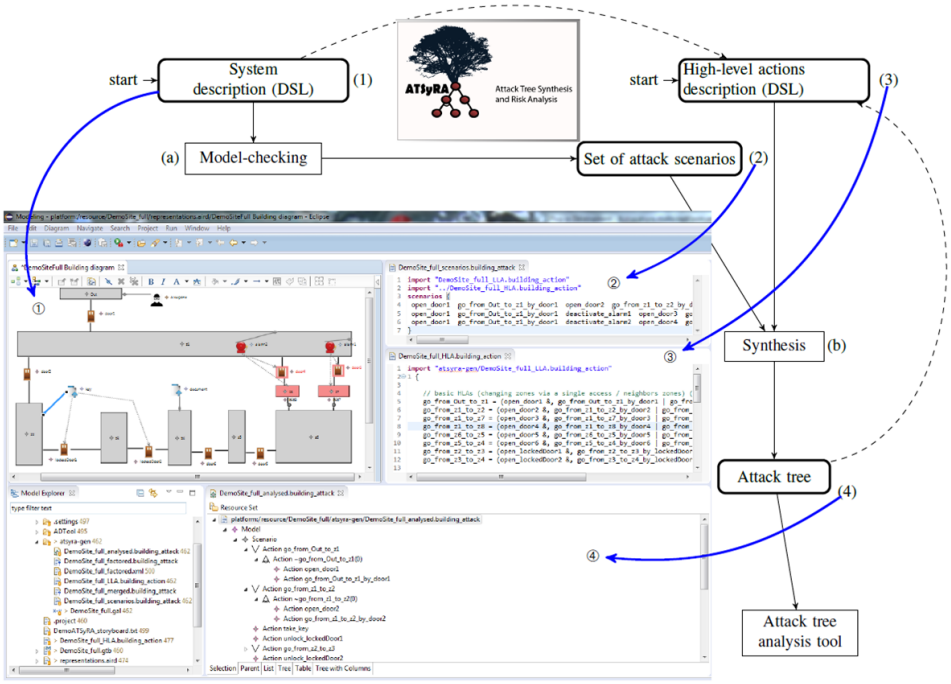


Figure VI.7: Architecture AtSyRa.

With "its-reach" functional properties could be checked as well as quantitative measures such as worst-case lifetime analysis.

In the Neoppod project [Cho+10a] the CTL component was used to verify response and consistency properties of a protocol for a distributed database.

Inria's Atsyra project [PAV14] computes attack defense trees from a DSL using a model-to-model transformation to GAL. Their expertise with Kermeta and metamodeling made it easy for them to use the EMF compliant GAL metamodel and connect it to their own DSL. Figure VI.7 shows the architecture of the tool.

In terms of raw benchmark power, ITS-tools participated in several editions of the model-checking contest at Petri nets conference, ranking first place in several categories. It is compared favorably to LTSmin and to SAT solver Superprove on the benchmark BEEM[Col+13]. It outperformed the symbolic tool Smart using its own benchmark models in [TM+09]. On timed models, comparisons to Uppaal show that we tend to scale better in number of clocks, but are more sensitive to large bounds on clocks, something that was reported in previous similar experiments [BLN03].

VI.4 Evaluation

The ITS-tools is composed of 148*kloc* of Java for the front-end and 123*kloc* of C++ for the back-end (a *kloc* is one thousand lines of code). A lot of configuration and

build files bring this total to about 300*kloc* of manually built code. We started from a fork of *libddd* by Jean-Michel Couvreur and Denis Poitrenaud written in 2001, for which we don't have version history, but we do have data since 2004. ITS-tools includes contributions by 36 committers, amounting to 6600 commit operations since 2004. This includes quite a few internships of Master students that I supervised and that contributed modules for language support. I am the author 5149 of these commits, amounting to 78% of the activity. My former PhD students Maximilien Colange (384 commit) and Alexandre Hamez (144 commit) are the next authors in terms of both volume and importance of the contributions (commits in the kernel).

Building practical tools in academia is a challenge. From experience, we try to always make consensual technology choices, supported by industry and a strong user base. It is essential to reuse existing components and rely on frameworks to leverage the thousands of man years of development effort that are made freely available in projects such as Eclipse and EMF. Maintenance and upgrades to match evolution of the framework are thus necessary, so continuity in the development team is desirable. Student projects can help bootstrap an idea or test a framework, but integration in a running code base requires a lot of testing, debugging and manual inspection. So having some key developers with a permanent position seems necessary to build and maintain solid tools in academia.

VI.5 Conclusion

All the contributions presented in this manuscript have been integrated in the freely available at `http://ddd.lip6.fr/` open-source symbolic model-checking suite ITS-tools. It uses Java and leverages EMF to provide a user friendly front-end both for direct modeling and definition of model transformation from a DSL. The back-end uses optimized C++ for better time and memory efficiency, and leverages Spot for LTL model-checking[DL14].

Part D

General Conclusion

Chapter VII

Conclusion and Perspectives

VII.1Conclusion

This manuscript presented my main research axis over the last decade : developing the theory and practice of symbolic model-checking. We identified key contributions to definition of a symbolic kernel in part A, then described some symbolic model-checking algorithms that use the kernel in part B. Part C presented in the ITS framework for modeling hierarchy, and its complement GAL as a semantic assembly language to describe concurrent semantics. We provide a free open-source implementation of all the algorithms presented in this memoir, within ITS-tools, one of the most efficient and general symbolic model-checkers available.

I'm a co-author of 26 conference papers including some of the best conferences of the domain (3 TACAS, 1 CAV) and 9 papers in journals. I have also participated in writing book chapters on formal verification and a course book on UML [COTM10] for graduate students (3 editions, so relatively successful).

I have co-supervised 4 PhD students :

- Alexandre Hamez [Ham09] (with F. Kordon) on parallel and distributed explicit model-checking [Ham+07] and the definition of rewriting rules for homomorphisms [HTMK08] which were presented in II.2.

- Yan Zhang [Zha13] (with B. Berard) on control theory [Zha+10], and its integration in a software development process [Zha+14]. To this end we developed a DSL VeriJ that builds on a fragment of Java and a transformation chain to support its analysis. We built an explicit solution engine, since some concepts (stack, heap) could not be easily mapped to the hypothesis of GAL. We were able to design a solution that is competitive with other Java model-checkers such as JPF [HP00].

- Yann Ben Maissa [BM13] (with F. Kordon and S. Mouline) on modeling and analysis of wireless sensor networks (WSN). We defined a DSL VeriSensor [BM+13] adapted to WSN, then analyzed properties such as worst case

time to failure using a translation to ITS tools. These models involved both real time constraints and resource constraints such as energy. The generated models used the ITS features of instantiation and label synchronization heavily, and were large quite large real time models as a result (dozens of clocks).

- Maximilien Colange [Col13] (with F.Kordon and S. Baarir) on the symbolic symbolic approach initially (see [Col+12] and IV), then on expression evaluation in a symbolic setting (see [Col+13] and II.3).

Since we focused in this memoir on symbolic methods, the work with Yann Ben Maissa and Yan Zhang was not fully presented. In both of these thesis, we explored the applicative side, by defining high-level DSL designed for a specific domain, then analyzing their behavior using model transformations. Thus they were positioned in the role of a client of formal verification, and helped to design and stress test the translation approach and verification tools presented in this manuscript.

VII.2Perspectives

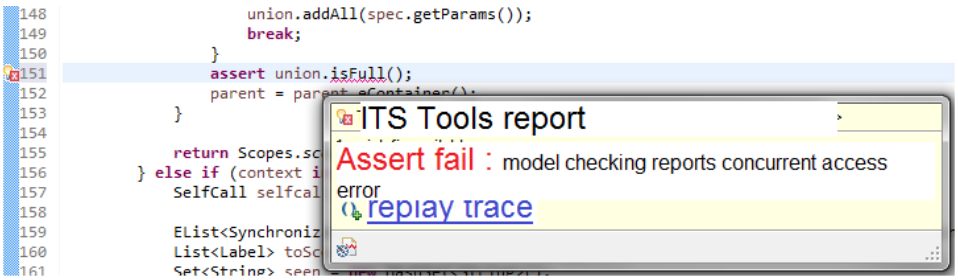


Figure VII.1: ITS-tools 3.0, circa 2025.

As a long-term goal, we would like to handle verification of concurrent code, using a model extraction approach that embeds automated abstractions. Figure VII.1 is an idealized view of ITS-tools in a few years : directly embedded in the IDE, running the behavioral analysis on rich DSL for real-time embedded systems that feature code fragments, and reporting diagnosis and traces to the user transparently with respect to underlying solver technologies.

Several challenges still need to be solved however. Each perspective presented here could be developed during a PhD thesis.

Multi-solver Integration

While ITS-tools is an efficient model-checker, the intermediate language GAL and the translation approach we have developed supports introduction of other solution engines.

Connection to explicit multi-core solutions such as LTSmin seems particularly desirable to complement our symbolic engine. We think a hybrid between GAL and the PINS input of LTSmin could benefit both : extending PINS with ITS compositions and sums of behaviors, and extending GAL with opaque functions embedding code.

Because GAL currently expresses the whole system semantics, it is also possible to export the transition relation to SAT or SMT solvers such as Z3 to perform bounded model-checking or other verifications.

A concurrent portfolio featuring multiple solvers would benefit from multi-core evolutions of modern hardware.

Stack and Heap

While GAL can model systems with a fixed set of integer variables and arrays, there is currently no support for modeling a procedural call stack or lists with a variable number of elements such as a heap.

We currently use bounded inlining for procedures or a bound on maximal size of a dynamic arrays to translate these concepts, but their native support would be both more flexible and powerful. To treat such cases, we need to extend GAL with support for dynamic lists, but also relax the assumptions of the solver(s).

For the specific case of a list representing a heap of objects allocated concurrently, we need to exploit symmetries between logically equivalent configurations to limit the state explosion.

Abstractions and Rewriting

Abstraction is a fundamental part of modeling, but is also necessary for verification. Experiments with DSL such as VeriJ [Zha13] for controller design or VeriSensor [BM13] to study wireless sensor networks have shown that exploiting the domain knowledge allows to perform many automatic abstractions during the translation of a formal model, while ensuring preservation of the property of interest. We have also experimented some abstractions for HLPN in the context of the model-checking contest with promising results.

The challenge is to express such abstractions in a general way, as rewriting rules from GAL to GAL with some property specific preconditions governing when they can be applied, thus profiting all input formalisms.

Bibliography

[AHI98] K. Ajami, S. Haddad, and J.-M. Ilié. "Exploiting Symmetry in Linear Time Temporal Logic Model Checking: One Step Beyond". In: *First International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*. Vol. 1384. LNCS. Springer-Verlag, 1998, pp. 52–67 (cit. on p. 49).

[AMS03] F. Aloul, I. Markov, and K. Sakallah. "FORCE: a fast and easy-to-implement variable-ordering heuristic". In: *13th ACM Great Lakes symposium on VLSI*. ACM. 2003, pp. 116–119 (cit. on p. 76).

[Arn02] A. Arnold. "Nivat's processes and their synchronization". In: *Theor. Comp. Sci.* 281.1-2 (2002), pp. 31–36. ISSN: 0304-3975 (cit. on pp. 60, 62).

[Bah+93] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. "Algebraic Decision Diagrams and Their Applications". In: *Proceedings of the 1993 IEEE/ACM International Conference on Computer-aided Design*. ICCAD '93. Santa Clara, California, USA: IEEE Computer Society Press, 1993, pp. 188–191 (cit. on p. 8).

[Bar+10] J. Barnat, L. Brim, M. Češka, and P. Ročkai. "DiVinE: Parallel Distributed Model Checker (Tool paper)". In: *Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC)*. IEEE, 2010, pp. 4–7 (cit. on pp. 28, 59).

[Bar+13] J. Barnat, L. Brim, V. Havel, J. Havlíček, J. Kriho, M. Lenčo, P. Ročkai, V. Štill, and J. Weiser. "DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs". In: *Computer Aided Verification (CAV)*. LNCS 8044. Springer, 2013, pp. 863–868 (cit. on p. 77).

[BCN11] Y. Boichut, J.-M. Couvreur, and D.-T. Nguyen. "Functional Term Rewriting Systems Towards Symbolic Model-Checking". In: *Int. J. Crit. Comput.-Based Syst.* 2.3/4 (Sept. 2011), pp. 378–408. ISSN: 1757-8779 (cit. on p. 14).

[BCS09] G. Blair, T. Coupaye, and J.-B. Stefani. "Component-based architecture: the Fractal initiative". In: *annals of telecommunications - annales des télécommunications* 64.1 (2009), pp. 1–4 (cit. on p. 61).

[BCZ99] A. Biere, E. M. Clarke, and Y. Zhu. "Multiple State and Single State Tableaux for Combining Local and Global Model Checking". In: *Correct System Design*. Vol. 1710. LNCS. Springer-Verlag, 1999, pp. 163–179 (cit. on pp. 38, 44).

[BDH02] D. Bošnački, D. Dams, and L. Holenderski. "Symmetric Spin". In: *International Journal on Software Tools for Technology Transfer* 4.1 (2002), pp. 92–106 (cit. on p. 50).

[Beh+01] G. Behrmann, K. G. Larsen, O. Moller, A. David, P. Pettersson, and W. Yi. "Uppaal-present and future". In: *Decision and Control, 2001. Proceedings of the 40th IEEE Conference on*. Vol. 3. IEEE. 2001, pp. 2881–2886 (cit. on pp. 59, 77).

[Beh+99] P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier. "Météor: A Successful Application of B in a Large Project". In: *FM'99 — Formal Methods: World Congress on Formal Methods in the Development of Computing Systems, Proceedings, Volume I*. Springer Berlin Heidelberg, 1999, pp. 369–387 (cit. on p. 4).

[Ber+09] B. Berthomieu, J. Bodeveix, C. Chaudet, S. Dal-Zilio, M. Filali, and F. Vernadat. "Formal Verification of AADL Specifications in the Topcased Environment". In: *Ada-Europe*. Vol. 5570. LNCS. Springer, 2009, pp. 207–221 (cit. on p. 58).

[BET10] V. Beaudenon, E. Encrenaz, and S. Taktak. "Data decision diagrams for Promela systems analysis". In: *STTT* 12.5 (2010), pp. 337–352 (cit. on pp. 14, 15, 35, 67).

[Bey01] D. Beyer. "Improvements in BDD-Based Reachability Analysis of Timed Automata". In: *Formal Methods Europe (FME)*. LNCS 2021. Springer-Verlag, 2001, pp. 318–343 (cit. on p. 77).

[Bey15] D. Beyer. "Software Verification and Verifiable Witnesses (Report on SV-COMP 2015)". In: *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and of Analysis Systems (TACAS 2015, London, UK, April 13-17)*. LNCS 9035. Springer-Verlag, Heidelberg, 2015, pp. 401–416. ISBN: 978-3-662-46680-3 (cit. on p. 59).

[BHI04] S. Baarir, S. Haddad, and J.-m. Ilié. "Exploiting partial symmetries in well-formed nets for the reachability and the linear time model checking problems". In: *in Proc. of the 7th Workshop on Discrete Event Systems (WODES'04*. Citeseer. 2004 (cit. on p. 42).

[BLN03] D. Beyer, C. Lewerentz, and A. Noack. "Rabbit: A Tool for BDD-Based Verification of Real-Time Systems". In: *Computer Aided Verification (CAV)*. LNCS 2725. Springer-Verlag, 2003, pp. 122–125 (cit. on p. 83).

[BM+13] Y. Ben Maïssa, F. Kordon, S. Mouline, and Y. Thierry-Mieg. "Modeling and Analyzing Wireless Sensor Networks with VeriSensor: an Integrated Workflow". In: *Transactions on Petri Nets and Other Models of Concurrency (ToPNoC)* VIII (2013), pp. 24–47 (cit. on pp. 82, 86).

[BM10] R. K. Brayton and A. Mishchenko. "ABC: An Academic Industrial-Strength Verification Tool". In: *CAV*. Vol. 6174. LNCS. Springer, 2010, pp. 24–40 (cit. on p. 33).

[BM13] Y. Ben Maissa. "Contribution to the modeling and verification of wireless sensor networks". PhD thesis. Paris, France and Rabat, Morocco: Universite Pierre et Marie Curie and Universite Mohammed V Agdal, 2013 (cit. on pp. 86, 88).

[BP08] S. Blom and J. van de Pol. "Symbolic Reachability for Process Algebras with Recursive Data Types". In: *ICTAC*. Vol. 5160. LNCS. Springer, 2008, pp. 81–95 (cit. on p. 8).

[BPW10] S. Blom, J. van de Pol, and M. Weber. "LTSmin: Distributed and symbolic reachability". In: *Computer Aided Verification (CAV)*. Springer. 2010, pp. 354–359 (cit. on pp. 28, 33, 35, 72, 75).

[Bra+96] R. K. Brayton, G. D. Hachtel, A. L. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. A. Edwards, S. P. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. "VIS: A System for Verification and Synthesis". In: *Computer Aided Verification (CAV)*. Vol. 1102. Lecture Notes in Computer Science. Springer, 1996, pp. 428–432 (cit. on pp. 58, 76).

[Bry86] R. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Transactions on Computers* 35.8 (1986), pp. 677–691 (cit. on pp. 8, 9).

[BS+14] A.-E. Ben Salem, A. Duret-Lutz, F. Kordon, and Y. Thierry-Mieg. "Symbolic Model Checking of stutter invariant properties Using Generalized Testing Automata". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Vol. 8413. LNCS. Springer, 2014, pp. 440–454 (cit. on pp. 46, 76).

[Buc+10] D. Buchs, S. Hostettler, A. Marechal, and M. Risoldi. "AlPiNA: An Algebraic Petri Net Analyzer". In: *Tools and Algorithms for the Construction and Analysis of Systems: 16th International Conference, TACAS 2010*. Springer, 2010, pp. 349–352 (cit. on pp. 14, 15, 35).

[Bur+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. Hwang. "Symbolic model checking: 10^{20} States and beyond". In: *Information and computation* 98.2 (1992), pp. 142–170 (cit. on pp. 8, 19, 23).

[BV03] B. Berthomieu and F. Vernadat. "State Class Constructions for Branching Analysis of Time Petri Nets". In: *Tools and Algorithms for the Construction and Analysis of Systems – TACAS*. Vol. 2619. LNCS. Springer, 2003, pp. 442–457 (cit. on p. 80).

[BV06] B. Berthomieu and F. Vernadat. "Time petri nets analysis with tina". In: *Quantitative Evaluation of Systems (QEST)*. IEEE. 2006, pp. 123–124 (cit. on p. 58).

[Bér+08] B. Bérard, S. Haddad, L. M. Hillah, F. Kordon, and Y. Thierry-Mieg. "Collision Avoidance in Intelligent Transport Systems: towards an Application of Control Theory". In: *9th International Workshop on Discrete Event Systems (WODES'08)*. Goteborg, Sweden: IEEE Computer Society, May 2008, pp. 346–351 (cit. on p. 67).

[Cav+14] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. "The nuXmv Symbolic Model Checker". In: *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. New York, NY, USA: Springer-Verlag New York, Inc., 2014, pp. 334–342. ISBN: 978-3-319-08866-2 (cit. on p. 58).

[CC79] P. Cousot and R. Cousot. "Systematic design of program analysis frameworks". In: *In 6th POPL*. ACM Press, 1979, pp. 269–282 (cit. on p. 5).

[CDLP05] J.-M. Couvreur, A. Duret-Lutz, and D. Poitrenaud. "On-the-Fly Emptiness Checks for Generalized Büchi Automata". In: *Proceedings of the 12th International SPIN Workshop on Model Checking of Software (SPIN'05)*. Vol. 3639. LNCS. Springer, Aug. 2005, pp. 143–158 (cit. on p. 40).

[CGP99] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge MA, USA, 1999 (cit. on pp. 8, 37).

[Chi+90] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. "On Well-Formed Coloured Nets and their Symbolic Reachability Graph". In: *Proc. 11^{th} International Conference on Application and Theory of Petri Nets*. Paris, France, 1990 (cit. on pp. 50, 65, 66).

[Chi+93] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. "Stochastic well-formed colored nets and symmetric modeling applications". In: *Computers, IEEE Transactions on* 42.11 (1993), pp. 1343–1360 (cit. on pp. 47, 50).

[CHM16] L. Cabac, M. Haustermann, and D. Mosteller. "Renew 2.5 – Towards a Comprehensive Integrated Development Environment for Petri Net-Based Applications". In: *Application and Theory of Petri Nets and Concurrency: 37th International Conference, PETRI NETS 2016, Toruń, Poland, June 19-24, 2016. Proceedings*. Cham: Springer International Publishing, 2016, pp. 101–112 (cit. on p. 58).

[Cho+10a] C. Choppy, A. Dedova, S. Evangelista, S. Hong, K. Klai, and L. Petrucci. "The NEO Protocol for Large-Scale Distributed Database Systems: Modelling and Initial Verification". In: *Application and Theory of Petri Nets (ICATPN)*. LNCS 6128. Springer, 2010, pp. 145–164 (cit. on p. 83).

[Cho+10b] C. Choppy, A. Dedova, S. Evangelista, S. Hong, K. Klai, and L. Petrucci. "The NEO Protocol for Large-Scale Distributed Database Systems: Modelling and Initial Verification". In: *Petri Nets*. Vol. 6128. LNCS. Springer, 2010, pp. 145–164 (cit. on p. 14).

[Cim+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. "NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking". In: *Proc. International Conference on Computer-Aided Verification (CAV 2002)*. Vol. 2404. LNCS. Copenhagen, Denmark: Springer, 2002 (cit. on pp. 58, 69).

[Cla+03] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. "Counterexample-guided Abstraction Refinement for Symbolic Model Checking". In: *J. ACM* 50.5 (Sept. 2003), pp. 752–794. ISSN: 0004-5411 (cit. on p. 5).

[Cla+96] E. Clarke, R. Enders, T. Filkorn, and S. Jha. "Exploiting symmetry in temporal logic model checking". In: *Formal Methods in System Design* 9.1 (1996), pp. 77–104 (cit. on pp. 50, 51, 53, 56).

[Cla+98] E. Clarke, E. Emerson, S. Jha, and A. Sistla. "Symmetry reductions in model checking". In: *Computer Aided Verification*. Springer. 1998, pp. 147–158 (cit. on pp. 47, 49).

[CMS03] G. Ciardo, R. Marmorstein, and R. Siminiceanu. "Saturation unbound". In: *Tools and algorithms for the construction and analysis of systems*. Springer Varlag, LNCS 2619, 2003, pp. 379–393 (cit. on pp. 23, 25, 26, 43).

[Col+11] M. Colange, S. Baarir, F. Kordon, and Y. Thierry-Mieg. "Crocodile: a Symbolic/Symbolic tool for the analysis of Symmetric Nets with Bag". In: *32nd International Conference on Petri Nets and Other Models of Concurrency (ICATPN 2011)*. Vol. 6709. Lecture Notes in Computer Science. Springer, June 2011, pp. 338–347 (cit. on pp. 14, 15, 51).

[Col+12] M. Colange, F. Kordon, Y. Thierry-Mieg, and S. Baarir. "State Space Analysis using Symmetries on Decision Diagrams". In: *Application of Concurrency to System Design (ACSD)*. IEEE Computer Society, 2012, pp. 164–172 (cit. on pp. 29, 51, 55, 67, 76, 87).

[Col+13] M. Colange, S. Baarir, F. Kordon, and Y. Thierry-Mieg. "Towards Distributed Software Model-Checking using Decision Diagrams". In: *Computer Aided Verification (CAV)*. LNCS 8044. Springer Verlag, 2013, pp. 830–845 (cit. on pp. 32, 33, 72, 73, 77, 83, 87).

[Col13] M. Colange. "Symmetry Reduction and Symbolic Data Structures for Model Checking of Distributed Systems". PhD thesis. Paris, France: Universite Pierre et Marie Curie, 2013 (cit. on pp. 29, 35, 55, 76, 87).

[COTM10] B. Charroux, A. Osmani, and Y. Thierry-Mieg. *UML2 : Pratique de la Modelisation*. 3eme. Pearson Education, 2010, p. 288. ISBN: 2744074667 (cit. on p. 86).

[Cou+02] J.-M. Couvreur, E. Encrenaz, E. Paviot-Adet, D. Poitrenaud, and P.-A. Wacrenier. "Data decision diagrams for Petri net analysis". In: *Application and Theory of Petri Nets (ICATPN)* (2002), pp. 129–158 (cit. on pp. 8–10, 20, 21, 35, 67).

[Cou+91] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. "Memory-Efficient Algorithm for the Verification of Temporal Properties". In: *Proceedings of the 2nd international workshop on Computer Aided Verification (CAV'90)*. Vol. 531. LNCS. Springer-Verlag, 1991, pp. 233–242 (cit. on p. 37).

[CTM05] J.-M. Couvreur and Y. Thierry-Mieg. "Hierarchical decision diagrams to exploit model structure". In: *Formal Techniques for Networked and Distributed Systems (FORTE)* (2005), pp. 443–457 (cit. on pp. 11, 13).

[Dal+12] A. E. Dalsgaard, A. Laarman, K. G. Larsen, M. C. Olesen, and J. van de Pol. "Multi-core Reachability for Timed Automata". In: *Formal Modeling and Analysis of Timed Systems: 10th International Conference, FORMATS 2012, London, UK, September 18-20, 2012. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 91–106 (cit. on p. 72).

[DL+11] A. Duret-Lutz, K. Klai, D. Poitrenaud, and Y. Thierry-Mieg. "Self-loop aggregation product a new hybrid approach to on-the-fly LTL model checking". In: *Automated Technology for Verification and Analysis (ATVA)*. Springer, 2011, pp. 336–350 (cit. on pp. 43, 46, 76).

[DL+16] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. "Spot 2.0 — a framework for LTL and ω-automata manipulation". In: *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*. Vol. ???? LNCS. To appear. Springer, Oct. 2016, ??–?? (Cit. on pp. 43, 76).

[DL14] A. Duret-Lutz. "LTL Translation Improvements in Spot 1.0". In: *International Journal on Critical Computer-Based Systems* 5.1/2 (2014), pp. 31–54 (cit. on pp. 43, 76, 84).

[DLP04] A. Duret-Lutz and D. Poitrenaud. "SPOT: an Extensible Model Checking Library using Transition-based Generalized Büchi Automata". In: *Proceedings of the 12th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'04)*. Volendam, The Netherlands: IEEE Computer Society Press, Oct. 2004, pp. 76–83 (cit. on pp. 40, 43).

[EB10] M. Eysholdt and H. Behrens. "Xtext: Implement Your Language Faster Than the Quick and Dirty Way". In: *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. OOPSLA '10. Reno/Tahoe, Nevada, USA: ACM, 2010, pp. 307–309 (cit. on p. 77).

[EL87] E. A. Emerson and C.-L. Lei. "Modalities for Model Checking: Branching Time Logic Strikes Back". In: *Science of Computer Programming* 8.3 (June 1987), pp. 275–306 (cit. on p. 76).

[Ete99] K. Etessami. "Stutter-Invariant Languages, ω-Automata, and Temporal Logic". In: *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*. Vol. 1633. LNCS. Springer-Verlag, 1999, pp. 236–248 (cit. on p. 37).

[Fis+01] K. Fisler, R. Fraer, G. Kamhi, M. Y. Vardi, and Z. Yang. "Is There a Best Symbolic Cycle-Detection Algorithm?" In: *Proc. of TACAS'01 (TACAS'01)*. Vol. 2031. LNCS. Springer, 2001, pp. 420–434 (cit. on pp. 38, 43).

[Gam+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2 (cit. on p. 61).

[Gar+05] G. Gardey, D. Lime, M. Magnin, and O. H. Roux. "Roméo: A Tool for Analyzing time Petri nets". In: *17th International Conference on Computer Aided Verification*. Vol. 3576. LNCS. http://romeo.rts-software.org/. Springer, July 2005 (cit. on p. 78).

[Gil+04] F. Gilliers, F. Bréant, D. Poitrenaud, and F. Kordon. "Model Checking of Highlevelobject Oriented Specifications : The LfP Experience". In: *3rd Workshop on Modelling of Objects, Components, and Agents (MOCA '04)*. INT LIP6 MoVe. Aarhus, Denmark, 2004, pp. 149–168 (cit. on p. 67).

[Ham+07] A. Hamez, F. Kordon, Y. Thierry-Mieg, and F. Legond-Aubry. "dmcG: a Distributed Symbolic Model Checker Based on GreatSPN". In: *28th International Conference on Petri Nets and Other Models of Concurrency (ICATPN 2007)*. Lecture Notes in Computer Science (LNCS). INT LIP6 MoVe. Siedlce, Poland: Springer-Verlag, June 2007, pp. 495–504 (cit. on p. 86).

[Ham09] A. Hamez. "Generation efficace de grands espaces d'etats". PhD thesis. Paris, France: Universite Pierre et Marie Curie, 2009 (cit. on pp. 24, 32, 35, 86).

[Hen+04] M. Hendriks, G. Behrmann, K. Larsen, P. Niebert, and F. Vaandrager. "Adding Symmetry Reduction to Uppaal". In: *Formal Modeling and Analysis of Timed Systems: First International Workshop, FORMATS 2003*. Springer Berlin Heidelberg, 2004, pp. 46–59 (cit. on pp. 50, 65).

[Her+03] H. Hermanns, M. Kwiatkowska, G. Norman, D. Parker, and M. Siegle. "On the use of MTBDDs for performability analysis and verification of stochastic systems". In: *The Journal of Logic and Algebraic Programming* 56.1 (2003), pp. 23 –67. ISSN: 1567-8326 (cit. on p. 8).

[HIK04] S. Haddad, J.-M. Ilié, and K. Klai. "Design and Evaluation of a Symbolic and Abstraction-based Model Checker". In: *Proceedings of the 2nd International Symposium on Automated Technology for Verification and Analysis (ATVA'04)*. Vol. 3299. LNCS. National Taiwan University, Taiwan: Springer, Oct. 2004, pp. 198–210 (cit. on pp. 38, 42, 44).

[HMP92] T. A. Henzinger, Z. Manna, and A. Pnueli. "What good are digital clocks?" In: *Automata, Languages and Programming*. Springer, 1992, pp. 545–558 (cit. on p. 77).

[Hol97] G. J. Holzmann. "The model checker SPIN". In: *IEEE Transactions on Software Engineering* 23 (1997), pp. 279–295 (cit. on pp. 28, 59, 77).

[Hon+12] S. Hong, F. Kordon, E. Paviot-Adet, and S. Evangelista. "Computing a Hierarchical Static Order for Decision Diagram-Based Representation from P/T Nets". In: *Trans. Petri Nets and Other Models of Concurrency* 5 (2012), pp. 121–140 (cit. on p. 15).

[Hos11] S. Hostettler. "High-level Petri net model checking : the symbolic way". PhD thesis. Genève, Suisse: Universite de Genève, 2011 (cit. on p. 14).

[HP00] K. Havelund and T. Pressburger. "Model Checking JAVA Programs using JAVA PathFinder". In: *STTT* 2.4 (2000), pp. 366–381 (cit. on p. 86).

[HPV02] H. Hansen, W. Penczek, and A. Valmari. "Stuttering-Insensitive Automata for On-the-fly Detection of Livelock Properties". In: *Proceedings of the 7th International ERCIM Workshop in Formal Methods for Industrial Critical Systems (FMICS'02)*. Vol. 66(2). ENTCS. Málaga, Spain: Elsevier, July 2002 (cit. on p. 37).

[HST09] M. Heiner, M. Schwarick, and A. Tovchigrechko. "DSSZ-MC – A Tool for Symbolic Analysis of Extended Petri Nets". In: *Applications and Theory of Petri Nets: 30th International Conference, PETRI NETS 2009, Paris, France, June 22-26, 2009. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 323–332 (cit. on pp. 14, 58).

[HTMK08] A. Hamez, Y. Thierry-Mieg, and F. Kordon. "Hierarchical Set Decision Diagrams and Automatic Saturation". In: *Applications and Theory of Petri Nets (ICATPN)*. LNCS 5062. 2008 (cit. on pp. 24, 67, 72, 86).

[HTMK09] A. Hamez, Y. Thierry-Mieg, and F. Kordon. "Building Efficient Model Checkers using Hierarchical Set Decision Diagrams and Automatic Saturation". In: *Fundamenta Informaticae* 94.3-4 (2009), pp. 413–437 (cit. on pp. 24, 61, 67).

[INH96] H. Iwashita, T. Nakata, and F. Hirose. "CTL model checking based on forward state traversal". In: *Computer-Aided Design (ICCAD)*. IEEE/ACM. 1996, pp. 82–87 (cit. on p. 76).

[JK07] T. Junttila and P. Kaski. "Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs". In: *Proceedings of the Meeting on Algorithm Engineering & Expermiments*. Society for Industrial and Applied Mathematics, 2007, pp. 135–149 (cit. on p. 54).

[Jun03] T. Junttila. "On the Symmetry Reduction Method for Petri Nets and similar formalisms". PhD thesis. Espoo, Finland: Helsinki University of Technology, 2003 (cit. on pp. 47–50).

[KP08] K. Klai and D. Poitrenaud. “MC-SOG: An LTL Model Checker Based on Symbolic Observation Graphs”. In: *Application and Theory of Petri Nets (ICATPN)*. LNCS. Springer-Verlag, 2008, pp. 288–306 (cit. on pp. 38, 42, 76).

[KPR98] Y. Kesten, A. Pnueli, and L. on Raviv. “Algorithmic Verification of Linear Temporal Logic Specifications”. In: *Proceedins of the 5th International Colloquium on Automata, Languages, and Programming (ICALP’98)*. Vol. 1443. LNCS. Springer-Verlag, 1998, pp. 1–16 (cit. on p. 38).

[KV97] R. Kaivola and A. Valmari. “The Weakest Compositional Semantic Equivalence Preserving Nexttime-less Linear temporal Logic”. In: *Proc. of CONCUR’92*. Vol. 630. LNCS. Springer, 1997, pp. 207–221 (cit. on p. 37).

[LA04] C. Lattner and V. Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04)*. Palo Alto, California, 2004 (cit. on p. 68).

[LB15] E. Lopez Bobeda. “Symbolic Model-checking with Set Rewriting”. PhD thesis. Genève, Suisse: Universite de Genève, 2015 (cit. on pp. 14, 33, 35).

[LBCB14] E. López Bóbeda, M. Colange, and D. Buchs. “StrataGEM: A Generic Petri Net Verification Framework”. In: *Petri Nets*. Vol. 8489. LNCS. Springer, 2014, pp. 364–373 (cit. on pp. 14, 33).

[Ler09] X. Leroy. “Formal verification of a realistic compiler”. In: *Communications of the ACM* 52.7 (2009), pp. 107–115 (cit. on p. 4).

[Lin09] A. Linard. “Sémantique paramétrable des Diagrammes de Décision : une démarche vers l’unification”. PhD thesis. Université Pierre & Marie Curie, EDITE, 2009 (cit. on p. 8).

[LL95] F. Laroussinie and K. G. Larsen. “Compositional Model Checking of Real Time Systems”. In: *CONCUR*. Vol. 962. LNCS. Springer, 1995, pp. 27–41 (cit. on pp. 60, 64).

[MC99] A. S. Miner and G. Ciardo. “Efficient Reachability Set Generation and Storage Using Decision Diagrams”. In: *ICATPN*. Vol. 1639. LNCS. Springer, 1999, pp. 6–25 (cit. on p. 8).

[Min06] A. Miné. “The Octagon Abstract Domain”. In: *Higher Order Symbol. Comput.* 19.1 (Mar. 2006), pp. 31–100. ISSN: 1388-3690 (cit. on p. 5).

[MLR08] M. Magnin, D. Lime, and O. H. Roux. “Symbolic state space of Stopwatch Petri nets with discrete-time semantics”. In: *ICATPN*. Vol. 5062. LNCS. 2008, pp. 307–326 (cit. on p. 78).

[MT00] N. Medvidovic and R. N. Taylor. "A Classification and Comparison Framework for Software Architecture Description Languages". In: *IEEE Trans. Software Eng.* 26.1 (2000), pp. 70–93 (cit. on p. 61).

[NID96] C. Norris Ip and D. Dill. "Better verification through symmetry". In: *Formal methods in system design* 9.1 (1996), pp. 41–75 (cit. on pp. 47, 50, 65).

[PAV14] S. Pinchinat, M. Acher, and D. Vojtisek. "Towards Synthesis of Attack Trees for Supporting Computer-Aided Risk Analysis". In: *Workshop on Formal Methods in the Development of Software (co-located with SEFM)*. 2014 (cit. on p. 83).

[Pel07] R. Pelánek. "BEEM: Benchmarks for Explicit Model Checkers". In: *Model Checking Software, 14th Int'l SPIN Workshop*. Vol. 4595. LNCS. Springer, 2007, pp. 263–267 (cit. on p. 33).

[Pop06] L. Popova. "Time Petri Nets State Space Reduction using Dynamic Programming". In: *Journal of Control and Cynernetics* 35.3 (2006), pp. 721–748 (cit. on p. 78).

[Ran+95] R. K. Ranjan, A. Aziz, R. K. Brayton, B. Plessier, and C. Pixley. "Efficient BDD Algorithms for FSM Synthesis and Verification". In: *In IEEE/ACM Proceedings International Workshop on Logic Synthesis, Lake Tahoe (NV*. 1995 (cit. on p. 19).

[Rat+03] A. V. Ratzer, L. Wells, H. M. Lassen, M. Laursen, J. F. Qvortrup, M. S. Stissing, M. Westergaard, S. Christensen, and K. Jensen. "CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets". In: *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets*. ICATPN'03. Eindhoven, The Netherlands: Springer-Verlag, 2003, pp. 450–462. ISBN: 3-540-40334-5 (cit. on p. 58).

[RCP95] O. Roig, J. Cortadella, and E. Pastor. "Verification of asynchronous circuits by BDD-based model checking of Petri nets". In: *16th International Conference on the Application and Theory of Petri Nets*. Vol. 815. 1995, pp. 374–391 (cit. on pp. 23, 26).

[Sch03] K. Schmidt. "Distributed Verification with LoLA". In: *Fundam. Inform.* 54.2-3 (2003), pp. 253–262 (cit. on p. 58).

[Sim71] C. Sims. "Computation with permutation groups". In: *Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*. ACM. 1971, pp. 23–28 (cit. on pp. 47, 54).

[Som05] F. Somenzi. *CUDD: CU Decision Diagram Package (release 2.4.1)*, `http://vlsi.colorado.edu/fabio/CUDD/cuddIntro.html`. 2005 (cit. on p. 23).

[SRB02] F. Somenzi, K. Ravi, and R. Bloem. "Analysis of Symbolic SCC Hull Algorithms". In: *Proc. of FMCAD'02 (FMCAD'02)*. Vol. 2517. LNCS. Springer, 2002, pp. 88–105 (cit. on pp. 38, 43, 76).

[STC98] M. Silva, E. Terue, and J. M. Colom. "Linear algebraic and linear programming techniques for the analysis of place/transition net systems". In: *Lectures on Petri Nets I: Basic Models*. Springer Berlin Heidelberg, 1998, pp. 309–373 (cit. on p. 5).

[Ste+09] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0*. 2nd. Addison-Wesley Professional, 2009. ISBN: 0321331885 (cit. on pp. 72, 77).

[STV05] R. Sebastiani, S. Tonetta, and M. Y. Vardi. "Symbolic Systems, Explicit Properties: on Hybrid Approches for LTL Symbolic Model Checking". In: *Proceedings of 17th International Conference on Computer Aided Verification (CAV'05)*. Vol. 3576. LNCS. Edinburgh, Scotland, UK: Springer, July 2005, pp. 350–363 (cit. on p. 38).

[TH08] Y. Thierry-Mieg and L. Hillah. "UML behavioral consistency checking using instantiable Petri nets". In: *ISSE* 4.3 (2008), pp. 293–300 (cit. on pp. 61, 82).

[TM+09] Y. Thierry-Mieg, D. Poitrenaud, A. Hamez, and F. Kordon. "Hierarchical set decision diagrams and regular models". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* 5505 (2009), pp. 1–15 (cit. on pp. 61, 65, 66, 83).

[TM+11] Y. Thierry-Mieg, B. Bérard, F. Kordon, D. Lime, and O. H. Roux. "Compositional Analysis of Discrete Time Petri nets". In: *1st workshop on Petri Nets Compositions (CompoNet 2011)*. Vol. 726. CEUR, 2011, pp. 17–31 (cit. on pp. 62, 63, 67, 80).

[TMDM03] Y. Thierry-Mieg, C. Dutheillet, and I. Mounier. "Automatic Symmetry Detection in Well-Formed Nets". In: *Proc. of ICATPN 2003*. Vol. 2679. LNCS. Springer Verlag, June 2003, pp. 82–101 (cit. on p. 55).

[TMIP04a] Y. Thierry-Mieg, J.-M. Ilié, and D. Poitrenaud. "A Symbolic Symbolic State Space". In: *Proc. of the 24th IFIP WG 6.1 Int. Conf. on Formal Techniques for Networked and Distributed Systems (FORTE'04)*. Vol. 3235. LNCS. Madrid, Spain: Springer, 2004, pp. 276–291 (cit. on pp. 50, 51).

[TMIP04b] Y. Thierry-Mieg, J.-M. Ilié, and D. Poitrenaud. "A Symbolic Symbolic State Space Representation". In: *Formal Techniques for Networked and Distributed Systems – FORTE 2004: 24th IFIP WG 6.1*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 276–291 (cit. on p. 67).

[Var96] M. Y. Vardi. "An Automata-Theoretic Approach to Linear Temporal Logic". In: *Proceedings of the 8th Banff Higher Order Workshop (Banff'94)*. Vol. 1043. LNCS. Banff, Alberta, Canada: Springer-Verlag, 1996, pp. 238–266. ISBN: 3-540-60915-6 (cit. on p. 37).

[Vit+04] V. Vittorini, M. Iacono, N. Mazzocca, and G. Franceschinis. "The OsMoSys approach to multi-formalism modeling of systems". In: *Software and System Modeling* 3.1 (2004), pp. 68–81 (cit. on p. 60).

[Voe+13] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013, pp. 1–558. ISBN: 978-1-4812-1858-0 (cit. on pp. 59, 76).

[Wan04] F. Wang. "Formal Verification of Timed Systems: A Survey and Perspective". In: *IEEE* 92.8 (2004) (cit. on p. 13).

[Zha+10] Y. Zhang, B. Bérard, F. Kordon, and Y. Thierry-Mieg. "Automated Controllability and Synthesis with Hierarchical Set Decision Diagrams". In: *Workshop on Discrete Event Systems (WODES)*. Berlin, Germany: IFAC/Elsevier, Sept. 2010, pp. 291–296 (cit. on pp. 76, 86).

[Zha+14] Y. Zhang, B. Bérard, L. M. Hillah, F. Kordon, and Y. Thierry-Mieg. "Controllability for Discrete Event Systems Modeled in VeriJ". In: *International Journal of Critical Computer-Based Systems* 5.3/4 (Sept. 2014), pp. 218–240. ISSN: 1757-8779 (cit. on p. 86).

[Zha13] Y. Zhang. "Semi-Automatic Controller Design in a Java-like Language". PhD thesis. Paris, France: Universite Pierre et Marie Curie, 2013 (cit. on pp. 86, 88).