INFORMATIQUE, BIOLOGIE INTÉGRATIVE ET SYSTÈMES COMPLEXES

KADL Specification of The Cash Point Case Study

Pascal Poizat [1], Jean-Claude Royer [2]

[1] IBISC FRE 2873 CNRS - Université d'Evry Val d'Essonne
Tour Évry 2, 523 place des terrasses de l'Agora,
F-91000 Évry Cedex
Arles Project, INRIA, France

[2] OBASCO Group, École des Mines de Nantes - INRIA, LINA
4, rue Alfred Kastler, B.P. 20722,
F-44307 Nantes Cedex 3

— —

RESEARCH REPORT

N° 00.0

January 2007

IBISC, FRE 2873 CNRS - Université d'Évry Val d'Essonne, Genopole
Tour Évry 2, 523 place des terrasses de l'Agora
91000 Évry Cedex, France

Pascal Poizat, Jean-Claude Royer
KADL ***Specification of The Cash Point Case Study***
48 p.

KADL Specification of The Cash Point Case Study

Pascal Poizat, Jean-Claude Royer

Pascal.Poizat@inria.fr, Jean-Claude.Royer@emn.fr

Abstract

This report presents the cash-point case study and mainly describes its specifications with the KADL ADL. The language is a mixed of state transition diagrams, abstract datatype and modal logic. We emphasize the need for abstract and formal descriptions especially communication architectures. We also gives some proofs done using our specific tool based on symbolic transition systems. Last we discuss previous specifications for this case study.

Categories and Subject Descriptors: D.2, D.2.4, D 2.11 [**Software Engineering**]: Software Verification, Software Architectures

General Terms: Experimentation, Languages, Verification

Additional Key Words and Phrases: Architectural Description Language, Cash Point Case Study, Component Based Software Engineering, Symbolic Transition Systems, Abstract Data Type, Verification

1 The Cash-Point Case Study

This case study is based on the FM'99 cash-point service benchmark [16]. The *system* 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 with a code stored on the card. After successfully identifying themselves to the system, customers may either (i), make a cash withdrawal or (ii), ask for a balance of their account to be printed. Information on accounts is held in a *central database* and may be unavailable in case of network failure. In such a case, actions (i) and (ii) may not be undertaken. If the database is available, any amount up to the total in the account may be withdrawn. Withdrawals are subject to a daily limit, which means that the total amount withdrawn within a day has to be stored on cards. Daily limits are specific to each customer and are part of their bank account records. Another restriction is that a withdrawal amount may not be greater than the value of the till local stock.

Tills may keep "illegal" cards, *i.e.* cards which fail a key checking. Each till is connected to the *central* by a specific *line*, which may be down or up. The central handles multiple and concurrent requests. Once a user has initiated a transaction, it is eventually completed and preferably within several real time constraint. A given account may have several cards authorized to use it.

1.1 Hypotheses

With reference to the original case study we add the following extensions: a realistic authentication mechanism with three tests for the PIN, an abstract database for the bank, a precise management of the card and a daily limit which is specific to each customer. We do not take into account the real time constraints.

- We first consider that the communications are safe between the tills and the bank manager. In a second step we will consider some communications problems.
- The till may keep the client card, in such a situation the client has to get back his card. We do not model this last action in the client behaviour and the card is simply removed from the system.
- We assume a given number of tills and a given number of clients. One client is not obliged to use a given till, a till may have no client but no more than one. Since a till without client does not introduce any activity we only consider a till with a client.
- We do not formalise printing the balance and sending an account statement by post. We focus on the withdrawal activity which is the most complex and critical.
- We introduce a clock in the till which computes the date. The date is used to check the date of the last withdraw action. We reduced it to the values `0, 1` since we are interested in tagging the card with a date and checking if this date is equal or not to the current date.

The section 10 gives a more detail comparison with other approaches for this case study.

2 KADL notations

This section gives the main aspects of the KADL notations which are based on symbolic transition system and abstract data type descriptions. We also try to reuse some of the UML notations.

2.1 Component Interfaces

Components, both primitive and composite ones, are represented using *boxes* with well-defined interfaces. These interfaces are sets of *event ports*, *i.e.* some form of dynamic signature made up of names with offer parameters (as in LOTOS) and interaction typing (as in SDL). A value emission of a `D` data type is noted `!D` and a value receipt `?D`. The communication interface symbols we use are described in Figure 1.

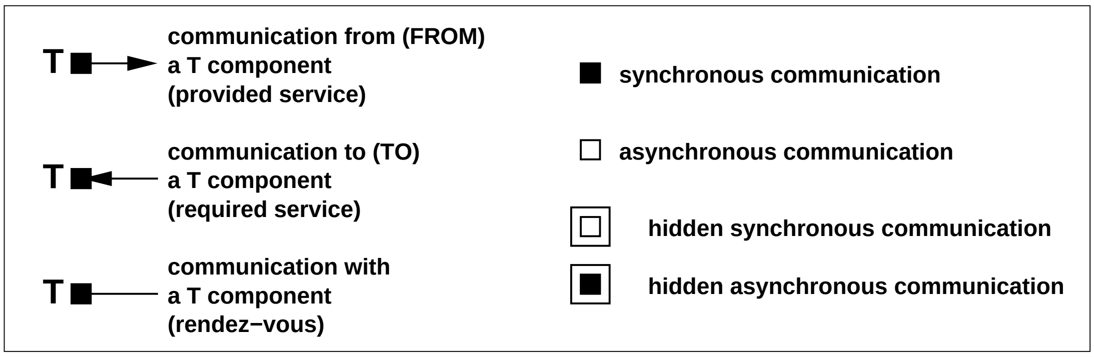
T
communication from (FROM)
a T component
(provided service)
communication to (TO)
a T component
(required service)
communication with
a T component
(rendez−vous)
synchronous communication
asynchronous communication
hidden synchronous communication
hidden asynchronous communication

Figure 1: Notations for Dynamic Interfaces

We have two kind of communication (synchronous or asynchronous). Hidden communication is used in composites to ensure that their external environment will not be able to communicate with them using specific (internal, hidden) ports. An event port may correspond either to a provided service (events received from the component environment), to a required service (events sent to the component environment) or to a synchronizing mechanism (rendez-vous, inherited from LOTOS). Only value receipts can be made on provided services, and only value emissions on required ones. Rendez-vous enable to use both but is restricted to synchronous communication. Component types can be associated to event ports thanks to the `TO`/`FROM` keywords. This enables one to state that any component to be glued on this signature has to satisfy (at least) a given protocol (see Inheritance, section 2.3 below).

Genericity and Patterns. Component interfaces may be generic on (possibly constrained) data values (*i.e.* constants, *e.g.* `N,M:Natural {1<N<M}`), data types (*e.g.* `MSG:Sort`, Fig. 24), event ports and component types (*e.g.* `MsgConnection`, Fig. 7). As in UML, this is denoted by dashed boxes in the top corners of the components (top left for event ports and top right for data values, data types and component types).

In conjunction with genericity on data types and component types, genericity on event ports yields the expressiveness of KADL to express patterns. As in LOTOS and object-oriented programming some idioms or patterns may be used in KADL to describe general and common architectures of systems.

2.2 Primitive Components

Primitive components are sequential components described with two aspects: a dynamic behaviour and a data type description integrated within a Symbolic Transition System (STS). This data type description may be given either using algebraic specifications [32, 9], model oriented specifications [4] or even Java classes as in a library for STS we are developing [23]. The next subsections illustrate the presentation of these aspects in KADL.

Symbolic Transition Systems (STS). STS are (possibly nondeterministic) symbolic labelled finite transition systems which have appeared under different forms in the litterature [22, 7, 9]. STS provide an expressive and abstract means to describe symbolically dynamic behaviours. The description of an STS may be given either in its graphical form (Fig. 8) or in its textual form [27] (better suited for tool processing).

Our STS have the following features. First of all, they rely on both a static description of a data type (denoted by `self`) and a dynamic event-oriented one. Transitions have the form: `[guard] event / action`. `guard` is a predicate on `self` and possibly received values which has to yield true for the transition to be fireable. `event` is a communication event (a communication port name together with reception variables denoted using `?` or/and emission terms denoted using `!`). `action` is the action to be done when the transition is fired. Both the `guard` and the `action` part can be empty. As in LOTOS an `i` action can be used to denote non observable (internal) events. In Figure 8 this is used to denote a timeout which may occur during the communication with the bank. Open terms, *i.e.* terms with variables, can be used in states (a predicate on `self`) and in the `guard` or the variables/term part of the `event` (this is the reason why our transition systems are symbolic ones).

The main interest with these transition systems is that (i) using open terms in transitions (received variables), they avoid state explosion problems, 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.

Abstract Data Types (ADT). An ADT is given for each STS, and transitions from this STS may use the operations defined in the ADT. The operations semantics are described using algebraic axioms. Note that B machines or Z schemas may be used instead or in conjunction with these algebraic specifications following the [4] principles. A sort corresponding to the STS is called *sort of interest*, and a term of this sort is denoted by `self` in the STS.

We here consider a simple approach where (i) actions are explicitly given in the STS (and then defined in the ADT), and (ii) the axioms are fully given by the specifier. See the section 3 for more details. In [32] we present a more automated approach which enables one to derive the operations, their profiles and parts of the axioms from the STS.

Graphical notations. *Composition diagrams* (Fig. 2) are used to represent the component structure of composition views.

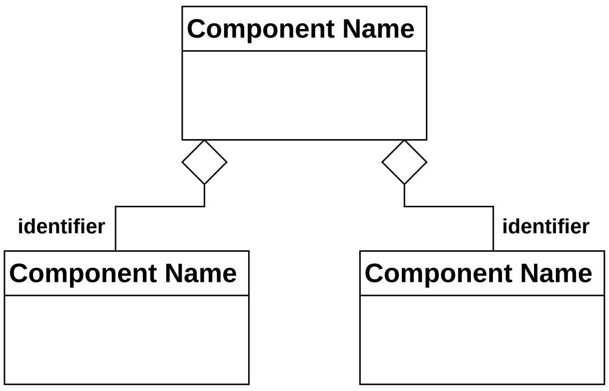

Figure 2: Composition diagrams.

Here we reuse the UML class diagram notation with aggregation relations (white diamonds) to denote concurrent composition of subcomponents into a composite. The UML aggregation notation has been chosen (in place of the composition notation for example) since the subcomponents of a composite (and more generally the components of an architecture) usually have independent life-cycles. As presented earlier on, we also reuse UML notations for templates/genericity (dashed boxes, see page 6) and for inheritance (white arrows, see subsection 2.3). We use the usual UML roles on aggregation relations to identify components and extend this notation using our range operator. A component interface may be associated with a composition by exporting some events of its subcomponents. Hence, composites are components too, and genericity and pattern issues (which we dealt with for primitive components in page 6) apply also to them.

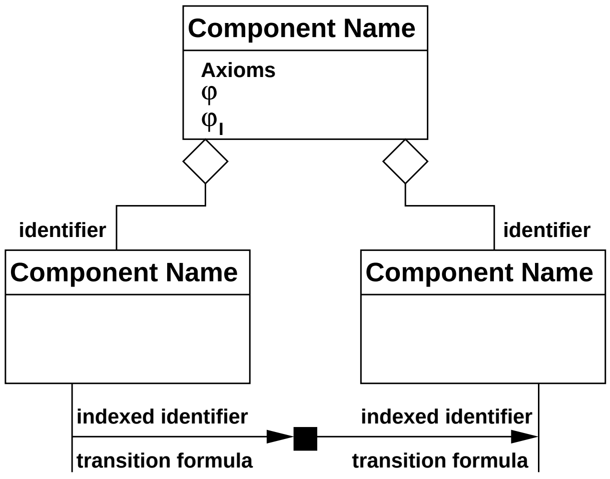

Figure 3: Communication diagrams.

Communication diagrams are composition diagrams complemented with glue rules (see Fig. 3). The axioms (Ax_Θ) and the state formulas ($\overline{\psi}$ and $\overline{\psi_I}$) are put in the composite component. Composition-oriented transition formulas ($\overline{\lambda}$) are represented as lines between the components to which the (indexed) subparts of the formula applies. For example, a formula such as $c_1.a_1 \Leftrightarrow c_2.a_2$ would be represented as a line between components c_1 and c_2.

A communication interface symbol (see Fig. 1) is given for each formula to give an information on the communication type. The identification part is put above the lines and the temporal formula is put below. In the previous example, one would have c_1 and c_2 above the line, and a_1 and a_2 below.

$$\xrightarrow[a_1]{c_1}\blacksquare\xrightarrow[a_2]{c_2}$$

If a range operator is used, it is also put above the line, on the side of the component to which it applies (*e.g.* with $server.send \Leftrightarrow \oplus i : [1..N](client.i.receive)$ a $\oplus$ would be put on the $client.i$ side)

$$\xrightarrow[send]{server}\blacksquare\xrightarrow[receive]{\oplus\ i{:}[1..N]\ client.i}$$

or above the communication interface symbol if it applies to the whole formula (*e.g.* with $\forall i : [1..N](server.send \Leftrightarrow client.i.receive)$ a $\forall$ would be put above the communication interface symbol).

$$\begin{tikzcd}
 & \forall\, i{:}[1..N] & \\
{} \arrow[r, "server", "send"'] & \blacksquare \arrow[r, "client.i", "receive"'] & {}
\end{tikzcd}$$

More details on graphical notations are given in [10] and in the notation appendix of [27].

2.3 Inheritance

In object-oriented programming, inheritance is one of the key concepts that enable the reuse of classes. Inheritance enables one to add methods to a class, and allows overloading and masking. Inheritance may also be used to add or strengthen constraints. In KADL we provide a simple form of inheritance for integration views. Our inheritance mechanisms are restricted to the adding of new states and new transitions. We do not allow overloading nor masking since this yields semantic complexity. This rather strict inheritance constraints simplify the dynamic descriptions of views, allow subtyping and ensure some kind of behavioural compatibility. Here, inheritance semantically corresponds to trace inclusion (the traces of the super-view are a subset of the traces of the sub-view). More complex behavioural subtyping relations could be used [26, 35] but would have to be extended for STS.

3 Algebraic presentations

This section gives some details about the datatype syntax and the specification principles for ADT. A datatype is either a predefined datatype (such as integer, or boolean) with some additional functionalities. We use genericity, for instance `List[Natural]`, and $\times$ represents the product of two datatypes. When more complex datatypes are needed we describe them using an algebraic approach. Such a description is summarised below for the card, where `/* */` is a comment. We have two main parts the signature and the axiom parts, see below the `Card` data type example.

```
/* ADT declaration */
Sort Card
/* imported types */
Imports Boolean, Natural, PinNumber, Money, Card, Ack, Info

/* operation profiles */
/* generator of a card */
newCard          : Ident x Money x Money x PinNumber x Date -> Card
/* update daily limit */
updateDailyLimit : Card x Money x Date -> Card
/* accessor for client id */
id               : Card                -> Ident
/* daily limit */
max              : Card                -> Money
/* daily amount */
sum              : Card                -> Money
/* PIN code */
code             : Card                -> PinNumber
/* date of the last withdraw */
last             : Card                -> Date
```

```
/* variable declaration */
Variables s,m,s1:Money; c,c1:Card; code : PinNumber; d,d1:Date

/* axioms declaration */
id(newCard(i, m, s, c, d)) = i
max(newCard(i, m, s, c, d)) = m
sum(newCard(i, m, s, c, d)) = s
code(newCard(i, m, s, c, d)) = c
last(newCard(i, m, s, c, d)) = d
d=d1 => updateDailyLimit(newCard(i, m, s, c, d), s1, d1)
                                            = newCard(i, m, s+s1, c, d)
d!=d1 => updateDailyLimit(newCard(i, m, s, c, d), s1, d1)
                                            = newCard(i, m, s1, c, d1)
```

Figure 4 presents an abstract grammar for our abstract datatypes. A terminal is written with `teletype` and a non terminal with SMALL CAPITAL. In the expressions or rules `*` denotes a list, binary | is alter-

1)	ADT	::= `Sort` TypeId `Imports` TypeId* Profile+ `Variables` Declaration* Axiom+
2)	Profile	::= OperationId `:` (`null` | TypeIdent) `->` TypeIdent
3)	Declaration	::= VariableId (, VariableId)* `:` TypeIdent
4)	TypeIdent	::= BasicId | TypeId | TypeIdent `x` TypeIdent | `List[` TypeIdent `]`
5)	Axiom	::= Condition `==>` Conclusion
6)	Condition	::= `null` | (Equation |Nequation) ∧ Condition
7)	Equation	::= Term `=` Term
8)	Nequation	::= Term `!=` Term
9)	Conclusion	::= OperationId(Term*) `->` Term
10)	Term	::= Constant | VariableId | OperationId(Term*)
11)	OperationId	::= lowerCaseLetter (letter | digit | _)*
12)	TypeId	::= upperCaseLetter (letter | digit | _)*

Figure 4: ADT Abstract Grammar

native, unary + denotes one or more elements and `null` the empty sequence. BasicId are for example `Integer, Boolean, Character` with literal Constant and operations OperationId. Of course these basic types may be defined as ADT.

Axioms are positive conditional axioms. We choose a rather operational approach which is simple to translate into programming language. We define a basic generator for the datatype and several constructors and observers. We expect that normal forms are unique, see [3] for related definitions and techniques. To manage partiality and errors we use special data values, for example the absence of the card is denoted by a special `noCard : Card` value. Algebraic specifications allow the definition of partial functions but we avoid this to simplify the datatypes.

4 Auxiliary Datatypes

We need some simple datatypes which are briefly described here, and they are provided with some functions.

`Money` **is** **`Natural`** a simple datatype for money.

`Date` **is** **`Natural`** a simple datatype for date.

```
inc : Date -> Date
```

`Ident` **is** **`Natural`** : it denotes the client identities as they are defined in the bank account.

`PinNumber` **is** **`Natural`** : the code PIN datatype.

```
crypt : PinNumber -> PinNumber
```

`Card = Ident` $\times$ **`Money`** $\times$ **`Money`** $\times$ **`PinNumber`** $\times$ **`Date`** : a card contains a client identity, the daily delivery limit, the current daily delivered amount, the PIN code, and the date of the last withdrawal.

```
updateDailyLimit : Card x Money x Date -> Card
id               : Card                -> Ident
max              : Card                -> Money
sum              : Card                -> Money
code             : Card                -> PinNumber
last             : Card                -> Date
```

The constant `noCard : Card` is a default card which denotes the absence of card.

MSG : is an abstract type with two concrete subtypes: `Info` and `Ack`.

`Info = Ident` $\times$ **`Money`** : type of messages denoting a client identity and the amount to withdraw. This datatype is used to communicate from the till to the bank to require withdraw authorisation.

```
client   : Info       -> Ident
sum      : Info       -> Money
```

`Ack` **is** **`Boolean`** : type of message denoting the reply of the bank manager to tills to allow or not a withdraw.

```
isOk   : Ack       -> Boolean
```

`Accounts = List[Ident` $\times$ **`Money]`** : the client accounts.

```
withdraw : Accounts x Ident x Money -> Accounts
account  : Accounts x Ident         -> Money
```

`Informations = List[Natural` $\times$ **`Ident`** $\times$ **`Money]`** : memorises the bank interface number, the client identity and the amount to withdraw in the database.

```
cons     : Natural x Ident x Money x Informations -> Informations
isIn     : Informations x Ident                   -> Boolean
hasKey   : Informations x Natural                 -> Boolean
sum      : Informations x Natural                 -> Money
remove   : Informations x Natural                 -> Informations
client   : Informations x Natural                 -> Ident
```

5 The System Architecture

The system architecture of the till system is given in the Figure 5 page 12 composition diagram. The till system is made up of `N` `TillLines` (*i.e.* a `Till` and its `MsgConnection` specialized communication lines) and the bank with its bank interfaces. The range operator is used to identify the `TillLines`: `tills.i:[1..N]`.

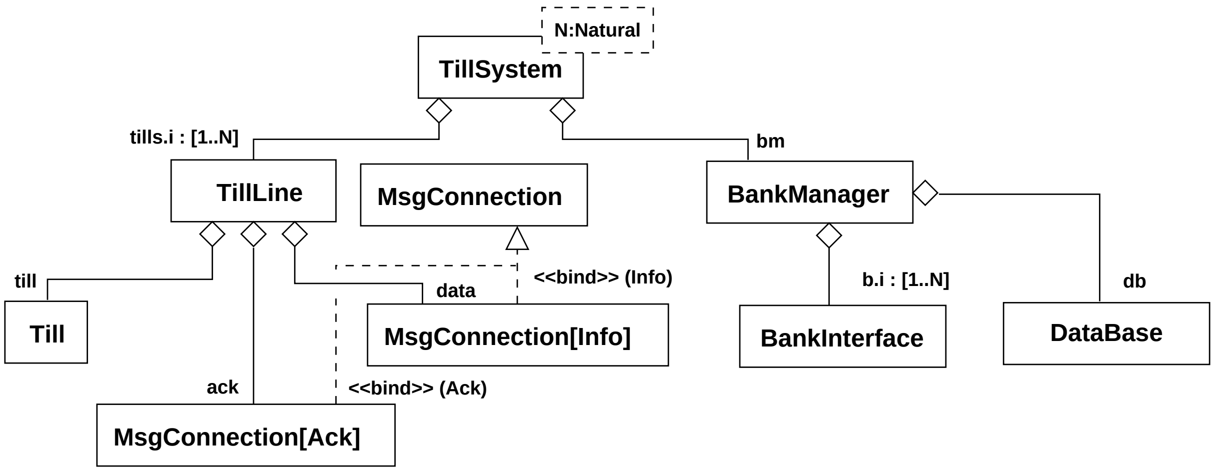

Figure 5: The System Architecture.

The communications are depicted in a communication diagram, see Figure 6, where graphic links are added to the system architecture.

6 The Component Specifications

6.1 The `Till` Specification

The till component interface is described in Figure 7 page 13. It has several event ports, `card ?Card` to insert the user card, `card !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 8 page 14. A transition such as `cash !sum / giveCash(self)` means that the till emits a sum of money and during this transition the card in the till is tagged (the daily limit is increased) and the `giveCash` operation updates the amount of money of the till datatype.

The `Till` ADT is given in Figure 9 page 15 (for the operations definitions) and in Figure 10 page 38 (for the corresponding axioms).

In our example, the till component is generic on a `MsgConnection` component type. This constrains it to be connected, as far as the `send` and `rec` event ports are concerned, with a `MsgConnection` component or any component that inherits from it. An example of genericity on datatypes is given in Figure 24 page 45, where the `MsgConnection` component is generic on the type of messages which transit in the connection.

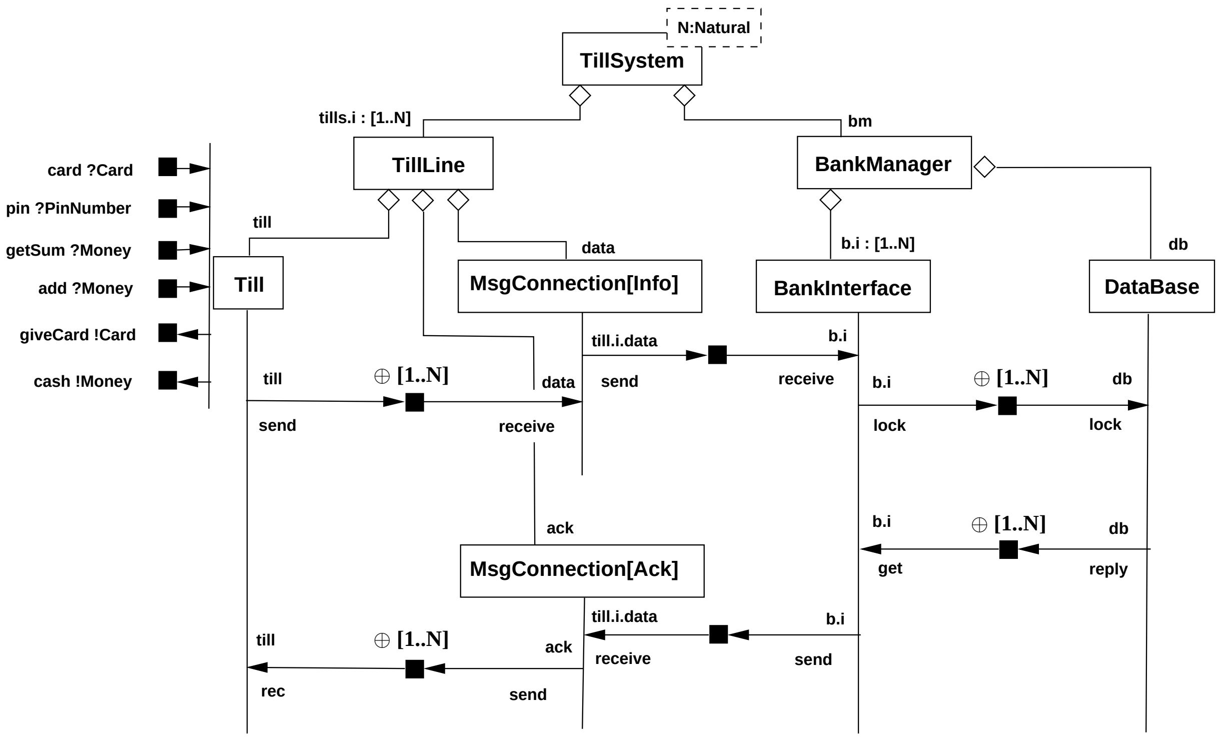

Figure 6: The Communication Diagram.

6.2 The `MsgConnection` Specification

From a given till to a bank interface there are two connections. The link from the till to the bank carries informations (`Info`) about the client and the amount to withdraw. The opposite link is used for acknowledgment (`Ack`) from the bank. The `MsgConnection` component describes a general and simple media which assumes safe communications.

6.3 The `Bank` Specification

This specification is achieved by a central database with several interfaces. The database (`DataBase`) has to abstract the client accounts and to manage N concurrent communications. Each one-way commu-

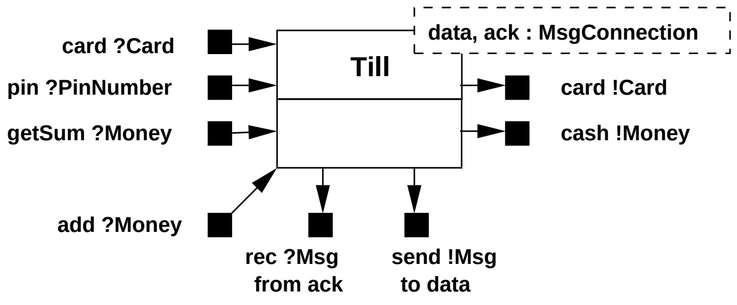

Figure 7: Component Interface Diagram of the `Till`.

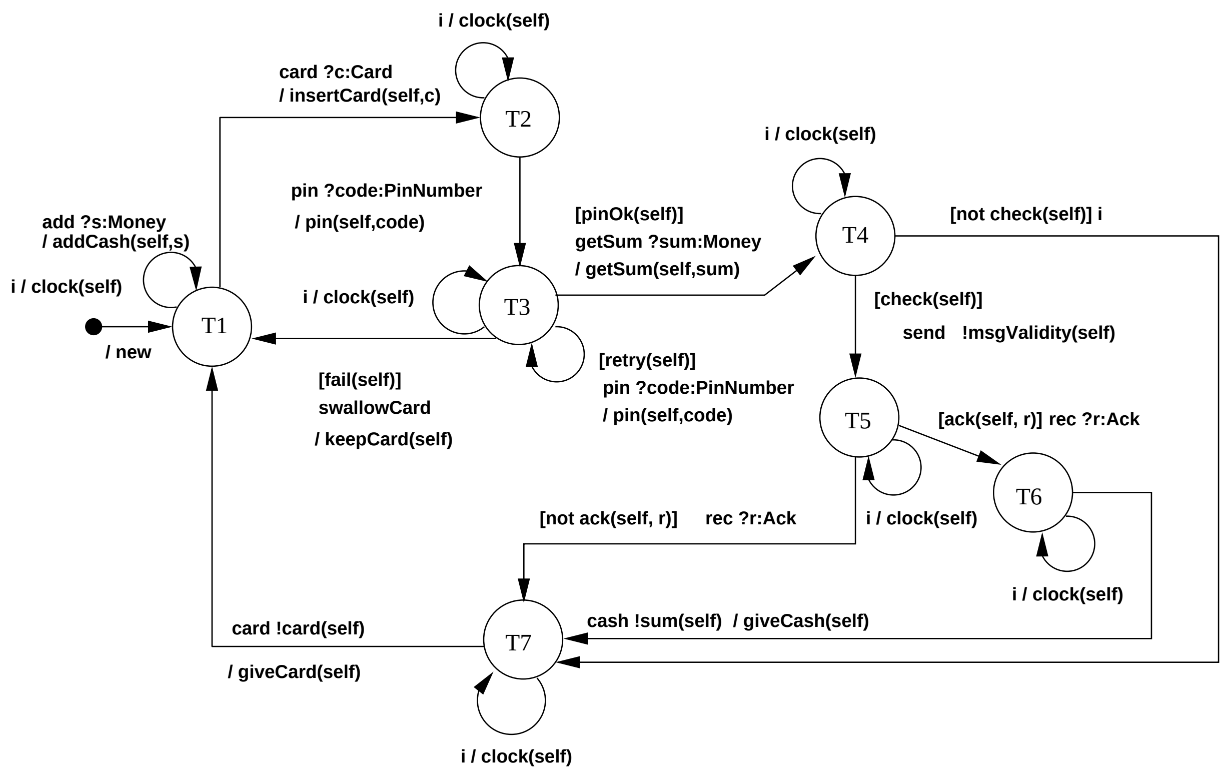

Figure 8: Symbolic Transition System of the `Till`.

nication is performed through a bank interface (`BankInterface`).

Note the special transitions `reply` in the database STS, Figure 14 page 40 (a similar situation occurs two times in the client STS). This transition stands for the following set of transitions:

```
Forall bi:Natural
     [checked(self, bi, False)]  reply !bi !False / unlock(self, bi)
```

Note that such a datatype description is possible for example in PVS and the * mechanism has been implemented in our Python prototype for STS. This is used several times to code a non-deterministic choice whose set of values can be computed by a function.

Note also that, in the system architecture Figure 5 page 12, there is only one port `reply` which is connected to the `get` port of the bank interfaces. The guard `iam` is used to check if the communication is really addressed to an interface (by checking its interface number).

6.4 A Client Specification

To perform some tests and verifications we use a simple abstract client with the STS described in Figure 19 page 43. It is not a closed system since we do not enforce a rigid client behaviour, the client may perform actions in any order and with various output values.

```
Sort Till
Imports Boolean, Natural, PinNumber, Money, Card, Ack, Info

Opns
/* generator for till */
newTill : Money x Card x PinNumber x Money Natural -> Till
/* add cash in the till */
addCash : Till x Money -> Till
/* card insertion */
insertCard : Till x Card -> Till
/* to type in the PIN */
pin : Till x PinNumber -> Till
/* choose a sum to withdraw */
getSum : Till x Money -> Till
/* get the cash */
giveCash : Till -> Till
/* give back card to user */
/* the till kept the card */
/* increase the clock number */
keepCard, card, clock : Till -> Till

/* variables observers */           /* guards */
sum :  Till -> Money               pinOk,retry,fail,check : Till -> Boolean
card : Till -> Card
code : Till -> PinNumber           ack : Till, Ack -> Boolean
counter : Till -> Natural

/* other observers */
amount : Till -> Money
msgValidity : Till -> Info
date  : Till -> Date
```

Figure 9: The `Till` Datatype (part I, operations).

6.5 Variables of Interest

This system has several interesting parameters which are useful for simulation or model-checking. These are: `MAX`: maximum withdrawal for the clients, `N`: number of till, `daily limit`: maximum daily limit, `till amount`: maximum of cash in a till, `max ident`: maximum number of clients, `account number`: maximum number of accounts, and `account max`: maximum cash in an account. If a till is not connected with a client it does not really participate in the global behaviour, the same is true with other pairs of components. Hence we are interested in a global system with a database, N bank interfaces, 2*N links, N tills and N clients. The value N=2 is critical in the sense that it is necessary and sufficient to check concurrent accesses to the bank accounts.

Our specification currently takes into account that a client may type in its PIN or a wrong one. He may also select several amounts to withdraw (1..`MAX`), hence we consider `daily limit = MAX`. It seems relevant to have `MAX` greater than both `till amount` and `account max`.

7 Verifications

7.1 The SyCLAP API

In [11], we have proposed a prototype environment dedicated to KADL which follows two main principles: openness and extensibility. According to these principles, it provides a library for STS and translation mechanisms from KADL into other mixed formalisms (*e.g.* LOTOS, LP) whose goal is to interface with tools (*e.g.* dynamic specification toolboxes such as CADP [20] or theorem provers such as the Larch Prover). Since targetable formalisms/tools are numerous and evolve, our framework is based on a class library reifying the different formalisms taken into account. This makes it *extensible*. Our goal was also to provide general tools which can be useful to other environments or formalisms. An example is the CLAP tool which can be used to compute synchronous compositions of any state-transition diagrams (automata, Petri Nets, symbolic transition systems). Another important feature of this environment is the ability to obtain concurrent object-oriented code (Active Java) from the KADL specifications [8].

The SyCLAP API is an implementation of STS, successor of CLAP, with the following functionalities:

- Definition of an LTS associated to a sequential system.
- Such an LTS may be completed by a datatype description (a Python class) yielding to a true STS.
- Such an STS allows, guards, emissions and receipts (n-ary but one-way), receipt on guards and the * notation.
- SyCLAP allows the uniform definition of STS and LTS and the system may mixed in various ways these notions. For instance to compute the configuration graph of an STS, this produces a LTS and to synchronise it with another STS.
- The synchronous product of STS has been implemented allowing the definition of complex system from architecture and sequential STS. One interesting aspect is the ability to keep inside all the structural information of the components.
- The configuration graph computation and a boundedness checking have been implemented.
- Some simple verification means have been implemented: deadlock, state reachability and trace computation.

Note that an ADT is manually translated into a Python class, since we are using a particular form of ADT such a link is not difficult to establish and to automate. Our ADT defined one generator and it corresponds to the class constructor (or class instanciation) we have in Python. Parameters of the generator leads to instance variables with the associated selectors, thus other functions may be defined as methods of this class. In addition we have some needs for the configuration computation, a deep equality of class instances, the implementation of emitter and generator associated to symbol `!` and `?`. Such a formal representation may be described using Hoare principles [21], one related references is [31]. As an illustration of the translation process, Subsection B gives the Python class corresponding to the `MsgConnection`. A more sophisticated example is the `DataBase`, in this case we have to implement the generators needing to process the `*` notations.

We have developed this prototype in Python, about 4000 lines of code and efficiency was not our primary goal. We have already applied successfully our approach (boundedness, decomposition and model-checking) to several examples: a simple flight reservation system, several variants of the bakery protocols, the slip protocol, several variants of a resource allocator, and a cash point service. Examples and use of this prototype may be found in [33, 23, 29].

7.2 Some verifications with SyCLAP

These examples have been done to illustrate the use of the prototype, some of them may be done more efficiently by interacting with a model-checker or using infinite state-based techniques. One interest of this approach is that it seems simple and well-suited to component systems. Sometimes it is not efficient but one known solution to get better results is to verify the property on the fly without computing the global product.

The Figure 20 page 43 gives the global STS with one client and one bank interface, it gives an abstract and readable view of the dynamic system. Such a view is useful to early check some errors in the dynamic behaviours especially related to the event synchronisations.

One may also check reachability of some configurations and to produce a graphic trace describing the event and the precise value context. As an example the Figure 21 page 44 depicts a situation where the client first gets some money and then he retries to withdraw, he fails to type in three times its PIN and its card is kept by the till.

In the following verifications we used sometimes the fact that abstracting one component of a composition defines an abstraction of the product.

1. We compute the global STS with one and two tills and then we calculate the configuration graph for some set of values. Since we do not completely model the client some bad situations occur. After a `swallowCard` the only outgoing transition is a `clock` which means that the system has a livelock. We verified that states with only one `clock` 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 (this is a lack in the requirements). They disappear with a more advanced client STS adding an action to get back the card. 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.

N=1, account number=2, daily limit=MAX, STS product=(11, 27)				
account max	till amount	MAX	size CFG	time (s)
1	1	1	(126, 266)	0.69
2	2	2	(698, 1586)	6.7
2	3	4	(2348, 5950)	66.3
3	3	3	(2292, 5576)	66.75
5	2	2	(698, 1586)	10.9
10	2	2	(698, 1586)	11.5
10	3	2	(1574, 3570)	27.21
10	2	3	(994, 2434)	15.11

N=2, account number=2, daily limit=MAX, STS product=(121, 594)					
account max	till amount	MAX1	MAX2	size CFG	time (s)
0	1	0	0	(324, 1224)	6.25
0	1	1	0	(1692, 6724)	50.27
0	1	1	1	(8872, 36992)	997.
1	1	1	1	(15912, 67176)	3434.

2. Our objective was to prove that the system ensures an exclusive access to any bank account (which is a safety property). 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. Experiments are not efficient so we apply an abstraction method presented in [28, 29]. First we remark that the database contains the client accounts and the informations related to communications. We observe that type `Informations` is equivalent `List[Natural x Ident] x List[Natural x Money]`. From this, a decomposition as defined in [29] exists. More precisely we keep the same system as above except the datatype of the database which is redefined to only operate on `List[Natural x Ident]`. Using the decomposition method we prove that the property yields with N=2, account number=10, MAX=3 and N=3, account number=4, MAX=2 and without a specific value for the max of accounts.

N	MAX	account number	size CFG	time (s)
2	2	1	(52, 118)	
2	2	2	(193, 564)	
2	2	10	(4561, 24580)	405.
2	3	1	(177, 484)	1.6
2	3	2	(713, 2568)	6.7
2	3	10	(17961, 145960)	10727.
3	2	1	(309, 966)	3.2
3	2	2	(2351, 9978)	120
3	2	4	(19461, 107292)	10419
3	3	1	(1895, 7290)	75.

3. Abstraction techniques such as [12, 5, 13, 25] may be used in our context, but currently with a manual transformation. Some abstractions are simple to perform on our STS either on the LTS 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 is proved by abstracting the data of the system into the card identity which is also the client id. The global product 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.

N	size CFG	temps (s)
N=1	(24, 56)	1.58
N=2	(576, 2688)	5.17
N=3	(13824, 96768)	3876.73

Our specification approach is able to take into account a variable number of components, however the bounded analysis is more difficult to perform. In such a case techniques for infinite systems, as for examples [14, 19] seem more adequate.

7.3 Other Verifications

Our approach based on SyCLAP may be viewed as a complementary technique to other existing ones: model-checking, abstractions, infinite system approaches and use of theorem provers.

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

8 Preliminary Comparisons with Model-Checking

The main objective of this section is to present some precise examples comparing our symbolic approach with more traditional finite state approaches. More specifically, we have done several experiments with the LOTOS language and the CADP toolbox. We begin with a first example and we try to justify our point of view. Our approach is simply a complementary way to analyse systems, there is no strict separation between both.

8.1 Application to Finite-State Systems

We consider a simple example, see Fig. 22, with two components which synchronise on `(emit, get)` and their bounded product. This example has been encoded in LOTOS. The global LTS computed using CADP with $M = 200$ is made up of 401 states and 400 transitions. Using our prototype, we get the same result in 1 second (product+unfolding).

CADP arbitrary bounds natural numbers to 256, this is the reason why we used a value smaller than 256 for M. But bounded analysis allows the computation of the two STS product and to check its boundedness for bigger values. Our prototype computes the product and builds the configuration graph of this example. Whenever the bounds set by model-checking tools are reached, the specifier does not know if its system is either too big for the tool or really unbounded. In such a case, bounded analysis may be successful and complements model-checking, for example it can provide the exact bounds to optimise the generation of the state system. The problem here is that the system is obviously manageable for bigger values than 256. But due to a lack of flexibility and parameterisation CADP cannot go further than this value. Our experience shows that it is simple and useful to try another way to proceed such an example.

8.2 A General View of the Two Approaches

In a first setting let us assume that the global computation of the system is needed. This is sometimes useful for some verifications. Figure 23 gives an overall picture of model-checking and STS.

This diagram uses two transformations, the synchronous product and the unfolding of STS. The path (a) takes several STSs and produces a configuration graph, that is the way we are illustrating here. The other way (b) is related to a more classical model-checking approach. Both ways are equivalent from a theoretical expressive power but from a practical point of view time and space may be different. First it is surely undecidable to know which way will be the more efficient in the general case. Results are needed here to provide guidelines for specifiers. The space problem is the following: the final configuration result may have a manageable size however one of its component is too wide or infinite. In this case model-checking will failed. The boundedness property may be used to know if a configuration graph is finite and it may also provide a more or less precise measure of its size.

Now if we consider on-the-fly model-checking this technique checks a property and only builds the required part of the configuration graph. Such technique improves efficiency but it seems also possible to apply it in the context of STS, thus avoiding the global computation of the synchronous product.

8.3 Application to Infinite State Systems

In [29] we have shown several examples where usual model-checking fails but our bounded analysis, as abstraction technique, can be useful on finite systems as well as on infinite ones. Whenever the bounds set by model-checking tools are reached, the specifier does not know if his system is either too big for the tool or really unbounded. In such a case, bounded analysis is successful and complements model-checking.

We have two examples related to a resource allocator, but the more demonstrative and simple one is a mutual exclusion algorithm protocol inspired by the ticket protocol as described in [14]. Our prototype succeeds in generating the global system (then checking mutual exclusion) up to 9 clients whereas CADP and SPIN (with the default configuration values and bounded data types, *e.g.* natural numbers bounded to 256) do not pass 6 clients. The resulting product (for 8 clients) is made up of 6561 states and 52488 transitions; the configuration graph contains 1280 states and 6656 transitions. A similar analysis is possible with other protocols, as we did with the Bakery mutual exclusion algorithm [15]. Finally, we stress out that our primary goal while implementing the prototype was the validation of the paper ideas. In particular, efficiency has to be improved, and is not satisfactory by now for several reasons: strongly dependent of Python high-level data structures, interpreted code wrt compiled code, no optimizing, etc.

The experiments done on the Cash Point case study show that our approach is useful. In fact the verification based on CADP failed due to to the high inter-relation between behavioural and the data types aspects in this case study.

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

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

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

9 Communication Problems

This section gives a simple view of the design of an unsafe communication link.

9.1 The DropMsgConnection Specification

The `DropMsgConnection` component is a simple media which may be down or up. An example of inheritance in KADL is given in the left-hand part of Figure 24 page 45 where the `MsgConnection` (media) and the `DropMsgConnection` (media with failure) are related using the UML notation for generalization (white arrow). Failure is modeled by a special `down` port. This method may also be used to take into account the creation or deletion of components. The inheritance relation between the components behaviours (STS) is given in the right-hand part of Figure 24 page 45.

From that we may detect wrong situations, for instance a successfully bank request but without the corresponding money delivered to the client. The problem is that communications may be down now. Thus a safe system must ensure that the account bank decreasing action and the cash delivery occur in a transaction.

10 Related Work

A special issue [16] is devoted to this case study, we quote below several of the approaches. One central point for us is the specification of communicating components and the need to express a communication architecture.

In [34] the banking system is described with different hierarchical descriptions for structure and behaviour. The model is based on a finite functional description of datatypes and behaviour are described by state transition diagrams. They do not explicit the database, but a communication architecture is given. The dynamic formalism used is closed to the STS notion. They used several tools mainly based on model-checking and simulation to verify the specifications.

In [17] a real-time, hierarchical and graphical formalism is used. There is a communication architecture and many time constraints are specified. The communication mode is asynchronous with channels. However the formalism with too many variables is not readable, furthermore we think that this approach needs too many time constraints not explicit in the case study. Several properties, using timed model-checking, are proved but there is nothing related to the global consistency of the system.

The used of UML for such a system have been criticised, see for example [24, 10, 6]. The description in [18] is too centralised and gives an abstract implementation structure with classes rather than a communication architecture. Another difficulty is that UML is complex and needs several complementary formalisms, the authors used Z and Lustre. One consequence is the absence of global consistency for the system. The authors think that it is a good point to do not precise the communication mode. However communication mode and runtime mode are important features which have a great impact on the semantics and the behaviour of a system.

[1] specifies this system with a functional language. There is no communication architecture and the hierarchy of modules is not sufficient for that. As with UML it explicits the design of the system not necessarily the communications between components. There is neither a graphical nor an abstract view of the system architecture. This style is not well-suited to dynamic system and some tricks are needed to complete the specifications.

With comparison to the previous work we add a more precise management of the card, the three tests for the pin and the checking of several values. Note also that our specifications catches several cases of non-determinism: right and wrong pin codes, withdraw of different values and wrong or right acknowledgment from the bank. This is expressed using the specific * notation. We avoid request canceled by the client since it does not appear in the informal requirements. We do not check and keep wrong cards, but this is easy to add as a new till transition.

11 Conclusion

Readability of our specifications is improved by the used of several complementary diagrams, special emphasis is put on the communication architecture and the STS. Note also that the range notation and genericity are interesting features. To specify an atomic system the conjunction of a state machine and a datatype allows a great flexibility, we may balance the control part or the computation part in a consistent manner. The use of the * notation permits to easily catch value non-determinism. STS (or process algebras) are surely the good mean to describe simple component. However to specify architecture of communicating components we consider that a modal logic approach is better since it expresses abstract properties of the global system. This is why we propose a modal logic, with some graphical notations, to express the meaning of communication. This modal logic is rather sophisticated generally a subset of the operators is needed.

With reference to other approaches we provide a structured and readable approach with an homogeneous semantics. Here we focus on tools based on model-checking but theorem provers are also possible, see [30, 2].

Some time constraints may be represented in our specifications, for instance the day changing or the timeout connection. To represent full time constraints seems possible and to adapt the principles of timed automata but is it still a future idea. Another area of improvements is to integrate abstraction techniques in our STS and the related tools.

References

[1] Timo Aaltonen, Pertti Kellomäki, and Risto Pitkänen. Specifying cash-point with disco. *Formal Aspects of Computing*, 12(4):231–232, 2000.

[2] Michel Allemand and Jean-Claude Royer. Mixed Formal Specification with PVS. In *Proceedings of the 15th IPDPS 2002 Symposium, FMPPTA*. IEEE Computer Society Press, 2002.

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

[4] Christian Attiogbé, Pascal Poizat, and Gwen Salaün. Integration of Formal Datatypes within State Diagrams. In *Fundamental Approaches to Software Engineering (FASE'2003)*, volume 2621 of *Lecture Notes in Computer Science*, pages 344–355. Springer-Verlag, 2003.

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

[6] Conrad Bock. UML2 Composition Model. *Journal of Object Technology*, 3(10):47–73, 2004.

[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. From Informal Requirements to COOP: a Concurrent Automata Approach. In J.M. Wing and J. Woodcock and J. Davies, ed., *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. Springer-Verlag, 1999.

[9] Christine Choppy, Pascal Poizat, and Jean-Claude Royer. A Global Semantics for Views. In T. Rus, ed., *International Conference on Algebraic Methodology And Software Technology, AMAST'2000*, volume 1816 of *Lecture Notes in Computer Science*, pages 165–180. Springer Verlag, 2000.

[10] Christine Choppy, Pascal Poizat, and Jean-Claude Royer. Specification of Mixed Systems in KORRIGAN with the Support of a UML-Inspired Graphical Notation. In Heinrich Hussmann, ed., *Fundamental Approaches to Software Engineering. 4th International Conference, FASE 2001*, volume 2029 of *LNCS*, pages 124–139. Springer, 2001.

[11] Christine Choppy, Pascal Poizat, and Jean-Claude Royer. The Korrigan Environment. *Journal of Universal Computer Science*, 7(1):19–36, 2001.

[12] 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.

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

[14] G. Delzanno. An Overview of MSR(C): A CLP-based Framework for the Symbolic Verification of Parameterized Concurrent Systems. In *Proc. of WFLP'02*, volume 76 of *ENTCS*. Elsevier, 2002.

[15] Giorgio Delzanno and Andreas Podelski. Model checking in CLP. In *TACAS*, volume 1579 of *Lecture Notes in Computer Science*, pages 223–239. Springer Verlag, 1999.

[16] T. Denvir, J. Oliveira, and N. Plat. The Cash-Point (ATM) 'Problem'. *Formal Aspects of Computing*, 12(4):211–215, 2000.

[17] Henning Dierks and Josef Tapken. Modelling and verifying of a 'cash-point service' using moby/-plc. *Formal Aspects of Computing*, 12(4):222–224, 2000.

[18] Sophie Dupuy and Lydie du Bousquet. A multi-formalim approach for the validation of uml models. *Formal Aspects of Computing*, 12(4):228–230, 2000.

[19] Javier Esparza. Verification of systems with an infinite state space. *Lecture Notes in Computer Science*, 2067:183–186, 2001.

[20] Hubert Garavel, Mark Jorgensen, Radu Mateescu, Charles Pecheur, Mihaela Sighireanu, and Bruno Vivien. CADP'97 - Status, Applications, and Perspectives. In *Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design*, June 1997.

[21] C.A.R. Hoare. Proof of Correctness of Data Representations. *Acta Informatica*, 1:271–281, 1972.

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

[23] Olivier Maréchal, Pascal Poizat, and Jean-Claude Royer. Checking Asynchronously Communicating Components Using Symbolic Transition Systems. In R. Meersman, Z. Tari, and al, eds., *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE*, volume 3291 of *Lecture Notes in Computer Science*, pages 1502–1519. Springer Verlag, 2004.

[24] Michael J. McLaughlin and Alan Moore. Real-time extensions to UML. *Dr. Dobb's Journal of Software Tools*, 23(12):82, 84, 86–93, December 1998.

[25] François Michel and François Vernadat. Maîtriser l'explosion combinatoire, réduction du graphe de comportement. *RAIRO, Technique et Science Informatiques*, 17:805–837, 1998.

[26] Oscar Nierstrasz. Regular types for active objects. In *Proceedings of the OOPSLA Conference*, volume 28 of *SIGPLAN Notices*, pages 1–15. ACM, October 1993.

[27] P. Poizat. *Korrigan: a Formalism and a Method for the Structured Formal Specification of Mixed Systems*. Doctoral thesis, Institut de Recherche en Informatique de Nantes, Université de Nantes, 2000. Available at `http://www.lami.univ-evry.fr/~poizat/documents/these.ps.gz`, in French.

[28] Pascal Poizat, Jean-Claude Royer, and Gwen Salaün. Symbolic Bounded Analysis for Component Behavioural Protocols. Technical report, Ecoles des Mines de Nantes, 2005. `http://www.emn.fr/x-info/~jroyer`.

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

[30] Jean-Claude Royer. Formal Specification and Temporal Proof Techniques for Mixed Systems. In *Proceedings of the 15th IPDPS 2001 Symposium, FMPPTA*, San Francisco, USA, 2001. IEEE Computer Society Press.

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

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

[33] Jean-Claude Royer and Michael Xu. Analysing Mailboxes of Asynchronous Communicating Components. In D. C. Schmidt R. Meersman, Z. Tari and al., eds., *On the Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, pages 1421–1438. Springer Verlag, 2003.

[34] Oscar Slotosch. Modelling and validation: AUTOFOCUS and quest. *Formal Aspects of Computing*, 12(4):225–227, 2000.

[35] Jean-Louis Sourrouille. A framework for the definition of behavior inheritance. *Journal of Object-Oriented Programming*, 9(1):17–21, 1996.

A The Cash-Point Description in LOTOS

```
(* messages:
0 : "take your card back"
1 : "take your money"
2 : "wrong pin code - card swallowed"
3 : "limit exceded or not enough money in till"
4 : "no authorization from bank"
*)

process TILL [card,pin,getSum,add,rec,send,cash,message](self:Till) : noexit :=

hide clock in (
   TILL_T1 [card,pin,getSum,add,rec,send,cash,message,clock](self)
)

where

   process TILL_T1 [card,pin,getSum,add,rec,send,cash,message,clock](self:Till) : noexit :=
      clock ; TILL_T1 [card,pin,getSum,add,rec,send,cash,message,clock](clock(self))
   []
      card?c:Card [(id(c) eq 2) and (sum(c) le max(c))] ;
                TILL_T2 [card,pin,getSum,add,rec,send,cash,message,clock](insertCard(self,c))
   endproc

   process TILL_T2 [card,pin,getSum,add,rec,send,cash,message,clock](self:Till) : noexit :=
      clock ; TILL_T2 [card,pin,getSum,add,rec,send,cash,message,clock](clock(self))
   []
      pin?code:PINNumber ;
           TILL_T3 [card,pin,getSum,add,rec,send,cash,message,clock](pin(self,code))
   endproc

   process TILL_T3 [card,pin,getSum,add,rec,send,cash,message,clock](self:Till) : noexit :=
      clock ; TILL_T3 [card,pin,getSum,add,rec,send,cash,message,clock](clock(self))
   []
      [c1(self)] -> getSum?sum:Money [sum gt 0];
                TILL_T6 [card,pin,getSum,add,rec,send,cash,message,clock](getSum(self,sum))
   []
      [not(c1(self))] -> pin?code:PINNumber ;
                TILL_T4 [card,pin,getSum,add,rec,send,cash,message,clock](pin(self,code))
   endproc

   process TILL_T4 [card,pin,getSum,add,rec,send,cash,message,clock](self:Till) : noexit :=
      clock ; TILL_T4 [card,pin,getSum,add,rec,send,cash,message,clock](clock(self))
   []
      [c1(self)] -> getSum?sum:Money [sum gt 0];
                TILL_T6 [card,pin,getSum,add,rec,send,cash,message,clock](getSum(self,sum))
   []
      [not(c1(self))] -> pin?code:PINNumber ;
                TILL_T5 [card,pin,getSum,add,rec,send,cash,message,clock](pin(self,code))
   endproc

   process TILL_T5 [card,pin,getSum,add,rec,send,cash,message,clock](self:Till) : noexit :=
      clock ; TILL_T5 [card,pin,getSum,add,rec,send,cash,message,clock](clock(self))
   []
      [c1(self)] -> getSum?sum:Money [sum gt 0];
                TILL_T6 [card,pin,getSum,add,rec,send,cash,message,clock](getSum(self,sum))
   []
      [not(c1(self))] -> message!2 of Nat ;
                TILL_T1 [card,pin,getSum,add,rec,send,cash,message,clock](keepCard(self))
  endproc

   process TILL_T6 [card,pin,getSum,add,rec,send,cash,message,clock](self:Till) : noexit :=
      clock ; TILL_T6 [card,pin,getSum,add,rec,send,cash,message,clock](clock(self))
   []
      [c2(self)] -> send!msgValidity(self) ;
                TILL_T7 [card,pin,getSum,add,rec,send,cash,message,clock](self)
   []
      [not(c2(self))] -> message!3 of Nat ;
                TILL_T9 [card,pin,getSum,add,rec,send,cash,message,clock](self)
   endproc

   process TILL_T7 [card,pin,getSum,add,rec,send,cash,message,clock](self:Till) : noexit :=
      clock ; TILL_T7 [card,pin,getSum,add,rec,send,cash,message,clock](clock(self))
   []
      rec?r:Ack [c3(self,r)] ;
                TILL_T8 [card,pin,getSum,add,rec,send,cash,message,clock](self)
   []
      rec?r:Ack [not(c3(self,r))] ; message!4 of Nat ;
                TILL_T9 [card,pin,getSum,add,rec,send,cash,message,clock](self)
   endproc

   process TILL_T8 [card,pin,getSum,add,rec,send,cash,message,clock](self:Till) : noexit :=
      clock ; TILL_T8 [card,pin,getSum,add,rec,send,cash,message,clock](clock(self))
   []
      message!1 of Nat ; cash!sum(self) ;
                TILL_T9 [card,pin,getSum,add,rec,send,cash,message,clock](giveCash(self))
   endproc

   process TILL_T9 [card,pin,getSum,add,rec,send,cash,message,clock](self:Till) : noexit :=
      clock ; TILL_T9 [card,pin,getSum,add,rec,send,cash,message,clock](clock(self))
   []
      message!0 of Nat; card!card(self) ;
                TILL_T1 [card,pin,getSum,add,rec,send,cash,message,clock](giveCard(self))
   endproc

endproc


type Till is PINNumber, Money, Card, Ack, Info

sorts Till

opns
   newTill (*! constructor *) : Money, Card, PINNumber, Money, Date -> Till

   (* transition constructors *)
   addCash : Till, Money -> Till
   insertCard : Till, Card -> Till
   pin : Till, PINNumber -> Till
   getSum : Till, Money -> Till
   giveCash,
   keepCard,
   giveCard,
   clock : Till -> Till

   (* variable observers -- selectors *)
   amount : Till -> Money
   card : Till -> Card
   code : Till -> PINNumber
   sum : Till -> Money
   date : Till -> Date

   (* other observers *)
   msgValidity : Till -> Info

   (* transition guards *)
   c1 : Till -> Bool
   c2 : Till -> Bool
   c3 : Till, Ack -> Bool
   (* helpers *)
   enoughMoneyInside : Till -> Bool
   cardConditions : Till -> Bool
   sameDate : Till -> Bool

eqns
forall self:Till,
       a,a1,sum,sum1:Money,
       c,c1:Card,
       code,code1:PINNumber,
       today:Date,
       r:Ack

   (* transition constructors *)
   ofsort Till
      addCash(newTill(a,c,code,sum,today),a1) = newTill(a+a1,c,code,sum,today);
      insertCard(newTill(a,c,code,sum,today),c1) = newTill(a,c1,code,sum,today);
      pin(newTill(a,c,code,sum,today),code1) = newTill(a,c,code1,sum,today);
      getSum(newTill(a,c,code,sum,today),sum1) = newTill(a,c,code,sum1,today);
      giveCash(newTill(a,c,code,sum,today)) =
                             newTill(a-sum,update(c,sum,today),code,sum,today);
      giveCard(newTill(a,c,code,sum,today)) = newTill(a,newCard(0,0,0,0,0),code,sum,today);
      keepCard(newTill(a,c,code,sum,today)) = newTill(a,newCard(0,0,0,0,0),code,sum,today);
      (today eq 0) => clock(newTill(a,c,code,sum,today)) = newTill(a,c,code,sum,1);
      (today eq 1) => clock(newTill(a,c,code,sum,today)) = newTill(a,c,code,sum,0);

   (* variable observers -- selectors *)
   ofsort Money
      amount(newTill(a,c,code,sum,today)) = a;
   ofsort Card
      card(newTill(a,c,code,sum,today)) = c;
   ofsort PINNumber
      code(newTill(a,c,code,sum,today)) = code;
   ofsort Money
      sum(newTill(a,c,code,sum,today)) = sum;
   ofsort Date
      date(newTill(a,c,code,sum,today)) = today;

   (* other observers *)
   ofsort Info
      msgValidity(self) = newInfo(id(card(self)),sum(self));

   (* helpers *)
   ofsort Bool
      sameDate(self) = last(card(self)) eq date(self);
      enoughMoneyInside(self) = (sum(self) le amount(self));
      cardConditions(self) = (sameDate(self) and
                            ( (sum(card(self))+sum(self) ) le max(card(self)) ))
                or (not(sameDate(self)) and ( sum(self) le max(card(self)) ));

   (* transition guards *)
   ofsort Bool
      c1(self) = (crypt(code(self)) eq code(card(self)));
      c2(self) = (enoughMoneyInside(self) and cardConditions(self));
      c3(self,r) = isOk(r);

endtype



process ACKCONNECTION [receive,send](self:AckConnection) : noexit :=

   ACKCONNECTION_D1 [receive,send](self)

where
```

```
   process ACKCONNECTION_D1 [receive,send](self:AckConnection) : noexit :=
      receive?m:Ack ; ACKCONNECTION_D2 [receive,send](push(self,m))
   endproc

   process ACKCONNECTION_D2 [receive,send](self:AckConnection) : noexit :=
      send!top(self) ; ACKCONNECTION_D1 [receive,send](pop(self))
   endproc

endproc

type AckConnection is Ack

sorts AckConnection

opns
   newAckConnection (*! constructor *) :                    -> AckConnection
   push             (*! constructor *) : AckConnection, Ack -> AckConnection
   pop              : AckConnection                         -> AckConnection
   top              : AckConnection                         -> Ack

eqns
forall self:AckConnection, m:Ack

   ofsort AckConnection
      pop(push(self,m)) = self;

   ofsort Ack
      top(push(self,m)) = m;

endtype


process INFOCONNECTION [receive,send](self:InfoConnection) : noexit :=

   INFOCONNECTION_D1 [receive,send](self)

where

   process INFOCONNECTION_D1 [receive,send](self:InfoConnection) : noexit :=
      receive?m:Info ; INFOCONNECTION_D2 [receive,send](push(self,m))
   endproc

   process INFOCONNECTION_D2 [receive,send](self:InfoConnection) : noexit :=
      send!top(self) ; INFOCONNECTION_D1 [receive,send](pop(self))
   endproc

endproc

type InfoConnection is Info

sorts InfoConnection

opns
   newInfoConnection (*! constructor *) :                      -> InfoConnection
   push              (*! constructor *) : InfoConnection, Info -> InfoConnection
   pop               : InfoConnection                          -> InfoConnection
   top               : InfoConnection                          -> Info

eqns
forall self:InfoConnection, m:Info

   ofsort InfoConnection
      pop(push(self,m)) = self;
   ofsort Info
      top(push(self,m)) = m;

endtype


process DATABASE [lock,check,reply] (self:Database) : noexit :=

   DATABASE_DB1 [lock,check,reply] (self)

where

   process DATABASE_DB1 [lock,check,reply] (self:Database) : noexit :=
      check?n:Nat?s:Money [isLocked(self,n)] ;
                    DATABASE_DB1 [lock,check,reply] (check(self,n,s))
   []
      lock?n:Nat?id:Ident [not(locked(self,id))] ;
                    DATABASE_DB1 [lock,check,reply] (lock(self,n,id))
   []
      reply?n:Nat!nok [not(isOk(checked(self,n)))] ;
                    DATABASE_DB1 [lock,check,reply] (unlock(self,n))
   []
      reply?n:Nat!ok  [isOk(checked(self,n))] ;
                    DATABASE_DB1 [lock,check,reply] (unlock(withdraw(self,n),n))
   endproc

endproc


type Database is Accounts, Informations, Ack

sorts Database

opns
   newDatabase (*! constructor *) : Accounts, Informations -> Database

   lock                           : Database, Nat, Ident   -> Database
   check                          : Database, Nat, Money   -> Database
   unlock                         : Database, Nat          -> Database
   withdraw                       : Database, Nat          -> Database

   locked                         : Database, Ident        -> Bool
   isLocked                       : Database, Nat          -> Bool
   checked                        : Database, Nat          -> Ack

eqns
forall accounts:Accounts, infos:Informations, n:Nat, id:Ident, s:Money
   ofsort Database
      lock(newDatabase(accounts,infos),n,id) =
                newDatabase(accounts,insert(newInformation(n,id,0),infos));
      check(newDatabase(accounts,infos),n,s) =
                newDatabase(accounts,assign(infos,n,s));
      unlock(newDatabase(accounts,infos),n)  =
                newDatabase(accounts,remove(infos,n));
      withdraw(newDatabase(accounts,infos),n)=
                newDatabase(withdraw(accounts,client(infos,n),sum(infos,n)),infos)
      (* withdraw should be called before unlock *)
   ofSort Bool
      locked(newDatabase(accounts,infos),id) = isIn(infos,id);
      isLocked(newDatabase(accounts,infos),n) = hasKey(infos,n);

   ofSort Ack
      ((hasKey(infos,n)) and ((sum(infos,n))>0)) and
      ((account(accounts,client(infos,n))) >= (sum(infos,n)))
                => checked(newDatabase(accounts,infos),n) = ok;
      not(((hasKey(infos,n)) and ((sum(infos,n))>0)) and
         ((account(accounts,client(infos,n))) >= (sum(infos,n))))
               => checked(newDatabase(accounts,infos),n) = nok;

endtype


process BANKINTERFACE [receive,lock,check,get,send] (self:BankInterface) : noexit :=

   BANKINTERFACE_B1 [receive,lock,check,get,send] (self)

where

   process BANKINTERFACE_B1 [receive,lock,check,get,send] (self:BankInterface) : noexit :=
      receive?info:Info ; BANKINTERFACE_B2 [receive,lock,check,get,send] (receive(self,info))
   endproc

   process BANKINTERFACE_B2 [receive,lock,check,get,send] (self:BankInterface) : noexit :=
      lock!ident(self)!client(info(self)) ;
                   BANKINTERFACE_B3 [receive,lock,check,get,send] (self)
   endproc

   process BANKINTERFACE_B3 [receive,lock,check,get,send] (self:BankInterface) : noexit :=
      check!ident(self)!sum(info(self)) ;
                   BANKINTERFACE_B4 [receive,lock,check,get,send] (self)
   endproc

   process BANKINTERFACE_B4 [receive,lock,check,get,send] (self:BankInterface) : noexit :=
      get?bi:Nat?a:Ack [iam(self,bi)] ;
                   BANKINTERFACE_B5 [receive,lock,check,get,send] (get(self,a))
   endproc

   process BANKINTERFACE_B5 [receive,lock,check,get,send] (self:BankInterface) : noexit :=
      send!ack(self) ; BANKINTERFACE_B1 [receive,lock,check,get,send] (self)
   endproc

endproc


type BankInterface is Natural, Info, Ack

sorts BankInterface

opns
   newBankInterface (*! constructor *) : Nat, Info, Ack      -> BankInterface

   receive                             : BankInterface, Info -> BankInterface
   get                                 : BankInterface, Ack  -> BankInterface

   ack                                 : BankInterface       -> Ack
   ident                               : BankInterface       -> Nat
   info                                : BankInterface       -> Info

   iam                                 : BankInterface, Nat  -> Bool

eqns
forall self:BankInterface, n,n1:Nat, info,info1:Info, a,a1:Ack
   ofSort BankInterface
      receive(newBankInterface(n,info,a),info1) = newBankInterface(n,info1,a);
      get(newBankInterface(n,info,a),a1) = newBankInterface(n,info,a1);

   ofSort Ack
      ack(newBankInterface(n,info,a)) = a;
   ofSort Nat
      ident(newBankInterface(n,info,a)) = n;
   ofSort Info
      info(newBankInterface(n,info,a)) = info;

   ofSort Bool
      iam(self,n1) = (ident(self) eq n1);

endtype


process CLIENT [card,pin,getSum,cash,message] (c:Card,hasCard:Bool) : noexit :=

   [hasCard] => card!c!true ; CLIENT [card,pin,getSum,cash,message] (c,false)
[]
   pin!pin(c) ; CLIENT [card,pin,getSum,cash,message] (c,hasCard)
[]
   getSum?s:Money [s gt 0] ; CLIENT [card,pin,getSum,cash,message] (c,hasCard)
[]
   [not(hasCard)] => card?newc!false ; CLIENT [card,pin,getSum,cash,message] (newc,true)
[]
   cash?s:Money ; CLIENT [card,pin,getSum,cash,message] (c,hasCard)
[]
   message?info:Nat ; CLIENT [card,pin,getSum,cash,message] (c,hasCard)

endproc


(*

SYSTEM TILL

v1

N = 1
(un seul Till, une seule TillLine, une seule BankInterface)

Structuration de l’architecture modifiée

*)

specification SYSTEM [card,pin,getSum,add,cash,message (* TILL *)
                     ] : noexit

library
   (* basic imports *)
   Boolean, Natural,
   (* simple ADT *)
   Money, Date, Ident, PINNumber, Card,
   Info, Ack, Account, Information,
   Accounts, Informations,
   (* component ADT *)
   Till,
   AckConnection, InfoConnection, (* use genericity instead ?? *)
   BankInterface,
   Database
endlib

behaviour

   let account1:Account = newAccount(1,1),
       account2:Account = newAccount(2,2)
   in
   let
       initial_accounts:Accounts = insert(account1,insert(account2,newAccounts))
   in (
   hide X5,X6,X7 in (
      (hide X3, X4 in (
         (hide X1,X2 in (
            TILL           [card,pin,getSum,add,X2,X1,cash,message]
                                          (newTill(2,newCard(0,0,0,0,0),0,0,0))
         |[X1,X2]|
            (
            INFOCONNECTION [X1,X3]                    (newInfoConnection)
            |||
            ACKCONNECTION  [X4,X2]                    (newAckConnection)
            )
         ))
         |[X3,X4]|
            BANKINTERFACE  [X3,X5,X6,X7,X4]            (newBankInterface(0,newInfo(0,0),ok))
      ))
      |[X5,X6,X7]|
         DATABASE          [X5,X6,X7]
                                         (newDatabase(initial_accounts,newInformations))
   )
   )

where
   library
      PROC_TILL,                                     (* card,pin,getSum,add,rec,send,cash *)
      (* CONNECTIONS, use genericity instead ?? *)
      PROC_ACKCONNECTION,                            (* receive,send *)
      PROC_INFOCONNECTION,                           (* receive,send *)
      PROC_BANKINTERFACE,                            (* receive,lock,check,get,send *)
      PROC_DATABASE                                  (* lock,check,reply *)
   endlib

endspec
```

B Representation in Python

The principles to translate an STS in SyCLAP are to build a textual representation of the transition system (`.aut` file) and a Python class representing the ADT (`.py` file). The translation from an ADT into a class Python is straightforward as illustrated with the pattern describes below. One important difference between both is that ADTs have a pure functional semantics while the Python classes have an imperative one. Before applying an action to an object the configuration graph algorithm creates a copy of the instance, applies the action and returns the result. Thus this achieves that the class with the copy mechanism has a pure functional semantics. However, as the reader may see it in the two next examples, some additional features are needed. These features are required by the SyCLAP system to compute configurations and to resolve the communications between the components. A transition action in the STS has an associated method with the same name, and if it is a receiver, it has one parameter for each value receipt. A transition guard is implemented by a pure boolean function with the same name. Emissions are done using pure functional methods returning a list of values. Communications are one way and multiple emissions are groups into lists of values. The * operator is viewed as an iteration mechanism over lists of values. It is implemented by the used of loops, slicing and map operators.

B.1 The Translation Pattern

The previous grammar (cf Section 3) is a general one, we adopt some methodological rules to build the ADT. In few words, we expect them defining total functions and we consider only one basic generator. Criterion coming from sufficient completeness requires that we defined the operations over the generator. In our prototype, see below, we define the `Name` sort and some imported datatypes. There is only one generator (`newName`), one constructor, one observer and an equality function.

```
/* a prototype  ADT example */

Sort Name

Imports Boolean, T1, ..., Tn

Opns
/* generator of the sort */
newName :  R1 x ... x Rs -> Name

/* a constructor operation */
op1 : Name x A1 x ... x Al -> Name

/*  an observer */
op2 : Name x B1 X .. x Bm   -> Ti

/* an equality boolean function */
equals : Name x Name -> Boolean

Variables

self : Name ; x1 : R1 ;  ... ; xs : Rs ; y1 : R1 ;  ... ; ys : Rs ;
 a1 : A1 ;  ... ; al : Al ; b1 : B1 ;  ... ; bm : Bm ;

Axioms

cond1 => op1(self, a1, ..., al)) =  newName(rt1, ..., rts)

cond2 => op2(self, b1, ..., bm)) =  rtz

equals(newName(x1, ..., xs), newName(y1, ..., ys))
                            = equals(x1, y1) and ... and equals(xs, ys)
```

We have axioms for each operation, in the general case there are more than one per operation, `condi` and `rti` are algebraic terms. If we note `translate` the translation of an algebraic term into its Python equivalent expression we get the following Python class.

```
#-----------------------
# Name.py
# 3/10/2006
# the translation of the prototype ADT
#---------------------

import T1
    ...
import Tn
import R1
    ...
import Rs
import A1
    ...
import Al
import B1
    ...
import Bm
```

```
class Data(Data.Data):

    #---------------
    # the x1 instance variable
    def set_x1(self, t):
        self.x1 = t
    # -----

    ...

    #---------------
    # the xs instance variable
    def set_xs(self, t):
        self.xs = t
    # -----

    #---------------
    # initialisation
    def __init__(self, values=[]):
        self.x1 = p1
        ...
        self.xs = ps
    # -----

    #---------------
    # textual representation
    def __str__(self):
        return str(self.x1) + " " + ... + " " + str(self.xs)
    #---------------

    #-------------------
    # deep equality to compare configurations
    def deep_equal(self, other):
        return self.x1 == other.x1 AND ... AND self.xs == other.xs
    #-------------------

    # -----------------------
    # op1 translation
    def op1(self,  a1, ..., al):
        if (translate<cond1>):
            self.set_x1(translate<rt1>)
            ...
            self.set_xs(translate<rts>)
    # -----------------------

    # -----------------------
    # op2 translation
    def op2(self,  b1, ..., bm):
        if (translate<cond2>):
            return translate<rt2>
    # -----------------------

#fin Data -----------------
```

We define a class with as instances variables the arguments of the generator. We also add a textual representation and an initialisation operator where `p1, ..., ps` are initial values coming from the global context. The equality function has a simple translation. An observer has a translation into a `if then return` structure. Multiple conditional axioms leads to `if then else if` control structures.

B.2 The Python Class: `MsgConnection`

```
#-----------------------
# MONCLAP3/CASH2/GLOBAL/MsgConnection.py
# 2/10/2006
#---------------------

__author__ = "JCR"
__version__ = "3.0"

import Dico

class Data(Dico.Dico):
    print 'Chargement de la classe CASH2/GLOBAL/MsgConnection'

    #----------------
    # msg is either Info or Ack, but Python is dynamically type-checked
    def set_msg(self, i):
        self.msg = i
    # ----- fin

    #----------------
    # initialisation
    def __init__(self, values=[]):
        self.msg = 0   # the void message
    # ----- fin init

    #----------------
    # textual representation
    def __str__(self):
        return " " + str(self.msg)
    #----------------

    #--------------------
    # deep equality to compare configurations
    def deep_equal(self, other):
        return self.msg == other.msg
    #--------------------

    # -----------------------
    # emitter of the top message
    def top(self):
        return [self.msg]
    # -----------------------

    # --- actions ----------
    # -----------------------
    # implementation of push
    def receive(self, i):
        self.set_msg(i)
    # -----------------------

    # -----------------------
    # implementation of pop
    def send(self):
        self.set_msg(0)
    # -----------------------

#fin Data -----------------
```

B.3 The Python Class: `DataBase`

```
#-----------------------
# MONCLAP3/CASH2/GLOBAL/DataBase.py
# 2/10/2006
#---------------------
```

```
__author__ = "JCR"
__version__ = "3.0"

import Dico
import Informations

class Data(Dico.Dico):
    print 'Chargement de la classe CASH2/GLOBAL/DataBase'

    #----------------
    # list of accounts
    def set_accounts(self, a):
        self.accounts = a
    # informations
    def set_info(self, inf):
        self.info = inf
    # ----- fin

    #----------------
    # initialisation
    def __init__(self, values=[]):
        self.set_accounts(values[0])
        self.set_info(Informations.Informations())
    # ----- fin init

    #----------------
    # external representation
    def __str__(self):
        return "DB|" + str(self.accounts) + " " + str(self.info) + "|"
    #----------------

    #--------------------
    # deep equality
    def deep_equal(self, other):
        return (self.accounts.deep_equal(other.accounts)) and  (self.info.deep_equal(other.info))
    #--------------------

    # ------------------------
    # a guard
    def notLocked(self, bi, cli, s):
        return not self.info.isIn(cli)
    # ------------------------

    # ------------------------
    # generator for *[bi True]
    def checked(self):
        tmp = []
        for biclis in self.info.tuples:
            if (biclis[2] > 0) and (biclis[2] <= self.accounts.account(biclis[1])):
                tmp = tmp + [[biclis[0], True]]
        return tmp
    # ------------------------

    # ------------------------
    # generator for *[bi True]
    def notChecked(self):
        tmp = []
        for biclis in self.info.tuples:
            if (biclis[2] > 0) and (biclis[2] > self.accounts.account(biclis[1])):
                tmp = tmp + [[biclis[0], False]]
        return tmp
    # ------------------------

    # ------------------------
    def lock(self, bi, cli, s):
        self.info.cons(bi, cli, s)
    # ------------------------

    # ------------------------
    # reply / unlock
    def replyU(self, bi, ack):
        self.info.remove(bi)
    # ------------------------

    # ------------------------
    # reply / widthdraw + unlock
    def reply(self, bi, ack):
            self.accounts.withdraw(self.info.client(bi), self.info.sum(bi))
            self.info.remove(bi)
    # ------------------------

#fin Data -----------------
```

```
Variables

a,sum,a1,sum1:Money; c,c1:Card; code,code1,code2:PinNumber;
r:Ack; ,today,today1,today2:Date ; cpt : Natural; self:Till

Axioms

addCash(newTill(a,c,code,sum,today,cpt),a1) = newTill(a+a1,c,code,sum,today,cpt)
insertCard(newTill(a,c,code,sum,today,cpt),c2) = newTill(a,c2,code,sum,today,0)
pin(newTill(a,c,code,sum,today,cpt),code2) = newTill(a,c,code2,sum,today,1+cpt)
getSum(newTill(a,c,code,sum,today,cpt),sum2) = newTill(a,c,code,sum2,today,cpt)
giveCash(newTill(a,c,code,sum,today,cpt)) =
           newTill(a-sum,updateDailyLimit(card(self),sum,today),code,sum,today,cpt)
giveCard(newTill(a,c,code,sum,today,cpt)) = newTill(a, noCard, code, sum,today,cpt)
keepCard(newTill(a,c,code,sum,today,cpt)) = newTill(a, noCard, code, sum,today,cpt)
clock(newTill(a,c,code,sum,today,cpt)) = newTill(a, noCard, code, sum,inc(today),cpt)

/* constant for initialisation */
new = newTill(0, noCard, 0, 0, 0, 0)
/* accessors for the card, the code, the required amount,
the counter of pin tests, the local amount, and the current date */
card(newTill(a,c,code,sum,today,cpt)) = c
code(newTill(a,c,code,sum,today,cpt)) = code
sum(newTill(a,c,code,sum,today,cpt)) = sum
counter(newTill(a,c,code,sum,today,cpt)) = cpt
amount(newTill(a,c,code,sum,today,cpt)) = a
date(newTill(a,c,code,sum,today,cpt)) = today

/* compute the message to allow withdraw */
msgValidity(self) = newInfo(id(card(self)), sum(self))

/* pin code control ok, control wrong and wrong after 3 tests */
pinOK(self) = equals(crypt(code(self)), code(card(self))) AND counter(self) <= 3
retry(self) = not equals(crypt(code(self)), code(card(self))) AND counter(self) < 3
fail(self) = not equals(crypt(code(self)), code(card(self))) AND counter(self) >= 3

/* local controls of the till and the card */
check(self) = (sum(self) <= amount(self)) AND
       ((equals(last(card(self)), date(self))
                   AND (sum(card(self)) + sum(self) <= max(card(self))))
     OR (not equals(last(card(self)), date(self))
                   AND (sum(self) <= max(card(self)))))
/* acknowledgment from the bank */
ack(self, r) = isOk(r)
```

Figure 10: The `Till` Datatype (part II, axioms).

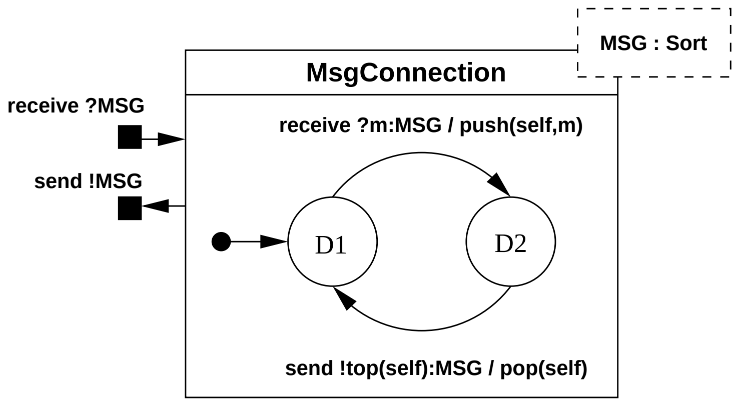
MSG : Sort
MsgConnection
receive ?MSG
send !MSG
receive ?m:MSG / push(self,m)
D1
D2
send !top(self):MSG / pop(self)

Figure 11: The `MsgConnection` STS.

```
Sort MsgConnection
Imports Boolean, MSG

Opns
/* generator for connection */
newMsgConnection : Msg -> MsgConnection
/* add a message */
push : MsgConnection x MSG  -> MsgConnection
/* remove the message */
pop :  MsgConnection -> MsgConnection
/* accessor for the message */
top : MsgConnection -> MSG

Constantes

voidMsg : Msg

Variables

m, m1 : MSG, self, self1 : MsgConnection

Axioms

push(newMsgConnection(m), m1) = newMsgConnection(m1)
top(newMsgConnection(m)) = m
pop(newMsgConnection(m)) = newMsgConnection(voidMsg)
```

Figure 12: The `MsgConnection Data` Type.

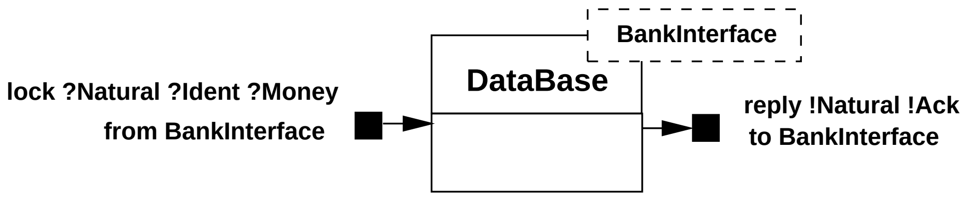

Figure 13: The `DataBase` Interface.

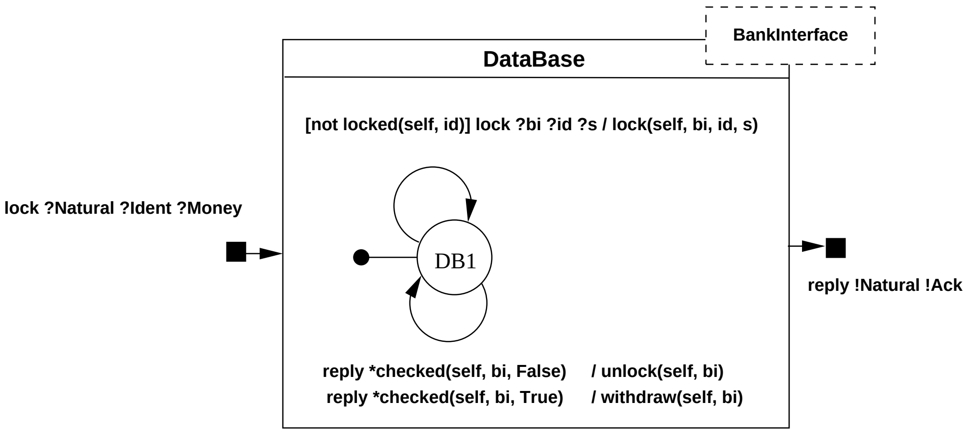
DataBase
BankInterface
[not locked(self, id)] lock ?bi ?id ?s / lock(self, bi, id, s)
lock ?Natural ?Ident ?Money
DB1
reply !Natural !Ack
reply *checked(self, bi, False) / unlock(self, bi)
reply *checked(self, bi, True) / withdraw(self, bi)

Figure 14: The `DataBase` STS.

```
Sort DataBase
Imports Boolean, Accounts, Informations, Natural, Ident

Opns
/* generator for database */
newDB :  Accounts x Informations -> DataBase
/* lock the database */
lock : DataBase x Natural x Ident x Money  -> DataBase
/* unlock it */
unlock : DataBase x Natural   -> DataBase
/* withdraw and unlock */
withdraw : DataBase x Natural   -> DataBase
/* is a client already locked ? */
locked : DataBase x Ident -> Boolean
/* is the client known and with sufficiently enough cash */
checked : DataBase  x Natural x Ack         -> Boolean

Variables

a, a1 : Accounts; i,i1 : Informations;  self, self1 : DataBase;
bi : Natural; id : Ident

Axioms

lock(newDB(a, i), bi, id, s) = newDB(a, cons((bi, id, s), i))
locked(newDB(a, i), id) = isIn(i, id)
checked(newDB(a, i), bi, a1) = (hasKey(i, bi) AND (sum(i, bi) > 0) AND
                            (a1 = (account(a, client(i, bi)) >= sum(i, bi))))
unlock(newDB(a, i), bi) = newDB(a, remove(i, bi))
withdraw(newDB(a, i), bi)
             = newDB(withdraw(a, client(i, bi), sum(i, bi)), remove(i, bi))
```

Figure 15: The `DataBase` Datatype.

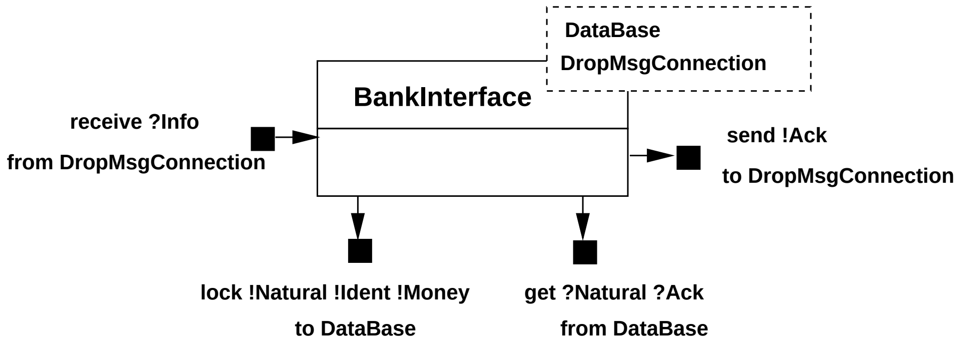

Figure 16: The `BankInterface` Interface.

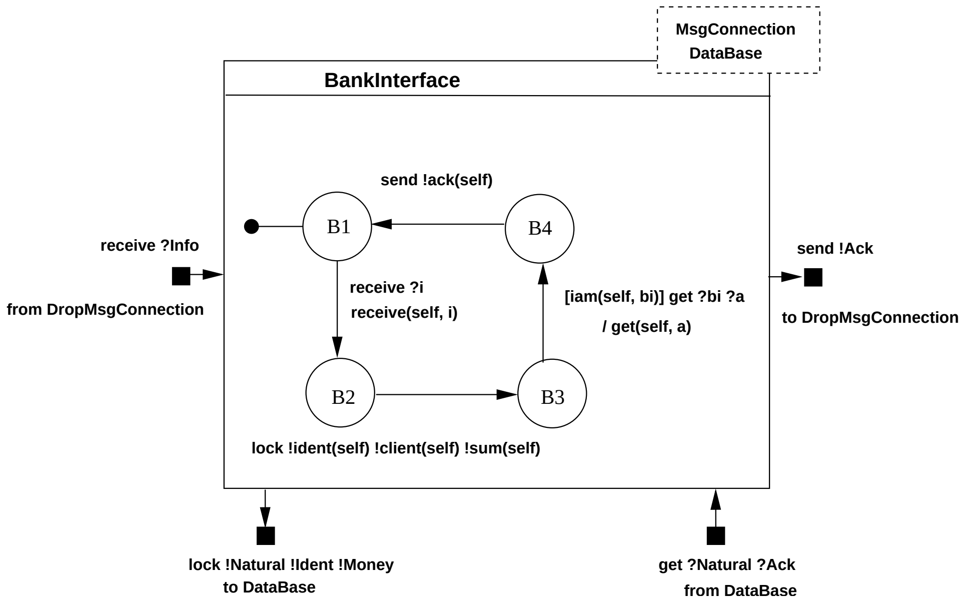

Figure 17: The `BankInterface` STS.

```
Sort BankInterface
Imports Boolean, Info, Natural, Ident

Opns
/* generator of a bank interface */
newBI :  Natural x Ident x Money x Ack -> BankInterface
/* accessor for interface identity */
ident : BankInterface -> Natural
/* accessor for client identity */
client : BankInterface -> Ident
/* accessor for required amount */
sum : BankInterface -> Money
/* accessor for  acknowledgment */
ack : BankInterface -> Ack
/* receipt of an information */
receive : BankInterface x Info  -> BankInterface
/*  receipt of an acknowledgment */
get : BankInterface x Ack  -> BankInterface
/* receipt test */
iam : BankInterface x Natural -> Boolean

Variables

self, self1 : BankInterface; a,a1 : Ack; bi, bi1, s, s1 : Natural;
id, id1 : Ident; in : Info

Axioms

ident(newBI(bi, id, s, a), in) = bi
client(newBI(bi, id, s, a), in) = id
sum(newBI(bi, id, s, a), in) = s
ack(newBI(bi, id, s, a), in) = a
receive(newBI(bi, id, s, a), in) = newBI(bi, client(in), sum(in), a)
get(newBI(bi, id, s, a), a1) = newBI(bi, id, s, a1)
iam(self, bi) = equals(ident(self), bi)
```

Figure 18: The `BankInterface` Datatype.

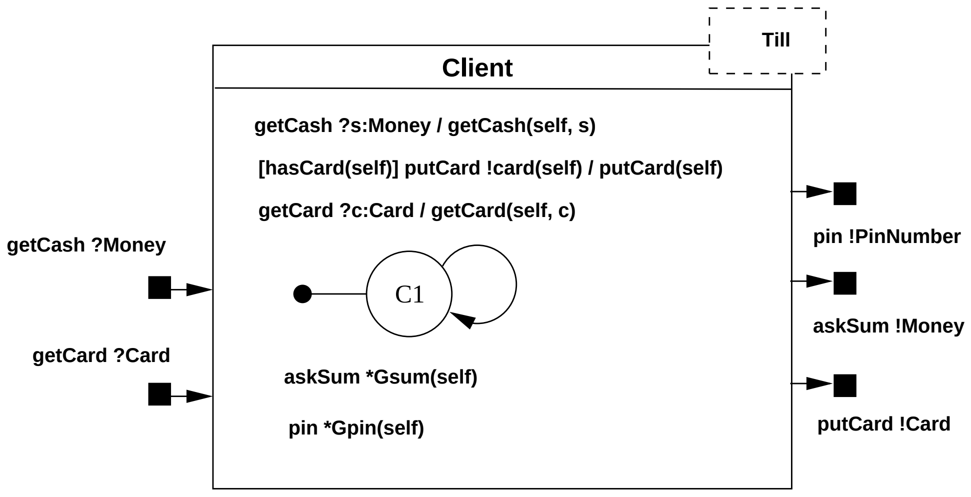
Till
Client
getCash ?s:Money / getCash(self, s)
[hasCard(self)] putCard !card(self) / putCard(self)
getCard ?c:Card / getCard(self, c)
C1
askSum *Gsum(self)
pin *Gpin(self)
getCash ?Money
getCard ?Card
pin !PinNumber
askSum !Money
putCard !Card

Figure 19: The `Client` STS.

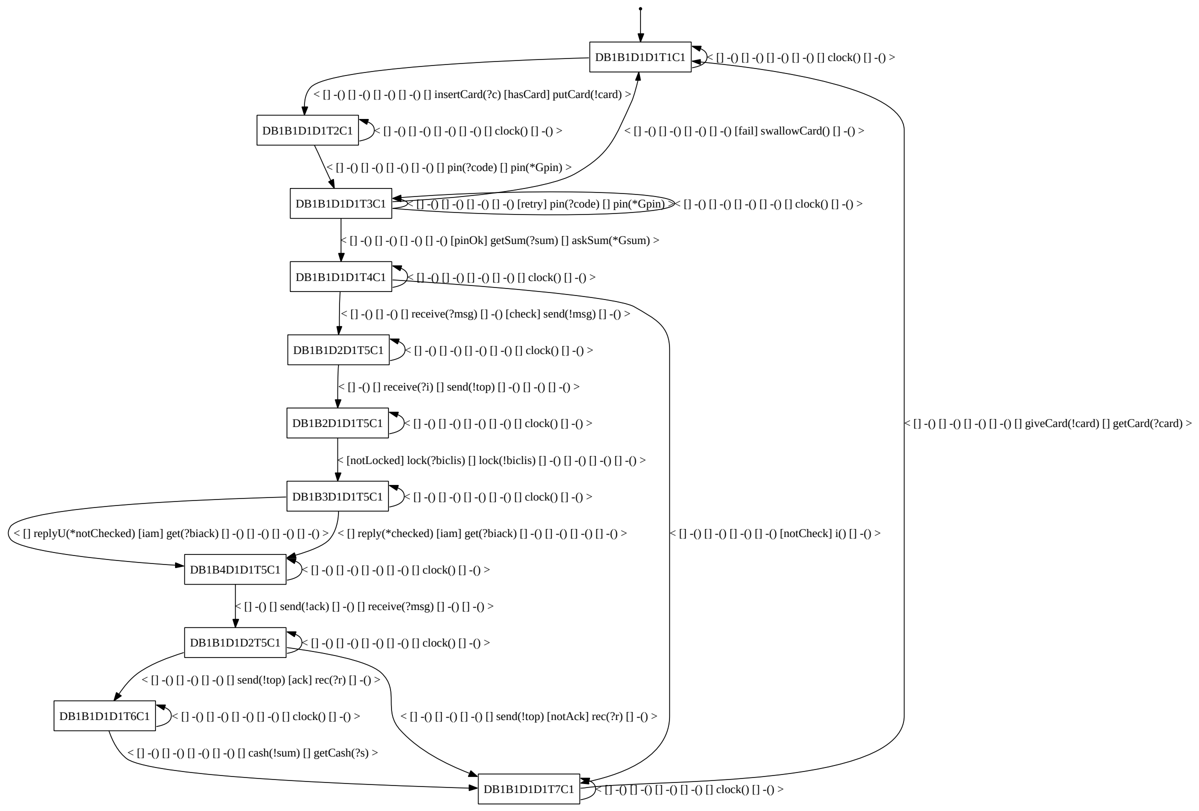
DB1B1D1D1T1C1
< [] -() [] -() [] -() [] -() [] clock() [] -() >
< [] -() [] -() [] -() [] -() [] insertCard(?c) [hasCard] putCard(!card) >
DB1B1D1D1T2C1
< [] -() [] -() [] -() [] -() [] clock() [] -() >
< [] -() [] -() [] -() [] -() [fail] swallowCard() [] -() >
< [] -() [] -() [] -() [] -() [] pin(?code) [] pin(*Gpin) >
DB1B1D1D1T3C1
< [] -() [] -() [] -() [] -() [retry] pin(?code) [] pin(*Gpin) >
< [] -() [] -() [] -() [] -() [] clock() [] -() >
< [] -() [] -() [] -() [] -() [pinOk] getSum(?sum) [] askSum(*Gsum) >
DB1B1D1D1T4C1
< [] -() [] -() [] -() [] -() [] clock() [] -() >
< [] -() [] -() [] receive(?msg) [] -() [check] send(!msg) [] -() >
DB1B1D2D1T5C1
< [] -() [] -() [] -() [] -() [] clock() [] -() >
< [] -() [] receive(?i) [] send(!top) [] -() [] -() [] -() >
DB1B2D1D1T5C1
< [] -() [] -() [] -() [] -() [] clock() [] -() >
< [] -() [] -() [] -() [] -() [] giveCard(!card) [] getCard(?card) >
< [notLocked] lock(?biclis) [] lock(!biclis) [] -() [] -() [] -() [] -() >
DB1B3D1D1T5C1
< [] -() [] -() [] -() [] -() [] clock() [] -() >
< [] replyU(*notChecked) [iam] get(?biack) [] -() [] -() [] -() [] -() >
< [] reply(*checked) [iam] get(?biack) [] -() [] -() [] -() [] -() >
< [] -() [] -() [] -() [] -() [notCheck] i() [] -() >
DB1B4D1D1T5C1
< [] -() [] -() [] -() [] -() [] clock() [] -() >
< [] -() [] send(!ack) [] -() [] receive(?msg) [] -() [] -() >
DB1B1D1D2T5C1
< [] -() [] -() [] -() [] -() [] clock() [] -() >
< [] -() [] -() [] -() [] send(!top) [ack] rec(?r) [] -() >
DB1B1D1D1T6C1
< [] -() [] -() [] -() [] -() [] clock() [] -() >
< [] -() [] -() [] -() [] send(!top) [notAck] rec(?r) [] -() >
< [] -() [] -() [] -() [] -() [] cash(!sum) [] getCash(?s) >
DB1B1D1D1T7C1
< [] -() [] -() [] -() [] -() [] clock() [] -() >

Figure 20: The global STS with N=1 (one client and one till).

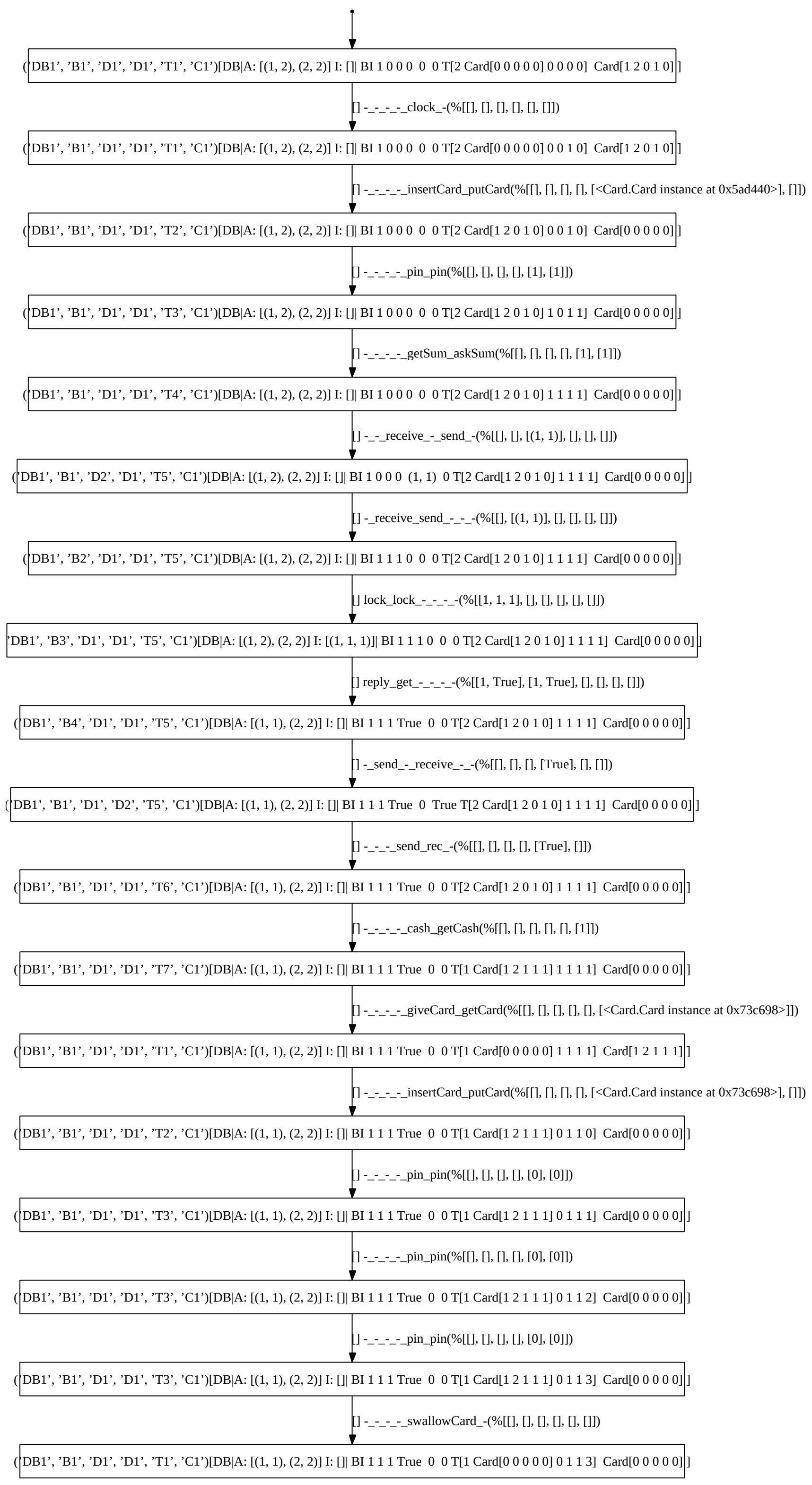
('DB1', 'B1', 'D1', 'D1', 'T1', 'C1')[DB|A: [(1, 2), (2, 2)] I: []| BI 1 0 0 0 0 0 T[2 Card[0 0 0 0 0] 0 0 0 0] Card[1 2 0 1 0]]
[] -_-_-_-_clock_-(%[[], [], [], [], [], []])
('DB1', 'B1', 'D1', 'D1', 'T1', 'C1')[DB|A: [(1, 2), (2, 2)] I: []| BI 1 0 0 0 0 0 T[2 Card[0 0 0 0 0] 0 0 1 0] Card[1 2 0 1 0]]
[] -_-_-_-_insertCard_putCard(%[[], [], [], [], [<Card.Card instance at 0x5ad440>], []])
('DB1', 'B1', 'D1', 'D1', 'T2', 'C1')[DB|A: [(1, 2), (2, 2)] I: []| BI 1 0 0 0 0 0 T[2 Card[1 2 0 1 0] 0 0 1 0] Card[0 0 0 0 0]]
[] -_-_-_-_pin_pin(%[[], [], [], [], [1], [1]])
('DB1', 'B1', 'D1', 'D1', 'T3', 'C1')[DB|A: [(1, 2), (2, 2)] I: []| BI 1 0 0 0 0 0 T[2 Card[1 2 0 1 0] 1 0 1 1] Card[0 0 0 0 0]]
[] -_-_-_-_getSum_askSum(%[[], [], [], [], [1], [1]])
('DB1', 'B1', 'D1', 'D1', 'T4', 'C1')[DB|A: [(1, 2), (2, 2)] I: []| BI 1 0 0 0 0 0 T[2 Card[1 2 0 1 0] 1 1 1 1] Card[0 0 0 0 0]]
[] -_-_receive_-_send_-(%[[], [], [(1, 1)], [], [], []])
('DB1', 'B1', 'D2', 'D1', 'T5', 'C1')[DB|A: [(1, 2), (2, 2)] I: []| BI 1 0 0 0 (1, 1) 0 T[2 Card[1 2 0 1 0] 1 1 1 1] Card[0 0 0 0 0]]
[] -_receive_send_-_-_-(%[[], [(1, 1)], [], [], [], []])
('DB1', 'B2', 'D1', 'D1', 'T5', 'C1')[DB|A: [(1, 2), (2, 2)] I: []| BI 1 1 1 0 0 0 T[2 Card[1 2 0 1 0] 1 1 1 1] Card[0 0 0 0 0]]
[] lock_lock_-_-_-_-(%[[1, 1, 1], [], [], [], [], []])
'DB1', 'B3', 'D1', 'D1', 'T5', 'C1')[DB|A: [(1, 2), (2, 2)] I: [(1, 1, 1)]| BI 1 1 1 0 0 0 T[2 Card[1 2 0 1 0] 1 1 1 1] Card[0 0 0 0 0]]
[] reply_get_-_-_-_-(%[[1, True], [1, True], [], [], [], []])
('DB1', 'B4', 'D1', 'D1', 'T5', 'C1')[DB|A: [(1, 1), (2, 2)] I: []| BI 1 1 1 True 0 0 T[2 Card[1 2 0 1 0] 1 1 1 1] Card[0 0 0 0 0]]
[] -_send_-_receive_-_-(%[[], [], [], [True], [], []])
('DB1', 'B1', 'D1', 'D2', 'T5', 'C1')[DB|A: [(1, 1), (2, 2)] I: []| BI 1 1 1 True 0 True T[2 Card[1 2 0 1 0] 1 1 1 1] Card[0 0 0 0 0]]
[] -_-_-_send_rec_-(%[[], [], [], [], [True], []])
('DB1', 'B1', 'D1', 'D1', 'T6', 'C1')[DB|A: [(1, 1), (2, 2)] I: []| BI 1 1 1 True 0 0 T[2 Card[1 2 0 1 0] 1 1 1 1] Card[0 0 0 0 0]]
[] -_-_-_-_cash_getCash(%[[], [], [], [], [], [1]])
('DB1', 'B1', 'D1', 'D1', 'T7', 'C1')[DB|A: [(1, 1), (2, 2)] I: []| BI 1 1 1 True 0 0 T[1 Card[1 2 1 1 1] 1 1 1 1] Card[0 0 0 0 0]]
[] -_-_-_-_giveCard_getCard(%[[], [], [], [], [], [<Card.Card instance at 0x73c698>]])
('DB1', 'B1', 'D1', 'D1', 'T1', 'C1')[DB|A: [(1, 1), (2, 2)] I: []| BI 1 1 1 True 0 0 T[1 Card[0 0 0 0 0] 1 1 1 1] Card[1 2 1 1 1]]
[] -_-_-_-_insertCard_putCard(%[[], [], [], [], [<Card.Card instance at 0x73c698>], []])
('DB1', 'B1', 'D1', 'D1', 'T2', 'C1')[DB|A: [(1, 1), (2, 2)] I: []| BI 1 1 1 True 0 0 T[1 Card[1 2 1 1 1] 0 1 1 0] Card[0 0 0 0 0]]
[] -_-_-_-_pin_pin(%[[], [], [], [], [0], [0]])
('DB1', 'B1', 'D1', 'D1', 'T3', 'C1')[DB|A: [(1, 1), (2, 2)] I: []| BI 1 1 1 True 0 0 T[1 Card[1 2 1 1 1] 0 1 1 1] Card[0 0 0 0 0]]
[] -_-_-_-_pin_pin(%[[], [], [], [], [0], [0]])
('DB1', 'B1', 'D1', 'D1', 'T3', 'C1')[DB|A: [(1, 1), (2, 2)] I: []| BI 1 1 1 True 0 0 T[1 Card[1 2 1 1 1] 0 1 1 2] Card[0 0 0 0 0]]
[] -_-_-_-_pin_pin(%[[], [], [], [], [0], [0]])
('DB1', 'B1', 'D1', 'D1', 'T3', 'C1')[DB|A: [(1, 1), (2, 2)] I: []| BI 1 1 1 True 0 0 T[1 Card[1 2 1 1 1] 0 1 1 3] Card[0 0 0 0 0]]
[] -_-_-_-_swallowCard_-(%[[], [], [], [], [], []])
('DB1', 'B1', 'D1', 'D1', 'T1', 'C1')[DB|A: [(1, 1), (2, 2)] I: []| BI 1 1 1 True 0 0 T[1 Card[0 0 0 0 0] 0 1 1 3] Card[0 0 0 0 0]]

Figure 21: A Graphic Trace Example, Data Description

Data: `[Database BankInterface MsgConnection MsgConnection Till Client]`

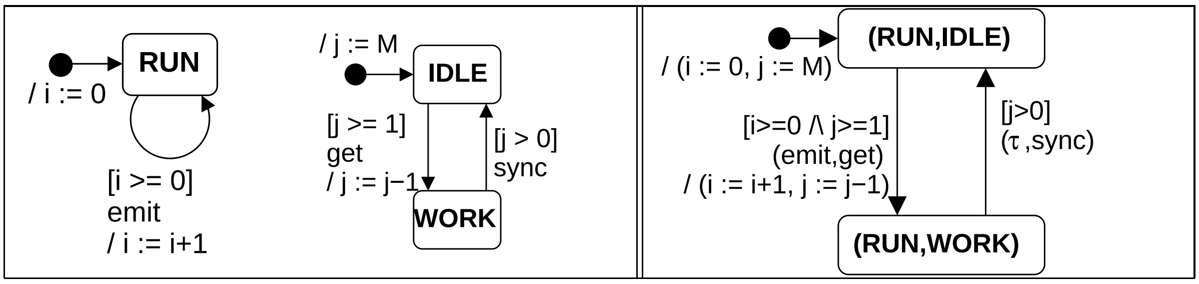

Figure 22: A Finite State System Example and Its Product

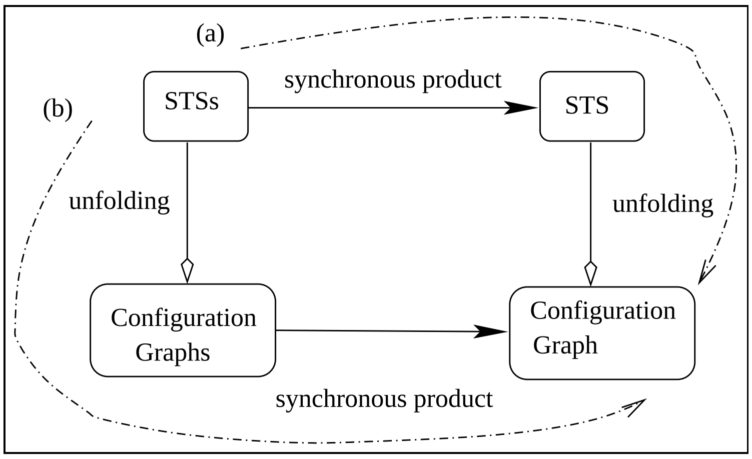

Figure 23: Relating STS and Model-Checking

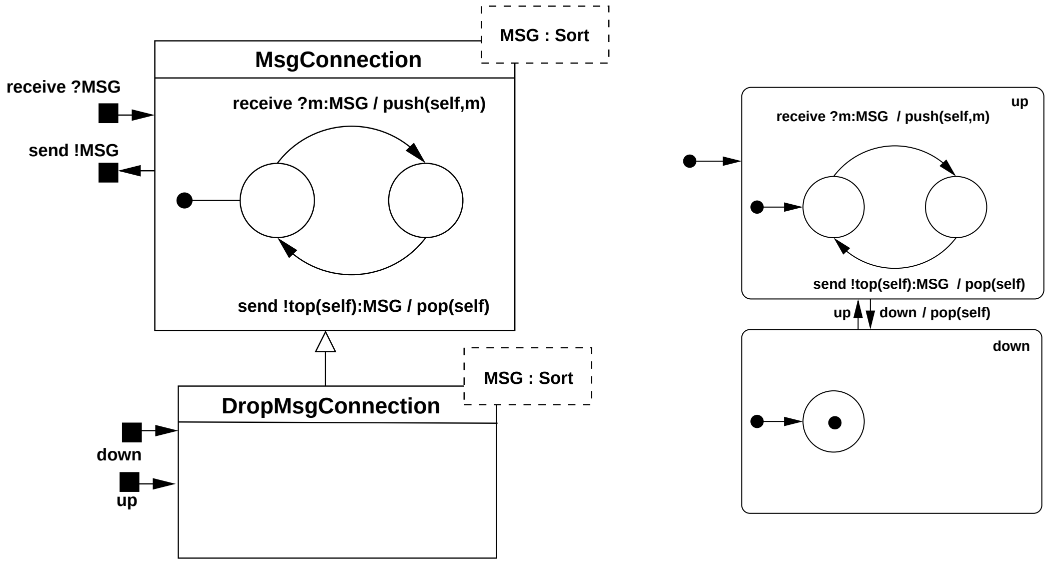
MSG : Sort
MsgConnection
receive ?MSG
send !MSG
receive ?m:MSG / push(self,m)
send !top(self):MSG / pop(self)
MSG : Sort
DropMsgConnection
down
up
up
receive ?m:MSG / push(self,m)
send !top(self):MSG / pop(self)
up
down / pop(self)
down

Figure 24: The `MsgConnection` and `DropMsgConnection` STS.

```
Sort DropMsgConnection
Imports Boolean, MSG

Opns
/* generator constructors (related to transitions) */
newDropMsgConnection : Msg -> DropMsgConnection
/* other constructors  (related to transitions) */
push : DropMsgConnection, MSG  -> DropMsgConnection
pop :  DropMsgConnection -> DropMsgConnection
up :  DropMsgConnection -> DropMsgConnection
down :  DropMsgConnection -> DropMsgConnection
/* observers */
top : DropMsgConnection -> MSG
/* equality */
equals : DropMsgConnection, DropMsgConnection -> Bool

Constantes

voidMsg : Msg

Variables

m, m1 : MSG, self, self1 : DropMsgConnection

Axioms

top(newDropMsgConnection(m)) = m
push(newDropMsgConnection(m), m1) = newDropMsgConnection(m1))
pop(newDropMsgConnection(m)) = newDropMsgConnection(voidMsg)
down(newDropMsgConnection(m)) = newDropMsgConnection(voidMsg)
up(newDropMsgConnection(voidMsg)) = newDropMsgConnection(voidMsg)


/* equality */
equals(newDropMsgConnection(m), newDropMsgConnection(m1)) = equals(m, m1)
```

Figure 25: The `DropMsgConnection` Data Type.

INFORMATIQUE, BIOLOGIE INTÉGRATIVE ET SYSTÈMES COMPLEXES

KADL Specification of The Cash Point Case Study

Pascal Poizat, Jean-Claude Royer

Abstract

This report presents the cash-point case study and mainly describes its specifications with the KADL ADL. The language is a mixed of state transition diagrams, abstract datatype and modal logic. We emphasize the need for abstract and formal descriptions especially communication architectures. We also gives some proofs done using our specific tool based on symbolic transition systems. Last we discuss previous specifications for this case study.

Categories and Subject Descriptors: D.2, D.2.4, D 2.11 [**Software Engineering**]: Software Verification, Software Architectures

General Terms: Experimentation, Languages, Verification

Additional Key Words and Phrases: Architectural Description Language, Cash Point Case Study, Component Based Software Engineering, Symbolic Transition Systems, Abstract Data Type, Verification

IBISC, FRE 2873 CNRS - Université d'Évry Val d'Essonne, Genopole
Tour Évry 2, 523 place des terrasses de l'Agora
91000 Évry Cedex, France