Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 215 (2008) 131–149
www.elsevier.com/locate/entcs

The STSLib Project: Towards a Formal Component Model Based on STS [1]

OBASCO project
École des Mines de Nantes-INRIA, LINA
4 rue Alfred Kastler, 44307 Nantes cedex 3, France

Abstract

We present the current state of our STSLib project. This project aims at defining an environment to formally specify and execute software components. One important feature is that our components are equipped with a protocol description, namely a Symbolic Transition System. These descriptions glue together a protocol with guards and input/output notations and a data type part. These sophisticated protocols are well-suited to the design of concurrent and communicating systems but verification remains a difficult challenge. We expect to narrow the gap between the design level and the programming level by providing a runtime support for STS. We give in this paper the main objectives of the STSLib project and overview its current state. We address the formal description of a component model, a specific approach to verify these systems and a survey of the operational level to execute them. These features are illustrated on a cash point case study.

Keywords: Software component, Behavioural protocol, Symbolic transition system, Verification, Java code generation.

1 Introduction

Software engineering is still evolving in several directions. The first direction is to provide a better modularization and a separation of concerns. Examples are the numerous works around software architecture, component based programming and aspect oriented programming. A second preoccupation has been

[1] This work was partly supported by the STREP AMPLE project (`www.project-ample.net`) and the CAPES program from Brazil.
[2] Email: `Fabricio.Fernandes@emn.fr`
[3] Email: `Jean-Claude.Royer@emn.fr`

doi:10.1016/j.entcs.2008.06.025

to provide formal semantics to models and programming features as well as verification means. Associated to this, there is the need for tools and automation when possible. A perfect illustration of these trends is trusted software components [18]. Using the terminology of this paper, we are particularly concerned with the *high road*: "the high road is intended to lead components with fully proved properties. The ambition of this goal implies that it's more long-term, and that its realization must start with relatively fine-grain (but practically critical) components such as library of classes."

Following this road, we are focusing on the formal specification of concurrent software components and the generation of Java code from these specifications. Furthermore, we consider mixed specifications that are mixing protocol and data type descriptions. The STSLib project aims at providing a framework to define formal components. The general objective is to define a Java tool support allowing the formal design of software components and their execution. We aim a powerful and concise formalism for both dynamic behaviour (control, concurrency, and communications) and data part with precise semantics. We expect a formalism between process algebras with values [15,7], but in a more visual way, and UML Statecharts, but simpler and more rigorous. We need to connect this formalism with usual verification means (general prover and model-checkers) but also to develop complementary ways to check the specifications. We require a Java code translation which would be as automated as possible, furthermore a real runtime support not only a specification simulator. One first feature is the formalism we use for primitive components, which is a mix of protocol description and algebraic data type: the Symbolic Transition System notion [23] (or STS for short). We develop a specific way to check the components based on an extension of the synchronous product and the configuration graph computation. This is applied to a cash point case study and we describe some results related to our verification experiments. Currently, we are also defining a Java interpreter to execute the component descriptions. Component data parts are translated into imperative Java code and a runtime support implements a n-ary rendezvous allowing the synchronization of primitive and composite components.

The outline of this paper is as follows. Section 2 will present related work. Section 3 is devoted to an introduction of the STS formalism, the synchronous product, and the configuration graph. The STS notion is illustrated with the till component of the cash point case study. Section 4 describes some experiments we have made about the verification of this case study. Section 5 shows the implementation principles to define a real Java runtime support for our components. Lastly we conclude and discuss future work.

2 Related Work

Related work may be classified into component programming languages, environments dedicated to the formal specification of concurrent systems, Java related tools or libraries and automatic translation into Java programming source code.

Java/A is an architectural programming language, which generates Java code [5]. It may be viewed as a step ahead from ArchJava [1], which defines an architectural extension of Java. Java/A integrates the notions of component, required and provided interface, port, connector and has some means to verify component assemblies. One important feature is that they have protocols inside ports which express the ordering of messages. The language is equipped with tools: namely a compiler and a model-checker for protocol consistency. The authors give a formalization of the abstract component model in terms of transition systems and states as algebras. They also prove a consistency result for assemblies, which provides the basis for reasoning on assemblies and port compatibility. The semantic model uses a states-as-algebras approach for representing the internals of components and assemblies, and I/O-transition systems for describing the observable behaviour. The semantics do not distinguish between simple components and composite components. They are both a component, which is defined through its internal algebraic state space, and the declaration of the ports it offers. In our case, we can distinguish the semantics of a composite from the semantics of a simple component, but we can also abstract away from this structural information. Our model supports some simple forms of dynamic reconfiguration but Java/A fully supports dynamic port changes. The behaviour of a component is given by an I/O-transition system and a state operator maps each control state to a data state which is an algebra over the state signature of the component. The labels of the transitions are either the internal label or I/O-labels corresponding to the messages sent and received via ports. One important difference with STS is that we have guards, which are not yet allowed in Java/A and this complicates the semantics models and the checking. However, due to the close relation between both semantic models it is possible to adapt the results of Java/A to our context but removing at least the guards.

Other related component references are: [12,4,16]. [12] provides a finite labelled transition model for behavioural interface of components and an automatic way to check their compatibility. The originality is that they consider an optimistic hypothesis and redefine the way to compose dynamic systems. In [16], the authors propose a way to model-check Java components by extracting a model of its environment. This can be seen as a variation of compositional model-checking but verifying specific properties of individual compo-

nents. The behavioural model of [4] is a subset of the STS model since it has only restricted data types and assignment actions. Additionally it provides a model to encode dynamic configuration of components and this is applied to ProActive Java code. However, none of these references provide tool support from specification to code.

CADP [14] ("Construction and Analysis of Distributed Processes") is considered a good representative of verification tools. CADP is a toolbox for the design of communication protocols and distributed systems based on the ISO language LOTOS. The CADP tool box provides a rich set of tools: equivalence checking, model-checkers for various temporal logics and mu-calculus and verification algorithms (enumerative verification, on-the-fly verification, symbolic verification using binary decision diagrams, etc). An interesting feature is that the tool box allows specification simulation (CAESAR tool). This simulation is based on C code for the data part and Petri nets for the concurrent and synchronization parts. Our current environment do not address classic model-checking, we rather propose some complementary tools and approaches. Another difference is that we do not provide a simulator but a real framework to execute software components automatically built from formal descriptions.

Considering Java source code generation, the constructive approach of Coglio and Green [10] is relevant. In this paper, the authors show that a constructive approach, generating code from specifications, can be a valuable alternative to usual code verification. The usual way is to verify legacy code by a post-hoc method of proving certain properties, or possibly functional correctness. But the combinatorial difficulty of a post-hoc approach has generally prevented the community from being able to prove full functional correctness, *i.e.* that the program actually does what it is intended to do. The goal of the authors is to provide a proof of functional correctness of the code with respect to its specification. The automated generation of such a proof, along with the code, is guided by the availability of the code generation/design process. A specification-first approach is made and uses user-friendly domain-specific notations, which simplifies code and proof generation. However, the input specification language is domain-specific and precludes certain features found in general-purpose languages (*e.g.* no recursion, no concurrency). The important point is that the target Java Card language is a strict subset of Java. In our case, concurrency is an essential feature.

3 The Component Model

Our current component model is a subset of the KADL model described in [8,21]. This model builds on the ADL ontology [17]: architectures or configurations are made of components with ports, and connections between component ports. We call our structures components but more precisely we are describing component types that can be instantiated. The features we are discussing in this paper are the definition and the use of symbolic transition systems to model software components both at the specification and programming level. KADL provides a rich set of modal operators to synchronize components but in STSLib we restrict it currently to a core subset. There are two categories of components: primitives and composites. A primitive component is described by means of an STS and composite components, reusable compositions of components (*i.e.* architectures), represented as communication diagrams. We first give the formal definitions of STS, configuration graphs and synchronous products.

3.1 Formal Definition of Symbolic Transition Systems

An STS is a dynamic behaviour coupled with a data type description. An Algebraic Data Type (ADT for short) is given for each STS, and transitions from this STS use the operations defined in the ADT. The operations semantics are described using algebraic axioms. Algebraic specifications abstract concrete implementation languages such as Java, C++, or Python (more details about these notations may be found in [3]). A *signature* (or static interface) Σ is a pair $(\mathcal{S}, F)$ where $\mathcal{S}$ is a set of *sorts* (type names) and F a set of function names equipped with *profiles* over these sorts. If R is a sort, then Σ_R denotes the subset of functions from Σ with result sort being R, Σ_R^D will be the subset of functions which have at least the D data type as first parameter type and whose resulting type being R. X is used to denote the set of all variables. From a signature Σ 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_Σ. An *algebraic specification* is a pair (Σ, Ax) where Ax is a set of axioms between terms of $T_{\Sigma,X}$.

Definition 3.1 [STS] An STS is a tuple $(D,\ (\Sigma, Ax),\ S,\ L,\ s^0,\ T)$ where: (Σ, Ax) is an algebraic specification, D is a sort called *sort of interest* defined in (Σ, Ax), $S = \{s_i\}$ is a finite set of states, $L = \{l_i\}$ is a finite set of event labels, $s^0 \in S$ is the initial state, and $T \subseteq S \times \Sigma^D_{Boolean} \times Event \times \Sigma^D_D \times S$ is a set of transitions.

Events denote atomic activities that occur in the components. Events are either: *i*) the stuttering event noted $-$, *ii*) hidden events: τ the internal event, and l, with $l \in L$, *iii*) silent events: l, with $l \in L$, *iv*) emissions: $l!e$, with $e \in \Sigma_R^D$, or *iv*) receipts: $l?x : R$ with $x \in X$. An event label l may be exclusively a hidden, a silent, a receipt or an emission event. The stuttering event is used to denote asynchronous activity of the component in case of concurrent composition. This stuttering event is not explicitly used by the user and occurs only in transitions $(s,\ true,\ -,\ Id_D,\ s)$, where *true* is the boolean constant function and Id_D is the do-nothing action. 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 receipts allow value communications. STS transitions are tuples $(s,\ \mu,\ \epsilon,\ \delta,\ t)$ for which s is called the source state, t the target state, $\mu \in \Sigma^D_{Boolean}$ the guard, ϵ the event and $\delta \in \Sigma_D^D$ the action. In forthcoming figures, transitions will be labelled as follows: $[\mu]\ \epsilon\ /\ \delta$.

3.2 Configuration Graphs

The semantics of STS is formalized using configuration graphs. They are obtained applying jointly the unfolding of receipts and the reduction of ground terms to their normal forms. Our model assumes that normal forms exists and are unique and they are noted with the $\downarrow$ operator.

Definition 3.2 [Unfolding] The unfolding of an STS $(D,\ (\Sigma, Ax),\ S,\ L,\ s^0,\ T)$, in $v^0 \in T_{\Sigma_D}$, is the STS $(D,\ (\Sigma, Ax),\ S',\ L,\ (s^0, v^0\downarrow),\ T')$. The sets $S' \subseteq S \times D$ and T' are inductively defined by: $(s^0, v^0\downarrow) \in S'$ and for each $(s, v) \in S'$:

- if $(s,\ true,\ -,\ Id_D,\ s) \in T$ then $(s',\ true,\ -,\ Id_D,\ s') \in T'$, where $s' = (s, v)$,
- if $(s,\ \mu,\ l,\ \delta,\ t) \in T$ with l a hidden or a silent event and $\mu(v)\downarrow = true$ then $s' = (t,\ \delta(v)\downarrow) \in S'$ and $((s, v),\ true,\ l,\ Id_D,\ s') \in T'$,
- if $(s,\ \mu,\ l!e,\ \delta,\ t) \in T$ and $\mu(v)\downarrow = true$ then $s' = (t,\ \delta(v, e(v))\downarrow) \in S'$ and $((s, v),\ true,\ l!e(v)\downarrow,\ Id_D,\ s') \in T'$, and
- if $(s,\ \mu,\ l?x : R,\ \delta,\ t) \in T$ then for each $r : R$ such that $\mu(v, r)\downarrow = true$, there is $s' = (t,\ \delta(v, r)\downarrow)) \in S'$ and $((s, v),\ true,\ l!r,\ Id_D,\ s') \in T'$.

Pairs (s, v) are *configurations* where s is the *control state*. Let d be an STS, its unfolding in a v^0 term, $G(d, v^0)$, is called a *configuration graph*. A configuration graph is a particular STS without receipt, where guards are all equal to *true*, emission terms are in normal form and actions are Id_D.

3.3 STS *synchronous product*

We extend the synchronous product originating from [2] to incorporate with STS. A synchronization vector is a tuple of ports, one for each component, which denotes a synchronization between the transitions associated to the port events. Note that hidden events cannot appear in a synchronization vector. The special stuttering event $(-)$ is used when a component does not synchronize. Two components synchronize at some transitions if their respective labels are synchronous (*i.e.* belong to the vector) and if the *label offers* are compatible. Offer compatibility follows simple rules: type equality and emission/receipt matching.

Definition 3.3 [Synchronous Product] The synchronous product (or product for short) of two STS $d_i = (D_i,\ (\Sigma_i, Ax_i),\ S_i,\ L_i,\ s_i^0,\ T_i)$, $i \in \{1, 2\}$, 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 $T \subseteq S \times (\Sigma^{D_1}_{Boolean} \times \Sigma^{D_2}_{Boolean}) \times (Event_1 \times Event_2) \times (\Sigma^{D_1}_{D_1} \times \Sigma^{D_2}_{D_2}) \times S$ are inductively defined by the rules:

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

As the reader may see, this product defines an STS with pairs of states and pairs of events. This synchronous product can be extended to a n-ary product and to any depth. The extension of the synchronous product of automata contains in the result the structure of the composite. Thus, we not only have states and transitions, but composite states, composite transitions, composite events and so on. This is valuable to get an exact understanding about the events and the conditions occurring in a complex system.

3.4 The Till Specification

To illustrate our model, we will consider the FM'99 cash-point service benchmark [26]. The system described in this paper is composed of several *tills*, which can access a central resource containing the detailed records of customers' bank accounts. A *till* is used by inserting a card and typing in a Personal Identification Number (PIN), which is encoded by the till and compared

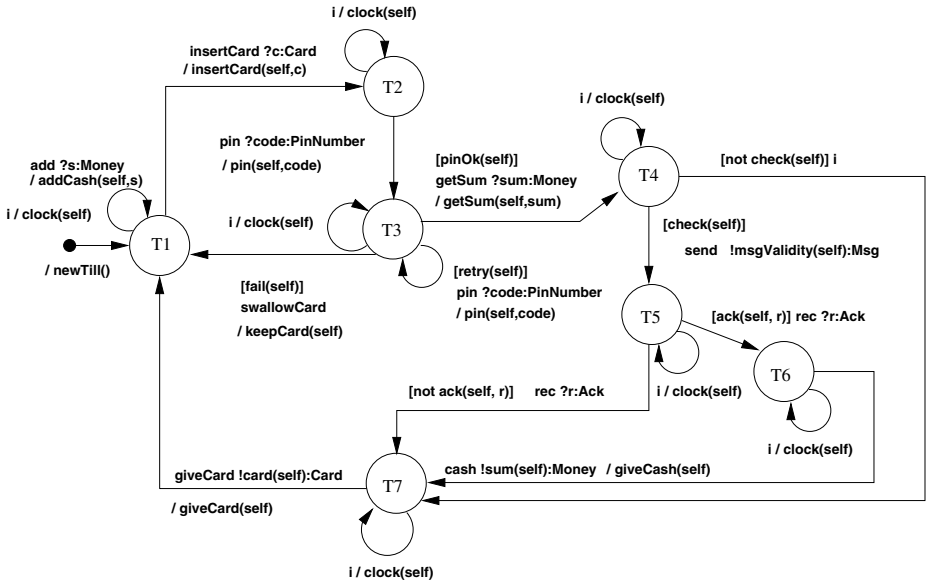

Fig. 1. Symbolic Transition System (Dynamic Part) of the Till.

with a code stored on the card. After successfully identifying themselves to the system, customers may make a cash withdrawal. Here we only present the till parts. A comprehensive report in [22] provides the full KADL specifications, verification results and a full LOTOS specification. A component interface defines the visible ports of the component. As an example, the ports of the till are: `insertCard ? Card` to insert the user card, `giveCard ! Card` to eject the card, `pin ? PinNumber` to enter the PIN, `getSum ? Money` to enter the desired cash amount, `cash ! Money` to get money, `add ? Money` to allow an operator to add money to the till available amount, `rec ? Msg` to receive a message from the connection, and `send ! Msg` to send a message. The dynamic behaviour of the till is depicted in Figure 1. A transition such as `cash !sum(self) : Money / giveCash(self)` means that the till emits a sum of money and during this transition, the `giveCash` operation updates the information of the till data type. Some axioms of the `Till` data type are given in Figure 2. We do not give the Till interface and the full ADT here due to space limitations.

STS are (possibly non-deterministic) symbolic labelled finite transition systems, which have appeared in various forms in the literature [15,7]. STS provide an expressive and abstract means to describe dynamic behaviour symbolically. The main interest with these transition systems is that (i) using receipt variables and guards in transitions, they control the system size and shape, and (ii) using an open term in states (`self`), they define equivalence classes (one per state) and hence strongly relate the dynamic and the static

(algebraic) representation of a data type.

```
VARIABLE a,sum:Money; c:Card; code:PinNumber; today:Date ; cpt : Natural; self:Till
AXIOM

        # generator for till
newTill : Money Card  PinNumber  Money  Date  Natural -> Till

       # card selector
 card : Till -> Card
>> card(newTill(a,c,code,sum,today,cpt)) -> c

        # guard to check PIN code
fail : Till -> boolean
>> fail(self) ->
     and(not(equals(crypt(code(self)), codecard(card(self)))), supLarge(counter(self), three))

giveCash : Till int -> Till
>> giveCash(newTill(a, c, code, sum, today, cpt), sum1) ->
     newTill(sub(a, sum), updateDailyLimit(card(self),sum,today),code,sum,today,cpt)
```

Fig. 2. Some Axioms and Signatures of the `Till` Data Type.

3.5 Primitive and Composite

A component defines an interface which is a set of ports with possible communication: emission, receipt and the associated types. We do not use the usual classification of required and provided ports since this is is only meaningful in a strict client-server context. In our case, we can define more complex interactions (n-ary and symmetric rendezvous) and this terminology is not always relevant here. A primitive component is described by means of an STS (the dynamic and the data part) as we have defined in Definition 3.1, and a list of exported events. Exported events are visible outside of the component, receipt events must be in this list, silent and emission events may be visible, but stuttering and hidden events cannot be visible. A composite component is an assembly of sub-components (primitive or composite) and a set of communications. A composite is also a kind of STS: i) its data type is the free product of the data type sub-components, and ii) its dynamic part is built from the synchronous product, as defined in Definition 3.3 and the synchronization vectors. The graphical definition of a composite component or a hierarchy of components is based on a UML composition diagrams enriched with communication notations.

4 Verifications with STSLib

One way to verify STS is to use model-checking, for instance the CADP tool after translating our STS and composites into LOTOS specifications. Another

way is to use a theorem prover for instance the PVS system, see [25]. However we experiment a third approach which is described below.

4.1 The STSLib *Verification Approach*

The approach we currently develop in [23,21] is based on the notion of configuration graph, *i.e.* a possible infinite state machine resulting from the unfolding of an STS as defined in Definition 3.2. Our specific approach to architecture verification can be summarized as follows: i) compute the synchronous product, ii) process some symbolic analysis, and iii) compute the configuration graph and prove properties. This approach may be related to on-the-fly model-checking, in fact this technique is also possible in our case but we have not yet implemented such algorithms. In the current discussion, we consider a situation where on-the-fly checking is not possible or inefficient. Such a case may occur if we are interested in proving several properties on the same system, computing the synchronous product first then verifying the properties may be more efficient than several on-the-fly model-checking.

Figure 3 gives an overall picture showing how to relate classic model-checking and our STS specific verification mean. This diagram uses two transformations, the synchronous product and the unfolding of STS. In Figure 3, the path (a) takes several STS and produces a configuration graph, that is the way we will illustrate in Subsection 4.3. The other way (b) is related to a more classical model-checking approach. Both ways a) and b) are equivalent from a theoretical expressiveness point of view but from a practical point of view, time and space may be different. First, it is undecidable which way will be the most efficient in the general case. The space problem is the following: the final configuration result may have a manageable size, however one of its component may be too wide or infinite. In this case, classic model-checking will fail or only give a partial response. Here the boundedness property may be critical to know whether a configuration graph is finite and it may also provide a more or less precise measure of its size. One challenge is to find useful symbolic analysis, for instance boundedness, guard evaluation, decomposition and so on. This relates in some new way theorem proving and model-checking. We have successfully experimented with several examples, which we can process with our approach but not with CADP or Spin, see [22,21]. One of this is our cash point case study. The next section draws some conclusions on our experiments. The STSLib specific verification approach may be viewed as a valuable, yet complementary technique, to other existing ones: classic model-checking, abstractions, infinite system approaches and the use of theorem provers.

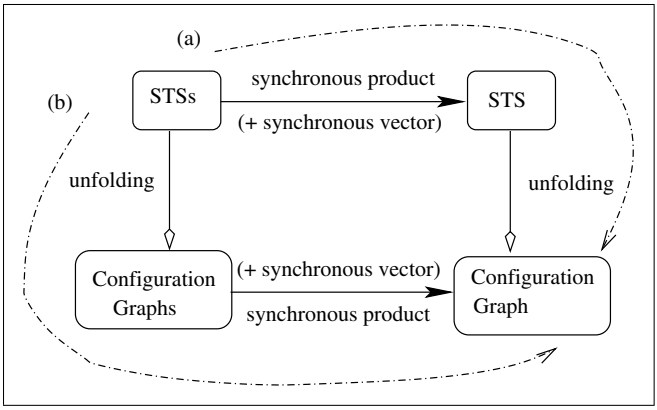

Fig. 3. Relating STS and Model-Checking

4.2 The STSLib *API*

The STSLib API is an implementation of the STS concept with the following functionalities. It supports the definition of the dynamic and the data type part of an STS. Such an STS allows guards, emissions and receipts (n-ary, one-way, and multiple), receipt on guards and the * notation. Architecture can be built from existing components and the synchronous product can be computed. This produces a structured STS, allowing analysis of complex architectures. STSLib is able to compute the configuration graph associated with an STS. It provides a uniform definition of STS and configuration graphs, thus the system may mix in various ways these notions. For instance, we can compute the configuration graph of an STS and synchronize it with another STS. Some simple verification means have been implemented: deadlock checking, state reachability and trace computation. Properties to be checked are expressed as Java predicates of the Java data part class. A boundedness checking was designed, it implements a general algorithm however currently restricted to STS with a set of integer counters as data types. Examples and uses of this prototype may be found in [23]. We already applied successfully our approach (boundedness and configuration graph) to several case studies: a simple flight reservation system, several variants of the bakery protocols, the slip protocol, several variants of a resource allocator, and the cash point service. We have developed this prototype in Python, about 4000 lines of code. We are currently rewriting it in Java 1.5 under Eclipse to get better performances, a wider diffusion and to add nice graphical interfaces.

4.3 Application to the Cash-Point

In the following, we illustrate some experiments done on the cash-point example. These tests have been done to illustrate the use of the Python prototype.

We compute the global STS with up to $N = 4$ tills (an architecture with nearly 20 components) and then we calculate the configuration graph for some set of values. The global synchronous product gives an abstract and concise view of the dynamic system. Such a view is useful to check early for some errors in the dynamic behaviours especially related to the event synchronizations or communications. One may also check reachability of some configurations and to produce a graphic trace describing the events and the precise data value context. We verified that after a `swallowCard` (see the till in Figure 1) the only outgoing transition is a `clock` transition which means that the system has a livelock. We verified that states with only one `clock` transition are exactly targets of a `swallowCard` event. But these cases are only due to the fact that the till keeps the client card after three successive wrong PINs (there was a lack in the requirements). We also checked three additional properties: the PIN counter is equal to three after a `swallowCard`, the database amount and the till amount are always greater or equal than zero. We check these properties for $N = 1, 2$ and small values for the other variables.

Our second objective was to prove that the system ensures an exclusive access to any bank account (which is a safety property). In the following verifications, we used the fact that abstracting one component of a composition defines an abstraction of the product. We check the part corresponding to the database and bank interfaces and abstract the rest of the system. We define a component devoted to the simulation of the tills, the clients and the communication links. A bad situation would be two clients with the same account number withdrawing *via* two distinct interfaces. First, we remark that the database contains the client accounts and the informations related to communications. We observe that the associated `Informations` type is equivalent to `List[Natural x Ident] x List[Natural x Money]`. From this, we apply the method defined in [23]. The principle is to keep the same system as above, except the data type of the database, which is redefined to only operate on `List[Natural x Ident]`. Using this decomposition, we prove that the property yields with MAX=3 and N=2, account number up to 10, MAX=2 and N=3, account number up to 4, and without a specific value for the max of accounts, see Table 1.

Abstraction techniques such as [9,6,11] may be used in our context, but are currently only possible with a manual transformation step. Some abstractions are simple to perform on our STS either on the dynamic part or the data part, a comprehensive analysis is under study. For example, we want to check that an existing card is either owned by the proper client or by its connected till or lost. This safety property was proved by abstracting the data of the system into the card identity which is also the client id. The global product

Table 1
Exclusive Access to the Bank Account Property Verification

N	MAX	account number	Configuration graph size	time (s)
2	2	10	(4561, 24580)	405.
2	3	10	(17961, 145960)	10727
3	2	2	(2351, 9978)	120
3	2	4	(19461, 107292)	10419
3	3	1	(1895, 7290)	75.

has been done for $N = 1, 2$ and 3 without choosing effective numbers for the other parameters. The configuration graphs are bounded and the property is checked using an ad-hoc procedure.

A design of this case study has been done with LOTOS and the CADP toolbox. The LOTOS description of the processes appeared in [22]. The LOTOS description is closest to our STS description and an automated translation is even possible. However CADP bound the data types, we use really strict bounds and we cannot compute the BCG representation (the internal LOTOS representation) even with one client and one till.

5 Runtime Implementation Overview

Our long term objective is to provide a Java compiler which is able to translate STS, both the state machine part and the data part, and architectures into Java code. Currently we defined an experimental interpreter. There are two main parts in this: the implementation of the rendezvous we have at the specification level and the translation of the STS data part into Java. The first part was detailed in [13] and this section describes the principles of the second part. The specification process is the following: the user generates, from the STS dynamic part, a skeleton of the ADT with the signatures, then he fills the axioms part and finally a code generator produces the full Java class and its interface.

5.1 ADT Hypotheses

Currently we consider only a strict subset of ADT, we list here the various assumptions we made. First of all, there is a straight link between the dynamic part of the STS and the interface of the ADT, in fact a generator produces this interface. The rule behind this generator are as follows: i) an emission is done by a function, called an emitter, of the current data type and as resulting

type the emitted type, ii) a guard is a boolean function of the current data type, and an action is a function of the current data type with as resulting type this current data type. In case of a receipt the guard and the action takes as additional parameters the typed receipt variables. In case of an emission, the action takes as additional parameters variables corresponding to the emitted types. In the usual terminology of ADT, guards and emitters are called *observers* and actions belongs to the *constructor* category.

Our automatic translation relies on some hypotheses about the ADT part: i) there is only one generator called `newT`, where `T` stands for the STS name and the sort of interest of the ADT, ii) there is one selector associated to each parameter type of the generator, iii) conditional axioms are assumed to be oriented into left-to-right rewriting rules, and iv) left part of the conclusion has a simple form as in functional languages: either `f(`$x_{1\leq i\leq n}$`)` or `f(newT(`$x_{1\leq i\leq n}$`),` $y_{1\leq j\leq m}$`)`, where x_i, y_j are variables.

5.2 Implementation of a Primitive Component

A global picture of our intra-component implementation is depicted in Figure 4. It represents the different elements defining a primitive component. The representation of the finite state machine is described in a `.sts` file which contains the states, the transitions and names. These names represent the guards, the events, the receipt variables, the emitters 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 follows similar rules than the ADT interface. Note that we expect to produce "real" codes and we consider that guards and emitters will become pure functional methods but an action will be an imperative method with the resulting type `void`. So a primitive component results from the combination of a protocol and existing Java code, 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 of the passive object implementing the component data part.

5.3 The Data Part Class

Our STSLib library provides a `Data` class which is the inheritance root for the data parts of the primitive components. This class defines some general services: to create, to copy, to compare, and to view the representation of the data part. This class uses the Java reflexive API to provide services for finding

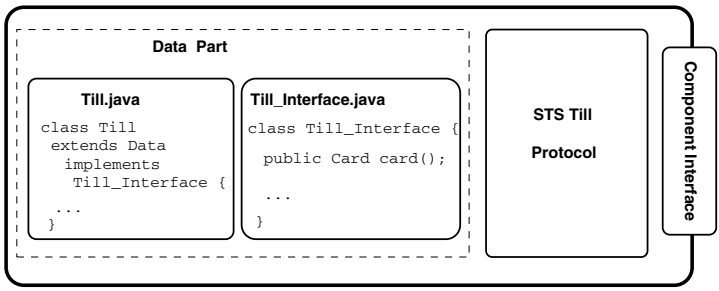

Fig. 4. Implementation of the Till Primitive Component

and executing the methods known by the class and to execute them on a data instance. We have defined a LL(1) grammar (we use ANTLR [19]) for the STS dynamic part and STSLib is able to generate the interface and a skeleton of the data class from the STS dynamic description, then the user has to fill it. However, the code can be automatically generated from an explicit and formal description, [20,24] are some related references to this. In the sequel of this section, we sketch the translation we are experimenting on.

5.4 The Translation Mechanism

A grammar has been defined for the axiom data part, it is a LL(2) one. A parser and an AST builder have been built, and on top of it we have implemented a Java code generator. The generated Java class contains: private fields, fields selectors, predefined constructors and the translation of the ADT axioms. To cope with the Java syntax and the imported data types, we use a dictionary of translation. This dictionary is filled with methods which translates functional calls of the ADT into the equivalent Java expression. It is responsible for finding type names correspondence, for instance `Money` will be implemented by `int`, and the correct translation of operators call and constants. This allows to change more easily the target language by providing another dictionary. A function `translateSimpleTerm` has the responsibility to walk through the AST of an expression and to build the corresponding Java string. It copes with the `this` argument, the dotted notation, variables and field selectors translation. See, for example, the `fail` expression in Figure 2 and its translation in Figure 5. The translation rules for an axiom are the following: i) equations in conditions are translated into equalities, ii) conditions give the test part of an `if` structure, iii) algebraic terms in equations are translated thanks to the `translateSimpleTerm` function. The delicate part is the translation of the axiom conclusion. The left term is traversed and a variable context is built to identify variable and field occurrences. The translation of the right part depends on the fact that it is an observer, a con-

structor term or a generator call. In the two first cases, a translation with `translateSimpleTerm` is done. In the last case we have to identify the arguments to assign to the object fields, and to translate these arguments, see the `giveCash` example in Figure 5. From the axiom the translator identifies that the field `amount` and `card` have to be assigned with new values.

```
// fail : [Till] -> boolean
  public boolean fail() {
      return (( ! (crypter(this.code()) == this.card().code()))
            && (this.counter() => 3));
}

// giveCash : [Till] -> Till
  public void giveCash() {
      this.amount = (this.amount() - this.sum());
      this.card = this.card().updateDailyLimit(this.sum(), this.date());
}
```

Fig. 5. The `fail` and `giveCash` Translation Examples

Our experimental generator relies on some hypotheses but the original thing was to generate full imperative Java code from data type description. The current hypotheses are a prefixed grammar, and the lack of operator overloading. While these features do not impose technical difficulties, they will complicate the grammar. Another restriction we are able to relax is the mono-generator constraint since we have already investigated this problem in [24]. This previous work explored object-oriented class generation representing a data type with several generators. Currently our translation process preserves the axiom ordering. We have to investigate a less strict approach allowing more general left conclusion terms and a support to check axiom exclusivity (for instance using critical pairs computation).

5.5 Implementation of a Composite

Our library provides a `CompositeSTS` Java class, which defines a list of sub-components, the locations of the dynamic and data parts, the internal connections, with the modal operator and the external connections. From the internal connections and the modal operator, a list of synchronization vectors is computed. These vectors serve to build the locks as needed to manage the rendezvous mechanism. In addition to this, the composite defines a scope which may hide or export some ports (and the associated events) to outside. It has a similar role to the hiding operator of process algebras. A parser and a loader have been designed for the composite structure which are able to handle complex architecture descriptions. This raises the issue of defining a global context class which memorizes the component (primitive or composite) al-

ready loaded in the current session. From a process point of view a composite may be viewed as a tree of threads, associated to primitive components, and interacting thanks to the Java implementation of the rendezvous mechanism we have at the specification level (see [13]).

6 Conclusion and Future Work

The STSLib project aims at providing a powerful way to design software components with protocol descriptions. One strong preoccupation is to narrow model for verification and programming code. Our environment currently proposes the STS notation, dynamic and algebraic parts, and composite description using communication diagrams. Our approach considers a specific verification means which acts on the symbolic representation of the behaviour rather than on a finite state approximation. However we think that openness and interfacing with other verification tools is an essential aspect of such an environment support. Currently, several tools have been implemented. The verifications enable the computation of synchronous products and configuration graphs. We also study a true runtime support: generation of interfaces and Java class skeletons, generation of ADT signatures and full translation of a simple ADT into imperative Java code. Lastly, our environment defines an implementation of the rendezvous mechanism which is able to synchronize components.

One important and future task on the verification side is to define and implement abstraction techniques. The second task is to elaborate a concrete syntax for hierarchical components and to implement a Java compiler based on our experimental interpreter. Another future perspective is to prove the translation process into Java code and the correctness of our rendezvous mechanism.

References

[1] 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.

[2] André Arnold. *Finite Transition Systems.* International Series in Computer Science. Prentice-Hall, 1994.

[3] Egidio Astesiano, Bernd Krieg-Brückner, and Hans-Jörg Kreowski, editors. *Algebraic Foundation of Systems Specification.* IFIP State-of-the-Art Reports. Springer-Verlag, 1999.

[4] Tomás Barros, Rabéa Boulifa, and Eric Madelaine. Parameterized models for distributed java objects. In David de Frutos-Escrig and Manuel Núñez, editors, *FORTE*, volume 3235 of *Lecture Notes in Computer Science*, pages 43–60. Springer, 2004.

[5] Hubert Baumeister, Florian Hacklinger, Rolf Hennicker, Alexander Knapp, and Martin Wirsing. A Component Model for Architectural Programming. In *Proc. 2nd Int. Wsh. Formal Aspects of Component Software (FACS'05)*, volume 160 of *ENTCS*, pages 75–96, 2005.

[6] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-Aided Verification, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, pages 319–331, Vancouver, Canada, June 1998. Springer-Verlag.

[7] 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.

[8] Christine Choppy, Pascal Poizat, and Jean-Claude Royer. Specification of Mixed Systems in Korrigan with the Support of an UML-Inspired Graphical Notation. In *Fundamental Approaches to Software Engineering (FASE'2001)*, volume 2029 of *Lecture Notes in Computer Science*, pages 124–139. Springer-Verlag, 2001.

[9] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model-Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[10] Alessandro Coglio and Cordell Green. A constructive approach to correctness, exemplified by a generator for certified Java Card applets. In *Proc. IFIP Working Conference on Verified Software: Tools, Techniques, and Experiments*, 2005.

[11] Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract Interpretation of Reactive Systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, 1997.

[12] Luca de Alfaro and Thomas A. Henzinger. Interface Automata. In *Proc. of ESEC/FSE'01*, pages 109–120. ACM Press, 2001.

[13] Fabricio Fernandes, Robin Passama, and Jean-Claude Royer. Components with symbolic transition systems: A java implementation of rendez-vous. In *Proceedings of the Communicating Process Architecture Conference*, 2007.

[14] Hubert Garavel, Frédéric Lang, and Radu Mateescu. An Overview of CADP 2001. *EASST Newsletter*, 4:13–24, 2001.

[15] Anna Ingolfsdottir and Huimin Lin. *A Symbolic Approach to Value-passing Processes*, chapter Handbook of Process Algebra. Elsevier, 2001.

[16] Pavel Jezek, Jan Kofron, and Frantisek Plasil. Model Checking of Component Behavior Specification: A Real Life Experience. *Electronic Notes in Theoretical Computer Science*, 160:197–210, 2005.

[17] 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.

[18] Bertrand Meyer. The grand challenge of Trusted Components. In IEEE Computer Society, editor, *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 660–667, Washington, DC, USA, 2003.

[19] Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25(7):789–810, July 1995.

[20] Pascal Poizat, Christine Choppy, and Jean-Claude Royer. From Informal Requirements to COOP: a Concurrent Automata Approach. In J.M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems*, volume 1709 of *Lecture Notes in Computer Science*, pages 939–962, Toulouse, France, 1999. Springer-Verlag.

[21] Pascal Poizat and Jean-Claude Royer. A Formal Architectural Description Language based on Symbolic Transition Systems and Modal Logic. *Journal of Universal Computer Science*, 12(12):1741–1782, 2006.

[22] Pascal Poizat and Jean-Claude Royer. KADL Specification of The Cash Point Case Study. Technical report, IBISC, Université d'Evry Val d'Essonne, 2006. http://www.emn.fr/x-info/~jroyer.

[23] Pascal Poizat, Jean-Claude Royer, and Gwen Salaün. Bounded Analysis and Decomposition for Behavioural Description of Components. In Springer Verlag, editor, *FMOODS*, number 4037 in Lecture Notes in Computer Science, pages 33–47, 2006.

[24] Jean-Claude Royer. An Operational Approach to the Semantics of Classes: Application to Type Checking. *Programming and Computer Software*, 27(3):127–147, 2002.

[25] Jean-Claude Royer. The GAT Approach to Specify Mixed Systems. *Informatica*, 27(1):89–103, 2003.

[26] Springer Verlag, editor. *The Cash-Point (ATM) Problem*, volume 12 of *Formal Aspects of Computing Science*. 2000.