Bounded Analysis and Decomposition for Behavioural Descriptions of Components

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@ibisc.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 interfaces are now accepted as a mandatory feature of components to address architectural analysis. Behavioural interface description languages should be able to deal with data types and with rich communication means. Symbolic Transition Systems (STS) support the definition of component models which take into account control, concurrency, communication and data types. However, verification of components described with protocol modelled by STS, especially model-checking, is difficult since they possibly involve different sources of infinity. In this paper, we propose the notions of *bounded analysis* and *bounded decomposition*. They can be used to test boundedness of systems and to generate finite simulations for them so that standard model-checking techniques may be applied 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, detecting 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, 8, 20] or automata-based formalisms [4, 24]. In the context of a national project, ACI DISPO, our researches are interested in checking components and resources or services availability.

Components may exchange data with service requests, or may internally compute data values on which behaviours depend, yielding compositions which

* 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 component availability.

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. Unfortunately, this is known to yield state explosion problems when verifying models, especially with model-checking. Research on Symbolic Transition Systems (STS) [10, 18, 19] aims at providing a model and dedicated verification techniques to deal with these issues.

In this paper we develop a model of communicating components based on STS, together with specific analysis techniques. First, we formalise our notion of communicating and concurrent STS, with a proper semantics based on configuration graphs. We also link our STS with LTS using interpretations and we state properties relating interpretations and STS composition. This provides a unified framework where STS and LTS can be both defined and composed. Second, we present a decidable boundedness procedure (*bounded analysis*) which tests the boundedness of communicating components architectures (called systems). Model-checking techniques can be used thereafter to prove properties. It is common that a system handles both bounded variables and unbounded variables. Enumerative model-checking will arbitrarily bound all the variables. Whenever the bounds set by model-checking tools are reached, the specifier does not know if the system is either too big for the tool or really unbounded. For instance, a system which deadlocks for every n smaller than 10, does not imply anything about the behaviour for greater values of n. Bounded analysis may therefore be viewed as a complementary debugging means to detect possible flaws that model-checking may miss in the presence of data types. Next, we develop a decomposition technique (*bounded decomposition*) that is used to split systems into parts which can be separately tested for boundedness and if so, checked separately. This approach may not solve every problem related to infinite data types, but it is especially worthy with the (numerous) systems involving bounded resources (*i.e.,* where parts associated to the resources are bounded) and with systems where the number of components is bounded.

The paper is organised as follows. Section 2 formally defines STS, configuration graphs, relations between STS and LTS, and communication between STS. Sections 3 and 4 present, respectively, boundedness analysis and bounded decomposition and illustrate them on examples. Section 5 reviews related work. Finally, Section 6 draws up some concluding remarks. More details about our approach and formal definitions can be consulted in [26].

2 Formalising Components as Symbolic Transition Systems

This section states some definitions we use thereafter to introduce our approach. First of all, we consider algebraic specifications as an abstraction of concrete implementation languages like Java, C++, or Python. A *signature* (or static interface) Σ 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. X is used to denote

the set of all variables, it contains a distinguished variable, $Self_D$, whose goal is much like explicit receivers in Object-Oriented languages (*e.g.,* `this` in Java). From a signature Σ and from X, one may obtain *terms*, denoted by $T_{\Sigma,X}$. An *algebraic specification* is a pair (Σ, Ax) where Ax is a set of axioms between terms of $T_{\Sigma,X}$. Let r be a ground term, $r\downarrow$ denotes the normal form or normalization (assumed to be unique) of r. $v : R$ means that v has type R and $v(u)$ denotes the application of v to term u.

2.1 Symbolic Transition Systems

Symbolic Transition Systems [10, 18, 19] 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.

Definition 1 (STS). *An STS is a tuple* $(D,\ (\Sigma, Ax),\ S,\ L,\ s^0,\ T)$ *where:* (Σ, Ax) *is an algebraic specification,* D *is a sort called* sort of interest *defined in* (Σ, 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.*

Note that *countable* means that the set may be infinite but can be enumerated. *Events* denote atomic activities that occur in the components. Events are either: *i*) hidden (or internal) events: τ, *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 $x \in X \backslash \{Self_D\}$. Internal events denote internal actions of the components which may have an effect on its behaviour yet without being observable from its context. Silent events are pure synchronising events, while emissions and receptions naturally correspond, respectively, to requested and provided services of the components. 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, μ the guard, ϵ the event and δ the action. Each action is denoted by a term with variables where at least $Self_D$ occurs. A do-nothing action is simply denoted by $Self_D$. In forthcoming figures, transitions will be labelled as follows: `[`μ`]` ϵ `/` δ.

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

Pairs (s,v) are *configurations* where s is the *control state*. Let d be an STS. Its unfolding in a v^0 term, $G(d,v^0)$, 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 do-nothing actions denoted by $Self_D$.

2.3 Interpretations

Configuration graphs and STS can be interpreted as LTS[4]. Such mappings enable one to use existing model-checkers, such as SPIN [17] or CADP [16], to verify these models. We introduce two LTS interpretations based on the following rules:

- $(rule_1)$ any STS transition $(x,\ \mu,\ \epsilon,\ \delta,\ y)$ is reduced to an LTS transition $(x,\ l,\ y)$, where l is the label of the event ϵ;
- $(rule_2)$ 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* $rule_1$ *and discarding* D *and* (Σ, Ax)*. The weak interpretation,* W_{LTS}*, of an STS, is an LTS computed with* $rule_2$ *and discarding* D *and* (Σ, Ax)*.*

We use $\supseteq$ for the transition relation inclusion and $\sqsupseteq$ for 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 $\mathcal{R}$ included in $S_A \times S_B$ 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 then $\exists s_B \in S_B$ such that $s_A \mathcal{R} s_B$ and s_B is initial, and iii) $\forall (s_A,\ l,\ t_A) \in T_A, \forall s_B \in S_B,\ s_A \mathcal{R} s_B \Rightarrow (\exists t_B \in S_B,\ \exists (s_B,\ l,\ t_B) \in T_B \wedge t_A \mathcal{R} t_B)$.

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

1. $W_{LTS}(d) \supseteq W_{LTS}(G(d,v^0))$,
2. $I_{LTS}(d) \succeq I_{LTS}(G(d,v^0))$,
3. $I_{LTS}(d) \sqsupseteq I_{LTS}(G(d,v^0))$.

Point 2 above defines a simulation which in turn implies trace inclusion (point 3). Previous works [22, 12] have shown that simulation preserves a subset of μ-calculus, namely safety properties. The above relations could be later extended to other existing abstractions, such as [11, 22, 12, 5].

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

2.4 Concurrency and Communication

Concurrent communicating components can be described with protocols modelled by STS, and synchronous products adapted from the LTS related definition [2] can be used to obtain the resulting global system. 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 the 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 to be non-synchronised or *asynchronous*. Corresponding transitions are triggered independently and have independent running steps. The formal definition of the synchronous product of STS can be found in [26]. The synchronous product operator is noted $\otimes_V$ and is extended to a n-ary product and to any depth.

The configuration graph and the standard interpretation have compatibility properties with the synchronous product, which are formalised below.

Proposition 2. *Let d_1 and d_2 be two STS, V a synchronisation vector, $v_1 \in T_{\Sigma_{D_1}}$ 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. $I_{LTS}(d_1 \otimes_V d_2) \succeq I_{LTS}(G(d_1, v_1) \otimes_V d_2) \succeq I_{LTS}(G(d_1 \otimes_V d_2, (v_1, v_2)))$.

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 $I_{LTS}(d_1 \otimes_V d_2)$. These results are used in Section 4 to apply the standard interpretation to composite systems.

3 Bounded Analysis

Enumerative model-checking works on state spaces which are generated from specifications written in high-level languages such as process algebras. Symbolic model-checking techniques rely on different techniques (such as 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 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, in presence of concurrency, to avoid the generation of the whole global system [23]. However, a shortcoming of all these approaches is that model-checking is applied to a restricted finite state system and full correctness cannot be ensured. Accordingly we present in the sequel of this section an approach preserving symbolic values. Our objective is not to replace existing

model-checking techniques and tools. Bounded analysis has to be viewed as a complementary debugging means to detect possible flaws that model-checking may miss.

3.1 Principle

It is common that a system handles both bounded variables and unbounded variables. Enumerative model-checking will arbitrarily bound all the variables and it may be insufficient to assert properly a given property for the whole system. Bounded analysis begins by checking if a system is bounded, *i.e.,* testing if its configuration graph is finite or not. The system taken into account may be either made up of a single component or several communicating components. Whenever the system is bounded, bounds for variables can be computed or at least estimated (see [21] for example) and the configuration graph may thereafter be generated. Boundedness checking mainly traverses the system configuration graph to seek accumulating cycles, *i.e.,* a cycle of control states with a greater data value at the end. When bounds are known, the generation of the bounded system can be tuned such that the entire system can be computed. We experimented the previous case with CADP, see [26] for an example. If the system is not bounded, verification techniques developed for infinite systems are relevant, see [5, 14, 13, 3, 7] for example. In the sequel we describe another way to abstract infinite component systems using boundedness analysis. This approach is particularly relevant on component systems, because of Proposition 2. This proposition states that a composite system may be abstracted by checking the boundedness of one component and if bounded, the synchronous product of the bounded configuration graph with other components is computed. This approach is illustrated in the next subsection and extended in Section 4 with a notion of decomposition.

Definition 4 (Bounded STS). *An STS is bounded, for an initial value v_0, iff its configuration graph is finite.*

Checking boundedness is a semi-decidable problem and a semi-algorithm computing the configuration graph has been implemented in our prototype. This algorithm completely unfolds the system and merges identical configurations. However boundedness is decidable for some specific classes of STS. Our prototype therefore implements a decision procedure for one of them, *counter STS*, which are an adequate abstraction for many systems, see [15, 3] for related definitions. They are particularly convenient in the context of component availability properties since counter STS can describe dynamic systems allocating finite amounts of resources.

Definition 5 (Counter STS). *A counter STS, is an STS where: i) the data type is restricted to natural numbers (counters) c_i $1 \leq i \leq m$, 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, and iii) actions are $c_i := \Sigma_{j=1}^{m} a_j * c_j \pm p_i$, where a_j, p_i are natural numbers and at least one a_j is greater than 0.*

Counter STS are as powerful as generalized transfer nets [15] which extend Petri nets with both duplication and transfer arcs. They admit neither reseting nor equality testing, since their boundedness would then not be decidable.

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

The principle of the procedure is to find an accumulating cycle in the configuration graph. The proof relies on the following fact: the effect of all the transitions may be viewed as an affine increasing function on the vector of counters. This defines a well-structured transition system and boundedness is therefore decidable following a general theorem for them [15].

3.2 Application: a Resource Allocator (V1)

This subsection describes the application of bounded analysis to an infinite system. Whenever the bounds set by model-checking tools are reached, the specifier does not know if the system is either too big for the tool or really unbounded. For instance a system which deadlocks for every n smaller than 10, does not imply anything about the behaviour for greater values of n. 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 the bounded STS, the product with the other STS, then the LTS interpretation of the finite resulting system. We recall with reference to Proposition 2.2 that it is a finer interpretation than simply computing the product and interpreting it afterwards.

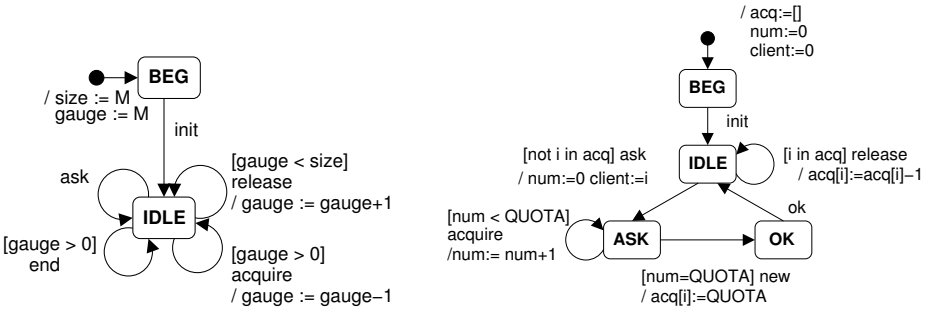

Fig. 1. Resource allocator system (left: allocator, right: client system)

As an illustration, let us take a resource allocator system with two components: the allocator and the client system. Figure 1 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 `M`. Variable `gauge` is used to keep track of the allocated resources. On the other hand, one *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`) and related to client `i`, accept a resource unit (`acquire`), release a resource 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 (with `QUOTA`$\leq$`M`). Variable `num` stores the current number of resources while acquiring them, and the `acq` vector stores the acquired resources for the clients. Synchronisations are `(init,init)`, `(release,release)`, `(acquire,acquire)`, `(ask,ask)` and `(end,ok)`.

Under the hypotheses of a given `M` and a given `QUOTA`, the system is not finite since the number of clients is not known. Model-checking techniques must set this number, and hence find a deadlock. Indeed, after a while, resources will lack since resources acquired by the clients 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 (`M` is a constant and `gauge`$\leq$`M` is a global constraint). In this example the weak interpretation of the STS ($W_{LTS}(allocator \otimes_V client\ system)$) 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 ($G(allocator, M) \otimes_V client\ system$) 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 and then build an abstraction. 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 6 (Decomposable STS). *An STS* $(D,\ (\Sigma, Ax),\ S,\ L,\ s^0,\ T)$ *is decomposable if and only if: i)* D *can be decomposed into* $D_1 \times D_2$*, ii) 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*

μ_i a guard for D_i, iii) 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 δ_i a function on D_i.

When d is decomposable we may define two successive partial unfoldings, 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, however the principle extends to other kinds of events. 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.

Proposition 4. *Let d be a decomposable STS. The configuration graph G of d can be 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 7 (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 preserve 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 decompositions. Propositions 1.2 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 $G(d, (v_1^0, v_2^0))$.

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

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 [13]. 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 their critical section. This counter is used to check the mutual exclusion property. STS associated to process and server are described in Figure 2. Synchronisations are summarized in the following vectors: `(think,givet)`, `(use,gives)`, and `(end,end)`.

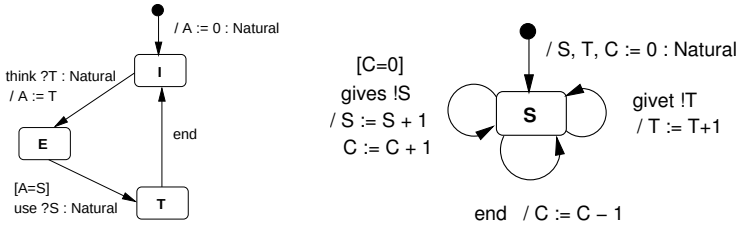

Fig. 2. STS descriptions: process (left) and server (right)

This system is unbounded since variables `S, T`, and `A` can store arbitrary large values. We split the variables into {} and {`A`} for the process, and {`C`} and {`T,S`} for the server. With these subsets we can easily check the decomposition of definition 6. Then, this decomposition produces a partial configuration graph on the `C` counter, on which boundedness is checked.

From such a finite system, safety properties like mutual exclusion can be checked. Mutual exclusion appears as the absence of the situation with more than one process in state `T` or as the fact that `C`$\leq$`1`. Our prototype succeeds in generating the global system, checking the boundedness, computing the configuration graph and then checking mutual exclusion for up to 8 processes within about three minutes. The resulting product (for 8 processes and the server) is made up of 6561 states and 52488 transitions; the configuration graph contains 1280 states and 6656 transitions. However CADP and SPIN, with the default configuration values and bounded data types, *e.g.,* natural numbers bounded to 256, do not pass 6 processes.

4.3 Application: a Resource Allocator (V2)

This section illustrates the use of bounded decomposition on a more elaborated variant (Fig. 3) 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 `M`$\geq$`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

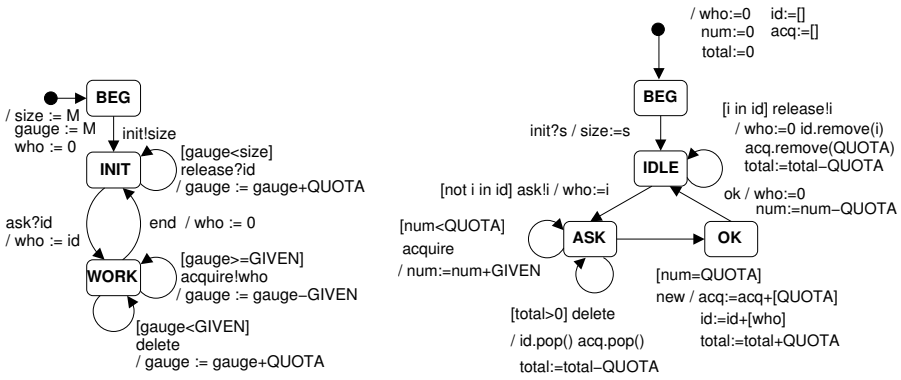

Fig. 3. Revisited resource allocator system (left: allocator, right: client system)

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 bounded. 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. Regarding the client, its decomposition is based on a partition between variables {`size, num, total`} and variables {`who, acq, id`}. Figure 4 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 are hidden in the decomposition.

Fixing values for `M`, `QUOTA`, and `GIVEN`, the boundedness is checked to be true for this decomposition. We have carried out experiments on the system for various values of `size`, `QUOTA`, and `GIVEN`. As an example with `M`=1000, `QUOTA`=2, and `GIVEN`=1, a configuration graph of 2503 states and 3004 transitions is built. Experiments show that if `GIVEN` does not divide `QUOTA` the system deadlocks. In the state `(WORD, ASK)` only three transitions are possible: `(acquire, acquire)`, `(delete, delete)`, and `(-, new)` (see Fig. 4). Since `GIVEN` does not divide `QUOTA`, the condition `num`>`QUOTA` will be eventually true and `num`=`QUOTA` will never be 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 an example of a safety property we have checked on the bounded decomposition.

On the other hand, if `GIVEN` divides `QUOTA` then the bounded decomposition has no deadlock. However, this fact is not sufficient to ensure that the global system is deadlock free. A thorough look at the bounded decomposition shows

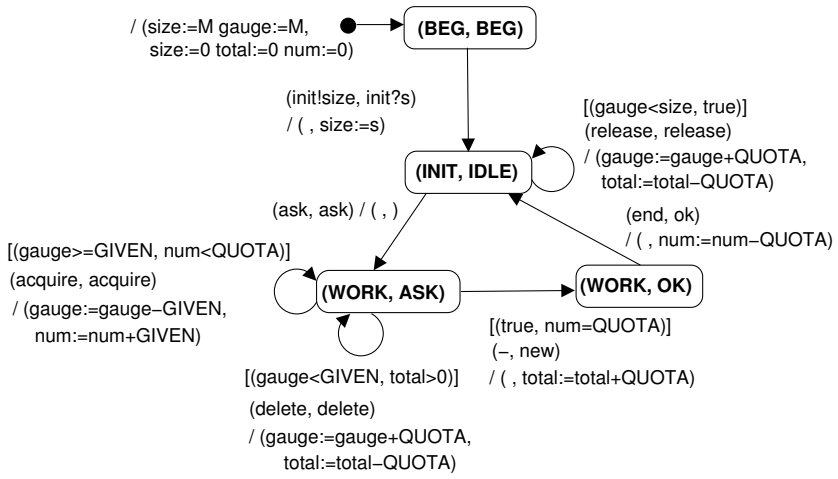

Fig. 4. The system decomposition view for {`size, gauge`} and {`size, num, total`}

that the guards left to evaluate in the G_2 step are `[(gauge<size, i in id)]` and `[(true, not i in id)]`. At least one of these guards is true since either `[not i in id]` or `[i in id]` is true and an allocation has been done thus `gauge<size` is true. 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 [27], availability properties with bounded waiting time are pure safety properties. Thus, bounded decompositions can be applied to check them. Assuming that each action has a maximum duration, we may be 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 the longest sequence property. However one may 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 as discussed above.

5 Related Work

Our model of concurrent components can be related to Architectural Description Languages (ADL). It allows one to describe behavioural interfaces of both atomic and composite components. In addition, components can handle data types within their protocols, and communicate synchronizing on messages. However, our focus in this paper is not to provide a new ADL but to tackle analysis

issues. [25] presents a formal ADL for which analysis is possible using techniques presented here.

Enumerative model-checking techniques usually bound all the sources of infinity. Similarly, bounded model-checking [6] searches for counterexamples in executions bounded by some length k. Therefore, let us focus on abstraction techniques and approaches dedicated to the verification of STS or parameterized systems.

Several works use abstraction techniques to verify state-based systems [11, 22, 12, 5]. For instance, in [11], the authors show how to extract abstract finite state machines from finite state programs using techniques similar to abstract interpretation. Our notions of abstraction and simulation are 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 can be built from it. Note that Proposition 2.2 in conjunction with a boundedness procedure gives an automatic way to approximate an infinite system. 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 focus on the use of verification in the design phase and our bounded decomposition automatically builds an abstraction mapping. Components are specified directly with STS, then we try to unfold them partially to use usual verification techniques.

Many approaches have been proposed for symbolic model-checking of various kinds of infinite state systems, such as [14, 13, 3, 7]. A formalism similar to our symbolic system is described in [13]. 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 are known to be equivalent to constraints written as Horn clauses. Compared to this work, our objectives are slightly different since rather than replacing model-checking approaches we propose to complement them for some specific systems (decomposable and bounded). We also emphasize [3] which computes reachability sets of counter automata. These sets, defined by Presburger formulae, are represented by automata and the authors propose an algorithm to increase convergence computation. Boundedness is equivalent to the property of finite reachability set. A counter automata is a counter STS allowing $c \leq M$ guards and this provides a general semi-algorithm for reachability.

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 STS 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 results in state explosion problems in the 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 LTS. In addition, we have also presented specific analysis techniques, namely bounded analysis and bounded decomposition, and demonstrated how they may complement model-checking. We have developed a prototype in Python (about 4000 lines) which supports STS description, configuration graph computation, product computation and the boundedness checking. 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.

Future work aims at extending our techniques on boundedness checking and boundedness decomposition. For instance, the selection of counter variables guiding the decomposition should be assisted by slicing techniques [9]. 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.

Acknowledgments. We would like to thank the reviewers for their useful comments and suggestions.

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. S. Bardin, A. Finkel, and J. Leroux. FASTer Acceleration of Counter Automata in Practice. In *Proc. of TACAS'04*, volume 2988 of *LNCS*, pages 576–590. Springer, 2004.
4. 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.
5. S. Bensalem, Y. Lakhnech, and S. Owre. Computing Abstractions of Infinite State Systems Compositionally and Automatically. In *Proc. of CAV '98*, volume 1427 of *LNCS*, pages 319–331. Springer-Verlag, 1998.
6. 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.
7. A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract Regular Model Checking. In *Proc. of CAV'04*, volume 3114 of *LNCS*, pages 372–386. Springer-Verlag, 2004.
8. A. Bracciali, A. Brogi, and C. Canal. A Formal Approach to Component Adaptation. *Journal of Systems and Software*, 74(1), 2005.
9. 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.

10. 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.
11. 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.
12. D. Dams, R. Gerth, and O. Grumberg. Abstract Interpretation of Reactive Systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, 1997.
13. 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.
14. 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.
15. A. Finkel, P. McKenzie, and C. Picaronny. A Well-Structured Framework for Analysing Petri Nets Extensions. *Information and Computation*, 195(1-2):1–29, 2004.
16. H. Garavel, F. Lang, and R. Mateescu. An Overview of CADP 2001. *EASST Newsletter*, 4:13–24, 2001.
17. G. J. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.
18. A. Ingólfsdóttir and H. Lin. *A Symbolic Approach to Value-passing Processes*, chapter 7 of Handbook of Process Algebra. Elsevier, 2001.
19. 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.
20. 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.
21. 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.
22. 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.
23. 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.
24. S. Moschoyiannis, M. W. Shields, and P. J. Krause. Modelling Component Behaviour with Concurrent Automata. In *Proc. of FESCA'05*, volume 114(3) of *ENTCS*, pages 199–220, 2005.
25. P. Poizat and J.-C. Royer. Korrigan: a Formal ADL with Full Data Types and a Temporal Glue. Technical Report 88-2003, LaMI, CNRS et Université d'Evry Val d'Essonne, September 2003.
26. P. Poizat, J.-C. Royer, and G. Salaün. Symbolic Bounded Analysis for Component Behavioural Protocols. Technical report, Écoles des Mines de Nantes, 2005. Available at `http://www.emn.fr/x-info/jroyer/rrBounded.pdf`.
27. F. B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.