Checking Asynchronously Communicating Components using Symbolic Transition Systems

Olivier Maréchal[1] and Pascal Poizat[2] and Jean-Claude Royer[1]

[1] OBASCO Group, École des Mines de Nantes - INRIA, 4, rue Alfred Kastler - BP 20722, F-44307 Nantes Cedex 3
(Olivier.Marechal,Jean-Claude.Royer)@emn.fr
[2] LaMI, UMR 8042 CNRS - Université d'Évry Val d'Essonne, Genopole - Tour Évry 2, 523 place des terrasses de l'Agora, F-91000 Évry Cedex
Pascal.Poizat@lami.univ-evry.fr

Abstract. Explicit behavioural interface description languages (BIDLs, protocols) are now recognized as a mandatory feature of component languages in order to address component reuse, coordination, adaptation and verification issues. Such protocol languages often deal with synchronous communication. However, in the context of distributed systems, components communicating asynchronously through mailboxes are much more relevant. In this paper, we advocate for the use of Symbolic Transition Systems as a protocol language which may deal also with this kind of communication. We then present how this generic formalism, specialized with different mailbox protocols, may be used to address verification issues related to the component mailboxes.

1 Introduction

Component Based Software Engineering (CBSE) has reached its maturity: a large number of different component based proposals deals with the different aspects of CBSE systems to be taken into account. A distinction can be made between component platforms (*e.g.* EJB, COM, [21]), components languages (Fractal, ArchJava [11, 1, 4]) and Architectural Description Languages (Wright, Rapide, [5, 32, 33]) dedicated to component architectures design. In both cases, components are described in a first glance using Interface Description Languages (IDL) which provide means to have them interact. These IDL are mainly signature-based ones: required/provided services, methods or ports. However, checking that architectures contain only components bound using compatible signature-based interfaces may not prevent them from deadlocking if for example the *protocols* of these components are not compatible. Such verifications can only be achieved using behavioural IDL (BIDL). The need for BIDL has also been recognized for better component reuse, coordination and adaptation.

Yet, few proposals take protocols into account as a public part associated to interfaces. SOFA [30] defines a component language with regular expressions (patterns) used to describe these protocols, but CBSE proposals taking BIDL into account are mainly design oriented ones (see [36] for recent references and for a description of this field of research in the WebServices framework). The usual means to describe BIDL in these proposals is using directly Labelled Transition Systems (LTS) or deriving them from abstract descriptions (patterns, process algebras or temporal logics). Transitions systems have a major

drawback: the absence of control over the state and transition explosion problem. It can be solved using extended state machines of different kinds (Symbolic Transitions Systems [28, 14, 16] developed to give (finite) semantics to value-passing process algebras, Statecharts [25], extended state machines [2, 15], AltaRica [7]). Their common features are symbolic states, value-passing events, the use of data types and guarded transitions.

We have been investigating the use of Symbolic Transition Systems (STS) in formal specifications with the Graphic Abstract data Type (GAT) approach [38, 39] and the KORRIGAN model [18, 16, 17, 3] for a few years now. We are currently developing a comprehensive component approach based on these STS which should take into account: means to model components and architectures, mechanisms to describe explicitly protocols in component languages and verification procedures.

In this paper we focus on the BIDL aspect of component models. Protocol languages for BIDL usually take only synchronous communication into account. However, in the context of distributed systems, components communicating through mailboxes are much more relevant. We present a generic form of STS and how asynchronous communication can be taken into account using them. We show that our STS can be specialized using different mailboxes protocols. We then give some theoretical results on the issue of verifying mailboxes of asynchronous communicating components modelled with STS, together with concrete algorithms dedicated to this verification. Mailbox analysis is a valuable step in the design of components and architectures as mailbox boundedness may for example lead to automatic simplifications in the protocol and component descriptions. Our algorithms are a proposal to overcome the actual lack for adequate tools related to this analysis.

The paper is organised as follows. Section 2 presents our STS and some associated definitions. We also compare our STS with related languages. Section 3 gives some results on the computability of mailbox contents. We define a general algorithm for checking dico boundedness which we prove to be a sufficient condition for fifo boundedness. Section 4 then presents, on a shared resource controller example, how mailbox analysis may help in the design of a correct system. Section 5 extends the application of our techniques to automatic simplification of specifications and to the analysis of some non asynchronous examples. Finally a conclusion finishes this presentation.

2 Symbolic Transition Systems

Symbolic Transition Systems (STS) [28, 14] 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 STS are a generalisation of these, associating a symbolic state and transition system with a data type description [38, 16, 10]. This data type description may be given either using algebraic specifications [38, 16], model oriented specifications [10] or even Java classes as in a library for STS we are developing.

In the context of this work, we take into consideration abstract descriptions (of mailbox protocols) given using algebraic specifications. We will here give only the necessary insight into these specifications, see [9] for more details. A *signature* (or static interface) Σ is a couple $(\mathcal{S}, F)$ 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 Σ_{R} denotes the subset of functions from Σ with result sort being R. X is used to denote the set of variables. From a signature Σ and from X, one may obtain *terms*, denoted by $T_{\Sigma,\mathrm{X}}$. The set of *closed terms* (also called ground terms) is the subset of $T_{\Sigma,\mathrm{X}}$ without variables, denoted by T_{Σ}. As usual, $\mathcal{P}()$ is used to denote the powerset of a set, and $\vec{x}$ denotes a vector. An *algebraic specification* is a pair (Σ, Ax) where Ax is a set of axioms between terms of $T_{\Sigma,\mathrm{X}}$. Models of specifications are Σ-algebras. They relate each sort s in Σ to a set of values denoted by $s^\sharp$ (interpretation of s), and each function name f in Σ to a function over these values denoted by $f^\sharp$ (interpretation of f). We are only interested in *reachable* algebras, that is those where each value can be obtained from (at least) a ground term in T_{Σ}. It is assumed that the data type provides a computable equality function to check whether two ground terms represent the same value.

We may then give a syntactic definition of our STS.

Definition 1 (STS). *A Symbolic Transition System (STS) is a tuple* $< D,\ (\Sigma,\ Ax),\ S,\ s_0,\ v_0,\ T >$ *where:*

- (Σ, Ax) *is an algebraic specification,*
- D *is a sort called* sort of interest *defined*[3] *in* (Σ, Ax)*,*
- $S = \{s_i,\ 0 \leq i \leq N\}$ *is a finite set of states,*
- $s_0 \in S$ *is the initial state,*
- $v_0 \in D$ *is the initial value,*
- $T \subseteq S \times \Sigma_{\mathrm{Boolean}} \times \Sigma_{\mathrm{D}} \times \mathcal{P}(X) \times S$ *is a finite set of transitions,*
- Σ *contains a set* $\{P_{s_i}\}$ *of state predicates which are functions which associate a state in* S *to each value in* D*,*
- $\forall v, v' \in D^\sharp,\ v \approx_{D^\sharp} v' \mathrel{\widehat{=}} (\exists! s_i \in S,\ v \in s_i \wedge v' \in s_i)$*, is an equivalence relation.*

STS transitions are tuples $(s,\ \mu,\ \lambda,\ \vec{x},\ t)$ for which s is called the source state, t the target state, μ the guard, λ the label and $\vec{x}$ the variables. LTS labels are closed terms. Our STS consider labels as operation calls. Moreover, there are guards in transitions which denote boolean functions. Hence a STS is a LTS if and only if the λ in transitions are functions from D to D, all the $\vec{x}$ are empty, and all the μ guards are $true$.

The semantics of STS is given in terms of configurations and transitions between configurations.

Definition 2 (Configuration). *A* configuration *of an STS* $< D,\ (\Sigma,\ Ax),\ S,\ s_0,\ v_0,\ T >$ *is a pair* (s, v) *with* $s \in S$ *(control state), and* $v \in D^\sharp$ *(value).*

[3] More exactly, (Σ, Ax) is an *algebraic presentation* for sort *D* [9].

Equivalence between two configurations is defined as equality on control states and equality on values (this is why a computable equality function is needed in the algebraic specification part of the STS). The set of all possible configurations of an STS is denoted $G(STS)$. Let $\vec{x}$ be a vector of variables, $\sigma_{\vec{x}}$ denotes a substitution over this vector, $x\sigma$ with $x \in \vec{x}$ denotes the value of x in this substitution and $\vec{x}\ \sigma$ denotes the application of σ to the whole vector (that is a vector of values).

Definition 3 (Configuration Transition). *A (fireable) configuration transition for a configuration* $(s,\ v)$ *of a STS* $< D,\ (\Sigma,\ Ax),\ S,\ s_0,\ v_0,\ T >$ *is any tuple* $((s,\ v), \mu,\ \lambda,\ \sigma_{\vec{x}},\ (t,\ v'))$ *such that* $(s,\ \mu,\ \lambda,\ \vec{x},\ t) \in T$, $\mu^\sharp(v, \vec{x}\ \sigma) = true_{\mathrm{Boolean}}$, *and* $v' = \lambda^\sharp(v, \vec{x}\ \sigma)$.

The usual notions over LTS (*e.g.* traces, reachability tree and reachability graph) are naturally extended to STS using configuration transitions and taking $(s_0, v_0^\sharp)$ as initial configuration. The semantics associated to an STS is a configuration relation which is the set of the configuration transitions of the STS.

Computation of the reachable configuration graph. The computation of the *reachable configuration graph* of an STS is done by computing the configuration tree using configuration transitions and merging nodes labelled with equivalent configurations. This is the principle of the semi-algorithm bound which is a semi-algorithm since this computation process may not terminate. Termination is ensured if and only if the set of reachable configurations is finite. Proposition 1 below is a general form of proposition 1 in [29], a reachable value being any $D^\sharp$ value which appears in the configurations.

Proposition 1. *The reachable configuration graph of an STS is finite if and only if the set of reachable values is finite.*

This comes from the fact that the reachable configuration graph will be infinite if and only if there is an infinite branch of the configuration tree labelled by different configurations. Since the number of control state is finite (by Def. 1), the set of configurations is finite if and only if the set of reachable values is finite. This often makes the study of the set of values *boundedness* a mandatory task before doing other analyses. However, our STS model being generic and able to simulate a Turing machine, boundedness is not decidable.

Relation between our STS and concurrent communicating systems. To define the semantics of concurrent communicating systems we adapt to STS the synchronous product principle [8]. In [38] we suggest to compute a (global) structured STS, that is an STS with structured states and structured transitions which memorise the system structuring. It is important in a component perspective to be able to retrieve information about the subcomponent STS from a global STS. Nevertheless an interesting property of our STS it that it is always possible to transform a structured STS into a bisimilar flat STS with a more complex data type.

Relation between our STS and ADT. The main interest with our STS is that (i) they avoid state explosion problems, and (ii) they define equivalence classes (one per state) and hence strongly relate the behavioural and the algebraic representations of a data type. Whenever the STS complexity decreases, the associated data type and the guards complexity increases and *vice versa*. Up to 50 states and 50 transitions, state and transition complexity can be manually handled. Structuring means (such as Statecharts hierarchical or orthogonal states) and automatic tools helps to increase this limit, but then time and space computation becomes more prominent. One way to overcome this scalability problem is the following. Once a compound system has been defined, the global structured STS can be replaced by a more abstract and bisimilar one. Of course the associated data type would have then to be recomputed, but we can expect this to be achieved with the help of algorithms.

Partial evaluation. A more general form of the `bound` algorithm is sometimes useful. It carries the idea of a partial evaluation of the STS. This may be used to constrain mailboxes to given contents, to a given size, or to evaluate specific guards associated to communications. Let `eval` be a function from D to $Boolean$ which terminates. We define `partial` as the computation of graph of the reachable configurations which satisfy `eval`.

STS decomposition. It is sometimes possible to analyse an STS with a two-step process as for example studying first the communications and then the datatypes, or analyzing the communications of two subcomponents (in a global structured STS) separately. This needs the STS to be decomposable.

Definition 4. *A STS is decomposable if and only if*

- $D = D_1 \times D_2$,
- *Each* $\mu = \mu_1 \ \wedge \ \mu_2$ *where* μ_i *is a guard for* D_i
- *Each* $\lambda = \lambda_1 \ \times \lambda_2$ *where* λ_i *is a function on* D_i

In such a case the following diagram explains the decomposition of a configuration transition: $(s,(v_1,v_2)) \stackrel{\mu\ \lambda\ \vec{x}}{\longrightarrow} (s',(v_1',v_2'))$ is equivalent to the sequence $(s,(v_1,v_2)) \stackrel{\mu_1\ \lambda_1\ \vec{x}}{\longrightarrow} (s'',(v_1',v_2))$ and $(s'',(v_1',v_2)) \stackrel{\mu_2\ \lambda_2\ \vec{x}}{\longrightarrow} (s',(v_1',v_2'))$. This decomposition can be extended to $n > 2$.

Comparing STS with other behavioural languages. There are three main ways to specify component BIDL: state machines, process algebras and temporal logic. The three of them are interrelated since state machines are usual models for process algebras and temporal logics, and since process algebras equivalences and specific temporal logics may be proven adequate [12]. Note that pattern languages can be seen as some form of process algebra without parallel operators.

Both process algebras and state machines are executable formalisms. Temporal logics can of course be used to describe components at a very abstract (declarative) level but

are rather adapted to check properties against the component descriptions. When it comes to specify components with data, value-passing process algebras (such as LOTOS) or extended state machines (such as STS, models of value-passing process algebras) are useful to avoid the state and transition explosion problem.

We choose STS since we want both data-abstraction and readability. STS may be related to Statecharts [25] but they are simpler and stricter on the semantic side. Moreover, our STS enable one to define different formal datatypes for the D sort. We may therefore take very different extended state machines semantics into account (D can model substitutions to simulate the LOTOS STS [14] model but also different mailbox protocols, as presented in the sequel, to simulate different existing Communicating Finite State Machines (CFSM) [42]).

3 Algorithms for Mailbox Analysis

A long term goal of our work is to provide a powerful framework based on STS. It should integrate various state machines semantics, together with (adaptations of) known algorithms and verification techniques. This section demonstrates the ability of STS to handle asynchronous communications. The main difference between the synchronous communication semantics and the asynchronous one is the use of mailboxes (or queues) to memorise messages. In the general case this increases complexity. Deadlock checking for example becomes undecidable. We also lose finite waiting time. We consider here simple mailbox protocols (safe communication, no message loss or message duplication, see [15, 2] for related work on this). However, such extensions can be taken into account in our model thanks to specific data types.

STS easily captures mailbox management and the associated guards. We first introduce three kinds of operation labels:

- `>op`: denotes the *receipt* of an `op` message in the mailbox
- `op>`: denotes the *execution* of the corresponding operation `op`.
- `op`: denotes an *autonomous operation*, *i.e.* the operation may be executed but no receipt is needed.

We then define two kinds of guards:

- `[&op]` means that an `>op` receipt has been received before. This is used as a condition to trigger the corresponding `op>` action.
- `[not fullMailbox]` may appear on receipt transitions (`>op`) and denotes that a mailbox may be bounded.

A *mailbox* may then be defined as a computable data structure which memorises message receipts not already taken into account in a given state. We assume that mailboxes and associated guards are independent from other data types, that is the STS is decomposable as

defined in Definition 4. Our experiments have shown that this is a sensible choice in several classes of applications. A mailbox will be *bounded* if the size of the associated buffer is bounded, else it will be *unbounded*. This definition naturally extends to composed entities (a composed component or a whole system). The size of mailbox is a positive integer if the mailbox is bounded, else it is noted ω. As a simple complexity measure for STS we consider $N \times M :: [l_1, \ldots, l_n]$ where N is the number of states, M the number of edges and l_i are the sizes of the different subcomponent buffers (in case of a global structured STS).

Definition 5. *A fifo is a mailbox protocol in which message receipts are stored in their arrival order and messages are taken out in the same order (first in - first out). Hence* op> *is possible if and only if* op *is the next message to be taken out of the mailbox.*

Definition 6. *A dictionary of services (or dico) is a mailbox protocol which abstracts away from receipt order. A natural number is associated to each possible message. This number corresponds to the number of such a message in the mailbox.* op> *is possible if and only if the natural corresponding to* op *is greater than zero.*

The fifo notion is usual, the interest of dico will be illustrated from both theoretical and practical point of views in this paper. Expressions *dico bounded* or *dico unbounded* (respectively *fifo bounded* or *fifo unbounded*) are used to refer boundedness properties of these two specific mailbox protocols.

We want to study the mailboxes evolution during computation. This is relevant since, as we show in the next sections, such an analysis will sometimes demonstrate that we have bounded mailboxes. In such a case we may then reuse techniques available for finite labelled transition systems. A specialisation of the transition system may be defined removing the need for explicit unbounded buffers. Some comments on the interesting properties of bounded systems may be found in [29, 31]. Indeed, real systems are always bounded. Finding out the correct buffer sizes is thus important. We will describe in the remaining of this section the general problem and give some useful results.

3.1 Mailbox Computability

A first comment can be made on mailbox computability: obviously if all paths are finite, then mailboxes are computable. However, this is not the case with infinite paths, which is the general case.

Proposition 2. *Depending on the STS and on the mailbox protocol, mailboxes may be not computable.*

Proposition 3. *For a fifo protocol, the contents of mailboxes may be not computable. However, for a dico protocol, mailboxes are always computable.*

Take for example the following STS. Its set of infinite paths is the set of infinite sequences over {0, 1}. It is not enumerable nor computable. If we consider a fifo mailbox then we do not have a computable mailbox since its content is isomorphic to the paths. However, with more abstract mailbox protocols such as dico or if we only take the mailbox size into account then mailboxes are computable.

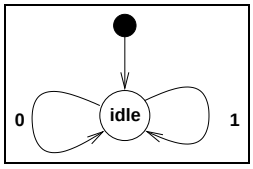

Fig. 1. Non Computable Infinite Paths

Example in Fig. 1 shows that infinite paths are not always computable. This is also true for fifo since mailboxes are isomorphic to paths. With dico, mailboxes memorise the messages names and, for each one, its number of occurrences within the paths. The number of services is finite and statically known, and the number of occurrences is an integer or ω. This shows that the set of the reachable values for a dico (taking ω into account) is finite. This theoretical result has immediate practical implications. For example, in [40], we consider a flight reservation system with dico mailbox protocol. We have presented there an algorithm which always computes a finite representation of the system behaviour. It is not always possible to do that with fifo. Hence, we have to split the algorithm into two parts: a `checking` one and a `bound` one.

3.2 Mailbox Boundedness Decidability

We are interested in situations where the system has bounded mailboxes.

Proposition 4. *Checking boundedness of STS with dico mailboxes is decidable.*

This comes from the fact that an STS with dico can be represented as some particular Petri net (using places for control and for message names, see example in Fig. 2) and from Petri nets boundedness decidability [34]. In this specific case the coverability graph contains only reachable configurations, even if the Petri net is not bounded. We will see later another way to check boundedness, more adequate to our STS and without the use of Petri net.

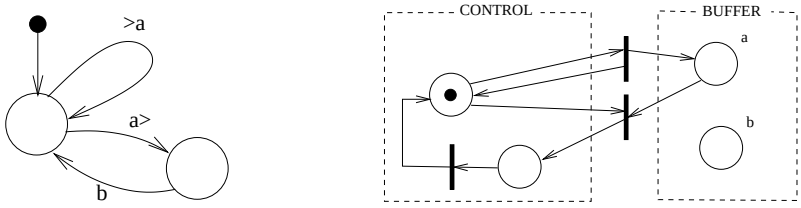

Fig. 2. A Simple STS and its Petri Net Representation

Proposition 5. *Checking boundedness of STS with fifo mailboxes is undecidable.*

It is easy to see that STS with fifo are particular CFSM systems (and thus a Turing complete formalism). Conversely a two CFSM system is obviously a two STS system with fifo. If there are more CFSM ($n > 2$), each of them will be represented by a synchronous product of $n - 1$ STS with fifo. The general boundedness problem of fifo communicating state machines is undecidable [13]. However there are some related and decidable, but incomplete, techniques, for instance [31, 29].

3.3 Checking Dico Boundedness

The role of the `checking` algorithm is to analyse an STS and to check if the system is dico bounded or not. The algorithm principle is to search the simulation of the system behaviour for execution cycles. When such a cycle is found a comparison is done between the mailbox contents of the two state occurrences. If the mailbox content at the end of the cycle is strictly greater than the mailbox content of the same component at the beginning of the cycle then this component is unbounded. The algorithm explores the reachable configurations with, for instance a depth first traversal. Whenever a cycle is found there are four cases depending on the comparison between the new and the old mailboxes. The $>_{dico}$ operator compares, for each kind of message, the number of its occurrences. More formally $d >_{\text{dico}} d' \;\widehat{=}\; \forall e \in \Sigma_D,\ (d[e] \geq d'[e]) \wedge \exists e \in \Sigma_D,\ (d[e] > d'[e])$. Moreover, $d =_{\text{dico}} d' \;\widehat{=}\; \forall e \in \Sigma_D,\ (d[e] = d'[e])$, and $d <>_{\text{dico}} d'$ if neither $d =_{\text{dico}} d'$ nor $d >_{\text{dico}} d'$ nor $d <_{\text{dico}} d'$.

1. If $new >_{\text{dico}} old$ then `checking` stops, a dico is unbounded.
2. If $new =_{\text{dico}} old$ then there is a loop in the configuration graph, the traversal continues with pending configurations.
3. If $new <_{\text{dico}} old$ then the traversal continues with pending configurations.
4. If $new <>_{\text{dico}} old$ then the traversal continues with the new configuration.

The traversal stops either with an unbounded dico or it explores a subset of the reachable configurations and argues that dicos are bounded.

Proposition 6. *The `checking` algorithm decides if an STS is dico bounded or not.*

The proof of this proposition is done in Appendix A.

This algorithm can cope with size constraints on the mailboxes (implementing the `[not fullMailbox]` guard) and it may be used to find the minimal requirements on the mailbox sizes to get a bounded system. A variant of this algorithm computes the different cycles where a mailbox accumulation exists. This may suggest the localization of errors to the specifier. The checking efficiency is not the point here. Since it is close to Petri nets [34], similar optimisation techniques, for instance matrix-equations, are possible.

3.4 Checking Fifo Boundedness

Let α be the function from fifo to dico which abstracts from the arrival order of messages. It may be inductively defined on the structure of fifo: $\alpha(\emptyset_{\mathrm{fifo}}) = \emptyset_{\mathrm{dico}}$, $\forall op$, $\alpha(> op, f) = add(> op, \alpha(f))$, and $\alpha(op >, f) = sub(op >, \alpha(f))$. Note that α is a surjective homomorphism from fifo to dico.

Proposition 7. *There is a simulation from the STS with fifo configuration relation to the STS with dico configuration relation.*

The notion of simulation is for instance formally defined on LTS in [8]. Our simulation is based on α, and it is sufficient to prove that for all configuration traces in the STS with fifo there exists a related configuration trace in the STS with dico. Take an STS with fifo configuration trace: $(s_0,\ \emptyset_{\mathrm{fifo}}) \xrightarrow{w_1} \ \ldots \ \xrightarrow{w_n} (s_n, f_n)$, where w_j are action labels with implicit guards. We have to prove that $(s_0,\ \emptyset_{\mathrm{dico}}) \xrightarrow{w_1} \ \ldots \ \xrightarrow{w_n} (s_n, \alpha(f_n))$ is a corresponding (through α) STS with dico configuration trace. By induction on n, the case $n = 0$ is obvious. Taking the hypothesis to be true for $l-1$, we must prove it for l. We have $(s_{l-1}, f_{l-1}) \xrightarrow{w_l} (s_l, f_l)$, we prove that w_l is fireable from $(s_{l-1}, \alpha(f_{l-1}))$ and reaches $(s_l, \alpha(f_l))$.

- w_l autonomous: w_l is fireable from $(s_{l-1}, \alpha(f_{l-1}))$ and reaches (s_l, d_l), with $d_l =_{\mathrm{dico}} \alpha(f_l) =_{\mathrm{dico}} \alpha(f_{l-1})$.
- w_l receipt: w_l is fireable from $(s_{l-1}, \alpha(f_{l-1}))$ and reaches (s_l, d_l). We have $d_l[w_j] = \alpha(f_{l-1})[w_j] = \alpha(f_l)[w_j]$ for all $w_j <> w_l$. For w_l we have $d_l[w_l] = \alpha(f_{l-1})[w_l] + 1 = \alpha(f_l)[w_l]$.
- w_l execution: $\alpha(f_{l-1})[w_j] > 0$ since w_l is fireable from f_{l-1} hence w_l is fireable from $(s_{l-1}, \alpha(f_{l-1}))$. We have $d_l[w_j] = \alpha(f_{l-1})[w_j] = \alpha(f_l)[w_j]$ for all $w_j <> w_l$. For w_l we have $d_l[w_l] = \alpha(f_{l-1})[w_l] - 1 = \alpha(f_l)[w_l]$.

Such a simulation relation is useful to extend properties of dico systems to fifo systems.

Proposition 8. *The* `checking` *algorithm is a sufficient condition to get the fifo boundedness of STS.*

If the STS is dico bounded, the previous proposition ensures that the set of fifo configurations is also bounded. Note that finding a cycle with a fifo accumulation (using the prefix ordering) is neither a sufficient nor a necessary condition for unboundedness.

Some unboundedness criteria maybe sometimes useful, one simple example follows.

Definition 7. *A STS has a simple accumulating cycle if and only if there exists a cycle in the state machine which does not consume any message but adds messages in the mailboxes.*

A simple accumulating cycle does not have `op>` on its transitions, it has only `>op` and autonomous operations. It is a very simple decidable case of the general theorem 2 of [29]. As we will see in the next examples this is a common situation which implies unboundedness of the STS.

3.5 On the Use of these Algorithms

There are different possible use of these algorithms depending of the computability of mailboxes, the decidability of boundedness and the bounded nature of the system. The more general way is to use some algorithms to check the system for bounded or unboundedness.

- If the system is bounded then `bound` may be used, or `partial` with constraints on mailboxes (size, contents, ...).
- If the system is not bounded or if no boundedness information exists, then the `partial` algorithm must be used with size constraints.

In case of computable mailboxes the same as above is possible. However, as illustrated with dictionaries in [40], a `bound+checking` algorithm can be applied and yields an interesting compact description. Sections 4 and 5 show application examples of these algorithms.

4 The Controller Example

As an example consider a simple system which provides exclusive access to a shared resource to some units, see Fig. 3 for an architectural description. A component is graphically represented by a box with receipt pins at left and execution/emission pins at right. Given some component, if a right pin has no corresponding (same name) left pin, then it is an autonomous operation for this component.

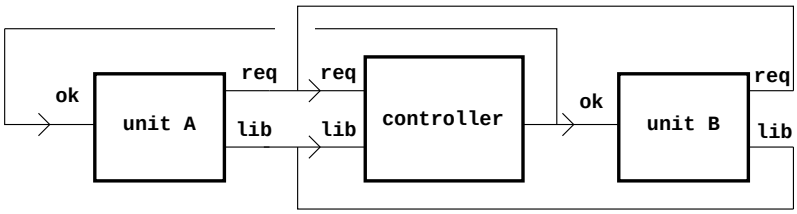

Fig. 3. An Architecture with two Units and a Controller

In Figure 4, we have the interface and the dynamic behaviour of the unit and the controller components. The controller manages the mutual exclusion of the different units. Units are simple components which try to access the resource and release it afterwards. The meaning of the message labels are the following:

- `ok` means that a unit is allowed to access the shared resource,
- `req` denotes the access request for a unit,
- `lib` stands for the release of the shared resource by a unit.

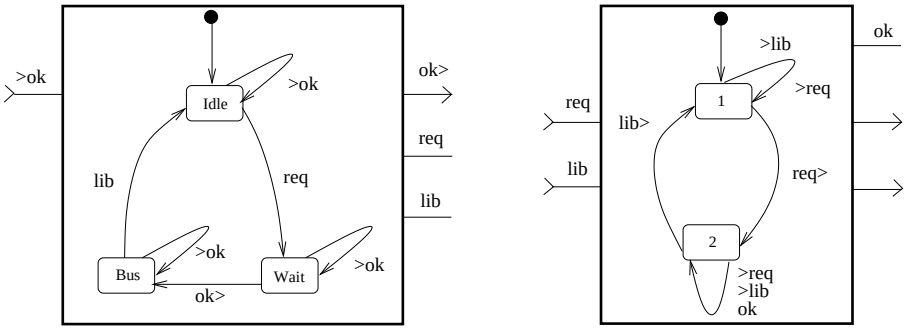

Fig. 4. Unit and Controller Dynamic Behaviours

The use of the `checking` algorithm on the system resulting from the architecture described in Fig. 3 shows that this system is not bounded with dico mailboxes. On one hand the units ask for a request and wait, on the other hand the controller tries to serialise the requests. Using the `partial` algorithm with fifo and size constraints (for example 1 for the controller, 2 for unit A and 0 for unit B) shows that unit A received two `ok` messages. The fifo unboundedness may be proved in this case since we have a cycle labelled only by receipts and autonomous operations (see Definition 7).

One would expect that such a system, with n units, should have bounded buffers: one for the units and n `req` plus one `lib` messages for the controller. Here the problem is that the controller in Fig. 4 may emit several `ok` messages, which is not correct. The definition of a correct dynamic behaviour for a distributed system may be a hard task. Even if our example is rather simple, the detection and the correction of such errors are definitively not obvious tasks in an asynchronous context. Mailbox analysis may provide relevant information to ease it. Here, mailbox analysis suggests to change the controller behaviour, as in the left side of Fig. 5.

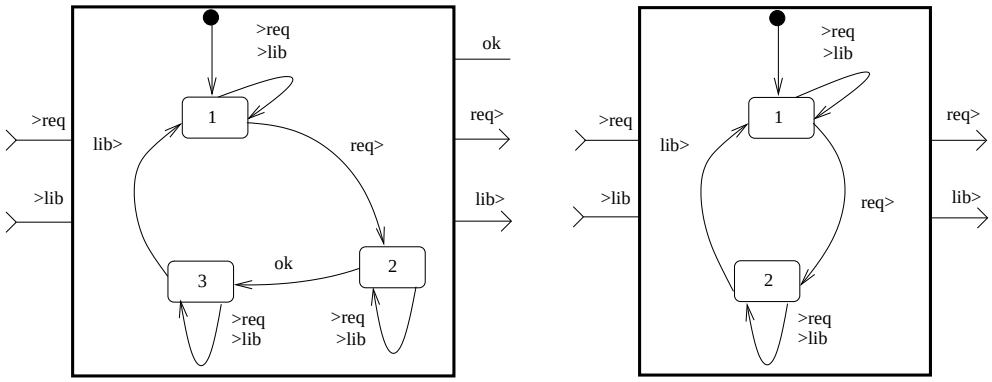

Fig. 5. The Three and the Two States Controllers

The analysis with fifo builds a 24×39::[3, 1, 1] bounded system and mutual exclusion for resource access is satisfied. A simple deadlock analysis finds one deadlock, with a mailbox content equal to `req lib req`. The controller received a first request and it sent `ok` to the corresponding unit. But before to receive the `lib` message from this unit, the other unit has sent a `req` leading to a deadlock. The result is depicted in Figure 6 thanks to the graphviz tools [20, 19].

One idea here would be to constrain the controller buffer size to avoid this deadlock. However, reusing the `partial` analysis with size constraints shows that the deadlock problem remains even with a size of 1. This convinces us that we have, maybe, a wrong system. We may then change the communications, the asynchronous semantics, the buffer policy or the component behaviours. There is no safe architecture changes, but asynchronous semantics leads to interesting remarks. A solution is to mimic synchronous communication. We have checked such a solution but we discarded it for being too synchronous (hence less realistic). Another idea is to change the priority of messages, by example switching the fifo buffer policy to dico. If we run a new analysis with dico, we obtain a 21×37:[3, 1, 1] result with mutual exclusion and no deadlock.

One may also observe that in the left side of Fig. 5 the `ok` message may be emitted during the `req>` execution. This lifts the need for state 3 and simplifies the controller as in the right-hand side of Fig. 5. The checking and bound analysis with dico gives a result of 18×32:[3, 1, 1] with mutual exclusion and no deadlock (see Fig. 7).

A correct solution with fifo is simple but not obvious. It needs to change the dynamic behaviour of the controller. We consider the controller of Fig. 5 left-hand side with two additional loops labelled by `req>` on states `2` and `3`. The checking analysis shows that it is fifo bounded (35×61:[3, 1, 1]), with mutual exclusion and no deadlock.

These experiments demonstrate that our algorithms may be valuable tools to help designing communicating components. Preliminary works had already been used in [40] with a simple flight reservation system. However we provide here a more general and more rational approach (constraint sizes, buffer policies, dedicated algorithms).

5 Other Applications

This section presents two different and complementary applications. The first one is related to optimisation in a general sense and the second one is devoted to the analysis of non purely asynchronous systems.

5.1 Automatic Optimisations and Simplifications

The results given by the previous algorithms may be used to optimise component implementation at deployment step. These algorithms may also be useful to simplify specifications and verifications. We report here a part of the conclusion of an experimentation in PVS [41], see [35] for more details. The problem is to specify a simple flight reservation

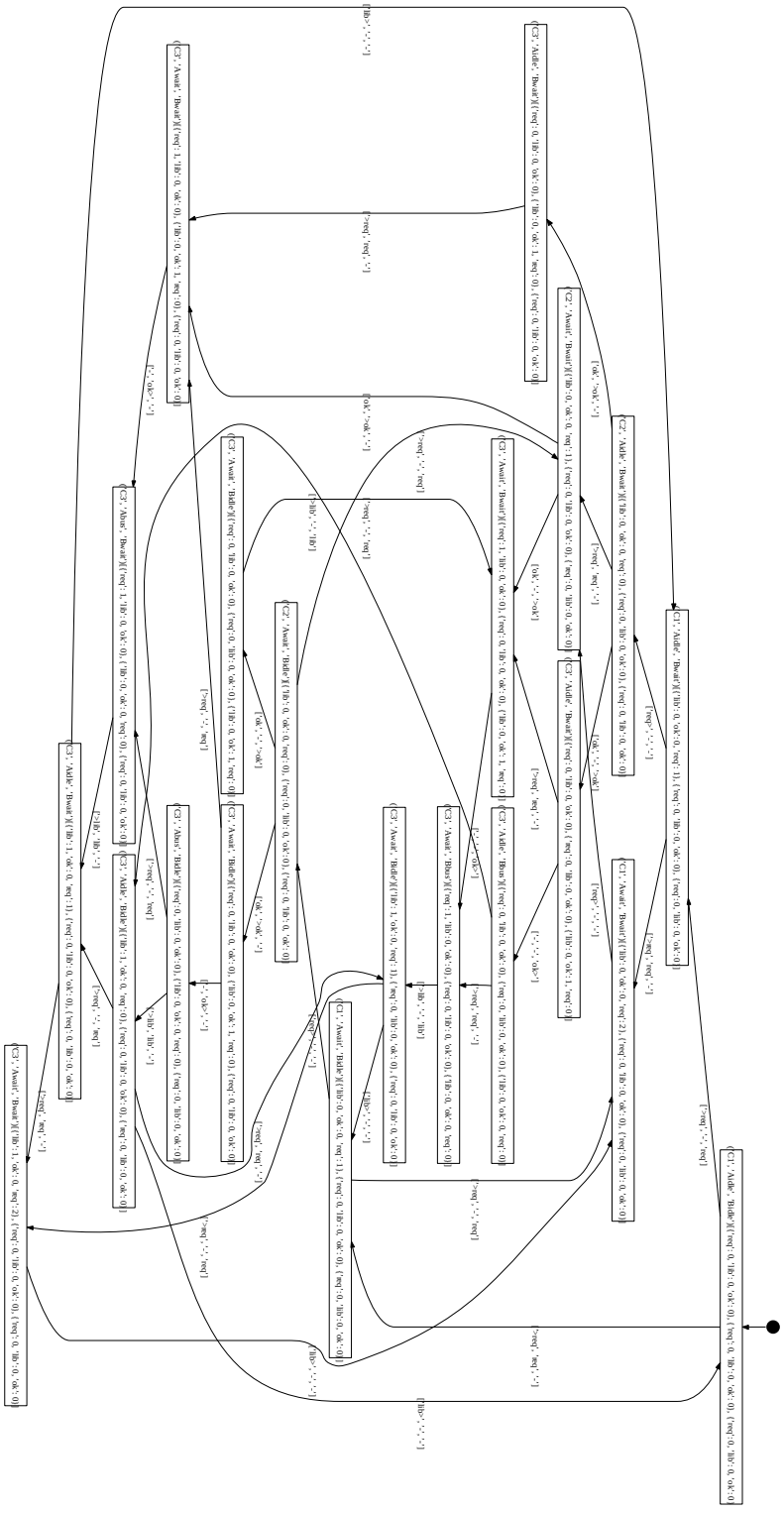

Fig. 6. The Controller Behaviour with a Fifo of Services

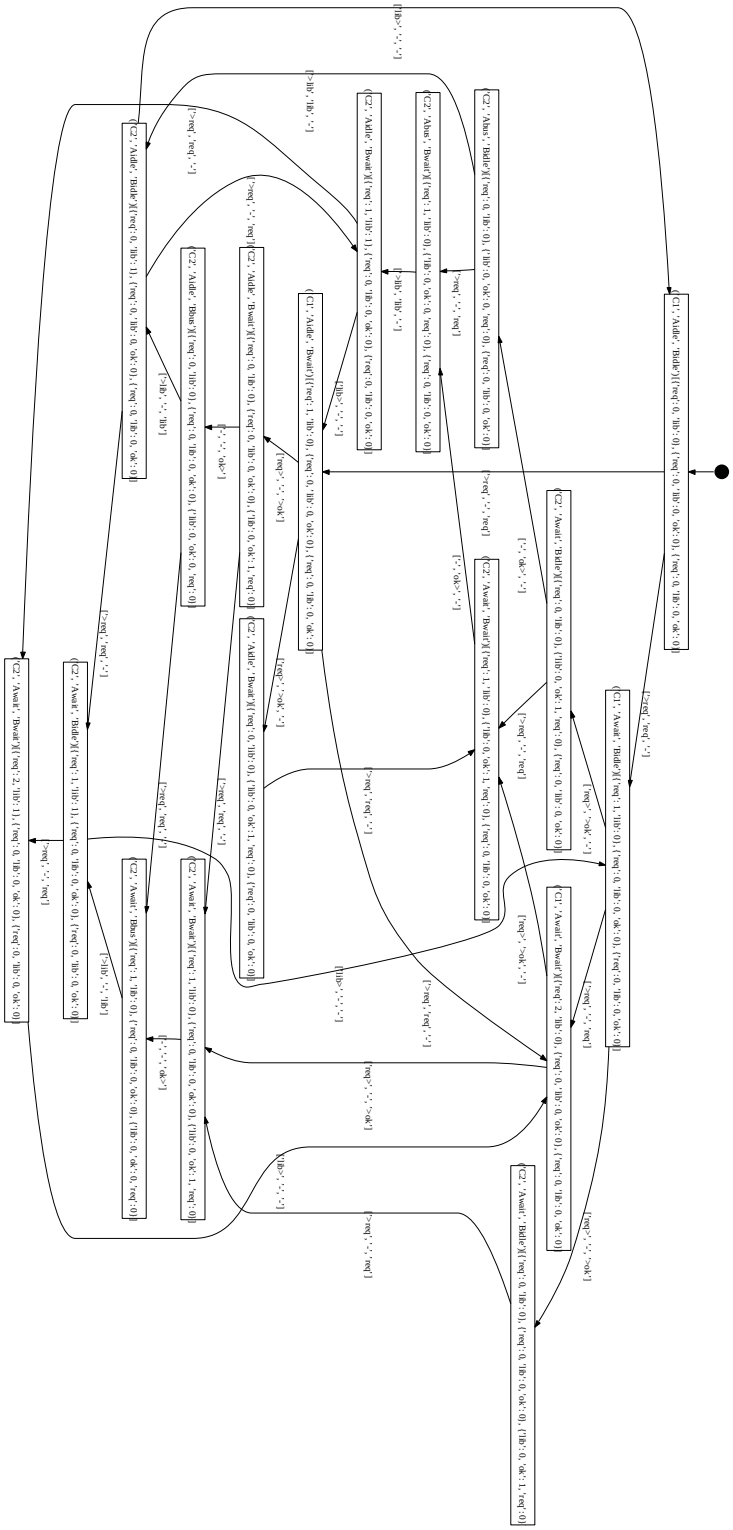

Fig. 7. The Controller Behaviour with a Dictionnary of Services

system [40]. However the goal is rather verification and especially proving that time duration between the user request and the receipt of its ticket is bounded. The first thing to do is to specify the system in a acceptable way for a prover (we chose PVS).

The PVS translation of a component follows a general scheme [38], but we have to add the management of the mailboxes and the associated guards. Unfortunately, this has proven complex even for such a rather simple example. We have tried two different approaches but both raise similar conclusions. The specification of asynchronous components is more complex than for synchronous ones, and as previously said, asynchronous communication does not ensure finite waiting time.

A first idea is to use the `checking` and `bound` algorithms to compute an exhaustive and bounded simulation of the system. Once this is done we extract from this analysis some constraints about the component behaviour. For example it shows that a component reaches the initial state either with an empty mailbox or with a mailbox containing an ordering message. Form this information and from the original STS, the `partial` algorithm builds a new STS. The algebraic specification associated to this new STS is simpler. There is no more need for mailboxes, hence for buffer computations and for the related guards and operations. One additional benefit is that the building of the simplified specification may be automatically generated from the original component description and the execution constraints.

From that it was possible to prove the bounded time property and even to get automatically a maximum bound. This experiment also makes explicit a way to automatically simplify the specification of asynchronous components depending on the architectural context.

5.2 Non Purely Asynchronous Systems

These algorithms are also useful to analyse other examples of applications even if they do not use explicitly asynchronous communications. This generally means to redesign the system in such a way that asynchronous communications appear. This is possible whenever there is one action which adds some data in a component and if another action removes it.

The Bakery Protocol. The bakery protocol [23], for example, may be redesigned with components and asynchronous communications. Figure 8 presents the architecture made up of a ticket machine and a counter-desk. The meaning of interactions is:

- `enter`: someone enters the shop,
- `take`: (s)he takes a ticket,
- `process`: (s)he is served at the counter-desk,
- `leave`: (s)he leaves the shop.

The checking algorithm shows that this system is dico unbounded. There is a simple accumulating cycle hence it is also fifo unbounded and then we have to choose a bound

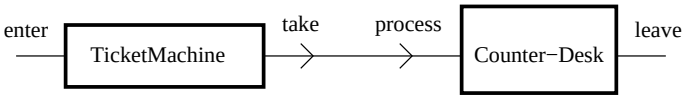

Fig. 8. An Asynchronous Architecture for the Bakery Protocol

size for the counter queue. The `partial` simulation shows that with a size $l \geq 1$ the complexity of the result is: $(4l+4)\times(8l+4)$::[0,l]. From these experiments we may infer the global structure of the behaviour and prove for example the deadlock freeness property for any l.

The SLIP Protocol. A similar approach can be applied to the SLIP protocol [37]. This is a simple protocol to send bytes on a serial link. An asynchronous architecture for it is described in Fig. 9. In the original specification we have a channel of size 1 which is implemented here using a mailbox. The meaning of interactions is:

- `byte`: read any byte distinct from `esc` and `end`.
- `esc`: read an escape character
- `end`: read an ending character

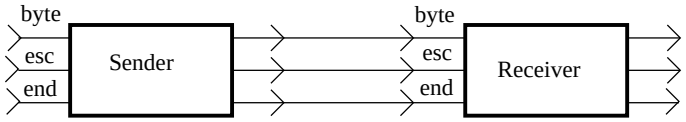

Fig. 9. An Asynchronous Architecture for SLIP

This example is dico unbounded and has a simple accumulating cycle thus it is also fifo unbounded. The `partial` algorithm gives simulation results for various channel sizes. For size 1 the result has a complexity of 8×10::[0,1] and the system is depicted in Fig. 10.

We have checked various synchronous versions. The minimum result is for three STS (18×33::[1,1,1]) and the maximum one is for three LTS (50×88::[1,1,1]). The complexity gain with the asynchronous approach comes from the fact that we have less component than with the synchronous versions. Another important aspect is that we can more precisely balance the use of abstraction (guards and variables) and the use of concrete events.

This protocol is expected to be reliable, *i.e.* each output sequence results from the same input sequence. This may be proved by weak bisimulation between the previous state machine and a state machine which reads a byte and then directly outputs it. One interesting paper about this verification example is [24], but there the way to prove reliability is quite complex (it uses μCRL). Looking at the result in Fig. 10 weak bisimulation is rather obvious. This is also true with a channel of size n, but model-checking would

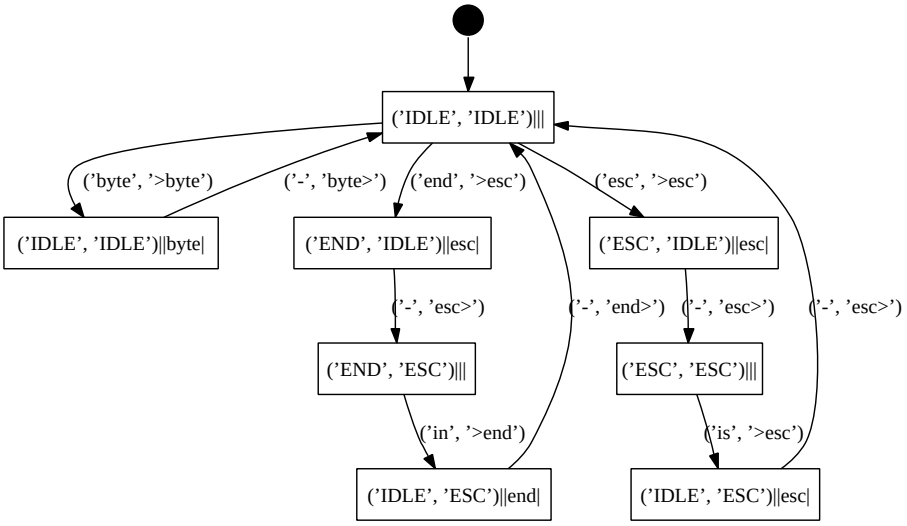

Fig. 10. The Bounded Simulation

be a more safe and efficient way to check it from the resulting analysis of the `partial` algorithm.

6 Related Work

Related approaches to STS are [23] with I/O automata and symbolic transition graphs of [26]. Our STS formalism is more general than both, since our states are more abstract, they are not only tuples of variables, and the action semantics may be defined by conditional axioms. In addition to this point, our work shows a natural link between first-order logic and CTL* (see [39]). This formalism clearly extends LTS, extended automata [2], push-down automata and also, but less obviously, Petri nets and timed automata. Various Petri net extensions may be embedded, transition systems with fifo [29] or communicating finite state machines [42, 13].

Boundedness is decidable for Petri Nets [34], but not for fifo communicating state machines [13], even with lossy fifo channels. However there are some decidable techniques, for instance [31, 29]. In [31] the authors present an algorithm to check boundedness of UML-RT models. It is based on several abstractions and the resolution of linear equations with positive coefficients. One important advantage is that it provides scalability of the test. Their approach uses an over approximation which relies on a data structure equivalent to our dico, but they do not mention the result we are using here. However their approach has two drawbacks. First, it abstracts away the sequence prefixes of cycles. They do `structural boundedness` *i.e.* boundedness checking independently of the initial conditions. Another restriction is that they consider independent cycles on the component state machines. We think that for these reasons our criterion may be useful

but actually it is less efficient. A first difference is the use of STS which allows us to control the size of the state machine. A first optimisation was to develop a checking algorithm avoiding the explicit computation of the synchronous product. One future way is to explore the computation of the cycles of the product from the cycles of the subcomponents, maybe using some ideas coming from [31] or matrix-equations.

Another work is [29], which defines a general criterion to check unboundedness. The algorithm developed there seems a bit restrictive on the form of the STS components, since it requires word linearity of the output languages.

Various problems of interest in verifications of unreliable fifo channels have been considered in [15]: message lost, duplication, error insertion and some combinations of these behaviours. The verification of interest are: reachability, unboundedness, deadlock and model-checking against CTL*. All these problems are undecidable for communicating machines with reliable fifo channels. But, surprisingly some of these problems are decidable with unreliable channels, for instance the finite termination problem is solvable for machines capable of lossiness errors.

Some other related papers are [27, 2, 22], they deal with related questions about classifications of symbolic state systems, reachability analysis of fifo machines and model-checking infinite systems.

There is related work on formal components and architectures, too. [33] is one of the main reference about ADL. A main difference with our work is the use of asynchronous communications while most of the time ADL promotes synchronous communications. Rapide, [32], is an event-based concurrent, object-oriented language for prototyping system architectures. A main design principle is to get an executable model in which the concurrency, synchronisation, dataflow and timing properties are explicit. The execution model is based on partially ordered event sets. Rapide promotes simulation and tests but with some static analysis aspects.

It is also interesting to compare our approach with WRIGHT [6, 5]. WRIGHT is a formal architectural description language with first class components and connectors. It may be seen as relook of CSP, since the notations and the semantics are inspired by this process algebra. This has the advantages to get effective model checking for CSP and related work about behavioural refinement. WRIGHT proposes a deep analysis about automatic checking for architectural languages. It allows to check connector consistency, configuror consistency, and attachment consistency using mainly techniques to prove deadlock freedom and behavioural refinement. However, most of these verifications are limited by the value-passing state explosion problem hence consider simple data types.

7 Conclusion

Behavioural Interface Description Languages are an important feature of CBSE languages in order to address reuse, adaptation and verification issues. BIDL often take only synchronous communication into account. However, synchronous and asynchronous com-

munications is a more realistic context with reference to real component models. Yet, this kind of communication is also known to yield difficult issues. In this paper we have demonstrated how Symbolic Transition Systems may take into account asynchronous communication and then may be used to address verification issues related to mailbox boundedness. Our STS are symbolic transition systems related to an abstract description of a data type. It provides a uniform and general framework to reason and compare different but related state machines. This may be used to express different mailbox policies for components. In this paper this property has been used to related boundedness decidability results between fifo and dico mailbox policies.

Often designing architectures needs guards and parameters to code complex communications between several instances of the same component type, for instance controller systems with several clients or telephony systems. One future work is to study specific communication checking to help the design and analysis of such systems. We have yet some preliminary results about deadlock checking for STS. We have to prove these results and to compare them with model-checking techniques. One last work is to continue the development of our Java library to integrate some already known algorithms, for instance some results of [29, 15, 2, 31].

A Proof of the dico checking

Proposition 9. *The* `checking` *algorithms decides if an STS is dico bounded or not.*

The algorithm is based on a classical depth-search traversal with history to find cycles in the state machine. The non termination may only result from the infinite execution of the case $new <> old$. It means that a (d_i) infinite sequence of incomparable dico exists. Note that $w \leq_{word} w' \supset dico(w) \leq_{\mathrm{dico}} dico(w')$, where w, w' are words built on the receipt alphabet, $\leq_{word}$ is the subword ordering and $dico$ computes the corresponding dictionary of a word. This result extends to n components, see a similar use in [15]. A Higman's theorem states that if a set of words consists of mutually incomparable elements according to $\leq_{word}$ then this set is finite. Note also that the $dico$ mapping is not injective: one dico has several corresponding words. If we have a set of incomparable dico, we have a greater set of incomparable words, thus Higman's theorem states that this set is finite which contradicts the infinity hypothesis.

If `checking` stops with an unbounded response then it found an accumulating cycle. It means a configuration trace as: $(s_0, \emptyset_{\mathrm{dico}}) \xrightarrow{*} (s_i, d_i) \xrightarrow{w_1} \ldots \xrightarrow{w_k} (s_i, d_{i+k})$, where w_j are action labels with implicit guards, $d_{i+k} >_{\mathrm{dico}} d_i$ and $k > 0$. In such a case the cycle may be run any number of times and it implies that we have an infinite sequence of distinct configurations. It remains to prove that if the cycle is run once it may be rerun. More formally we have $(s_i, d_i) \xrightarrow{w_1} \ldots \xrightarrow{w_k} (s_i, d_{i+k})$, where $d_{i+k} >_{dico} d_i$ and $k > 0$ and we prove $\forall 0 \leq j < k,\ w_{j+1}$ is fireable from (s_i, d_{i+j+k}) and $d_{i+j+1+k} >_{\mathrm{dico}} d_{i+j+1}$. A proof by induction on j follows. The basic case, $j = 0$, by hypothesis w_1 is fireable

from (s_i, d_i) and $d_{i+k} >_{dico} d_i$. Analysing each component and the three kinds of events (autonomous, receipt, execution) it implies that w_1 is fireable from (s_i, d_{i+k}). A similar analysis and properties of naturals ensure that $d_{i+1+k} >_{dico} d_{i+1}$.

Assume that the induction hypothesis is true for $0 \leq j < l < k$ we prove that it is also true for $j = l$. This follows from a similar analysis than in the basic case. We analyse the different cases for w_l and for each we prove the expected properties.

- w_l autonomous: There is no precondition, w_l is fireable from (s_{i+l-1}, d_{i+l-1}) it is fireable from $(s_{i+l-1+k}, d_{i+l-1+k})$ too, and there is no postcondition change.
- w_l receipt: There is no precondition, w_l is fireable from $(s_{i+l-1+k}, d_{i+l-1+k})$. We have $d_{i+l}[w_l] = d_{i+l-1}[w_l]+1$, $d_{i+l+k}[w_l] = d_{i+l-1+k}[w_l]+1$, and $d_{i+l-1+k}[w_l] \geq d_{i+l-1}[w_l]$ hence $d_{i+l+k}[w_l] \geq d_{i+l}[w_l]$.
- w_l execution: there is a precondition $d_{i+l-1+k}[w_l]$ has to be > 0, but $d_{i+l-1}[w_l] > 0$, since w_l is fireable from (s_{i+l-1}, d_{i+l-1}), and $d_{i+l-1+k}[w_l] \geq d_{i+l-1}[w_l]$, thus w_l is fireable from $(s_{i+l-1+k}, d_{i+l-1+k})$. We have $d_{i+l}[w_l] = d_{i+l-1}[w_l] - 1$, $d_{i+l+k}[w_l] = d_{i+l-1+k}[w_l] - 1$, and $d_{i+l-1+k}[w_l] \geq d_{i+l-1}[w_l]$ thus $d_{i+l+k}[w_l] \geq d_{i+l}[w_l]$.

The last point to note is, by hypothesis it exists at least w_j such that $d_{i+l-1+k}[w_j] > d_{i+l-1}[w_j]$, and in the three cases above this property is propagated to $d_{i+l+k}[w_j] > d_{i+l}[w_j]$.

To complete the proof we argue that if the STS is dico unbounded then the checking returns true (there is an accumulating cycle). If the system is unbounded then the configuration tree has an infinite sequence of distinct configurations. Since the set of states is finite we have in fact an infinite sequence (s, d_i) of distinct configuration with $s \in S$. If it exists $i < j$ and $d_i <_{dico} d_j$ then we have an accumulating cycle. If it exists $i < j$ and $d_i >_{dico} d_j$ then we may prove that the sequence is finite since there is a strictly decreasing sequence of positive naturals. Let's assume there is no accumulating cycle, we have an infinite sequence of d_i with incomparable dictionaries and this is not possible since the algorithm terminates.

References

1. *The Fractal Composition Framework*, June 2002. `www.objectweb.org`.
2. Parosh Aziz Abdulla, Aurore Annichini, Saddek Bensalem, Ahmed Bouajjani, Peter Habermehl, and Yassine Lakhnech. Verification of infinite-state systems by combining abstraction and reachability analysis. In Nicolas Halbwachs and Doron Peled, editors, *Computer-Aided Verification, CAV '99*, volume 1633 of *Lecture Notes in Computer Science*, pages 146–159, Trento, Italy, July 1999. Springer-Verlag.
3. M. Aiguier, F. Barbier, and P. Poizat. A Logic with Temporal Glue for Mixed Specifications. In *FOCLASA'2003 – Foundations of Coordination Languages and Software Architectures*, volume 97 of *Electronic Notes in Theoretical Computer Science*, pages 155–174. Springer-Verlag, 2005.
4. Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pages 187–197. ACM Press, 2002.
5. Robert Allen, Remi Douence, and David Garlan. Specifying and Analyzing Dynamic Software Architectures. In *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)*, volume 1382 of *Lecture Notes in Computer Science*, pages 21–37. Springer-Verlag, 1998.

6. Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.
7. André Arnold, G. Point, Alain Griffault, and Antoine Rauzy. The altarica formalism for describing concurrent systems. *FUNDINF: Fundamenta Informatica*, 34:109–124, 2000.
8. André Arnold. *Finite Transition Systems*. International Series in Computer Science. Prentice-Hall, 1994.
9. E. Astesiano, B.Krieg-Bruckner, and H.-J. Kreowski Eds., editors. *Algebraic Foundations of System Specification*. IFIP State-of-the-Art Reports. Springer Verlag, 1999. ISBN 3-540-63772-9.
10. Christian Attiogbé, Pascal Poizat, and Gwen Salaün. Integration of Formal Datatypes within State Diagrams. In *FASE'2003 - Fundamental Approaches to Software Engineering*, volume 2621 of *Lecture Notes in Computer Science*, pages 344–355. Springer-Verlag, 2003.
11. Françoise Baude, Denis Caromel, and Matthieu Morel. From Distributed Objects to Hierarchical Grid Components. In D. C. Schmidt R. Meersman, Z. Tari and al., editors, *On The Move to Meaningful Internet Systems 2003: Coopis, DOA, and ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, pages 1226–1242. Springer-Verlag, 2003.
12. J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of Process Algebra*. Elsevier, 2001.
13. Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, April 1983.
14. Muffy Calder, Savi Maharaj, and Carron Shankland. A Modal Logic for Full LOTOS Based on Symbolic Transition Systems. *The Computer Journal*, 45(1):55–61, 2002.
15. G. Cécé, A. Finkel, and S. Purushothaman Iyer. Unreliable channels are easier to verify than perfect channels. *Information and Computation*, 124(1):20–31, 1996.
16. Christine Choppy, Pascal Poizat, and Jean-Claude Royer. A Global Semantics for Views. In T. Rus, editor, *International Conference on Algebraic Methodology And Software Technology, AMAST'2000*, volume 1816 of *Lecture Notes in Computer Science*, pages 165–180. Springer Verlag, 2000.
17. Christine Choppy, Pascal Poizat, and Jean-Claude Royer. Formal Specification of Mixed Components with Korrigan. In *Proceedings of the 8th Asia-Pacific Software Engineering Conference, APSEC'2001*, pages 169–176. IEEE, 2001.
18. Christine Choppy, Pascal Poizat, and Jean-Claude Royer. Specification of Mixed Systems in KORRIGAN with the Support of a UML-Inspired Graphical Notation. In Heinrich Hussmann, editor, *Fundamental Approaches to Software Engineering. 4th International Conference, FASE 2001*, volume 2029 of *LNCS*, pages 124–139. Springer, 2001.
19. John Ellson, Emden Gansner, Eleftherios Koutsofios, John Mocenigo, Stephen North, Gordon Woodhull, David Dobkin, and Vladimir Alexiev. Graphviz. `http://www.research.att.com/sw/tools/graphviz/`.
20. John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz — open source graph drawing tools. *Lecture Notes in Computer Science*, 2265:483–484, 2002.
21. Wolfgang Emmerich and Nima Kaveh. F2: Component technologies: Java beans, COM, CORBA, RMI, EJB and the CORBA component model. In Volker Gruhn, editor, *Proceedings of the Joint 8th European Software Engeneering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engeneering (ESEC/FSE-01)*, volume 26, 5 of *SOFTWARE ENGINEERING NOTES*, pages 311–312. ACM Press, 2001.
22. Javier Esparza. Verification of systems with an infinite state space. *Lecture Notes in Computer Science*, 2067:183–186, 2001.
23. W. O. D. Griffioen and H. P. Korver. The bakery protocol: A comparative case-study in formal verification. In J. C. van Vliet, editor, *CSN'95 (Computer Science in the Netherlands)*, pages 109–121. Stichting Mathematisch Centrum, 1995.
24. Jan Frisco Groote, François Monin, and Jan van de Pol. Checking Verifications of protocols and Distributed Systems by Computer. In David Sangiorgi and Robert de Simone, editors, *CONCUR'98*, volume 1466 of *Lecture Notes in Computer Science*, pages 629–655. Springer-Verlag, 1998.
25. David Harel. Statecharts: A visual formulation for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
26. M. Hennessy and H. Lin. Symbolic Bisimulations. *Theoretical Computer Science*, 138(2):353–389, 1995.
27. Thomas A. Henzinger and Rupak Majumdar. A classification of symbolic transition systems. *Lecture Notes in Computer Science*, 1770:13–34, 2000.
28. A. Ingolfsdottir and H. Lin. *A Symbolic Approach to Value-passing Processes*, chapter Handbook of Process Algebra. Elsevier, 2001.

29. Thierry Jéron and Claude Jard. Testing for unboundedness of fifo channels. *Theoretical Computer Science*, 113:93–117, 1993.
30. Tomáš Kalibera and Petr Tuma. Distributed Component System Based on Architecture Description: The SOFA Experience. In D. C. Schmidt R. Meersman, Z. Tari and al., editors, *On The Move to Meaningful Internet Systems 2002: Coopis, DOA, and ODBASE*, volume 2519 of *Lecture Notes in Computer Science*, pages 981–994. Springer Verlag, 2002.
31. Stefan Leue, Richard Mayr, and Wei Wei. A scalable incomplete test for the boundedness of uml rt models. In *TACAS 2004*, volume 2988 of *Lecture Notes in Computer Science*, pages 327–341. Springer-Verlag, 2004.
32. David C. Luckham, John L. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.
33. Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.
34. Tadao Murata. Petri nets: properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.
35. Jacques Noyé, Sébastian Pavel, and Jean-Claude Royer. A PVS Experiment with Asynchronous Communicating Components. In *17th Workshop on Algebraic Development Techniques*, Barcelona, Spain, 2004. `www.emn.fr/x-info/jroyer/rrWADT04.pdf.gz`.
36. Pascal Poizat, Jean-Claude Royer, and Gwen Salaün. Formal Methods for Component Description, Coordination and Adaptation. In *WCAT'2004 - Int. Workshop on Coordination and Adaptation Techniques for Software Entities*, 2004.
37. J. Romkey. SLIP Protocol Specification. Web document. `www.faqs.org/ftp/rfc/pdf/rfc1055.txt.pdf`.
38. Jean-Claude Royer. The GAT Approach to Specify Mixed Systems. *Informatica*, 27(1):89–103, 2003.
39. Jean-Claude Royer. A framework for the gat temporal logic. In ISCA, editor, *Proceedings of the 13th IASSE'04 Conference*, 2004. ISBN: 1-880843-52-X.
40. Jean-Claude Royer and Michael Xu. Analysing Mailboxes of Asynchronous Communicating Components. In D. C. Schmidt R. Meersman, Z. Tari and al., editors, *On The Move to Meaningful Internet Systems 2003: Coopis, DOA, and ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, pages 1421–1438. Springer Verlag, 2003.
41. N. Shankar. PVS: Combining specification, proof checking, and model checking. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 257–264. Springer-Verlag, 1996.
42. Gregor von Bochmann. A General Transition Model for Protocols and Communication Services. *IEEE Transactions on Communications*, 28(4):643–650, April 1980.