Components with Symbolic Transition Systems: a Java Implementation of Rendezvous

Fabricio FERNANDES, Robin PASSAMA and Jean-Claude ROYER

OBASCO Group, École des Mines de Nantes – INRIA, LINA
4 rue Alfred Kastler, 44307 Nantes cedex 3, France.

Abstract. Component-based software engineering is becoming an important approach for system development. A crucial issue is to fill the gap between high-level models, needed for design and verification, and implementation. This paper introduces first a component model with explicit protocols based on symbolic transition systems. It then presents a Java implementation for it that relies on a rendezvous mechanism to synchronize events between component protocols. This paper shows how to get a correct implementation of a complex rendezvous in presence of full data types, guarded transitions and, possibly, guarded receipts.

Keywords. Component-Based Software Engineering, Behavioural Interfaces, Explicit Protocols, Symbolic Transition Systems, Rendezvous, Synchronization Barriers

Introduction

Component-Based Software Engineering (CBSE) is becoming an important approach for system development. As large distributed systems become always more critical, the use of formal analysis methods to analyze component interactions arises as a crucial need. To this end, explicit protocols have been integrated to component interfaces to describe their behaviour in a formal way. Behavioural interface description languages are needed in component models to address architectural analysis and verification issues (such as checking component behavioural compatibility, finding architectural deadlocks or building adapters to compensate incompatible component interfaces) and also to relate efficiently design and implementation models. Nevertheless, explicit protocols are often dissociated from component codes: they are ”pure” abstractions of the way components behave. This is really problematic, since nothing ensures component execution will respect protocols rules. So, a critical issue is to fill the gap between high-level formal models and implementation of protocols to ensure consistency between analysis and execution phases.

In this field, our long term-goal is to define a component programming language with explicit executable protocols, coupled with a formal ADL (Architectural Description Language) and associated analysis tools. To make a strong link between specification or design models and programming languages for implementation, there are two possible ways: (i) automated translation of models into programming code, and (ii) extraction of abstract model and protocol information from programming code. We focus on the first approach. The features of the target language are object-orientation, multi-threading and facilities for synchronization.

Acknowledgment. This work was partly supported by the AMPLE project www.ample-project.net, and the CAPES grant from Brazil.
As an instance, we consider Java 1.5. The second way, from code to model, is a bit different both on concepts and tools; see for example [1,2]. Our development process is decomposed into two steps: the first one is the description of components and architectures with our ADL formalism, and the second one is to represent, in Java, the state machine, the synchronization and to implement the data type part with Java classes. In the realization of this process, our current objective is to provide support for implementing component protocols in such a way that their execution respects the semantics of the protocol description language.

The chosen protocol description language is the Symbolic Transition System (STS) formalism [3]. STSs are finite state and transition machines with unrestricted data types and guards. The STS formalism is a general model of computation which may be seen as a strict subset of UML statecharts or as a graphical model of a process algebra with value passing and guards. It is adequate for formal design and verification, although this latter is still a difficult challenge. Various ways to verify these systems already exist: using a general prover, calculating abstractions or interfacing with classical model-checkers (the interested reader may look at [4,5,6,7]). Our approach for the verification of these systems relies on the interaction with efficient model-checkers and on the use of specific techniques for symbolic systems. For several examples, the use of boundedness and decomposition techniques we developed are described in [8]. The STS formalism has many advantages for design: it improves readability and abstraction of behavioural descriptions compared to formalisms with restricted data types. It helps to control state explosion with the use of guards and typed parameters associated to the transitions. Lastly, it allows the description of message exchange with asynchronous or synchronous communication mode.

Implementing STS requires to manage different development steps: (i) implementing the data part (ii) representing the protocol (iii) gluing the data part and the protocol into a primitive component (intra-component composition), and (iv) implementing components synchronization and communication mechanism (inter-component composition). The three first steps may be viewed from either a code generation or a code reutilization perspective. On the one hand, code generation from formal specification [9,10] is a problem related to compilation, but with a greater abstract gap between the source and the target language than for general purpose programming languages. On the other hand, code reutilization may be done with a more or less intrusive approach, a related reference is [11]. Whatever the way STSs are created, the central need is a generic mechanism to execute and synchronize the activities of STSs and to make them communicate.

The main proposal of this paper is to present this mechanism; we avoid here the discussion related to data type representation. Such a mechanism is important to get components that can be composed in a safe manner – with a direct link to the formal semantics level. More precisely, we focus on synchronous communication mode (see [12,11] for a related work on asynchronous communications). For the time being, we only consider one-to-many, one-way synchronous communications. As quoted in [13], during a synchronous communication the sender, to resume its own execution, waits for (i) completion by the receiver of the invoked method execution and then (ii) the return statement of the replier. It is opposed to asynchronous communication in the sense that the sender does not resume its execution as soon as the message has been sent. STS synchronous communication is a bit more sophisticated: a message transmission is bound to the service execution both on sender and receiver side. Semantics models of STS, as for process algebras or finite state machines use an advanced rendezvous that strongly glues several participants of a communication executing their guarded actions simultaneously. This can be seen as a generalization of synchronous communication modes of object-oriented programming languages with concurrent features. Coupled to guards on receipts, this allows complex interactions between components to be described. However, previous protocol implementations, for instance [14,15,16], only propose variants such as Remote Procedure Calls (RPCs) or synchronous message sending.
The STS formalism supports this composition semantics based on concurrency and event synchronizations [8]. The rendezvous is a synchronization point between several actions which may involve communications. The proposed mechanism implements an $n$-ary rendezvous, with receipt on guarded events and allows independent synchronizations to processes in the same time. We restrict communication to one sender of a value and several receivers, which provides a basic and powerful mechanism. Such a rendezvous requires two synchronization barriers, one for entering and one for leaving the rendezvous. The synchronization barrier principle for multiple threads is the following: all threads must arrive at the barrier before any of them are permitted to proceed past the barrier.

We show here how to get a correct synchronization mechanism build on top of a synchronization barrier, with respect to STS synchronization specificities. Our approach is achieved with four progressive steps: (i) we start with a simple rendezvous for Labelled Transition Systems (LTS) and a central arbiter, (ii) we then split the arbiter into several lock objects associated to the synchronizations, (iii) we improve the solution allowing independent synchronizations to enter simultaneously the barriers, and (iv) we add full data types, communications and guards.

The paper is organized as follows. Section 1 reviews related work. Section 2 presents the main features of our component model and an example of a system design. Section 3 introduces our hypotheses for the model implementation in Java. Section 4 describes the synchronization barrier principles and discusses how to implement communications and guards in the rendezvous. Finally, Section 5 draws concluding remarks and discusses future work.

1. Related Work

In the last decade, formal component models with behavioural descriptions have been proposed either on their own [17,18] or in the context of software architectures [19]. Different behavioural models have been used, such as process algebras [20,19] or automata-based formalisms [1,21]. However, if they propose different analysis mechanisms for component architectures, they do not address the issue of taking protocols into account within the implementation, which is a mandatory issue for seamless CBSE development. Discussions in this section focus on approaches with a strong coupling between message sending and service execution. Thus we do not discuss purely asynchronous approaches or synchronization by need (readers can refer to [13] for more details). Discussions are also directed towards approaches that propose a direct link between formal models and code.

The STS formalism [22,3] has initially been developed as a way to control the state and transition explosion problem in value-passing process algebras using substitutions associated to states and symbolic values in transition labels. The STS formalism we use, [8], is a generalization of this latter, associating a symbolic state and transition system with a data type description. The data type description is given using algebraic specifications [23,24]. The STS semantics provides concurrent composition of STSs with event synchronizations, namely the rendezvous notion introduced by CSP [25]. In a previous work, we extended the synchronous product of LTSs to STSs. The principles and the precise definition may be found in Section 2 and [23,8]. This formal basis for STS composition is helpful to implement a correct synchronization mechanism for STSs. We have previously done some experiments on translating formal behavioural specifications, namely LOTOS, in Java [9]. The code generation was based on a Java dialect providing condition activations and state notifications. It proposes a central monitoring mechanism with controllers for each node in the tree structure of the specification. In [11], we presented how to compound components with STS protocols, thanks to asynchronous communications links. The asynchronous communications are implemented with channels. Our current work extends these first proposals with a more precise
proposition to glue the STS protocol and the data type part and introduces the possibility for complex synchronization mechanisms between components with STSs.

In the concurrent object-oriented community, the use of explicit behavioural protocols at the language level is not new. PROCOL [14], SOFA [15] and Cooperative Objects [16] are three representative proposals. To describe protocols, PROCOL and SOFA employ regular expressions denoting traces – i.e. sequences of events (required, provided, and internal calls). Cooperative Objects employs Object Petri-Net-like notations for the same reason. Both formalisms are less readable and user-friendly than STSs. PROCOL and Cooperative Objects protocols consider data types and guards. SOFA and Cooperative Objects synchronous communications can be reduced to simple 1-1 RPC calls. PROCOL allows basically 1-1 communication, it separates message sending and service execution, and only message sending implies synchronization. The receiver waits for the message and, once received, the sender resumes and then the receiver executes the associated service. STSs composition semantics, as in LOTOS [22], allows one to express the synchronization of actions executed by one sender and several receivers. As far as we know, current object or component-oriented languages do not provide such a native synchronization feature.

A related work is [26] that provides methods to link Finite State Processes (FSP) and Java constructions. FSP is a recent process algebra originally proposed to design software architectures and is based on CSP. FSP can define constants, ranges, sets and simple data types like integers and strings. It also provides the classic construction to define processes and to compose them. The synchronization is based on the rendezvous mechanism and the common actions of processes. Important facilities of FSP are renaming of actions and a powerful notation for labels. FSP is a different model from STS for several reasons. The most important one is that FSP considers only finite state systems. The semantics of STS is based on configuration graphs which are not necessarily finite labelled transition systems as in FSP. Knowing that a system is finite is useful to generate the state space exhaustively; this is not generally possible with STSs which provide more general semantics. STSs also support unrestricted data types and the synchronization uses an external vector of synchronization and no explicit renaming. As in LOTOS, we provide the notion of guard with receipt (post guard in LOTOS) as a primitive mechanism. There is no direct support in FSP for this kind of guard, there are only classical guards. FSP does not provide an interpreter of process algebras but the LTSA book details the Java implementation of rendezvous: it is a synchronous message, thus it is more basic than our rendezvous notion.

JCSP is a pure Java class library designed by Welch and Austin and provides a base range of CSP primitives and a rich set of extensions, see [27] for more details. One main interest is that it conforms to the CSP model of communications and there is a long experience of tools and many practical case studies. The Java monitor thread model is rather easy to understand; however, it is more difficult to use safely as soon as examples are not small. Thus JCSP is indeed a safer alternative than the built-in monitor model of Java threads. The use of explicit shared channels is a simple way to synchronize processes. We have no explicit channel. Processes synchronize on any service execution – not only on read and write operations. Our prototype is not strictly based on CSP but may be viewed as an operational framework for a LOTOS like model of concurrency. Other differences are, as with FSP, support for full data types and guards with receipt. Our approach is oriented to the development of a true language supporting components, rather than a library for Java. One other important reason to quote this work is that it provides a CSP model for the Java thread model [28,29]. This formal model has been used to prove the correctness of a non trivial example. Thus we expect to reuse this model as one of the tools to prove that our rendezvous mechanism is correct. With the same purpose, CTJ [30] is another Java library based on CSP concepts with the additional property of providing support for real-time software via a built-in kernel. Both libraries provide access to the CSP model and have some similarities (see [31] for a comparison).
The aim of our work being to implement the STS synchronization mechanism, we need to define a complex synchronization structure, based on a more classic synchronization barrier. There are many algorithms to implement synchronization barriers. While we are interested in software implementation in Java, the two relevant references are [32,33]. In [33], the principles of these algorithms are explained and an overview of their cost is given. Several proposals are limited to two threads thus they are not sufficient enough for us. In [32], a precise analysis of the performances of several barrier algorithms are compared. The authors note that synchronized is needed to get a safe barrier, but that this feature and the wait-notify mechanism reduce performance. The wait-notify is a safe contention-free wakeup method, but it is slow compared to the Butterfly or the Static f-way barriers. Our basic barrier mechanism (Section 4.1) is fundamentally the same used, for example in [34], to synchronize an aspect with its base code. However, our approach differs from this, not only on the formalism used and the context, but also on the additional synchronization mechanisms presented here.

2. STS-oriented Component Model

Our component model is a subset of the Korrigan model described in [35,36]. This model builds on the ADL ontology [37]: architectures or configurations made of components with ports, and connections between component ports. The specifics we are discussing here are the use of Symbolic Transition Systems and the rendezvous semantics.

There are two categories of component: primitive and composite. We will present the description and the implementation principles of primitive components in the next section. Composite components are reusable compositions of components (i.e., architectures). In this paper they are reduced to a simple assembly of primitive components without entering in the detail of hierarchical decomposition of architectures. The runtime support for compositions of components is the main focus of Section 4.

2.1. Formal Definition of Symbolic Transition Systems

An STS is a dynamic behaviour coupled with a data type description. In our previous formal definition, we use abstract data type (see [8]). In this section, the data type part is described with an informal algorithmic language which is refined in Java code in the next sections. A signature (or static interface) $\Sigma$ is a pair $(S, F)$ where $S$ is a set of sorts (type names) and $F$ a set of function names equipped with profiles over these sorts. If $R$ is a sort, then $\Sigma_R$ denotes the subset of functions from $\Sigma$ with result sort being $R$. $X$ is used to denote the set of all variables. From a signature $\Sigma$ and from $X$, one may obtain terms, denoted by $T_{\Sigma,X}$. The set of closed terms (also called ground terms) is the subset of $T_{\Sigma,X}$ without variables, denoted by $T_{\Sigma}$. An algebraic specification is a pair $(\Sigma, Ax)$ where $Ax$ is a set of axioms between terms of $T_{\Sigma,X}$.

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

Events denote atomic activities that occur in the components. Events are either: i) hidden (or internal) events: $\tau$, ii) silent events: $l$, with $l \in L$, iii) emissions: $l(e)$, with $e \in T_\Sigma$, or iv) receipts: $l?x : R$ with $x \in X$. 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 synchronizing 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, \(\mu\) the guard, \(\epsilon\) the event and \(\delta\) the action. Each action is denoted by a program with variables. A do-nothing action is simply denoted by \(-\). In forthcoming figures, transitions will be labelled as follows: \([\mu] \epsilon / \delta\).

2.2. Connections and Synchronizations

A primitive component, for example the server component in Figure 1, is made of ports and a protocol described in the STS formalism. The STS has states and transitions between states. The general syntax of an STS transition is \([\text{guard}] \text{event} / \text{action}\), where guard is a condition to trigger the transition, event is a dynamic event (possibly with emission \(!\) or receipt \(?\)) and action is the action performed. An action corresponds to the call of a sequential operation. An event corresponds to the (external) notification of action execution. Ports are component connection points, each port externalizes the triggering of a given event in the STS protocol.

Connections are primitive bindings between ports rather than complex connectors. They denote synchronous communications between components. When ports are connected, their corresponding events are synchronized. Synchronizing several events means triggering them in any real order, but in the same logical time: this is the rendezvous principle. In case of communication (\(!\) and \(?\) events), the rendezvous takes place but the sender necessarily initiates a value computation which is communicated to receivers during the rendezvous. An STS of a primitive component already involved in a synchronization cannot trigger any other event during this synchronization. This rendezvous provides execution of actions of all the participants as well as a 1 to \(n\) communication.

This composition model proposes three ways for components to interact: (i) asynchronous activity: one component executes an action independently (i.e. without interaction), (ii) rendezvous without communication: \(n\) components execute a given action in the same logical time, and (iii) rendezvous: in addition to the latter case, a component emits a value
and the others receive it during the rendezvous. In this case, we consider that every receiver guard may check the emitted value, that is we have a guard with receipt (see Section 2.5).

### 2.3. Global Semantics

One way to define the global semantics of such a system is to compute the synchronous product of STSs [38] or the concurrent composition of processes [26]. These computations rely both on primitive component protocols and connections, so they can be automated from an architecture specification. They take as input STSs defining protocols and synchronization vectors defined by connections to produce semantic models. Fig. 2 shows the synchronous product of the three STSs in Fig. 1.

![Figure 2. The STS Global Product of the Ticket Protocol (s × p1 × p2)](image)

A synchronization vector is a vector of events that denotes a possible synchronization, at runtime, between a set of events. Synchronization vectors are computed according to the connections between component ports and defined according to an arbitrary ordering of primitive components. Each connection defines a given computation of synchronization vectors depending on connected ports. The three connections in the architecture of Figure 1 use the same communication operator. For example, one of these connections connects think ports of processes components with givet port of server component. It defines synchronizations that are binary between server and processes and exclusive between processes (denoted by the ⊗ symbol in Fig. 1). If we admit that possible synchronizations are denoted using synchronization vectors with ordering (s, p1, p2), then this connection produces two synchronization vectors: (givet, think, -), (givet, -, think). The - symbol is the stuttering notation to denote asynchronous (i.e. independent) activities of components. So, the resulting vectors express that the givet event of server s synchronizes with think event of process p1 or process p2, but not with these two think events in the same time. We have to notice that many different connections can be described to produce various computations of synchronization vectors.
Once all synchronization vectors are computed for a given architecture, they are used to compute the semantic model of the system, by combining STSs. Then, verification methods can be used to check the semantic model, but this is out of the scope of this paper. What we show in this paper, is that synchronization vectors are also useful for configuring runtime support of components.

Concurrent communicating components can be described with protocols modelled by STS and synchronous products, adapted from the LTS definition [38], can be used to obtain the resulting global system. Given two STSs with sets of event labels \( L_1 \) and \( L_2 \) and a set \( V \) of synchronization vectors, there is a set of pairs \((l_1, l_2)\), called synchronous events, such that \( l_1 \in L_1 \) and \( l_2 \in L_2 \). Hidden events cannot participate in a synchronization. Two components synchronize at some transition if their respective events are synchronous (i.e. belong to the vector) and if the event offers are compatible. Offer compatibility follows simple rules: type equality and emission/receipt matching. An event label \( l \) such that there is no pair in \( V \) which contains \( l \), is said to be asynchronous. Corresponding transitions are triggered independently.

**Definition 2 (Synchronous Product)** The synchronous product (or product for short) of two STS \( d_i = (D_i, (\Sigma_i, Ax_i), S_i, L_i, s^0_i, T_i) \), \( i = 1, 2 \), relatively to a synchronization vector \( V \), denoted by \( d_1 \otimes V d_2 \), is the STS \((D_1 \times D_2, (\Sigma_1, Ax_1) \times (\Sigma_2, Ax_2), S, L_1 \times L_2, s^0, T)\), where the sets \( S \subseteq S_1 \times S_2 \) and \( T \subseteq S \times T_{\Sigma_{\text{Boolean}}-X} \times (\text{Event}_1 \times \text{Event}_2) \times T_{\Sigma_{D}-X} \times S \) are inductively defined by the rules:

- \( s^0 = (s^0_1, s^0_2) \in S \),
- \((s_1, s_2) \in S, (s_1, \mu_1, \epsilon_1, \delta_1, t_1) \in T_1, (s_2, \mu_2, \epsilon_2, \delta_2, t_2) \in T_2, \) then
  - if \((l_1, l_2) \in V \) then \((s_1, s_2), \mu_1 \land \mu_2, (\epsilon_1, \epsilon_2), (\delta_1, \delta_2), (t_1, t_2) \) \( \in T \) and \((t_1, t_2) \) \( \in S \).
  - if \( l_1 \) is asynchronous then \((s_1, s_2), \mu_1, (\epsilon_1, \tau), (\delta_1, \text{Sel}_f_{D_2}), (t_1, s_2) \) \( \in T \) and \((t_1, s_2) \) \( \in S \).
  - if \( l_2 \) is asynchronous then \((s_1, s_2), \mu_2, (\tau, \epsilon_2), (\text{Sel}_f_{D_1}, \delta_2), (s_1, t_2) \) \( \in T \) and \((s_1, t_2) \) \( \in S \).

The synchronous product operator can be extended to an n-ary product and to any depth.

### 2.4. The Ticket Protocol Example

The example depicted in Figure 1 illustrates an architecture of primitive components with a mutual exclusion protocol inspired by the ticket protocol [5]. Processes and Server components are organized following a client-server architectural style. However, our version differs from the one in [5] since we deal with distributed components communicating by rendezvous, and not processes operating on a shared memory. We also distinguish entering (use event) and leaving (end event) the critical section.

In the example, there are six synchronization vectors computed according to connections between component ports: (givet, think, -), (givet, -, think), (gives, use, -), (gives, -, use), (end, end, -), and (end, -, end). Note that, whenever an event of a component does not occur in any synchronization vector, it is an asynchronous event, which can be triggered independently of others. Here, we note that processes p1 and p2 have asynchronous activities, either outside the critical section (activityOut) or inside it (activityIn). The server gives a ticket number to the process which memorizes it in its variable A. This synchronization step is represented by synchronization vectors (givet, think, -) or (givet, -, think), depending on respectively p1 or p2 enters the critical section. Then, to enter in critical section, the process p1 or p2 checks if its variable A is equal to the ticket S of the server. This synchronization step is represented by synchronization vectors (use, gives, -) or (use, -, gives), depending on respectively p1 or p2 enters the critical section. If all guards succeed, then the one process.
enters in critical section (state $T$). Then the process leaves critical section on the end event. This synchronization step is represented by synchronization vectors (end, end, -) or (end, -, end), depending on respectively $p_1$ or $p_2$ enters the critical section.

Figure 2 was calculated with our STS tool to illustrate the global behaviour of our example. The picture is simplified since actions are not depicted, but they may be easily inferred from the component STS. Note that something like $[<C==0.A==S.->]$ is a compound guard expressing that $s$ and $p_1$ evaluate their guards while $p_2$ has a default true guard. The same thing applies for the compound events which glues three events each one coming from a component. The reader may see that processes have asynchronous activities which are expressed by transitions like $<-.activityOut.->$ or $<-.activityIn.$ The semantics provides concurrent composition of components with event synchronizations, namely the rendezvous notion introduced by CSP [25]. This synchronization mode is not generally what we find in programming languages, for instance in the PROCOL, SOFA or Cooperative Objects approaches. Thus to relate the formal level with the operational one we want to implement the concurrent composition of STSs. This construction takes several STSs and the synchronization vectors which link events of the input STSs.

2.5. Guard with Receipt

One reason to introduce the ticket example is that it shows a complex communication with guarded receipt during the (gives, use, -) or (gives, -, use) synchronization. Guards with possible receipt is an important construction with a specific semantics: components can conditionally receive and synchronize on a value in the same logical time. They correspond to post guards in the LOTOS language. One benefit is to increase abstraction and reduce the size of the finite state machine. Note that, in such a communication, the emitter must have a guard without receipt.

Some translations of guarded transitions are possible. The $[A=S] \ ? \ use S:int$ transition of the STS process has a guard with receipt and no action. This complex transition may for example be split into three steps: a receipt, a guard checking and a null action. However this decomposition should be used with care since in place of a single event we get a sequence of three events. In other words, hiding for instance the guard checking is not preserving the observational semantics (it is only a strict behavioral abstraction). From a practical point of view, the consequence on the synchronization mechanism is that when a rendezvous occurs the sequences of these three steps have to be synchronous, not only one of them. This last point raises a major implementation issue to keep the model semantics and components execution consistent.

3. Model Implementation Overview

In this section, we detail our hypotheses related to the description of primitive components in Java. A global picture of intra-component implementation is depicted in Figure 3. It represents the different elements defining a primitive component.

In the component model of the Korrigan formal ADL, the finite state machine notations are mixed with the data type part description. This is convenient when we want an integrated model suited for verification purposes. However for the operational side, we think that it is better to separate the finite state machine and the data part. This simplifies a little the implementation and, moreover, separates the two aspects which makes the implementation more reusable. For example, we can reuse a given state machine with another data type implementation provided that some compatibility rules are ensured. The Java representation of the finite state machine is thus reduced to the states, the transitions and some names. These names represent the guards, the events, the receipt variables, the senders and the actions. The
data part is a Java class implementing the formal data type part. The exact role of the class is to give a real implementation, with methods, of the names occurring in the state machine part. Thus, both parts are glued thanks to a normalized Java interface which is automatically computed from the STS. An emitter is a pure function computing the emitted value in a given state of the component. Similarly, a guard is a boolean function implementing a condition.

So, in our current scenario, a primitive component results from the combination of a protocol and existing Java code (henceforth referred to as the data part), more precisely, a passive Java class implementing a specific Java interface. Each primitive component is implemented with an active object (thread in Java) in charge of both the STS protocol execution and the call to the passive object implementing the component data part. We choose to rely on an active object since it may simulate a passive one (a usual class), the reverse being false. Thus from now on, an STS defines the events, guards, emitters, and actions names related to the Java interface of the data part class.

The code may be either automatically generated from an explicit and formal description ([9,10]) or provided by the user writing some programs or reusing some classes. One important issue is the compatibility or coherence between the data part intrinsic protocol (i.e., the execution protocol) and the externally defined STS protocol. One way to address this issue is to provide a method that extracts a compatible data type from the STS description [9,24]. Another way is to develop the data part and the protocol separately and then to check compatibility between both parts. Behavioural compatibility has been addressed in process algebra [39] and in state machine [38,40] approaches. There exists related work on component compatibility (for instance [2,41]). We assume to rely on the technique presented in [41] which is compatible with the STS behavioural semantics. As an example, a Java interface and a Java class compatible with the STS process presented are described in Figures 4 and 5.

```java
public interface IProcess {
    public void think (int T);
    public boolean check (int S); // check for guard (A == S)
    public void use (int S);
    public void end ();
}
```

Figure 4. Java Interface for the Process STS
public class Process extends Data implements IProcess {
    protected int A;
    public Process () {
        this.A = 0;
    }
    public void think (int T) {
        this.A = T;
    }
    // guard with receipt
    public boolean check (int S) {
        return this.A == S;
    }
    // use action with receipt
    public void use (int S) {
        System.out.println("Enter critical section");
    }
    public void end () {
        System.out.println("Leaving critical section");
    }
}

Figure 5. Java Class for the Process STS

Figure 6 presents the translation rules for emission and receipt labels. Note that, in case of receipt, the guard and the action signatures of the receiver transition have to accept the received argument. However the methods have the possibility to forget this parameter if useless. Formally, the syntactic compatibility between the STS label information and the Java interface can be checked on the basis of the rules presented in Figure 6. The syntactic compatibility between the Java interface and the data part class follows the Java 1.5 type checking rules. In Figure 6, guard, action and emitter are Java method identifiers, var is a Java variable identifier and Type is a Java type identifier.

Architecture or component assembly relies on primitive and composite components and a glue mechanism to synchronize them. A direct way to compose properly components is to build their synchronous product according to synchronization vectors. This product represents the global execution of the component interactions as a global STS (e.g. Figure 2) and a compound data part may be built from the subcomponent data parts. However, one important drawback of this solution is the computation cost of the synchronous product, which is exponential in the size of the STS components. Another problem is that the resulting application will be centralized if we consider the global STS as a primitive component’s STS, since it will be executed on a single active object. Lastly, although this provides an equivalent simulation (as with [9]) of the compound system, the original components are not really reused.
That is why we choose to implement the concurrent composition of STSs. This construction takes as input several STSs and synchronization vectors that bind their events. It configures STS runtime support in such a way that STS execution conforms to the semantic model. The direct consequences are that each STS has its own execution thread and that all STSs have to be synchronized depending on synchronization vectors. In this implementation, a primitive component corresponds, at runtime, to a unique thread and a composite component corresponds to a collection of interacting threads. Synchronization of threads is supported by a specific rendezvous mechanism, presented in the next section.

4. A Java Implementation of Rendezvous

In this section, we present the principles to implement our rendezvous mechanism for Java components with STSs. While our solution is a general one we suggest to use it only in local networks since it may be a bottleneck in wide area networks due to communication delays. In wide area networks, asynchronous communications have to be used in place of synchronous communications [13]. Nevertheless this latter communication mode can be implemented by synchronous communication and channel or intermediate component, but this is out of the scope of this paper. We choose to implement our proper rendezvous mechanism in Java 1.5 using monitors. The two other alternatives were the join method or the CyclicBarrier class. Technically, when using the join method, threads are exiting and we need to start new ones; handling persistent state for data is more complex. A second remark is that the implementation of the rendezvous would require constructions similar to those we introduce later to cope with guards and communications. The CyclicBarrier seems to be a perfect candidate to synchronize the threads associated to our STSs. However, the problem is still the implementation of guards which are conditions to enter into the synchronization barrier. One may have one thread which is waiting on the barrier and another one which cannot since its guard is false. Thus we have to check all the involved guards before reaching the barrier. Except for the exiting barrier, the use of the CyclicBarrier does not really simplify our implementation. Lastly, we need to know precisely the synchronization mechanism since this first approach will be optimized later. In the following subsections, we present the implementation of component runtime in four progressive steps, from a simple barrier to the rendezvous with receipt on guards.

4.1. The Basic Barrier Principles

The basic mechanism described in this subsection is nearly the same as in [34]. In this first setting, a mechanism was implemented to synchronize LTS. As in FSP [26], a synchronization is possible between two actions if they have the same name. A central object, the arbiter, controls that synchronizations are correctly handled. The principle is to use a Java monitor to implement two synchronization barriers. Note that one synchronization barrier is generally not sufficient enough to ensure the correct rendezvous between actions. With only one barrier, an asynchronous action of an STS may be triggered in the same logical time as a synchronous action of another component. This would be inconsistent with the STS composition semantics. The right solution requires one barrier for entering in the synchronization area and another one for all participants to leave it.

Figure 7 gives the static class diagram of the solution. Actions and states are encoded by integers. An LTS\(^1\) is encoded with a list of actions and a matrix. In this matrix, for each state we have a vector (indexed by actions) of the target states. The LTS has also a reference

---

\(^1\)These details of implementations are provided to give to the reader an understanding of the synchronization mechanism. However, in the real implementation, things are much more complex and based on hash mappings.
to an Arbiter instance. The LTS class is active by implementing the Runnable interface and owning an instance of class Thread. The run method evaluates (using eval) an action until the thread is interrupted or the LTS reaches a state without outgoing transitions. The eval method of the LTS class checks whether the transition is synchronous. If the action is asynchronous, the LTS evolves independently of others. If synchronous, the arbiter is called with a synchronizeOnEntry for this action. Then, currentState is updated with the target matrix and the arbiter finishes the rendezvous with a synchronizeOnExit call.

The arbiter is a shared passive object which is called to synchronize actions. Its sync-ValueNumber variable defines, for each synchronization, the number of actions (and consequently the number of LTSs) to synchronize. The counter variable defines, for each synchronization, the number of LTS that have passed the entry barrier and wait for others LTS involved in the synchronization. The entry and exit barriers are implemented with two synchronized methods. The code of the entry barrier is shown in Figure 8, the synchronized qualifier ensures that only one thread is executing this call. The exit barrier has a similar implementation.

```java
synchronized public void synchronizeOnEntry (int action) {
    if (counter[action] < syncValueNumber[action] - 1) {
        counter[action]++;
        try {
            wait();
        } catch (InterruptedException e) {
        }
    } else {
        counter[action]=0;
        notifyAll();
    }
}
```

Figure 8. The Synchronization Barrier

All synchronized LTSs query the entry barrier in any order and their supporting threads then wait. When the last LTS queries the entry barrier, all threads are woken (notifyAll) and the synchronization counter is reset to 0. Then, all LTSs concurrently execute their respective actions. When an LTS ends its action, it queries the exit barrier and then waits. When the last LTS queries the exit barrier, all threads are woken and all LTSs can continue their execution independently: the synchronization of LTSs’ actions is ended. Actions require the barrier in any ordering and have to wait before starting to execute their actions until the last action is also ready to synchronize.
Often, it is recommended to enclose the `wait` in a `while` loop, see [26] for details. Here, this is not needed since, once awakened, a sleeping thread simply exits from the barrier. However we have to follow the Java documentation of the `wait` method, "waits should always occur in loops" because of the "spurious wakeups". Our implementation also takes into account protocol non-determinism by simulating a random choice of actions. It is not too difficult to write such a solution. However, we have to minimize the number and the size of the synchronized parts to increase concurrency between threads, while getting a correct solution.

4.2. Synchronization Vectors Representation

A first improvement is to relax the restriction on names for synchronization. Often design and component languages do not decouple the behavioural description from the communications, for instance PROCOL, FSP or UML. To reuse components, they have to be synchronized into various environments and there is no reason for port naming to be a global knowledge. To fight against name mismatch, the two classic solutions are renaming (as in FSP) or component adapter. We think that a solution based on synchronization vectors is most general since it does not need code modification or any additional programmable entity (i.e. adapter).

![Figure 9. Partial UML Class Diagram](image)

In this new setting, a set of synchronization vectors (cf. Section 2) is declared, each one representing a possible synchronization between some component events. An event name and an action name are associated inside a transition (class `Transition` in Fig. 9). A synchronization vector, denoting a set of synchronous events, indirectly defines a set of synchronous actions. The `LockSync` class, that represents a synchronization vector, is then introduced to the diagram. The synchronization barrier methods are moved from the `Arbiter` to this new class, and there are now two barriers for each synchronization vector.

The `eval` method is also changed. It first asks the arbiter to get the `LockSync` instance which concerns the current action. It uses the `isSynchronous` method to choose one `LockSync` object. Then a `synchronizeOnEntry` call is made and returns a boolean indicating if entering in the barrier succeeds or fails (see Fig. 10).

The first thread entering in the barrier must process two specific tests, `isPossible` and `isFree`, which are implemented in the `Arbiter` class. The `isPossible` method checks if a synchronization can occur from the current global state. Method `isFree` tests if a synchroniza-
4.3. Independent Synchronizations

One may observe that, in the basic barrier, two distinct synchronization entries or exits are always serialized since there is a single arbiter and the methods to enter and leave the barriers are synchronized. The definition of the LockSync class is a first attempt to break this centralized control.

The conflict of a synchronization is defined as the set of synchronizations which synchronize on, at least, a common component. In our example, all the synchronizations are mutually conflicting because of the central server component. A synchronization is independent from another one if it does not belong to its conflict set. The improvement, here, consists in implementing the conflicts set (class Conflict in Fig. 9) of each synchronization and to allow two (or more) independent synchronizations to enter simultaneously in the barrier (or to leave it).

We define a Flag class which contains, for each synchronization, a reference on the corresponding synchronization counter and a boolean (access) representing the possibility of accessing this counter (cf. Fig. 7). Two methods (freeze and relax) are defined with the synchronized qualifier; they have the responsibility to implement exclusive access to the vector of shared counters, by testing and setting the value of the access attribute. Now the isFree method is no longer synchronized. It tests if conflicting synchronizations are not already entering a barrier and, if OK, allows the current synchronization to proceed. The first thread entering the barrier – and only this thread – has to freeze the counters conflicting with...
the current synchronization; then isFree is called and, finally, the counters are released (cf. Fig. 10). This current solution minimizes the bottleneck access to the vector of counters with two small methods, defined as tests and assignments on a vector of boolean objects.

4.4. Guards and Communications Management

Since STS transitions are more complex than those represented until now, we need a richer class diagram to manage STS properties not already taken into account. Classes Guarded, Emission and Receipt are defined to represent the corresponding transitions (see Fig. 9). An abstract class Data contains methods, based on the reflexive Java API, to execute guards, emitters and actions on an instance. This class is specialized by the specific data part class of each component (cf. Fig. 3). The method run tries to trigger a possible transition if there is one. There is no busy waiting loop to check the guards, they are evaluated only when needed at the entry in the synchronization barrier.

The management of communication has to be implemented to conform to the STS model (cf. Section 2). Since there are guards with receipt, communications have to be evaluated before any synchronization and even before checking guards. Furthermore, all guards related to all synchronized actions have to be checked before the execution of these actions. So, the eval method is modified to manage synchronous actions with communication, in addition to the two previous cases (asynchronous actions and synchronous actions). A synchronous action with communication is initiated by the first thread entering the barrier, which is necessarily the sender. The local guard of the emitter transition (if any) is checked and the emitted value is then computed (see Fig. 10). The call to synchronizeOnEntry is then performed with the value communicated to the LockSync object (setEmittedValue(v) in Fig. 10). This object is an instance of the LockCom class (that specializes the LockSync class to introduce a specific version of the entry barrier for the communication case). In addition to LockSynch operations, it realizes a checkGuards method call to check if the guards associated with a synchronization vector are true, coping with value communicated to other STSs. The eval method of the STS class also retrieves the communicated value (getEmittedValue("use") in Fig. 10), to perform the execution of synchronized actions that use this value as argument.

4.5. Final Comments

The previous implementation provides an interpreter supporting rendezvous and allowing dynamic changes of STSs, data parts or even components (obviously with some care in stopping and restarting components). The current discussion is mainly directed to get a correct barrier with complex synchronization conditions allowing receipt on guards. Efficiency has been taken into account in two ways: distributing the central arbiter in several sets of objects (locks, conflicts and flags) and minimizing the synchronized parts. The guard checking, the emission computation (if needed) and freezing the flags are only done by the first thread that enters the synchronization barrier.

In this interpreter version, reflexivity is used to glue protocols and data parts. In the compiler version, protocols will do direct call to the data parts methods. Note also that exception handling, barrier delays and RMI have to be integrated to get a true usable system. The current version relies on a “wait and notify barrier”. An optimization is to use result from [32], for instance, to replace it with a Static f-way barrier. However, a major problem will be the distribution of the shared objects and the limitation of remote communications. We have also to fight against the global synchronization problem (see [42]). Here, we partially addressed this problem with the introduction of conflicts and locks and we will feature the balance between synchronous and asynchronous communications. A more comprehensive analysis has to provide a solution that scales up to wide distributed systems.
5. Conclusion and Future Work

In this paper, we provide a mechanism to synchronize components with protocols. We consider complex protocols, namely symbolic transition systems, with full data types, guards and communications. We allow non-determinism in the protocols and we provide a flexible naming notation to define event synchronizations. One original and powerful feature is the possibility to define conditional rendezvous taking into account the communicated values. These protocols are adequate for the formal specification of systems and our approach gives a means to execute them – thus relating verification and execution of component systems. We describe an implementation of a complex rendezvous based on two synchronization barriers, each of them implemented with the monitor and \texttt{wait/notifyAll} facilities of Java. One delicate thing is synchronization in presence of communications and guards. We show how to proceed in four steps to get a correct solution. This solution is general in the sense that we do not constrain the ordering of processes to enter, execute their action and leave the critical section. We also propose a first optimization to allow several independent synchronizations to process the barrier. This is a first way to distribute the central arbiter mechanism used to synchronize the components. Currently, this work provides an operational interpreter to program primitive components in Java with STSs and a powerful way to compose them.

Until now we have done tests, implemented various small and middle size examples and checked with our verification tool some specific parts of the mechanism. We have also implemented a dynamic check which verifies that events generated by the runtime are according to the synchronization rules and compatible with each running state machine. This defines a dynamic checking which is able to alert the user if some synchronizations are not correct and if state changes or transition triggering are not occurring at the right moment. While this checking is useful it is not sufficient to prove that our mechanism respect its specifications.

One thing we would prove in our future work is the correctness of the solution. First, we already reused the work of [28,29] which gives a CSP view of the Java monitoring mechanism. Rather than a CSP view, we get an STS description of the mechanism. We model the simple barrier with our STS tool and we try to do verifications on some simple examples. We are able to verify that the mechanism allows a correct entry and exit in the rendezvous area, but with only LTS behaviour. One result of this was a simplification of the two barriers which are the base of our actual mechanism. This was a first step, the second, yet future, is to design the full mechanism with STSs integrating the guard and communication mechanisms. We have also to model the locks and flags features, but these are passive objects. Then we will prove that, from a temporal logic point of view, our two barriers define an area of synchronization. That is a logical time area where synchronous actions occur inside (in any ordering) and synchronous components have no other activities. Last, our locks and conflicts own the following properties: (i) two different threads with a same synchronization vector cannot compete for entering the barrier since the \texttt{synchronizeOnEntry} is a synchronized method, and (ii) two different threads with different synchronization vectors can simultaneously start an area of synchronization iff the synchronizations are not conflicting. We think that it is sensible to get a full manual proof, however our STS tool will be used to check some examples. One final improvement will be to translate our specifications into PVS (see [24] for a related work) and to run the manual proof.

Future work will also consider the definition of a Java based language with STS, asynchronous and synchronous communications. We have to make precise the compilation mechanism as well as some optimization aspects. Amongst these, we expect to propose a solution to choose automatically between passive and active object implementations. Another feature is to elaborate a splitting mechanism for the central flags based on the analysis of synchronizations and communications in the deployed architecture.
References


