# Symbolic Bounded Analysis for Component Behavioural Protocols

Pascal Poizat*, Jean-Claude Royer**, and Gwen Salaün

[1] IBISC - FRE 2873 CNRS, Tour Évry 2,
523 place des terrasses de l'Agora, F-91000 Évry Cedex
Pascal.Poizat@lami.univ-evry.fr
[2] OBASCO Group, EMN - INRIA, LINA
4 rue Alfred Kastler, BP 20722, F-44307 Nantes Cedex 3
Jean-Claude.Royer@emn.fr
[3] VASY Project, INRIA Rhône-Alpes
655 Avenue de l'Europe, F-38330 Montbonnot Saint-Martin
Gwen.Salaun@inrialpes.fr

**Abstract.** Explicit behavioural protocols are now accepted as a mandatory feature of components to address architectural analysis. Behavioural protocol languages must be able to deal with data types and with rich communication means. Symbolic Transition Systems are an adequate component model which takes into account dynamic aspects and data types. However, verification of components described with STS protocols is difficult since they possibly involve different sources of infinity. In this paper, we propose a notion of *symbolic bounded analysis*. This approach tests boundedness of a possibly infinite system, and then generates a finite simulation for it. Afterwards, standard model-checking techniques can be used for verification purposes.

## 1 Introduction

Behavioural interface description languages and protocol descriptions are needed in component models to address architectural analysis and verification issues such as checking component behavioural compatibility, finding architectural deadlocks or building adaptors to compensate incompatible component interfaces, but also to relate efficiently design models and implementation ones. In this context, different behavioural models have been used, such as process algebras [1, 30, 11] or automata-based formalisms [41, 38, 6].

Components may exchange data with service requests, or may internally compute data values on which behaviours depend, yielding compositions which deadlock only for some specific values (*e.g.,* think of an arithmetic component which accepts two integers, $x$ and $y$ and denies service when $y$ is 0). Therefore, there is a need for component models integrating data types within behaviours. In order to

* Supported by the French national project RNRT STACS on abstract and compositional techniques for model-based testing.
** Supported by the French national project ACI DISPO on components availability.

avoid state explosion problems resulting from this extension, the usual models for behaviours, Labelled Transition Systems (LTSs), can be made symbolic on the data types, yielding models found under different names in the literature: Symbolic Transition Graphs (STGs) [26], *Symbolic Transition Systems* (STSs) [14, 4, 35] or Input-Output Symbolic Transition Systems (IOSTSs) [28]. Their common features are symbolic states, guarded transitions with value-passing events and underlying data types.

Formal verification of STS descriptions is a great issue since it may involve different sources of infinity: infinite domains of data structure, infinite number of concurrent entities, unbounded queues in case of asynchronous communication, etc. In this paper, we focus on dynamic systems with possibly infinite data types. Checking boundedness or reachability in dynamic systems with data types is an important area of research. One specific case is boundedness of asynchronous communicating systems where the data type is a FIFO buffer. The problem is not decidable [12] but some partial procedures exist [29, 32, 33, 35]. For instance, we give in [35] a first result on simple counter systems, where only $> 0$ guards and $+1$, $+0$, and -1 as actions are possible.

In this paper, we propose a more operational formalisation of the STS model, and a generalisation of its analysis mechanisms. We first define STS, counter STS and a decidable *boundedness* decision procedure for them. Our approach tests the boundedness of a system, and then provides a finite version of it if bounded. Hence, existing model-checking techniques and tools (based on standard models such as LTSs) can be used to prove properties on it. Our goal is to complement model-checking with reference to its limit in the presence of data types within behaviours. Next, we formalise a notion of *decomposition* to split a system into several parts. Boundedness is checked separately and verification is possible on bounded parts of the specification. Our symbolic boundedness analysis is especially worthy with systems (and they are numerous) involving bounded resources, *i.e.,* where a part of the system is bounded. All the material presented in this paper has been implemented in a Python prototype tool.

The paper is organised as follows. Section 2 formalises definitions of STS, configuration graphs, relations between STS and LTS, and synchronous product of STSs. Section 3 introduces our symbolic approach and presents results about the boundedness of counter STS. Section 4 defines bounded decomposition and illustrates it respectively on two examples: a ticket mutual exclusion protocol and a resource allocator. Section 5 reviews related work. Finally, Section 6 draws up some concluding remarks.

## 2 Preliminary Definitions

This section states some definitions we use thereafter to introduce our symbolic bounded approach.

### 2.1 Algebraic Specifications

We will here give only the necessary insight into algebraic specifications, see [3] for more details. We consider algebraic specifications since they can be abstracted from a concrete implementation language like Java, C++, or Python. A *signature* (or static interface) $\Sigma$ is a pair $(\mathcal{S}, F)$ where $\mathcal{S}$ is a set of *sorts* (type names) and $F$ a set of function names equipped with *profiles* over these sorts. If $R$ is a sort, then $\Sigma_R$ denotes the subset of functions from $\Sigma$ with result sort being $R$. $X$ is used to denote the set of all variables. It contains a distinguished variable, $Self$, whose goal is much like explicit receivers in Object-Oriented languages (*e.g.,* `this` in Java). From a signature $\Sigma$ and from $X$, one may obtain *terms*, denoted by $T_{\Sigma,X}$. The set of *closed terms* (also called ground terms) is the subset of $T_{\Sigma,X}$ without variables, denoted by $T_\Sigma$. An *algebraic specification* is a pair $(\Sigma, Ax)$ where $Ax$ is a set of axioms between terms of $T_{\Sigma,X}$. $r\downarrow$ denotes the normal form (assumed to be unique) of the ground term $r$. $[R]$ denotes the set of all normal form terms of sort $R$ and $r : R$ means that $r$ is in this set. $r(u)$ denotes the application of $r$ to $u$.

### 2.2 Symbolic Transition Systems

Symbolic Transition Graphs [26] have initially been developed as a solution to the state and transition explosion problem in value-passing process algebras using substitutions associated to states and symbolic values in transition labels. Our Symbolic Transition Systems (STS) are a generalisation of these, associating a symbolic state and transition system with a data type description which may be given either using algebraic specifications, model oriented specifications or even OO classes (Java or Python).

**Definition 1 (STS).** *An STS is a tuple* $(D,\ (\Sigma, Ax),\ S,\ L,\ s^0,\ T)$ *where:* $(\Sigma, Ax)$ *is an algebraic specification,* $D$ *is a sort called* sort of interest *defined in* $(\Sigma, Ax)$, $S = \{s_i\}$ *is a countable set of states,* $L = \{l_i\}$ *is a countable set of event labels,* $s^0 \in S$ *is the initial state, and* $T \subseteq S \times T_{\Sigma_{Boolean},X} \times Event \times T_{\Sigma_D,X} \times S$ *is a set of transitions.*

*Events* denote atomic activities that occur in the components. Events are either: *i*) hidden (or internal) events: $\tau$, *ii*) silent events: $l$, with $l \in L$, *iii*) emissions: $l!e$, with $e \in T_{\Sigma,\{Self_D\}}$, or *iv*) receptions: $l?x : R$ with $x \in X \backslash \{Self_D\}$. To simplify we only consider binary communications here, but emissions and receptions may be extended to n-ary emissions and receptions. STS transitions are tuples $(s,\ \mu,\ \epsilon,\ \delta,\ t)$ for which $s$ is called the source state, $t$ the target state, $\mu$ the guard, $\epsilon$ the event and $\delta$ the action. Each action is denoted by a term with variables where at least $Self$ occurs. A do-nothing action is simply denoted by $Self_D$. In forthcoming figures, transitions will be labelled as follows: `[`$\mu$`]` $\epsilon$ `/` $\delta$.

### 2.3 Configuration Graphs

The semantics of STS is formalised using configuration graphs. They are obtained applying jointly the unfolding of receptions and the reduction of ground terms to their normal forms.

**Definition 2 (Unfolding).** *The unfolding of an STS* $(D,\ (\Sigma, Ax),\ S,\ L,\ s^0,\ T)$, *in* $v^0 \in T_{\Sigma_D}$, *is the STS* $(D,\ (\Sigma, Ax),\ S',\ L,\ (s^0, v^0\downarrow),\ T')$. *The sets* $S' \subseteq S \times D$ *and* $T'$ *are inductively defined by the rules:*

- $(s^0, v^0\downarrow) \in S'$
- for each $(s, v) \in S'$
  - if $(s,\ \mu,\ \tau,\ \delta,\ t) \in T$ and $\mu(v)\downarrow = true$ then $s' = (t,\ \delta(v)\downarrow) \in S'$ and $((s, v),\ true,\ \tau,\ Self_D,\ s') \in T'$.
  - if $(s,\ \mu,\ l,\ \delta,\ t) \in T$ and $\mu(v)\downarrow = true$ then $s' = (t,\ \delta(v)\downarrow) \in S'$ and $((s, v),\ true,\ l,\ Self_D,\ s') \in T'$.
  - if $(s,\ \mu,\ l!e,\ \delta,\ t) \in T$ and $\mu(v)\downarrow = true$ then $s' = (t,\ \delta(v)\downarrow) \in S'$ and $((s, v),\ true,\ l!e(v)\downarrow,\ Self_D,\ s') \in T'$.
  - if $(s,\ \mu,\ l?x : R,\ \delta,\ t) \in T$ then for each $r : R$ such that $\mu(v, r)\downarrow = true$, there are $s' = (t,\ \delta(v, r)\downarrow)) \in S'$ and $((s, v),\ true,\ l!r,\ Self_D,\ s') \in T'$.

The unfolding principle is similar to the standard semantics of LOTOS [27] which considers $l?x : R$ as equivalent to the choice $[]_{r:R}\ l!r$. Pairs $(s, v)$ are called *configurations* and $s$ is the *control state*. Let $d$ be an STS, its unfolding in $v^0$, $G(d, v^0)$, is called a *configuration graph*. A configuration graph is a particular STS without reception, where guards are all equal to *true*, emission terms are in normal form and actions are only $Self_D$.

### 2.4 Interpretations

Configuration graphs and STS can be interpreted as LTS[1]. Such mappings enable one to use existing model-checkers, such as SPIN [25] or CADP [24], to verify these models. Hence, we introduce two LTS interpretations based on the following rules:

- $(r_1)$ any STS transition $(x,\ \mu,\ \epsilon,\ \delta,\ y)$ is reduced to a LTS one $(x,\ l,\ y)$, where $l$ is the label of the event $\epsilon$.
- $(r_2)$ any configuration $(s,\ v)$ is reduced to its *control state* $s$, and any STS transition $((s,\ v),\ \mu,\ \epsilon,\ \delta,\ (t,\ u))$ is reduced to a LTS one $(s,\ l,\ t)$.

**Definition 3 (LTS Interpretations).** *The standard interpretation,* $I_{LTS}$, *of an STS, is an LTS computed with the* $r_1$ *rule and discarding* $D$ *and* $(\Sigma, Ax)$. *The weak interpretation,* $W_{LTS}$, *of an STS, is an LTS computed with the* $r_2$ *rule and discarding* $D$ *and* $(\Sigma, Ax)$.

[1] We recall that an LTS is a structure $(S,\ L,\ s^0,\ T)$ with $T \subseteq S \times L \times S$.

We note $\supseteq$ the transition relation inclusion and $\sqsupseteq$ the trace inclusion of two LTSs. $d_1 \supseteq d_2$ means that $d_1$ and $d_2$ share the same set of states but the set of transitions of $d_2$ is a subset of the transitions of $d_1$. $d_1 \sqsupseteq d_2$ means that any $d_2$ trace is also a $d_1$ trace. As defined in [2] for LTS, $B = (S_B,\ L,\ b^0,\ T_B)$ is a *simulation* of $A = (S_A,\ L,\ a^0,\ T_A)$, noted $B \succeq A$, iff there is a relation $\mathcal{R}$ from $S_A$ to $S_B$ such that : i) $\forall s_A \in S_A,\ \exists s_B \in S_B$ such that $s_A \mathcal{R} s_B$, ii) if $s_A$ is initial $\exists s_B \in S_B$ such that $s_A \mathcal{R} s_B$ and $s_B$ is initial, and iii) $\forall (s_A,\ l,\ t_A) \in T_A, s_B \in S_B,\ s_A \mathcal{R} s_B \supset (\exists t_B \in S_B,\ (s_B,\ l,\ t_B) \in T_B \wedge t_A \mathcal{R} t_B)$.

**Proposition 1.** *Let $d$ be an STS:*

1. $W_{LTS}(d) = I_{LTS}(d)$ *if $d$ has only control states.*
2. $W_{LTS}(d) \supseteq W_{LTS}(G(d, v^0))$.
3. $I_{LTS}(d) \succeq I_{LTS}(G(d, v^0))$.
4. $I_{LTS}(d) \sqsupseteq I_{LTS}(G(d, v^0))$.

Point 3 above defines a simulation which in turn implies trace inclusion (point 4). Previous works [34, 17] have shown that simulation preserves a subset of $\mu$-calculus, namely safety properties. Therefore, model-checking may be used to prove or disprove such properties on STS using our interpretations.

### 2.5 Concurrency and Communication

Concurrent communicating components can be described with STS protocols, and a synchronous product (adapted from the LTS one definition) can be used to figure out the resulting global system. In addition, we handle structured STS, that is STS with tuple states and transitions, which memorize the system structuring.

Given two STS with sets of labels $L_1$ and $L_2$, a set $V$ of synchronisation vectors is a set of pairs $(l_1,\ l_2)$, called synchronous labels, such that $l_1 \in L_1$ and $l_2 \in L_2$. Hidden events cannot participate in a synchronisation. Two components synchronise at some transition if their respective labels are synchronous (*i.e.,* belong to a same vector) and if the label offers are compatible. Offer compatibility follows simple rules: type equality and emission/reception matching. A label $l$ such that there is no pair in $V$ which contains $l$, is said asynchronous. Corresponding transitions are triggered independently from other ones. We use overloading and $\times$ to represent the binary composition of states, data types, specifications, labels and events. The data type part of the STS product is a product of the component data types ($D = D_1 \times D_2$). The new constructor is built using the tuple constructor (with $v_1 : D_1$ and $v_2 : D_2$, this constructor builds the tuple $(v_1,\ v_2)$ for $D_1 \times D_2$), and accessors allow to reach basic data types $D_1$ and $D_2$. Operations of $D$ are defined combining operations of $D_1$ (resp. $D_2$) and accessors to it (*e.g.,* $op_D(d) = op_{D_1}(access_{D_1}(d))$ where $op_D : D \to T$, $op_{D_1} : D_1 \to T$, $access_{D_1} : D \to D_1$, and $d \in D$).

**Definition 4 (Synchronous Product).** *The synchronous product of two STS $d_i = (D_i,\ (\Sigma_i, Ax_i),\ S_i,\ L_i,\ s_i^0,\ T_i)$, $i = 1, 2$, relatively to a set $V$ of synchronisation vectors, denoted by $d_1 \otimes_V d_2$, is the STS $(D_1 \times D_2,\ (\Sigma_1, Ax_1) \times (\Sigma_2, Ax_2)$,*

*$S$, $L_1 \times L_2$, $s^0$, $T$), where the sets $S \subseteq S_1 \times S_2$ and $T \subseteq S \times T_{\Sigma_{Boolean},X} \times (Event_1 \times Event_2) \times T_{\Sigma_D,X} \times S$ are inductively defined by the rules:*

- $s^0 = (s_1^0, s_2^0) \in S$,
- for each $(s_1, s_2) \in S$, $(s_1,\ \mu_1,\ \epsilon_1,\ \delta_1,\ t_1) \in T_1$, $(s_2,\ \mu_2,\ \epsilon_2,\ \delta_2,\ t_2) \in T_2$ where $label(\epsilon_1) = l_1$, and $label(\epsilon_2) = l_2$
  - if $(l_1, l_2) \in V$ then $((s_1, s_2),\ \mu_1 \wedge \mu_2,\ (\epsilon_1, \epsilon_2),\ (\delta_1, \delta_2),\ (t_1, t_2)) \in T$ and $(t_1, t_2) \in S$.
  - if $l_1$ is asynchronous then $((s_1, s_2),\ \mu_1,\ (\epsilon_1, \tau),\ (\delta_1, Self_{D_2}),\ (t_1, s_2)) \in T$ and $(t_1, s_2) \in S$.
  - if $l_2$ is asynchronous then $((s_1, s_2),\ \mu_2,\ (\tau, \epsilon_2),\ (Self_{D_1}, \delta_2),\ (s_1, t_2)) \in T$ and $(s_1, t_2) \in S$.

The product is associative and commutative. Binary vectors and products can be extended to $n$-ary ones. Complex component architectures can be structured by the product and yield STS whose events are trees having component STS events as leaves. LTS interpretations can be used provided that these tree structured events can be flattened into simple ones. This can be achieved associating to each vector an event denoting the observable result of the synchronisation. Unfolding is extended to deal with asynchronous activities, synchronisation without communication, and synchronisation with communication. The first two cases are simple therefore we only give details for the third one. Transitions with synchronisation and communication have the form:

$$((s_1, s_2),\ \mu_1 \wedge \mu_2,\ (l_1!e, l_2?x : R),\ (\delta_1, \delta_2),\ (t_1, t_2))$$

where $(l_1!e, l_2?x : R)$ is a synchronous event. The unfolding rule in this case is:

if $s = ((s_1, s_2),\ (v_1, v_2))$ is a state of the configuration graph, $\mu_1(v_1)\downarrow = true$ and $\mu_2(v_2, e(v_1))\downarrow = true$ then

- $s' = ((t_1, t_2),\ (\delta_1(v_1)\downarrow, \delta_2(v_2, e(v_1))\downarrow))$ is a state of the configuration graph,
- $(s,\ true,\ (l_1!e(v_1)\downarrow, l_2!e(v_1)\downarrow),\ (Self_{D_1}, Self_{D_2}),\ s')$ is a transition of the configuration graph.

**Proposition 2.** *Let $d_1$ and $d_2$ be two STS, $V$ a set of vectors, $v_1 \in T_{\Sigma_{D_1}}$ and $v_2 \in T_{\Sigma_{D_2}}$:*

1. $G(d_1 \otimes_V d_2, (v_1, v_2)) \equiv G(G(d_1, v_1) \otimes_V d_2, v_2) \equiv (G(d_1, v_1) \otimes_V G(d_2, v_2))$.
2. $I_{LTS}(d_1 \otimes_V d_2) \succeq I_{LTS}(G(d_1, v_1) \otimes_V d_2) \succeq I_{LTS}(G(d_1 \otimes_V d_2, (v_1, v_2)))$.
3. $I_{LTS}(d_1 \otimes_V d_2) \sqsupseteq I_{LTS}(G(d_1, v_1) \otimes_V d_2) \sqsupseteq I_{LTS}(G(d_1 \otimes_V d_2, (v_1, v_2)))$.

Let $\Phi$ be the configuration isomorphism from $(S_1 \times S_2) \times (D_1 \times D_2)$ to $(S_1 \times D_1) \times (S_2 \times D_2)$. The symbol $\equiv$ stands for the isomorphism extension of $\Phi$ on STS structures, it implies weak bisimulation [2]. Proposition 2.1 gives three ways to compute the configuration graph of an STS product. Proposition 2.2 shows that the interpretation of $G(d_1, v_1) \otimes_V d_2$ is a finer simulation for $I_{LTS}(G(d_1 \otimes_V d_2, (v_1, v_2)))$ than $I_{LTS}(d_1 \otimes_V d_2)$. Proposition 2.3 states that the standard interpretation of an STS product defines a trace superset of the standard interpretation of its configuration graph. It also shows that a smaller trace superset is obtained when computing the configuration graph of a component.

### 2.6 Prototype

A prototype has been developed in Python (about 2000 lines of code) to support the previous notions and some presented in the next sections. Currently our Python software supports STS, their synchronous product, the boundedness checking and the configuration graph computation. Currently our checking algorithm is more general than the one described in this paper and do not use a Petri net translation. An STS is defined by a textual description of the protocol (states and transitions). A Python class describes the data type and its methods implement the guards and actions occurring in the dynamic description. For the sake of readability, we illustrate with small values for variables in the remainder. Nevertheless, some experiments have been carried out as well with bigger values. For instance, our approach is quite efficient compared to TLC [37], and the computation of a configuration graph of 5000 states and 5000 transitions takes nearly one minute. We have already applied successfully our approach (boundedness, decomposition and model-checking) to several examples : a flight reservation system, several variants of the bakery protocols, the slip protocol, several variants of a resource allocator, and a cash point service.

## 3 Bounded Analysis

This section presents our bounded analysis and illustrates how it complements model-checking techniques on finite and infinite systems. The CADP toolbox [24] will be taken as a representative implementation of these standard techniques.

Standard model-checking is based on enumerative approaches. State spaces are generated from specifications written in high-level languages such as process algebras. Bounded[2] model-checking techniques rely on BDD encodings to deal with big state spaces. However this is not sufficient when components encapsulate or exchange data. Possibly infinite data type domains must be restricted and free variables bound to avoid state explosion. For instance, reasoning on LOTOS specifications using CADP [24] may be performed in different ways. The underlying global LTS can be first generated and then verified. On-the-fly techniques can also be used to avoid the generation of the whole global system [36]. A shortcoming of both approaches is that model-checking is applied to a restricted finite state system. Accordingly, full correctness cannot be ensured. With regards to this shortcoming, we introduce in the sequel of this section an approach preserving bounded values. Our objective is not to replace existing model-checking techniques and tools which are efficient with regards to enumerative approaches. Bounded analysis has to be viewed as a complementary means to detect possible flaws that model-checking may miss.

[2] Note that the *bounded* term in our approach is related to the kind of systems it operates on, STS, and not to bounded model-checking.

### 3.1 Bounded Analysis Based on STS

Generally a system has some bounded variables and unbounded or "unknown" variables. Enumerative model-checking will arbitrarily bounds the variables and it may be insufficient to assert a given property for the whole system. The bounded analysis consists in computing the synchronous product of the STS and then studying properties of interest (first boundedness, and then, properties such as deadlocks and safety properties) over this global STS using unfolding and interpretations. If the system is not bounded, verification techniques developed for infinite systems are relevant, see [7, 9, 22, 19, 18, 5, 10] for examples. As an example relevant for components, we introduce in the next section the case of bounded decomposition. If the system is bounded, we can compute or estimate variable bounds which may be useful for both verification and implementation, see [29, 32, 35] for related algorithms. Knowing bounds can be used to optimise the generation of the bounded system, but we do not developed further this case here.

**Definition 5 (Bounded STS).** *An STS is bounded, for an initial value $v_0$, iff its configuration graph is finite.*

Checking boundedness is not decidable for STS since they provide a general computational model as powerful as Turing machines or Minsky machines. More precisely it is a semi-decidable problem and a semi-algorithm computing the configuration graph has been implemented in our prototype. However boundedness is decidable for some specific classes of STS. Thus, one may reuse various existing results [21, 31, 33]. Similarly, our prototype implements the semi-algorithm mentioned above and a decision procedure for counter STS.

We present here counter STS as an adequate abstraction for many systems, *e.g.,* they generalise the dictionary STS we are used in [35] to check component mailboxes in the context of asynchronous communicating systems. They are also convenient in the context of component availability properties since counter STS can describe dynamic systems allocating finite amount of resources.

**Definition 6 (Counter STS).** *A counter STS, is an STS where: i) the data type is restricted to natural numbers (counters) $c_i$, ii) guards are boolean conjunctions of the following atoms:* `true`, `false`, *$c_i > n_i$, or $c_i \geq n_i$, where $n_i$ is a natural, iii) actions are $c_i := c_i \pm p_i$, where $p_i$ is a natural, and iv) if a transition has $c_i > n_i$ (resp. $c_i \geq n_i$) in its guard and $c_i := c_i - p_i$ in its action then $n_i > p_i$ (resp. $n_i \geq p_i$).*

In this definition, counters are independent thus counter STS only allows constant communications between components, see [21, 16] for related results allowing bounded communications.

**Proposition 3.** *The boundedness of counter STS is decidable.*

The proof is achieved first translating counter STS into Petri nets [39] and then using the Petri net boundedness decision procedure. For illustration purposes, we describe in Figure 1 the translation of an STS transition with two

counters. The principle is to associate a place with each counter and each control state. The initial assignment defines the initial marking of the net. Transitions are defined as Petri transitions. Arcs are defined as described in Figure 2.

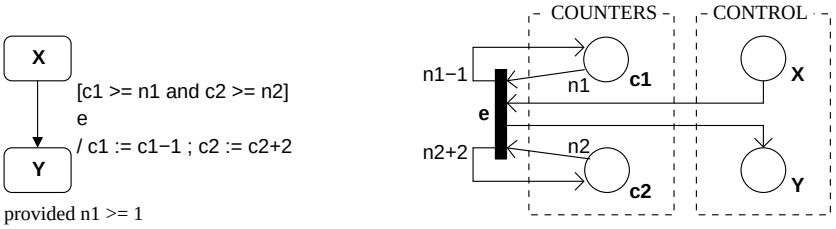


**Fig. 1.** A counter STS transition and its Petri net translation

The basic constraint to take into consideration is $\geq$. Guards of form $c \geq M$ are translated into input arcs with weight $M$. Actions $c := c \pm b$ are used to build output arcs. Subtraction is not always defined, and cases $M = b$ and $M > b$ must be taken into consideration.

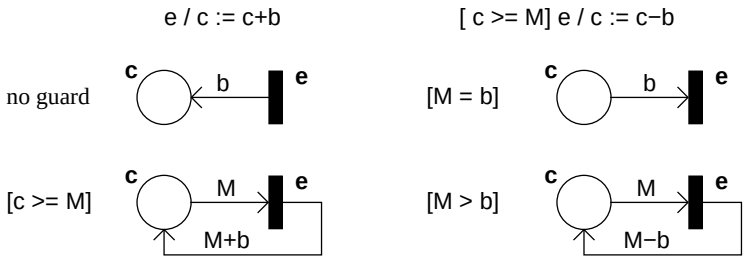


**Fig. 2.** Petri net translation for addition and subtraction

This Petri net translation does not enable $=$, $<$ or $\leq$ tests between a counter and a constant since it allows to test emptiness of places which is a theoretical limitation of the Petri net formalism. However, still using a Petri net approach, boundedness of STS counters with global $c = M$, $c < M$ and $c \leq M$ constraints on counters can be decided. To translate these constraints, we use complementary places. A $c^+$ counter is associated to each bounded place and encodes the bound. The constraint may then be represented as the invariant $c + c^+ = M$. Any transition acting on $c$, *i.e.,* $c := c \pm b$, has also to perform the complementary action on $c^+$.

### 3.2 Application to Finite-State Systems

We consider a simple example, see Fig. 3, with two components which synchronise on `(emit, get)` and their bounded product. This example has been encoded in LOTOS. The global LTS computed using CADP with $M = 200$ is made up of 401 states and 400 transitions. Using our prototype, we get the same result in 1 second (product+unfolding).

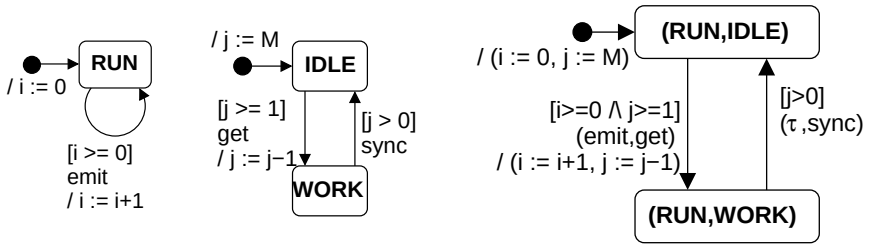


**Fig. 3.** Finite State System Example

CADP arbitrary bounds natural numbers to 256, this is the reason why we used a value smaller than 256 for $M$. But bounded analysis allows the computation of the two STS product and to check its boundedness for bigger values. Our prototype computes the product and builds the configuration graph of this example within reasonable time (*e.g.,* 21 seconds for $M = 1000$). Whenever the bounds set by model-checking tools are reached, the specifier does not know if its system is either too big for the tool or really unbounded. In such a case, bounded analysis may be successful and complements model-checking, for example it can provide the exact bounds to optimise the generation of the state system.

Model-checking is an efficient way to automatically prove properties on finite-state systems. However our analysis may provide the following benefits:

- an abstract and often readable description of the global system can be figured out,
- early checking can be performed on this bounded description (boundedness, deadlocks, ...), and in some cases,
- when model-checking fails on the initial specification, the configuration graph may be computed and model-checked.

### 3.3 Application to an Infinite State System

Bounded analysis, as abstraction technique, can be useful on finite systems as well as on infinite ones. Whenever the bounds set by model-checking tools are reached, the specifier does not know if his system is either too big for the tool or really unbounded. In such a case, bounded analysis is successful and complements model-checking. Let us consider an infinite global system in which some

components are finite (bounded), which has been proved using the method introduced above. Indeed, we compute the configuration graph of one component, the product, then the LTS interpretation. We recall with reference to Proposition 2.2 that it is a finer interpretation that simply computing the product and interpreting it afterwards.

As an illustration, let us take a resource allocator system with two components: the allocator and the client system. Figure 4 presents the STS descriptions of these components. The *allocator* can start (`init`), accept a request for a quantity (`ask`), send a resource unit (`acquire`), release a quantity (`release`), fulfill a request (`end`). The maximal amount of resources shared by the allocator is `size`. Variable `gauge` is used to keep track of the allocated resources. On the other hand, the *client system* centralizes the management of all the clients which are requiring resources. To simplify the presentation, we have omitted the client actions of entering and leaving the system. The client system can start (`init`), send a request for a quantity (`ask`), accept a resource unit (`acquire`), release the quantity acquired by the client `i` (`release`), terminate the request (`new`), return to the idle state (`ok`). The amount requested by one client is identified by the `QUOTA` constant which is defined smaller than `size`. Variable `num` stores the current number of resources while acquiring them, and the `acq` list stores the acquired resources for all clients. Synchronisations are `(init,init)`, `(release,release)`, `(acquire,acquire)`, `(ask,ask)` and `(end,ok)`.

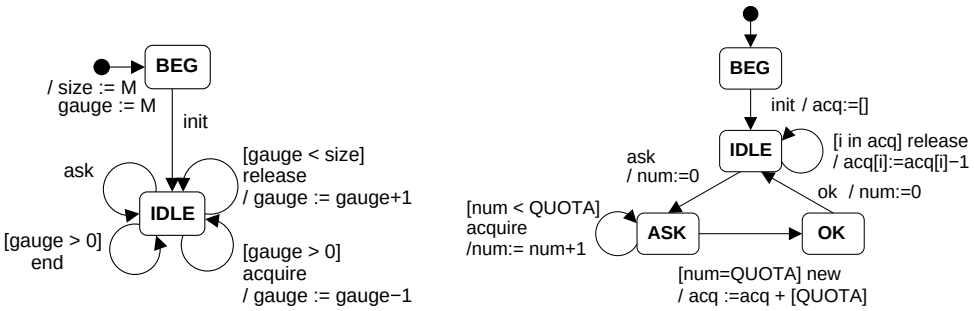


**Fig. 4.** Resource allocator system (left: allocator, right: client system)

Under the hypotheses of a given `size` and a given `QUOTA`, the system is not finite since the number of clients is not known. Model-checking must set this number, and hence will find a deadlock. Indeed, after a while, resources will lack since resources acquired by a client are not all released before starting a new request. Thus we can only assert that the allocator deadlocks for a given number of clients. However, bounded analysis can be performed, since the allocator is bounded (`size` is a constant and `gauge` $\leq$ `size` is a global constraint). In this example the weak interpretation of the STS ($W_{LTS}(allocator \otimes_V client)$) resulting from the synchronous product of the allocator and the client system STS,

and the configuration graph of the allocator STS ($G(allocator, M)$) are deadlock free. Finally, we can detect that the synchronous product of the allocator configuration graph and the client STS ($G(allocator, M) \otimes_V client$) deadlocks without choosing arbitrarily a specific number of clients.

## 4 Bounded Decomposition

Results of the previous section are extended by a notion of decomposition which allows in a first step to generate finite representations of bounded parts of a system, and to check them in a second step.

### 4.1 Principle

The idea is to choose a subset of the data and to do a partial evaluation of STS using it. The computation of the configuration graph is adjusted to only evaluate guards and actions related to the selected data. One can then analyse parts of an STS which can be bounded. This requires the STS to be decomposable. In this section, we introduce our definitions on a binary decomposition, even though the decomposition can be extended to $n > 2$ and can be iterated several times.

**Definition 7 (Decomposable STS).** *An STS* $(D,\ (\Sigma, Ax),\ S,\ L,\ s^0,\ T)$ *is decomposable if and only if:*

- $D$ *can be decomposed into* $D_1 \times D_2$,
- *for each* $(s,\ \mu,\ \epsilon,\ \delta,\ t)$ *in* $T$, *for each* $v = (v_1, v_2) : D$, $\mu(v) \equiv \mu_1(v_1)\ \wedge\ \mu_2(v_2)$, *with* $\mu_i$ *a guard for* $D_i$,
- *for each* $(s,\ \mu,\ \epsilon,\ \delta,\ t)$ *in* $T$, *for each* $v = (v_1, v_2) : D$, $\delta(v) \equiv (\delta_1(v_1), \delta_2(v_2))$, *with* $\delta_i$ *a function on* $D_i$.

When $d$ is decomposable we may define two successive partial unfolding, $G_1$ and $G_2$. $G_1$ simulates the system relatively to $D_1$ and keeps unchanged information related to $D_2$. $G_1$ can be viewed as a partial evaluation of the configuration graph. We focus here on emissions, the principle extends to other kinds of event. $G_1$ applies to transitions $(s,\ \mu,\ l!e,\ \delta,\ t)$ and values $v_1 : D_1$. If $\mu_1(v1)\downarrow = true$, $G_1$ generates a transition $((s, v_1), \mu_2,\ l!e,\ (Self_{D_1}, \delta_2),\ (t, \delta_1(v_1)\downarrow\ ))$. $G_2$ simulates $G_1(d, v_1^0)$ relatively to $D_2$. Hence, it applies to transitions generated by $G_1$ and values $v_2 : D_2$. If $\mu_2(v_2)\downarrow = true$, $G_2$ generates a transition $((s, (v_1, v_2)),\ true,\ l!e((v_1, v_2))\downarrow,\ (Self_{D_1}, Self_{D_2}),\ (t, (\delta_1(v_1)\downarrow, \delta_2(v_2)\downarrow))$. During the $G_1$ step, internal communications and (external) emissions are evaluated. However, receptions from $D_2$ must be delayed until the $G_2$ step takes place. We emphasize that such a decomposition principle always applies to counter STS.

**Proposition 4.** *Let* $d$ *be a decomposable STS, the configuration graph* $G$ *of* $d$ *is computed as follows:*

$$G(d, (v_1^0, v_2^0)) \equiv G_2(G_1(d, v_1^0), v_2^0)$$

On the left hand side, a transition such as $(s,\ \mu,\ l!e,\ \delta,\ t)$ with $v = (v_1, v_2)$, becomes $((s,(v_1,v_2)),\ true,\ l!e((v_1,v_2))\downarrow,\ (Self_{D_1}, Self_{D_2}),\ (t,(\delta(v_1,v_2)\downarrow)))$. On the right hand side, the transition is $((s,v_1),v_2),\ true,\ l!e((v_1,v_2))\downarrow,\ (Self_{D_1}, Self_{D_2}),\ (t,(\delta_1(v_1)\downarrow,\delta_2(v_2)\downarrow))$ if $\mu_1(v_1)\downarrow = true$ and $\mu_2(v_2)\downarrow = true$. Both results are equivalent taking into account the decomposition properties of $d$ and the state isomorphism from $S_1 \times (D_1 \times D_2)$ to $(S_1 \times D_1) \times D_2$.

**Definition 8 (Bounded Decomposition).** *If $d$ is a decomposable STS and $G_1(d, v_1^0)$ is finite then it is a bounded decomposition of $d$.*

Bounded decompositions define abstractions of STS which yet respect interesting properties with reference to the initial STS. These properties ensure that some analysis for the initial STS can be undertaken on one of its bounded decomposition. Propositions 1.3 and 4 ensure that the standard interpretation of the bounded decomposition $G_1(d, v_1^0)$ is a simulation of the standard interpretation of $G(d, (v_1^0, v_2^0))$. This means that safety properties involving only variables of the decomposition can be verified on the decomposition.

**Proposition 5.** *If $d$ and $d'$ are decomposable STS then $d \otimes_V d'$ is a decomposable STS.*

There are several possible decompositions for $d \otimes_V d'$. Note that the STS synchronous product naturally yields decomposable STS. However, a nontrivial decomposition is the following. If $D = D_1 \times D_2$ and $D' = D'_1 \times D'_2$ then the data type of $d \otimes_V d'$ is $(D_1 \times D_2) \times (D'_1 \times D'_2)$ which is isomorphic to $(D_1 \times D'_1) \times (D_2 \times D'_2)$. $d$ and $d'$ being decomposable, this isomorphism may guide a new decomposition of $d \otimes_V d'$.

### 4.2 Application: the Ticket Mutual Exclusion Protocol

We illustrate first the decomposition principle on a mutual exclusion protocol inspired by the ticket protocol as described in [18]. However, our version differs from that one since we deal with distributed components communicating by messages, and not processes operating on a shared memory. We also distinguish entering (`use`) and leaving (`end`) the critical section. Finally, a counter `C` and a guard `C=0` are added to the server which computes the number of processes in critical section. This counter is useful to check the mutual exclusion property. STSs associated to client and server are described in Figure 5. Synchronisations are summarized in the following vectors: `(think,givet)`, `(use,gives)`, and `(end,end)`.

This system is unbounded since variables `S, T`, and `A` can store arbitrary large values. We split the variables into {`C`} and {`T,S`} for the server, and {} and {`A`} for the client. Then, decomposition produces a partial configuration graph (on `C`) on which boundedness is checked. This configuration graph automatically dumped into a DOT format using our prototype is presented in Figure 6. Note that the left part of each state, *e.g.,*`[0]`, stands for the value of the `C` variable

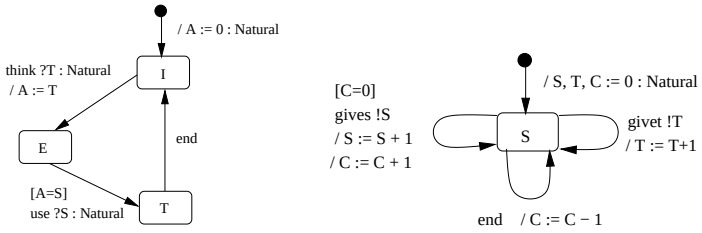


**Fig. 5.** STS descriptions: client (left) and server (right)

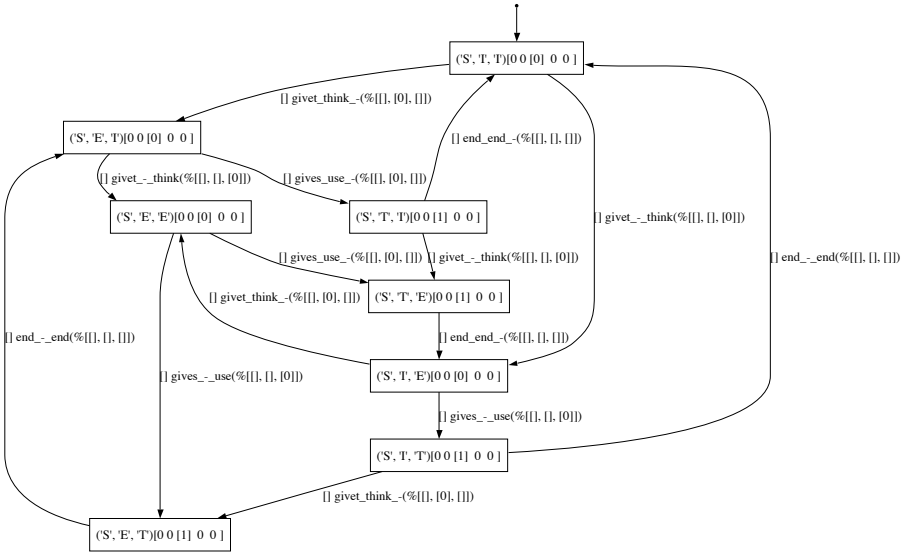


**Fig. 6.** Configuration graph for {C}: one server and two clients

which varies from `0` to `1` depending on the presence or not of a client in critical section.

From such a finite system, safety properties like mutual exclusion can be checked. In Figure 6, mutual exclusion graphically appears as the absence of the erroneous state (`S,T,T`) or as the fact that `C<=1`. Our prototype succeeds in generating the global system (then checking mutual exclusion) up to 8 clients whereas CADP and SPIN (with the default configuration values and bounded data types, *e.g.,* natural numbers bounded to 256) do not pass 6 clients. The resulting product (for 8 clients) is made up of 6561 states and 52488 transitions; the configuration graph contains 1280 states and 6656 transitions. A similar analysis is possible with other protocols, as we did with the Bakery mutual exclusion algorithm [19]. Finally, we stress out that our primary goal while implementing the prototype was the validation of the paper ideas. In particular, efficiency has to be improved, and is not satisfactory by now for several reasons: strongly dependent of Python high-level data structures, interpreted code wrt compiled code, no optimizing, etc.

### 4.3 Application: a Resource Allocator

This section illustrates the use of bounded decomposition on a variant (Fig. 7) of the Section 3 resource allocator. In this version client identities are communicated to the allocator which hence knows the client (`who`) and the requested quantity. The allocated (`GIVEN`) and the requested (`QUOTA`) amounts are natural number constants (not necessary equal). The constraint `size` $\geq$ `QUOTA` $\geq$ `GIVEN` $\geq$ `1` is assumed. The allocator communicates with the client system on the `delete` event when there are not enough free resources. Whenever this occurs, the client system releases the allocated resources owned by a client. Variable `num` stores the current quantity acquired by a client, while `total` accumulates the acquired resources for all clients. Variable `id` is used to store the client identities and `acq` the allocated quantities.

The global system is not bounded, and furthermore none of the components is. A possible decomposition is to separate actions on identities from actions on quantities as allowed by Proposition 5. Hence, one has on the one hand variables {`size, gauge`} and on the other hand the `who` variable. As regards the client, its decomposition is based on a partition between variables {`size, num, total`} and variables {`who, acq, id`}. Figure 8 presents the system decomposition view, which was obtained from the synchronous product of the allocator and the client system. Guards and actions not related to the variables {`size, gauge`} of the allocator and {`size, num, total`} of the client system get hidden in the decomposition.

Fixing values for `size`, `QUOTA`, and `GIVEN`, the boundedness is checked to be true for this decomposition. We carried out experiments on the system for various values of `size`, `QUOTA`, and `GIVEN`. As an example with `size`=1000, `QUOTA`=2, and `GIVEN`=1, a configuration graph of 2503 states and 3004 transitions is built. Bounded analysis experiments show first that if `GIVEN` does not divide `QUOTA` the system deadlocks. In the state `(WORD, ASK)` only three transitions are

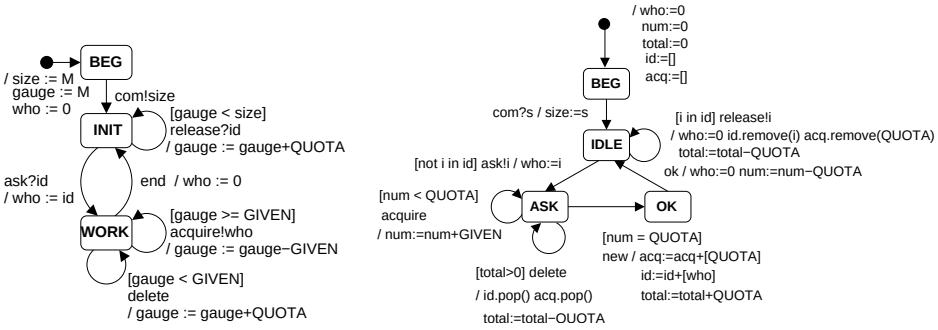


**Fig. 7.** Revisited resource allocator system (left: allocator, right: client system)

possible: `(acquire, acquire)`, `(delete, delete)`, and `(-, new)` (see Fig. 8). Since `GIVEN` does not divide `QUOTA`, the condition `num > QUOTA` will be eventually true and `num = QUOTA` will be never true. Note also that the condition `gauge < GIVEN` which enables `delete` becomes false after triggering this transition (since `QUOTA >= GIVEN`). Thus the sequence `delete ; delete` cannot occur, this is a safety property we checked on the bounded decomposition.

On the other hand, if `GIVEN` divides `QUOTA` then the bounded decomposition has no deadlock. However, the bounded analysis is not sufficient to ensure that the global system is deadlock free. A thorough look at the bounded decomposition shows that the guards left to evaluate in the $G_2$ step are `[(gauge < size, i in id)]` and `[(true, not i in id)]` (see Fig. 7). At least one of these guards is true since either `[not i in id]` or `[i in id]` is true, thus an allocation has been done and `gauge < size` is true. Let $(s, v)$ be a configuration of the global configuration graph, by the simulation it exists a corresponding configuration with the same control state in the bounded decomposition. If the control state is not `(INIT, IDLE)`, there is no guard to evaluate and since the bounded decomposition does not deadlock, the state has a successor state in the global configuration graph. If the control state is `(INIT, IDLE)`, the bounded decomposition has two outgoing transitions (labelled respectively `(ask, ask)` and `(release, release)`) with the guards `(not i in id, true)` (resp. `(i in id, gauge < size)`). The remark above shows that one of these guards is true thus a successor state exists also in this case. Therefore the global system is not blocking if `GIVEN` divides `QUOTA`.

Resource availability is an important property in such a system, generally it is a mix of safety and liveness properties. However, as stated in [40], availability properties with bounded waiting time are pure safety properties. Thus, they can be checked on bounded decompositions. Assuming that each action has a maximum duration, we are interested in the longest logical time sequence between a client request (`ask`) and its end (`end`). The longest sequence has form: `ask ; acquire`$^p$ `; delete ; acquire`$^r$ `; new ; ok`, where $p + r$ = (`QUOTA` % `GIVEN`). Therefore the global system satisfies this property. However one may

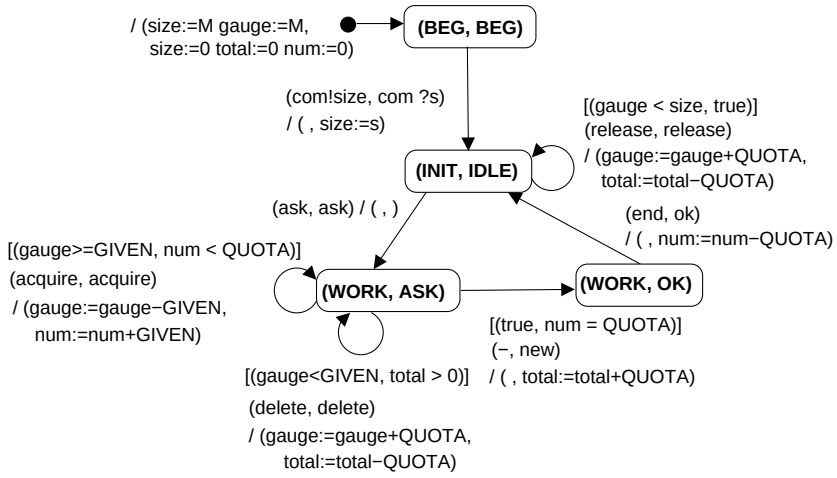


**Fig. 8.** The system decomposition view for {`size, gauge`} and {`size, num, total`}

expect to prove that: *for any client the longest sequence is* `ask ; acquire`$^p$ `; delete ; acquire`$^r$ `; new ; ok`. This is true but actually it requires an additional analysis observing that the client system freezes the client identity during the allocation and the system is not blocking.

This shows that bounded analysis and decomposition are useful, but often it must be connected with classic model-checking or other proof techniques.

## 5 Related Work

Standard model-checking techniques usually bounds all the sources of infinity. Similarly, bounded model checking [8] searches counterexamples in executions bounded by some length $k$. Therefore, let us focus on abstraction techniques and approaches dedicated to the verification of STSs or parameterized systems. Finally, we present some results on boundedness checking and counter automata.

CADP [24] is a rich toolbox made up of many state-of-the-art verification tools working on state space descriptions (LTS for short). As regards the verification of LOTOS specifications using CADP, LTS are computed flattening symbolic values appearing into the input abstract LOTOS specifications, a mechanism know as enumerative generation exploration. Concrete values are computed for each variable iterating on the domain of its type in case of free variables. As far as rich data types are taken into account, CADP faces the state explosion problem imposing some restrictions, such as bounds to avoid infinite enumerative generation. This is the main limitation of model-checking. However, existing algorithms and verification techniques (*e.g.*,, compositional verification) as encoded in CADP are quite efficient in many cases, especially when dealing with asynchronous systems and their underlying interleaving semantics. Hence, the combination of our STS analyses with CADP could yield interesting results.

Several works used abstraction techniques to verify state-based systems [15, 34, 17, 7]. For instance, in [15], the authors show how to extract abstract finite state machines from finite state programs using techniques similar to abstract interpretation. Our notion of abstraction or simulation is close to this work but our starting point is a state and transition based description of a program. In addition, our goal is to check if a bounded approximation may be built from it. Note that Proposition 2.2 in conjunction with a boundedness decision procedure gives an automatic way to approximate an infinite system. This is also true in case of counter STS with the notion of decomposition introduced in Definition 7. [15] proposes several abstraction mappings which are chosen by the specifier. In this work and its extensions the user selects the appropriate abstractions, while our bounded analysis automatically built an abstraction mapping. Let us stress that while most authors try to define abstractions over LTS (obtained from low level specification or code) and then address usual verification techniques on these abstracted LTS, we address the use of verification in the design phase. Components are specified directly with abstract LTS (our STS). Then we try to partially unfold them to use usual verification techniques.

Many approaches have been proposed for symbolic model-checking of various kinds of infinite state systems, such as [9, 22, 19, 18, 23, 10]. For example, a formalism similar to our symbolic system is described in [19, 18]. The authors define a general and concurrent system with a translation preserving-semantics into Constraint Logic Programming. They also present a method for verifying safety properties which is relevant to infinite state systems. While the formalism is different, our data types with positive conditional axioms is known to be equivalent to constraints written as Horn clauses. Compared to this work, our approach is slightly different since rather than replacing model-checking approaches we propose to complement them for some specific systems (decomposable and bounded).

The last point we want to discuss here are results about boundedness checking in dynamic systems with data types. Checking boundedness or reachability in such systems is an important area of research. One specific case is boundedness of asynchronous communicating systems where the data type is a FIFO buffer. The problem is not decidable [12] but some partial procedures do exist [29, 32, 35]. In [32], the authors propose an efficient way to check structural boundedness using linear programming. This test is useful but may compute inconclusive counterexamples in case of unboundedness. Hence, they extended their approach in [33] to determine spuriousness of counterexample and to exclude such situations. Other works focus on general decision procedures for some classes of dynamic systems. [35] gives a first result on simple counter systems, only $> 0$ guards and $+1$, $+0$, and -1 as actions are possible. The boundedness problem is also solvable for several Petri net extensions, especially double nets and transfer nets as proved in [20]. These works lead to future extensions of our boundedness decision procedure.

## 6 Concluding Remarks

Behavioural interfaces are required in component based software engineering to perform analysis and relate efficiently models and implementations. Most proposals in this area deal with LTS models. However, more expressive models such as STSs are needed to take data encapsulation and value passing into account. A major weakness of such models is the lack of dedicated analysis techniques. Direct mapping into standard model-checkers yield state explosion problems in presence of unbounded data types and hence is not directly applicable.

In this paper we proposed an analysis framework for STS based on configuration graphs and LTS interpretations. This enables one to use the usual verification techniques on these LTSs. In addition, we also experimented specific analysis techniques, namely STS symbolic analysis and bounded decomposition, and we demonstrated how they complement model-checking. We have developed a prototype in Python (about 4000 lines) which supports STS description, configuration graph computation, product computation and the analysis techniques presented before.

Future work aims at extending our techniques on boundedness checking. For instance, the selection of counter variables guiding the decomposition should be assisted by slicing techniques [13]. They can be applied to focus on a property one wants to check (which depends on variables), and then obtain the set of variables with a direct effect on this formula. Another perspective is to link our prototype with the verification tools CADP or SPIN.

## References

1. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.
2. A. Arnold. *Finite Transition Systems*. International Series in Computer Science. Prentice-Hall, 1994.
3. E. Astesiano, B. Krieg-Brückner, and H.-J. Kreowski, editors. *Algebraic Foundation of Systems Specification*. IFIP State-of-the-Art Reports. Springer Verlag, 1999.
4. C. Attiogbé, P. Poizat, and G. Salaün. Integration of Formal Datatypes within State Diagrams. In *Proc. of FASE'03*, volume 2621 of *LNCS*, pages 344–355. Springer-Verlag, 2003.
5. Sébastien Bardin, Alain Finkel, and Jérôme Leroux. FASTer acceleration of counter automata in practice. In Kurt Jensen and Andreas Podelski, editors, *Proceedings of the 10th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 576–590, Barcelona, Spain, March 2004. Springer.
6. T. Barros, L. Henrio, and E. Madelaine. Behavioural Models for Hierarchical Components. In *Proc. of SPIN'05*, volume 3639 of *LNCS*, pages 154–168. Springer-Verlag, 2005.
7. S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In Alan J. Hu and Moshe Y. Vardi, editors, *Proceedings of CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, pages 319–331, Vancouver, Canada, June 1998. Springer-Verlag.

8. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proc. of TACAS'99*, volume 1579 of *LNCS*, pages 193–207. Springer-Verlag, 1999.
9. B. Boigelot and R. Wolper. Symbolic Verification with Periodic Sets. In *Proc. of CAV'94*, volume 818 of *LNCS*, pages 118–131. Springer-Verlag, 1994.
10. A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract Regular Model Checking. In *Proceedings of CAV'04*, volume 3114 of *LNCS*, pages 372–386. Springer-Verlag, 2004.
11. A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *Journal of Systems and Software*, 74(1), 2005.
12. D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *Journal of the ACM*, 30(2):323–342, 1983.
13. I. Brückner and H. Wehrheim. Slicing an Integrated Formal Method for Verification. In *Proc. of ICFEM'05*, volume 3785 of *LNCS*, pages 360–374. Springer-Verlag, 2005.
14. M. Calder, S. Maharaj, and C. Shankland. A Modal Logic for Full LOTOS Based on Symbolic Transition Systems. *The Computer Journal*, 45(1):55–61, 2002.
15. E. M. Clarke, O. Grumberg, and D. E. Long. Model-Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.
16. H. Comon and Y. Jurski. Multiple Counters Automata, Safety Analysis and Presburger Arithmetic. In *Proc. of CAV'98*, volume 1427 of *LNCS*, pages 268–279. Springer-Verlag, 1998.
17. D. Dams, R. Gerth, and O. Grumberg. Abstract Interpretation of Reactive Systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, 1997.
18. G. Delzanno. An Overview of MSR(C): A CLP-based Framework for the Symbolic Verification of Parameterized Concurrent Systems. In *Proc. of WFLP'02*, volume 76 of *ENTCS*. Elsevier, 2002.
19. G. Delzanno and A. Podelski. Model Checking in CLP. In *Proc. of TACAS'99*, volume 1579 of *LNCS*, pages 223–239. Springer Verlag, 1999.
20. C. Dufourd, A. Finkel, and P. Schnoebelen. Reset Nets between Decidability and Undecidability. In *Proc. of ICALP'98*, volume 1443 of *LNCS*, pages 103–115. Springer-Verlag, 1998.
21. C. Dufourd, P. Jančar, and P. Schnoebelen. Boundedness of Reset P/T Nets. In *Proc. of ICALP'99*, volume 1644 of *LNCS*, pages 301–310. Springer-Verlag, 1999.
22. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient Algorithms for Model Checking Pushdown Systems. In *Proc. of CAV'00*, volume 1855 of *LNCS*, pages 232–247. Springer-Verlag, 2000.
23. A. Finkel and J. Leroux. How to Compose Presburger-Accelerations: Applications to Broadcast Protocols. In *Proc. of FSTTCS'02*, volume 2556 of *LNCS*, pages 145–156. Springer-Verlag, 2002.
24. H. Garavel, F. Lang, and R. Mateescu. An Overview of CADP 2001. *EASST Newsletter*, 4:13–24, 2001.
25. G. J. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.
26. A. Ingolfsdottir and H. Lin. *A Symbolic Approach to Value-passing Processes*, chapter Handbook of Process Algebra. Elsevier, 2001.
27. ISO/IEC. LOTOS: A Formal Description Technique based on the Temporal Ordering of Observational Behaviour. ISO/IEC 8807, International Organization for Standardization, 1989.

28. B. Jeannet, T. Jéron, V. Rusu, and E. Zinovieva. Symbolic Test Selection Based on Approximate Analysis. In *Proc. of TACAS'05*, volume 3440 of *LNCS*, pages 349–364. Springer Verlag, 2005.
29. T. Jéron and C. Jard. Testing for Unboundedness of FIFO Channels. *Theoretical Computer Science*, 113:93–117, 1993.
30. J. Kramer, J. Magee, and S. Uchitel. Software Architecture Modeling and Analysis: A Rigorous Approach. In *Proc. of SFM'03*, volume 2804 of *LNCS*, pages 44–51. Springer-Verlag, 2003.
31. S. Leue, R. Mayr, and W. Wei. A Scalable Incomplete Test for Message Buffer Overflow in Promela Models. In *Proc. of SPIN'04*, volume 2989 of *LNCS*, pages 216–233. Springer-Verlag, 2004.
32. S. Leue, R. Mayr, and W. Wei. A Scalable Incomplete Test for the Boundedness of UML RT Models. In *Proc. of TACAS'04*, volume 2988 of *LNCS*, pages 327–341. Springer-Verlag, 2004.
33. S. Leue and W. Wei. Counterexample-Based Refinement for a Boundedness Test for CFSM Languages. In *Proc. of SPIN'05*, volume 3639 of *LNCS*, pages 58–74. Springer-Verlag, 2005.
34. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property Preserving Abstractions for the Verification of Concurrent Systems. *Formal Methods in System Design*, 6(1):11–44, 1995.
35. O. Maréchal, P. Poizat, and J.-C. Royer. Checking Asynchronously Communicating Components Using Symbolic Transition Systems. In *Proc. of DOA'04*, volume 3291 of *LNCS*, pages 1502–1519. Springer Verlag, 2004.
36. R. Mateescu. A Generic On-the-Fly Solver for Alternation-Free Boolean Equation Systems. In *Proc. of TACAS'03*, volume 2619 of *LNCS*, pages 81–96. Springer Verlag, 2003.
37. Stephan Merz. TLA+ Case Study: a Resource Allocator. Technical report, LORIA, 2004.
38. S. Moschoyiannis, M. W. Shields, and P. J. Krause. Modelling Component Behaviour with Concurrent Automata. In *Proc. of FESCA'05*, 2005. To appear in *ENTCS*.
39. T. Murata. Petri Nets: Properties, Analysis, and Applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
40. F. B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.
41. D. M. Yellin and R. E. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.