

A PVS Experiment with Asynchronous Communicating Components

DRAFT

Jacques Noyé¹, Sebastian Pavel², and Jean-Claude Royer²

¹ INRIA, Campus Universitaire de Beaulieu Rennes, France

Jacques.Noye@emn.fr

² OBASCO Group, EMN - INRIA

Ecole des Mines de Nantes, 4, rue Alfred Kastler - BP 20722, F-44307 Nantes Cedex 3

(Sebastian.Pavel, Jean-Claude.Royer)@emn.fr

Abstract. In our previous work we defined an approach based on symbolic transition system and data type to specify and verify mixed systems. We applied this to components and architectures with full data types, and synchronous communications. However to fit distributed systems, it seems more realistic to consider asynchronous communications. It provides a more primitive communication protocol and maximize the concurrency. To take into account asynchronous communications we distinguish message receipt from message execution and we add mailboxes in our symbolic systems. When we tried to experiment proofs in such a system a difficulty was the presence of buffers and the fact that the receipt instant is distinct from the execution instant. This complicates the specifications and also the proofs. The main problem is that the logic instant to receive is not simply linked with the execution instant. We propose to use an algorithm which decides if the system has bounded mailboxes and computes the reachable mailbox contents of the system. This algorithm gives constraints which are used to specialised the dynamic behaviour of the components according to the current system configuration. Then we are able to generate a PVS specification coping with dynamic behaviour and data type which is simpler since it removes the need for some mailboxes. The component model, the algorithms and the proofs are illustrated on a simple flight system reservation.

KEYWORDS: Asynchronous Communication, Component, Architecture, Dynamic Behaviour, Unbound or Bounded Mailbox, PVS, Specialisation

1 Introduction

In the context of the DISPO project [1] we aim to add security properties to software components and to provide means to check these properties. In [17] the authors develop an axiomatic first-order model which introduces temporal logic, real time constraints, and deontic operators. Yu and Gligor, in [43], consider only a finite time policy and uses linear temporal logic. Thus we need a rather rich context to expect to prove something useful, so we choose to do a first experiment with the PVS prover. Starting from one simple example we expect to prove that the time duration between a request and the reply is bounded. This document relates these experimentations, and the difficulties we have found.

Software components, architectures and formal specifications are important trends in software engineering. Architectures and components [31, 20, 19, 27, 44, 4, 2, 33] are nowadays technologies in software development. They promote software architectures based on communicating software entities. Architectural

Description Languages like Wright [5], Rapide [30], or KORRIGAN [13] are formal languages dedicated to specification and verification of components and architectures. One important difference with component languages like Fractal, ArchJava and industrial proposals is the explicit use of protocols or dynamic behaviours. We think that in the future component languages will promote the use of dynamic behaviour as a part of the public interface of a component. This feature is one we are interested in, and we try to reuse our experiences with KORRIGAN and GAT [37] here.

In the context of distributed computing, asynchronism of communications should be the default policy, especially in wide area networks. [29] presents a comprehensive discussion about distributed computing and its main characteristics. Asynchronous communications are simpler and more primitive than synchronous one, even if each one can simulate the other. Asynchronism is the choice done by several theoretical approaches but less often by real platforms, for example client-server has generally synchronous communications. Many infrastructures or component languages have basically synchronous communications: EJB, CORBA, RMI [20], this is also true with several classic models and languages like ADA, CCS, CSP or LOTOS. Asynchronous communications are less constraining from a concurrency point of view but the emitter does not know if a message will be received by the receiver. Thus it implies more complex descriptions, we need to cope with more errors and it produces complex dynamic behaviours. To the contrary synchronous communications are more time consuming, more abstract and produce simpler dynamic behaviours. Recently some infrastructures, for instance EJB with JMS and CCM 2.0, introduce some means to do asynchronous calls. Thus verifications of asynchronous communicating systems will become a mandatory activity in order to provide secure and on-the-shelf components.

Our current work is based on some previous experiences about architectures and components. The KORRIGAN model [14, 13] is devoted to the structured formal specification of mixed systems through a model based on a hierarchy of views. It allows one to specify in a uniform and structured way both data types and behaviours using Symbolic Transition Systems (or STS) and algebraic specifications. Symbolic Transition Systems are finite state machines with guards and variables in addition to traditional labels. KORRIGAN is relevant to describe reusable components, architectures and communication schemes. The Graphic Abstract data Type model [38] provides an effective way to specify and to verify mixed systems. It proposes a general approach to prove properties for the system which is also successful to prove temporal logic properties. The technique [37, 3] uses first-order logic to write temporal properties.

We added asynchronous communications in our model and we experimented with the PVS proof assistant [35]. But as we may suspect it, formal specifications and proofs are even more difficult than in the case of synchronous systems. The reason is the need to differentiate emission and receipt of messages and also the use of buffers to memorize messages. Such buffers have to be non-empty before executing an action, or they are bounded and cannot receive messages. It adds conditions in the specifications and in the proofs. A simple bank component will help us to illustrate this point.

Once a formal specification with asynchronous communication has been written we may observe that an abstract interpretation about the received messages is relevant. This abstract interpretation operates on the global dynamic behaviour of the system. Reusing an algorithm dedicated to the computation of the mailbox sizes in such an asynchronous system [39] we may extract constraints on the subcomponent behaviours. From these constraints we automatically specialized the subcomponent STSs. The size of the new STSs may evolved, however it removes the need for mailboxes and the related guards. This process allows us

to regenerate a simplified formal specification. We will illustrate our approach on a simple flight reservation system with four components. Our approach illustrates the use of abstract information (STS), partial evaluation (computing the mailbox contents), and specialisation (using constraints for STS) conjointly with a general proof approach. We think that, in the future, verification techniques will increase the need for collaborations between automatic and efficient proof techniques (model checking, abstract interpretation, specialised algorithm) and proof assistants.

The paper is organised as follows. Section 2 presents the GAT principles and its translation inside the PVS prover. Section 3 shows an example of a simple asynchronous system and discuss the difficulty to specify and to prove in this context. Section 4 presents one solution which is based on an algorithm which automatically computes constraints on the system configuration. From these constraints a specialised behaviour for components can be built and from it a simpler PVS specification. Finally, related work is discussed and a conclusion finishes this presentation.

2 The GAT Principles

This Section introduces the concepts of *Symbolic Transition System* (STS) and *Graphical Abstract Data Type* (GAT). A GAT for a component is a STS and an algebraic specification of a data type. For a GAT component we consider two views: the dynamic view and the functional view. The dynamic view is a STS: a finite set of states and a finite set of labelled transitions. Classic finite transition systems, or Labelled Transition Systems (LTSs) have labels which are closed terms. Unlike LTS, our STS labels are operation calls with variables and guards. This concept is related to machines where states and transitions are not necessarily unique objects. A state may represent a set of either finite or infinite objects and a transition collects several state changes. This kind of state machine controls the state and transition explosion problems and makes dynamic behaviours more readable. The functional view is an algebraic specification of a partial abstract data type [11, 12]. We distinguish two kinds of GAT components: sequential components and concurrent components.

In the GAT process specification we suggest to start from the dynamic view of the components since it is the most abstract view. First, the specifier declares the operations, the conditions and the states of the component. These informations are represented graphically using a STS. Second, the semantics of the operations is provided (in the functional view) by an algebraic specification. In case of concurrent and communicating components, the synchronous product of STSs is used before generating the axioms. Figure 1 describes an overview of the GAT process and its semantics. An algebraic specification is extracted from a STS and its semantics is a partial abstract data type. The STS represents a graphic view of a partial equivalence relation over the data type.

2.1 Symbolic Transition System

The finite state machine formalism is well-known by practitioners. It is well-suited to the description of interactions and controls. One problem with such a formalism is the great number of states and transitions. For instance, one can combine states into super-states or aggregate states as in [24]. However when the system has not a finite or bounded number of state one must use more powerful concepts. It often happens if one has a mixed system with both control and data types. We defined the notion of finite and symbolic

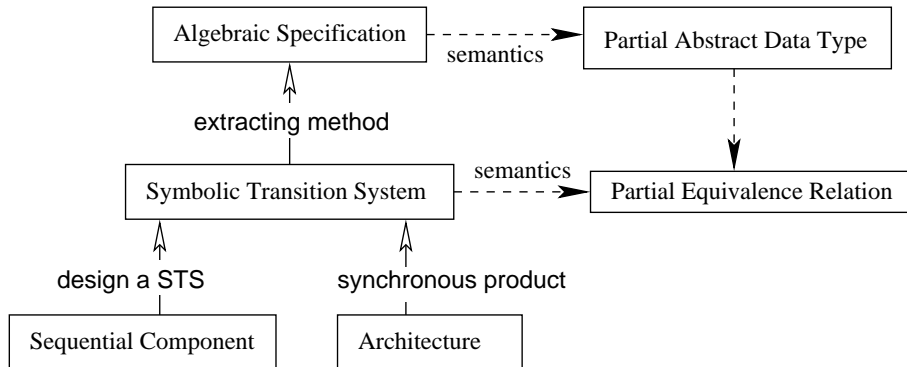


Fig. 1. The GAT Overall Process

transition system. This notion arises also from the need of a full semantics for language like LOTOS [40] and in the AltaRica formalism [7].

A symbolic transition systems is a finite set of states and a finite set of labelled transitions. The right part of Figure 2 illustrates the graphical presentation of such a description, the term label may have variables and the $[\dots]$ expression denotes a guard. *self* in this figure denotes a variable associated to the current sort. Symbols and terms occurring in the STS must be interpreted in the context of the algebraic specification. A transition corresponds to an internal operation (a constructor) with an interpretation formula based on state predicates. Our notion is more general than the symbolic transition graph defined in [25] or I/O automata as in [21]. We have more general states (not only tuples of conditions) and we have no restriction on variables occurring on transitions.

We also need finitely generated values, *i.e.* every values can be denoted by a finite sequence of generators [41]. We assume that each T internal operation is a generator. Thus op_B (respectively op_R) denotes a basis internal operation labelling an initial transition (respectively a recursive one labelling a non initial transition).

The axioms for the definedness, preconditions, and state predicates are automatically computed from the STS description. Thus the user has to manually complete the specification of the observers and the guards.

2.2 Concurrent and Communicating GATs

The composition scheme for components, in order to handle concurrency and communications, is described below. The synchronization list denotes the actions which are required to synchronize in each component. The semantics of synchronization is obtained from the synchronous product of STSs in a similar way than for the synchronous product of automata [8]. We have defined a notion of structured STS [15] which is a natural extension with structured states and structured transitions. A transition of the product is a tuple of transitions and the associated guard is the conjunction of the component guards. Firstly, we built the free product of the two STSs. Secondly we get out the pair of transitions which are not allowed by the list of synchronizations. This rule is similar to the LOTOS one, other rules may be possible, for example the CCS rule. Last, the synchronizations are enriched by communications. During a synchronization, some values

may be emitted or received. Communications may occur during synchronizations we use $?$ to denote a receipt and $!$ for an emission. A value is emitted by an observer and received by the mean of a variable. An algebraic specification is eventually built from the computed STS. Thus both synchronization and communication are integrated in an algebraic style. In case of a concurrent system without proper data computation at the concurrent node level, the specification can be automatically generated.

2.3 PVS Specification

An adequate semantic framework for GAT seems to use partial data type ([12]) but efficient prover generally support total deduction. However total first-order logic is an “universal” logic which may embed various partial logic, see [12] for a precise translations of partial first-order logic into total first-order logic. We choose to first experiment with Larch Prover [37], but recently we switch to PVS [35]. The main reasons are that PVS is an up-to-date and powerful prover with higher-order logic, facilities for model-checking, and type-checking constraints (tcc) improve the specification writing. We follow the same way than [3] but we try to simplify some parts when possible. In short the PVS translation for a T atomic component with only synchronous communications, is the following.

1. Define a total data type with as constructors the T internal operations.
2. Define a theory over this data type providing: the state predicates, the definedness predicate, the preconditions, the guards, and the observers.
3. The observers may be partial, we use predicate subtype in this case.

For a concurrent component we have to :

1. Choose names for the global operations which acts synchronously or asynchronously.
2. Compute the synchronous product of STS.
3. Define a theory over the subcomponents and which introduces the state predicates, the definedness, the preconditions, and the guards for the product in the same way than for an atomic component.
4. Define the observers for the subcomponents with positive conditional axioms, a simple way is to generate one axiom per transition in the STS of the product.

On these specifications exclusivity, complementary and some other properties are generally not too difficult to prove using mainly term rewriting and induction.

3 To Specify Asynchronous Component

In this Section we illustrate the application of GAT to an asynchronous component. We reused the idea of buffers but they are parts of our components. Asynchronous communication distinguishes message sending and its execution. If op is an operation call we note op_R the message sending and op_E its execution. The emitter does not memorize messages; this is done by the receiver in a specific buffer. This buffer acts as an asynchronous channel but it is part of the component as for an actor. This buffer (or message queue) contains the messages received by a component, it will be called a *mailbox* and we only consider a FIFO policy. Sometimes operation may be executed but no receipt is needed: we call them autonomous operations and simply note them op . An op *autonomous operation* is an action which does not need a receipt to be

executed. It takes into account the fact that a component may know sufficient enough information to trigger this action. A *message receipt* denotes the receipt of a message in the mailbox. An *action*, op_E , means that the op operation is triggered after the component received the corresponding message receipt (op_R). The fact that the component received the message is denoted by the guard $[\&op]$. Message sending occurs during the execution of an operation. The notation op_R stands for the usual provided service and op_E is linked to required service of many other component languages.

Synchronisation remains the basic way to communicate, it is possible in two forms between distinct components:

- Between two autonomous operations, it means usual “rendez-vous” with communication.
- From one autonomous operation or one operation execution ($_E$) to a receipt ($_R$), it is message sending.

Graphically a component is a box with *input pins* corresponding to message receipts and *output pins* corresponding to message sending. Input pins (resp. output pins) are put on the left of the component (resp. on the right). An autonomous operation has only a right output pin, other operations have a left input pin (receipt) and a right output pin (action and sending). For example the graphic bank component is represented in the left part of Fig. 2. There is an operation for message receipt $order_R$ with two arguments: the account number of the client ($?c:Client$) and the price of the ticket ($?p:Real$). The operations *fail* and *success* are autonomous. The different pins may be (or may not be) connected in a given architecture, it expresses the receipt of messages (on the left) or message sending (on the right).

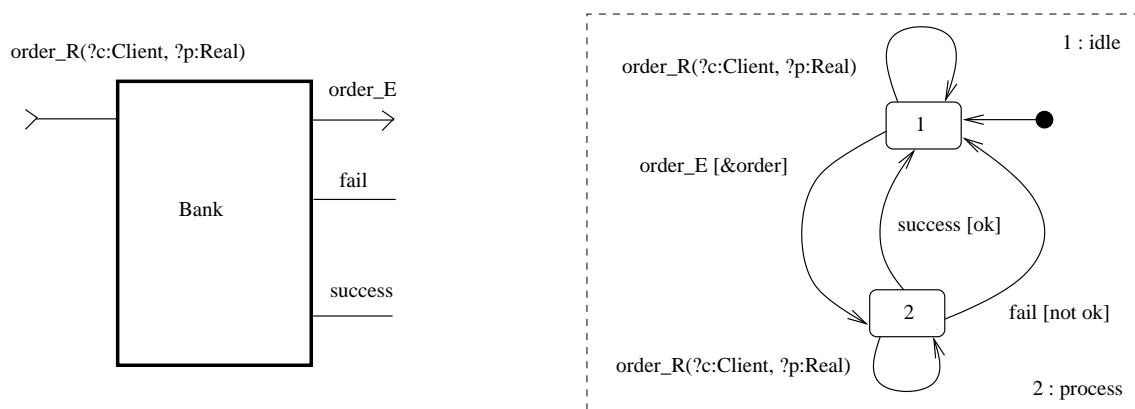


Fig. 2. The Bank Component and Its Dynamic Behaviour

The previous figure describes the interface part of the bank, we also take interest in the dynamic behaviours or protocols. Such a protocol is represented in the right part of Fig. 2. Note that each state has transition loops for message receipts, $order_R$ in the bank example. A sender does not block except if the buffer is full and nothing else is possible, this is also true for the receiver except if the mailbox is empty and without autonomous operation. Here we avoid a discussion about other possible choices, see [39] for more details. The description of the other components is done in Section 4. From now on we assume two hypothesis about our flight reservation system: there is only one client and the sizes of the mailbox buffers are not bounded. We will relax later these assumptions in Section 4.4.

3.1 The Bank Example in PVS

The PVS translation of the bank component follows the general scheme of Section 2. But we have to add the management of the mailboxes and the associated guards. Unfortunately, even, for such a simple example it was a bit complex. In addition to the mailbox management, another problem is that now the execution of an operation is loosely coupled with its message receipt. For example to define the semantics for the `ok` operation we have to find the message corresponding to the oldest `order_R`. The need to know the correspondence between the instant execution (`order_E(self)`) and the corresponding message (`order_R(self, id, p)`) appears also if we want to measure the time duration between these two events. We tried two different approaches which are reported below.

3.2 Version 1

This version does not use an explicit buffer for messages, the various functions are inductively defined on the execution steps. An execution step is a term built from the events of the STS, it represents the full history of the system. With our FIFO policy we have to define some auxiliary operators. The `rank` operator computes the rank of the `order_R` subterm in the expression corresponding to the execution (`order_E(self)`). The principle is to compute the number of messages less than the indice of the considered execution. The `raux` operator extracts the `order_R` subterm from the `self` expression. The `received` function gives the right subterm corresponding to a defined execution.

received: AXIOM Pdef(order_E(self)) \supset received(order_E(self)) = raux(self, rank(self, 1, 0))

From that the `ok` guard may be defined as:

ok1: AXIOM process?(self) \supset ok?(order_R(self, id, p)) = ok?(self)

ok2: AXIOM process?(order_E(self)) \supset ok?(order_E(self)) = IF member(self, PROJ_1(first(buffer(self)))) THEN (PROJ_2(first(buffer(self))) \leq account(self, PROJ_1(first(buffer(self)))) ELSE FALSE ENDIF

It generates some `tccs` and we need some theorems to discharge them. One is related to an additive property about `rank` and the other is the similar property for `raux`. We have done these proofs, even if they are not too complex they may disturb and make more heavy the specifications. Indeed the generation of `tcc` improves the safety, but it forces us to split the specification in three theories one for each operator (`rank`, `raux`, and `ok`). May be it is possible to simplify a bit these definitions coping with some extra conditions or specific buffer policies. However, in most of the cases, we have to live with such complexity.

The size of this bank specification without `rank`, `raux`, and `received` operators is a bit more longer than the synchronous bank version. The specifications of the `rank`, `raux`, and `received` operators doubles the size of the bank specification. In case of specifications with several message kinds the solution may become complex. The use of PVS higher-order constructions may simplify the situation.

3.3 Version 2

The second version computes a buffer of messages, this is an instantiation of a generic and classic data structure provided by PVS. The `buffer` operation definition may be generated automatically from the STS. The functions `msg_order_R`, `ok`, ... are simply defined using this data structure. The use of this data structure does not solve the definition of the correspondence between emission and receipt (the `received` function). Either we reuse the previous way, or we use another data structure, or we add informations in the

previous one. In all cases the benefits are not obvious, the specification is a bit more readable but it does not strongly simplify it.

These experiments convince us that such kinds of specifications would be complex in a real system. The core of the problem is that we expect to receive messages in the buffer less or more concurrently than their execution. Another point is that with our hypothesis it seems not possible to get bounded time. Indeed it is possible to receive an infinite number of messages before to process the first one. Bounded mailbox is a necessary condition to ensure the finite time property. However such a specification is the most general and can be seen as the reference description of the component. Thus we try another way, because we expect that the number of messages have to be bounded.

3.4 A Specialised Specification

We may note that sometimes the context constrains the dynamic behaviour of the system. For example in the bank behaviour one may assume that the allowed language for the dynamic events is $(\text{order_R} ; \text{order_E})^*$. In this case we may specialise the bank STS as in Fig. 3.

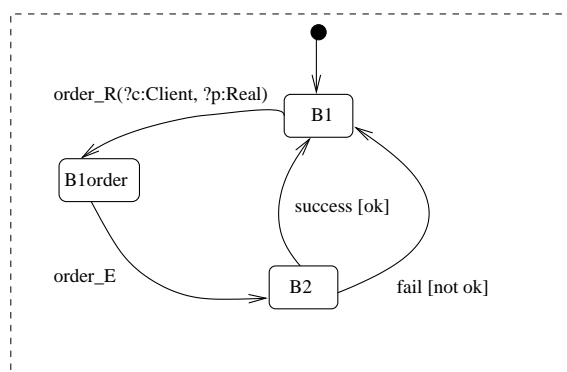


Fig. 3. A Specialised Dynamic Behaviour

The extracting process may be rerun and manually completed to build a new specification for the Bank component. The `BankSpec` has: a definedness predicate, the states predicates and the preconditions, which are automatically generated from the specialised STS. The specification is of course simpler, the definition of `ok` does not need the previous `rank`, and `raux` functions. The mailbox computation may be removed (`buffer` and `msg_order` functions) as well as the `&op`, and `not fullMailbox` guards definitions. The specifications of the two new observers associated to `ok?` and `received` are defined as:

`ok1: AXIOM one?(self) \supset ok?(order_E(order_R(self, id, p))) = IF member(self, id) THEN ($p \leq$ account(self, id)) ELSE FALSE ENDIF`

`received: AXIOM one?(self) \supset received(order_E(order_R(self, id, p))) = order_R(self, id, p)`

The size of the specification is a bit longer than the synchronous version. The next Section describes an automatic way to check and to handle this simplification more systematically.

4 A Bounded System

We present here a more systematic approach of the previous idea. As an example we model a simple system with four concurrent components, a previous version with synchronous communications is [36]. This is a part of a flight seat reservation system with a component for the seat reservation, one for simulating the bank, one for the flight company and a last one for the client. The client gives its account number when he requests a seat to the counter. The counter asks the company to know if there is a seat. This may fail or succeed, in this last case the seat reservation orders the price to the counter which resends it to the bank. The order may fail or if it succeeds then the counter prints a ticket and the company books the reservation. A comprehensive description of this example may be found in [39]. Fig. 4 represents the component for the counter and the meanings of its ports. The two other components: company and client are respectively depicted in Fig. 5 and 6. Note that in order to simplify the STS descriptions we forget the looping receipts on each state.

- request : it handles a user request,
- price ?p:Real : it receives the price from the company,
- noPlace : there is no place,
- fail : the bank ordering fails,
- success : the bank ordering succeeds
- printTicket : a ticket is delivered to the user.

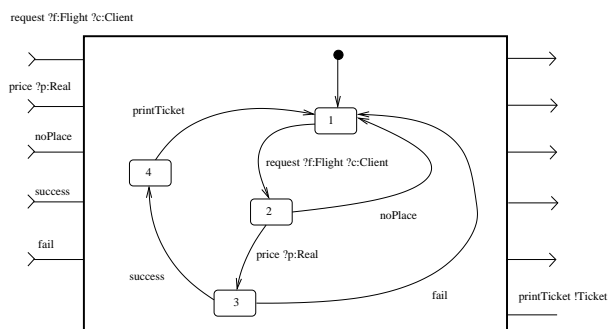


Fig. 4. The Counter Component

- request : the request has been received,
- checkPlace : it checks if there is some place,
- noPlace : there is no place,
- fail : the reservation fails (coming from the counter),
- book : the reservation succeeds.

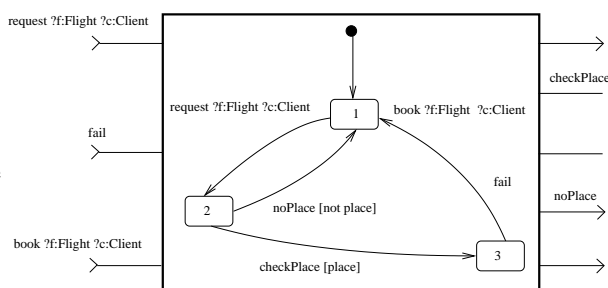


Fig. 5. The Company Component

Figure 7 represents the architecture of our flight reservation system. A message transmission is graphically denoted by a thin line from an output pin to an input pin. To simplify the figures we avoid some names, the effective arguments and the guards of messages. Some pins like `request_E` and `book_E` for the company, or `order_E` for the bank are not connected since they do not send messages in this configuration.

- request : the client asks for a flight reservation,
- ticket : it get its flight ticket,
- fail : the reservation fails.

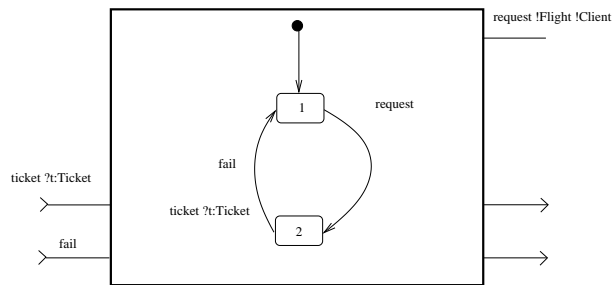


Fig. 6. The Client Component

One example of synchronisation concerns the emission of the `price_R` message to the counter by the

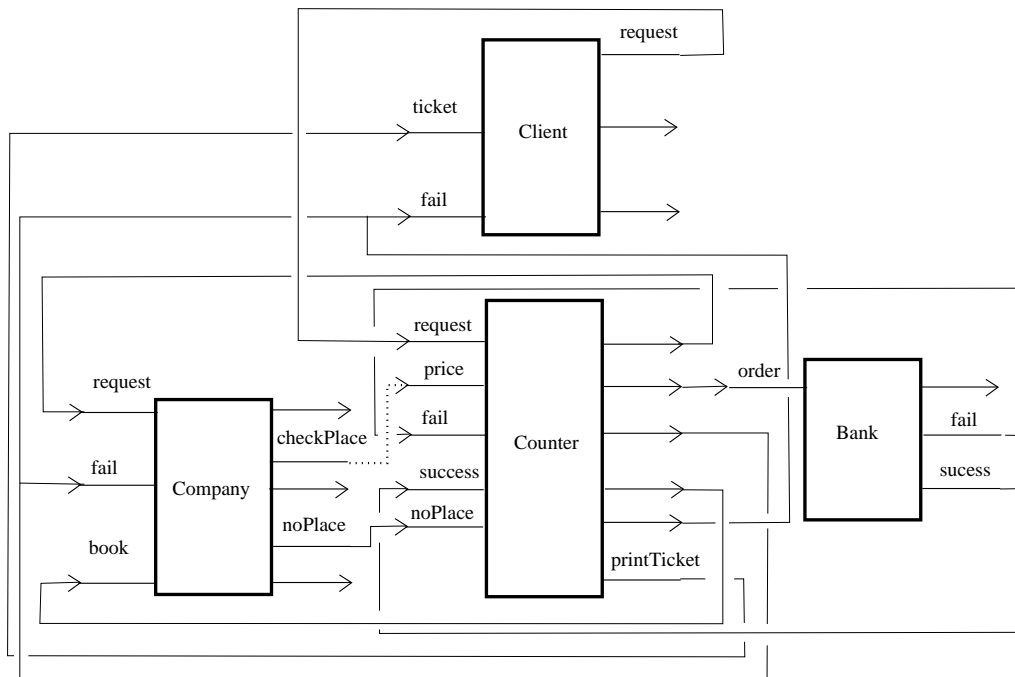


Fig. 7. The System Architecture

`checkPlace_E` operation of the company. This line is dotted in the picture. During this communication the first and the fourth component (the client and the bank) do nothing, this is a structured transition noted $(- \text{checkPlace}_E \text{ price}_R -)$. The global dynamic behaviour, a structured STS, has about 50 states and 300 transitions. This complexity comes the fact that we have a general asynchronous system. The similar example with synchronous communication mode, see [36], has nearly 10 states and 15 transitions. However we must precisely compare it with a synchronous system simulating asynchronous communication. A quick analysis shows that the number of states and transitions would be nearly the same.

One interesting situation is when the system has only bounded mailboxes. This may be decided using various ways, for instance the coverability algorithm of Petri nets. In [39] we propose a specific algorithm dedicated to our symbolic and structured systems and which computes the contents of the mailboxes. In this algorithm the mailboxes are dictionaries of messages. The algorithm searches in the dynamic system and computes the states with their mailbox contents. When a mailbox has a possible infinite contents, a star is put to avoid the construction of an infinite set of states.

4.1 Application to the Flight System

Here we consider a FIFO strategy for mailbox, and a related algorithm is able to check if the system has only finite mailboxes. If it is bounded another algorithm computes a system simulation, the result of this bound analysis is done in Fig. 8. We may note that mailboxes have a size lesser than two in this configuration.

From this observation we suggest the following process:

1. Compute the global behaviour of the asynchronous system using the synchronous product.
2. Use the checking and bound analysis to extract constraints for subcomponents (see Table 1).
3. Specialise the STS of the subcomponents with these constraints (see Fig. 3, 9, 11, and 10).
4. Generate the PVS specifications as in the synchronous case with these STSs.

Client	CL1 (\perp), CL2 (\perp , (fail), (ticket))
Company	C1 (\perp , (request)), C2 (\perp), C3 (\perp , (fail), (book), (request fail), (request book))
Counter	C1 (\perp , (request)), C2 (\perp , (noPlace), (price)), C3 (\perp , (success), (fail)), C4 (\perp)
Bank	B1 (\perp , (order)), B2 (\perp)

Table 1. The Subcomponent Dynamic Constraints

The result of step 2 is a set of allowed states for the subcomponents, automatically extracted from the result of the bound analysis. For each subcomponent we note the visited state and the mailbox contents, for instance, see Table 1, the CL2 state for Client is activated with an empty buffer (\perp), with a fail or with a ticket message. From these constraints a simple specialise algorithm, see Table 2, computes the minimal STS for the subcomponent which realises the constraints.

The resulting STS for the subcomponents are described in Fig. 9, 10, 11, and 3. Note also that the result of Fig 8 is isomorphic to the synchronous product of these specialised behaviours (following the rules given by the architecture). This may be explained following the computation of the synchronous product. The initial state is a member of the set of states of the two results. Let e a state in the specialised product and t a transition of the original product, the component transitions verify the synchronisation rules, their sources are states of the specialised behaviours, thus t is also a transition of the specialised product. Let t a transition of both results and e its target state, the component transitions are in the specialised behaviours and they verify the synchronisation rules, hence e is also a state of the specialised product. The reverse inclusion is similar.

```

specialise(GAT, listState)
stateResult <- [(initial state, [ ])]
result <- newSTS
WHILE (stateResult != [ ]) DO
  etatCour <- stateResult[0]
  etatCorresp <- stateResult[0][0]
  stateResult.pop(0)
  FOR oTran in self.voisins(etatCorresp):
    newTarget <- compute the target of Otran
    coping with receipt/emission and buffer
  IF (newTarget in listState)
    AND oTran executable from etatCour
  THEN IF newTarget already in resultat
    THEN add (etatCour,oTran,newTarget) in resultat
    ELSE add etatCour in resultat
    add (etatCour,oTran,newTarget) in resultat
    stateResult = stateResult + [newTarget]
  ENDIF
  ENDIF
ENDFOR
ENDWHILE
RETURN resultat

```

Table 2. The specialise algorithm

4.2 Automatic Specialisation of a GAT

The previous algorithm is able to specialise a component STS. The full specification remains to provide with the same process as in Section 2.3.

The specialised STS is a simulation, in a dynamic sense as defined in [6], of the original one. The specialised specification is also a simulation/representation, but with the Hoare meaning as defined in [34, 9], of the original specification. This is not new that both notions of simulation coincide. To disambiguate some operators they are prefixed by the theory name, `theoryBank` for the original specification and `theoryBankSpec` for the new specialised one. More precisely we have a partial embedding, with `theoryBankSpec` as abstraction invariant, and compound with an enrichment.

The buffer definition and other observers of the original specification are valid axioms in the context of `theoryBankSpec`. We have also some theorems about the contents of the mailboxes coming from the bound analysis. We have proved the following theorem. It was a bit hard, since the theorems which link the original specification and the simplified one are mutually dependent.

tout: THEOREM

$$\begin{aligned}
& (\text{theoryBankSpec} . \text{Pdef}(\text{self}) \supset \text{theoryBank} . \text{Pdef}(\text{self})) \wedge \\
& ((\text{theoryBankSpec} . \text{Pdef}(\text{self}) \wedge \text{theoryBankSpec} . \text{oneOrder?}(\text{order_R}(\text{self}, \text{id2}, p_2))) \supset \\
& \quad (\text{PROJ_1}(\text{first}(\text{buffer}(\text{order_R}(\text{self}, \text{id2}, p_2)))) = \text{id2})) \\
& \wedge \\
& ((\text{theoryBankSpec} . \text{Pdef}(\text{self}) \wedge \text{theoryBankSpec} . \text{oneOrder?}(\text{order_R}(\text{self}, \text{id3}, p_3))) \supset \\
& \quad (\text{PROJ_2}(\text{first}(\text{buffer}(\text{order_R}(\text{self}, \text{id3}, p_3)))) = p_3)) \\
& \wedge \\
& (\text{theoryBankSpec} . \text{Pdef}(\text{self}) \supset
\end{aligned}$$

$$\begin{aligned}
& (\text{theoryBankSpec} . \text{member}(\text{self}, \text{id}) = \text{theoryBank} . \text{member}(\text{self}, \text{id})) \\
\wedge \\
& (\text{theoryBankSpec} . \text{Pdef}(\text{self}) \supset \\
& \quad (\text{theoryBankSpec} . \text{account}(\text{self}, \text{id1}) = \text{theoryBank} . \text{account}(\text{self}, \text{id1}))) \\
\wedge \\
& ((\text{theoryBankSpec} . \text{Pdef}(\text{self}) \wedge \text{theoryBankSpec} . \text{two}?(\text{self}) \wedge \text{process}?(\text{self})) \supset \\
& \quad (\text{theoryBankSpec} . \text{ok}?(\text{self}) = \text{theoryBank} . \text{ok}?(\text{self}))) \\
\wedge \\
& (\text{theoryBankSpec} . \text{Pdef}(\text{self}) \supset \text{one}?(\text{self}) = (\text{idle}?(\text{self}) \wedge \text{null}?(\text{buffer}(\text{self})))) \wedge \\
& (\text{theoryBankSpec} . \text{Pdef}(\text{self}) \supset \\
& \quad \text{oneOrder}?(\text{self}) = (\text{idle}?(\text{self}) \wedge \text{length}(\text{buffer}(\text{self})) = 1)) \\
\wedge \\
& (\text{theoryBankSpec} . \text{Pdef}(\text{self}) \supset \\
& \quad \text{two}?(\text{self}) = (\text{process}?(\text{self}) \wedge \text{null}?(\text{buffer}(\text{self}))))
\end{aligned}$$

This proof is indeed an interesting result but it is not the most useful thing from the specifier point of view. In a component perspective, it would be better to automatically specialise the general specification of the component when it is needed. The remaining part is to automatically simplify the observers and auxiliary operations coping with the previous theorems. Several ideas are possible, one is to reuse technics coming from code specialisation. One idea we want to explore is the following:

- We have a new specification and we have to simplify the observers and guards definitions
- We have a new specialised STS with no need of buffer and the following property holds: if op_E is an execution then there is only one corresponding op_R transition in the specialised STS between the initial state and the execution. Of course the op_R occurrence depends on the mailbox strategy.
- The received state and the arguments of the call corresponding to an op_E execution are extracted from the specialised STS without a buffer computation.
- A given state corresponds to some well-defined call of the constructors.
- A property may be defined by a set of positive conditional axioms of the form:
$$\text{petat}(\text{self}) \text{ AND } \text{someConditions}(\text{self}, \dots) \supset \text{prop}(\text{self})$$

These facts, which remains to be proved, ensure that a part of the specification may be automatically simplified. At least it also gives a useful process to interactively help the simplification of the specification. For example the simplification of the $ok?$ observer may be done as following:

- To simplify an observer we start from the corresponding theorem in the representation. We trigger transitions in reverse direction, we take into account conditions and we use rewriting to simplify expressions.
- The representation property is: $\text{two}?(\text{self}) \supset \text{theoryBankSpec} . \text{ok}?(\text{self}) = \text{theoryBank} . \text{ok}?(\text{self})$
- We use PVS to transform this expression until to get the ok simplified axiom of Section 3.4.
- We have to trigger, in the opposite way, starting from the $two?$ state, the $order_E$ and $order_R$ transitions.

The main part of the PVS script is given in Section D, it mainly use rewriting technics. However this is only a manual example, we think that at least the process may help the specifier to simplify some definitions. A deeper study is needed here to get some simplification algorithms. One future work is to propose such an algorithm and to implement it as a PVS strategy.

4.3 The Full Specification

During the PVS generation we have no problem with buffer and guards associated to message, since they are completely removed in this specification. Note that the specification part corresponding to the synchronous product does not change between the general version and the simplified one.

We expect to prove the following theorem, that is the time duration between a `getTicket` and the corresponding `request`, without `fail` or `noPlace` action is bounded.

th: THEOREM

$$\text{Pdef}(\text{getTicket}(\text{self})) \supset (\text{time}(\text{getTicket}(\text{self})) - \text{time}(\text{received}(\text{getTicket}(\text{self})))) \leq 40$$

A similar proof may be done for the `request`, `fail` pair. Since we consider that every action has a duration, the mailboxes checking is a necessary condition to get a maximum waiting time policy. We have to define some time measure associated to each functions, however we only consider a number of tick units represented by integers. A more elaborated approach, compatible with our context is [28]. In this work the author describe a class of linear models of time which includes both discrete and dense models, with or without initial instant.

To prove the above theorem we have two ways. The first one is to define all the distinct paths from one `request` to a `getTicket`. It is not too difficult to define this set of paths, but it not easy and not efficient to compute and verify them. An efficient and automatic way is to extract these paths directly from the STS. In our examples we have 10 paths and using skolemization and rewriting we verified that these paths have a bounded time duration.

In some simple contexts the time durations may be expressed as valuations on the edges of the STSs. In this case, the bounded time checking is automatic using a maximum path algorithm. But we may need a prover for two reasons: we may have more complex time computations than simple bounds, or we need a more precise time boundary.

For an atomic component it is not too difficult to define a `time` function inductively on the structure of the events (given by the STS). For a concurrent component the same way is also possible thanks to the global STS. But another approach consists in decomposing the time function of the global actions in more elementary time pieces associated to sending, receipt, subcomponent activations and so on. We have checked both possible approaches, the latter leads to simpler proofs.

Other properties have been checked on the simplified and global specification:

- The complementary and exclusivity properties of states for the subcomponents
- The complementary and exclusivity properties of states for the global system
- Deadlock freeness of \parallel the components
- The decomposition of composite states into a conjunction of subcomponent states

4.4 Discussion

In this section we have to explain how to relax some assumptions and how to extend our experiments.

We have only one client, but we have checked this system with two and three clients. Of course the size of the bound system grows. The global system with n clients has also bound mailboxes, the reasons are explained below. On one hand, a client emits a request and waits a success or a fail action. On the other hand, the counter serialises the request, it asks the company and the bank before to process another request. Another way to see that is the following:

- Consider an abstraction of the n clients with a one-state STS and looping transition for each event.
- Build the synchronous product and check it for bounded mailboxes.

We observe that it is unbound, with FIFO and with method dictionary. However to get a bound system it is sufficient to bound the counter size.

Our checking algorithm is able to cope with size constraints on the mailboxes. The previous checking algorithm application consider unbound buffers and often it shows that buffers are bounded. The reason is that many useful systems send request and wait for responses. However malicious clients may ask for an infinite sequence of requests and then availability of service may fail. Note that our approach applies to the bounded parts of a system, even if the system is not globally bounded. However it requires that the buffers associated with infinite paths are computable. This is not always the case, for example with FIFO mailboxes. Indeed real systems are bounded since memory space is not infinite. However the state space becomes too large even with symbolic transition systems.

The synchronous product needs to know the maximum number of clients, a more serious problem is that it has an exponential complexity. One idea is to define more efficient algorithms which do not need to compute explicitly the synchronous product, see [26] for a related approach.

The STS allows us to control the number of states and transitions. The hard constraints are: at least one state and one looping transition per event. If the STS complexity decreases then the associated data type complexity increases and *vice versa*. Until this complexity is around 50 states and 50 transitions a human may manually manage it. Automatic tools help to enlarge this limit, but time and space computation becomes more prominent. One idea to overcome the scalability problem is the following: Once we have defined a compound system like our flight system we try to change the STS by a more abstract one. We have to recompute the associated data type but we expect to do that with the help of algorithms. For example it is automatic to associate a simpler STS to a product of n components.

Note also that our simplification technique is relevant with any system as soon as its bound analysis shows that some components have bounded mailboxes. We expect to use it to optimize the deployment of components. This way seems also possible with other proving techniques, for example with I/O automata as in [22] it will be rather immediate. It will be harder, but feasible, with other specification techniques for mixed systems.

To summarise the specification and simplification process:

- Specify the system with GAT and PVS.
- Analyze the mailboxes with the checking and bound algorithms.
- Extract the constraints and specialise the STS.
- Simplify the GAT specifications of the subcomponents

In the general case the main parts of the verification process are automatic, the parts which are hand-written are:

- The specification of the observers of the subcomponents.
- The ultimate simplification of the subcomponent observers and guards.

5 Related Work

It is impossible to be exhaustive here since our approach overlaps several important fields of researches.

Several approaches for mixed systems have been studied in the area of algebraic specifications [10, 23, 40, 16]. Some proposals separate the process calculi from the data calculi, others have a two layered algebraic specification. We will compare our approach with the [18] rewriting logic approach and with formal architectural description languages.

Of course some analysis may be conducted using Petri Net tools, model-checkers, or other automata related tools. Generally our systems are not adequate for this and a preliminary translation is needed. However the main reason to try another way is that we have structured systems with data types and guards. Thus we need a powerful approach taking into account the full description of the system. From this powerful approach it seems relevant to propose more specific but automatic tools. Our approach computes all the mailbox contents without hypothesis on the arrival rate of events. To the contrary queuing networks and various stochastic techniques are able to calculate the average size of buffers coping with message probabilities.

Our component and architecture description is related to architectural description languages (ADL), see [31] for a good survey. We have atomic and complex components with interfaces and dynamic behaviours. Our approach gives a way to specify mixed systems *i.e.* with both full data types and dynamic behaviours. Here we only present a graphical representation of the architectural description language, this is not generally sufficient for automatic processing and full code generation. A main difference is the use of both synchronous and asynchronous communications while most of the time ADLs only promotes synchronous communications.

In [18] the authors propose rewriting logic as an executable specification formalism for protocols. Rewriting logic is able to model object-oriented message passing, various concurrency mode and reflective aspects. They use the Maude rewrite engine to analyse all configurations of a system. However, as with model-checking, this works well if the system configuration is finite. The authors improve this with the narrowing technique which allows to start the analysis with several initial states. We have a less flexible approach which may be embedded into rewriting logic. But it is also more readable and close to programming practice. In this context we have also experimented the use of provers (LP and PVS) and an extension of the CTL* logic with data types.

At this stage it is interesting to compare our approach with WRIGHT [5, 4]. WRIGHT is a formal architectural description language with first class components and connectors. A component has a set of ports and a behaviour part. A connector defines a set of roles and a glue specification. Roles are expected to describe the local behaviour of the interacting parts. The glue describes how the local activities of the different parts are coordinated. The semantics of these constructions is defined by a translation into CSP.

This has the advantages to get effective model checking for CSP and related work about behavioural refinement. However, most of these verifications are limited by the state explosion problem and consider simple data types. WRIGHT proposes a deep analysis about automatic checking for architectural languages. It allows connector consistency, configurator consistency, and attachment consistency using mainly techniques to prove deadlock freedom and behavioural refinement. We improve readability by graphic notations, this is important for large scale applications. In our approach we consider both dynamic and functional properties not only dynamic properties with restricted data types. This is a first important difference but others are the use of symbolic transitions systems and asynchronous communications.

The use of Petri Net and the reachability/coverability algorithms [32] may solve our mailbox contents analysis, but it needs, at least, a translation into the Petri Net world. Our algorithm is different since we have buffers with various policies.

The bounded time property is related to real time constraints and quality of service. Quoting [?] we have doing a static analysis based on paths searching. However we are far from true and strict real time since we consider abstract specification.

The work of [42] is related to our approach but with LTS systems in place of STS, another important difference is the synchronous semantics. The related section of this paper contains also a set of useful remarks.

6 Conclusion

We provide an approach to design component and architecture with asynchronous communications and dynamic behaviours. To handle asynchronous communications we distinguish emission and receipt operations rather than the use of specific buffers to memorize messages. Our approach provides a uniform and formal way to express synchronous and asynchronous communications, it seems readable and close to software engineer practices. We show how to compute the global behaviour of an architecture and to represent it without lost of information.

In the case of asynchronous communications the specifications and the proofs become harder since we have to cope with the buffer managements. But another difficulty is that the receipt instants are not straightly linked with the execution instants. We propose a process to simplify the specifications and the proofs in this case. This process analyses the mailboxes of the architecture and if they are bounded it computes some constraints which are used to simplify the dynamic behaviour and the specifications of the components. The process has several automatic parts and may be supported by tools. In the case of maximum waiting time the approach remains general since bounded mailbox is a necessary condition.

One trend of future researches is to extend our approach to cope with other analysis, for example with guards in communications. Another one is to consider a variable number of components. These are important features to fit with more realistic systems. We have to improve our set of algorithms to automate when possible the various checking or specialisations. One useful aspect of PVS will be the higher-order facility, it may help us to automatically generate definedness, state predicate and preconditions.

A Fourre-tout

- Pour l’aspect “dispo” c’est juste une propriété simple du genre : je veux prouver que le temps est borné entre deux requêtes. Il y’a au moins trois façons d’exprimer ça :
 - Utiliser `prefixe` et des conditions annexes, moyen très général mais pas très opérationnel sur le composant global, voir sur l’exemple limité de la banque.
 - Utiliser l’opérateur `received`, ça donne des résultats mais il y’a encore des difficultés sur le système global
 - Calculer les chemins candidats : possible uniquement si ceux-ci sont explicites et c’est ce que permet la technique de `bound`+spécialisation. Si on garde les buffers tel quels sûrement plus difficile.
- Y’a un problème de “scalability”.
 - L’utilisation des STS permet d’avoir un contrôle précis du nombre d’état (au moins 1, le collage des états ne semble pas poser de problème).
 - Le nombre minimale de transitions est celui des événements observables.
 - La complexité du produit est directement dépendante de la complexité des composants ...
 - En pratique au-delà de 30X50 ça devient délicat à la main, l’utilisation d’outils est un plus mais ne résout pas complètement les difficultés de lisibilité.
 - Une idée à explorer est le changement de STS : à partir d’un système déjà spécifié je peux avoir une description plus simple qui se focalise sur des événements ou conditions particulières. En gros ressemble à un changement de représentation/abstraction, peut-être une piste sérieuse.
- Pour la preuve du théorème global un problème d’expression et de preuve (chemin infini et induction pas dans le bon sens. Une technique ad-hoc semble possible mais peut-on faire mieux ?
- STS = LTS compactifié, outils de simulation divers à voir.

B The Bank Specification in PVS

```
theoryBank: THEORY
BEGIN

IMPORTING Bank

self: VAR Bank

id, p1: VAR nat

la: VAR list[[nat, nat]]

idle?, process?: [Bank → bool]

Pdef: [Bank → bool]

P_order_E, P_success, P_fail: [Bank → bool]
```

$P_order_R: [[Bank, nat, nat] \rightarrow bool]$
 $msg_order: [Bank \rightarrow bool]$
 $ok?: [Bank \rightarrow bool]$
 $idle_newBank: AXIOM \text{ idle?}(newBank(la))$
 $idle_order_R: AXIOM \text{ idle?}(order_R(self, id, p_1)) = \text{ idle?}(self)$
 $idle_order_E: AXIOM \neg \text{ idle?}(order_E(self))$
 $idle_success: AXIOM$
 $\quad \text{idle?}(success(self)) =$
 $\quad \text{IF process?}(self) \text{ THEN ok?}(self) \text{ ELSE FALSE ENDIF}$
 $idle_fail: AXIOM$
 $\quad \text{idle?}(fail(self)) =$
 $\quad \text{IF process?}(self) \text{ THEN } \neg \text{ ok?}(self) \text{ ELSE FALSE ENDIF}$
 $process_newBank: AXIOM \neg \text{ process?}(newBank(la))$
 $process_order_R: AXIOM \text{ process?}(order_R(self, id, p_1)) = \text{ process?}(self)$
 $process_order_E: AXIOM$
 $\quad \text{process?}(order_E(self)) =$
 $\quad \text{IF idle?}(self) \text{ THEN msg_order}(self) \text{ ELSE FALSE ENDIF}$
 $process_success: AXIOM \neg \text{ process?}(success(self))$
 $process_fail: AXIOM \neg \text{ process?}(fail(self))$
 $Pdef_newBank: AXIOM Pdef(newBank(la))$
 $Pdef_order_R: AXIOM$
 $\quad Pdef(order_R(self, id, p_1)) = Pdef(self) \wedge P_order_R(self, id, p_1)$
 $Pdef_order_E: AXIOM Pdef(order_E(self)) = Pdef(self) \wedge P_order_E(self)$
 $Pdef_success: AXIOM Pdef(success(self)) = Pdef(self) \wedge P_success(self)$
 $Pdef_fail: AXIOM Pdef(fail(self)) = Pdef(self) \wedge P_fail(self)$
 $P_order_R: AXIOM$
 $\quad P_order_R(self, id, p_1) = (\text{ idle?}(self) \vee \text{ process?}(self))$

```

P_order_E: AXIOM P_order_E(self) = (idle?(self) ∧ msg_order(self))

P_success: AXIOM P_success(self) = (process?(self) ∧ ok?(self))

P_fail: AXIOM P_fail(self) = (process?(self) ∧ ¬ ok?(self))

msg_order_newBank: AXIOM ¬ msg_order(newBank(la))

msg_order_order_R: AXIOM msg_order(order_R(self, id, p1))

msg_order_order_E_newBank: AXIOM ¬ msg_order(order_E(newBank(la)))

msg_order_order_E_order_R: AXIOM
  msg_order(order_E(order_R(self, id, p1))) = msg_order(self)

msg_order_order_E_success: AXIOM
  msg_order(order_E(success(self))) = msg_order(self)

msg_order_order_E_fail: AXIOM
  msg_order(order_E(fail(self))) = msg_order(self)

msg_order_success: AXIOM msg_order(success(self)) = msg_order(self)

msg_order_fail: AXIOM msg_order(fail(self)) = msg_order(self)

exclu: THEOREM ¬ (idle?(self) ∧ process?(self))

comp: THEOREM Pdef(self) = (idle?(self) ∨ process?(self))

impl: THEOREM (idle?(self) ⊃ Pdef(self)) ∧ (process?(self) ⊃ Pdef(self))

noblk: THEOREM
  ∀ (self: Bank, id: nat, p1: nat):
    (Pdef(self) ⊃
      (P_order_R(self, id, p1) ∨
        P_order_E(self) ∨ P_fail(self) ∨ P_success(self)))

END theoryBank

theoryRank: THEORY
BEGIN

IMPORTING theoryBank

```

self, self2, tmp: VAR Bank

id, p1, i, j, e, r: VAR nat

la: VAR list[[nat, nat]]

rank: [[Bank, nat, nat] → nat]

rank_newBank: AXIOM

$(e > 0 \wedge r \geq 0) \supset \text{rank}(\text{newBank}(la), e, r) = r - e$

rank_order_R: AXIOM

$\text{Pdef}(\text{order_R}(\text{self}, id, p_1)) \wedge (e > 0 \wedge r \geq 0) \supset$
 $\text{rank}(\text{order_R}(\text{self}, id, p_1), e, r) = \text{rank}(\text{self}, e, 1 + r)$

rank_order_E: AXIOM

$\text{Pdef}(\text{order_E}(\text{self})) \wedge (e > 0 \wedge r \geq 0) \supset$
 $\text{rank}(\text{order_E}(\text{self}), e, r) = \text{rank}(\text{self}, 1 + e, r)$

rank_success: AXIOM

$\text{Pdef}(\text{success}(\text{self})) \wedge (e > 0 \wedge r \geq 0) \supset$
 $\text{rank}(\text{success}(\text{self}), e, r) = \text{rank}(\text{self}, e, r)$

rank_fail: AXIOM

$\text{Pdef}(\text{fail}(\text{self})) \wedge (e > 0 \wedge r \geq 0) \supset$
 $\text{rank}(\text{fail}(\text{self}), e, r) = \text{rank}(\text{self}, e, r)$

thRank: THEOREM

$\text{Pdef}(\text{self}) \wedge (i > 0 \wedge j > 0) \supset$
 $\text{rank}(\text{self}, i, j) - 1 = \text{rank}(\text{self}, i, j - 1)$

END theoryRank

theoryRaux: THEORY

BEGIN

IMPORTING theoryRank

self, self2, tmp: VAR Bank

id, p1, i, e, r: VAR nat

raux: [[Bank, nat] → Bank]

raux_order_R: AXIOM

$\text{Pdef}(\text{order_R}(\text{self}, id, p_1)) \supset$

```

raux(order_R(self, id, p1), i) =
  IF i = 0
    THEN order_R(self, id, p1)
  ELSE raux(self, i - 1)
  ENDIF

```

```

raux_order_E: AXIOM
  Pdef(order_E(self))  $\supset$  raux(order_E(self), i) = raux(self, i)

```

```

raux_success: AXIOM
  Pdef(success(self))  $\supset$  raux(success(self), i) = raux(self, i)

```

```

raux_fail: AXIOM Pdef(fail(self))  $\supset$  raux(fail(self), i) = raux(self, i)

```

```

thRaux: THEOREM Pdef(order_E(self))  $\wedge$  (i  $\geq$  0)  $\supset$  order_R?(raux(self, i))

```

```

END theoryRaux

```

```

theoryReceived: THEORY
BEGIN

```

```

  IMPORTING theoryRaux

```

```

  la: VAR list[[nat, nat]]

```

```

  self, self2, tmp: VAR Bank

```

```

  id, id1, p1, i, e, r: VAR nat

```

```

  search(la: list[[nat, nat]], id: nat): bool =
    some( $\lambda$  (c: [nat, nat]): PROJ_1(c) = id)(la)

```

```

  find(la: list[[nat, nat]], id: nat): RECURSIVE nat =
    CASES la
      OF null: 0,
         cons(cons1_var, cons2_var):
           IF (PROJ_1(cons1_var) = id) THEN PROJ_2(cons1_var) ELSE find(cons2_var, id) ENDIF
      ENDCASES
    MEASURE length(la)

```

```

  received: [(process?)  $\rightarrow$  Bank]

```

```

  member: [[Bank, nat]  $\rightarrow$  bool]

```

```

  account: [[Bank, nat]  $\rightarrow$  nat]

```

```

member_newBank: AXIOM member(newBank(la), id) = search(la, id)

member_order_R: AXIOM
  member(order_R(self, id1, p1), id) = member(self, id)

member_order_E: AXIOM member(order_E(self), id) = member(self, id)

member_success: AXIOM member(success(self), id) = member(self, id)

member_fail: AXIOM member(fail(self), id) = member(self, id)

account_newBank: AXIOM account(newBank(la), id) = find(la, id)

account_order_R: AXIOM
  account(order_R(self, id1, p1), id) = account(self, id)

account_order_E: AXIOM account(order_E(self), id) = account(self, id)

account_success: AXIOM account(success(self), id) = account(self, id)

account_fail: AXIOM account(fail(self), id) = account(self, id)

received: AXIOM
  Pdef(order_E(self))  $\supset$ 
    received(order_E(self)) = raux(self, rank(self, 1, 0))

ok1: AXIOM
  Pdef(order_E(self))  $\supset$ 
    ok?(order_E(self)) =
      (member(received(order_E(self)), id)  $\wedge$ 
       (p1  $\leq$  account(received(order_E(self)), id)))

ok2: AXIOM
  Pdef(order_R(self, id, p1))  $\supset$  ok?(order_R(self, id, p1)) = ok?(self)

END theoryReceived

```

C The Specialised Bank Specification in PVS

```

theoryBankSpec: THEORY
  BEGIN

  IMPORTING Bank

  IMPORTING listNat

```

```

self: VAR Bank

la: VAR list[[nat, nat]]

id, id1, p1: VAR nat

one?, oneOrder?, two?: [Bank → bool]

Pdef: [Bank → bool]

P_order_E, P_success, P_fail, P_init: [Bank → bool]

P_order_R: [[Bank, nat, nat] → bool]

ok?: [(two?) → bool]

member: [[Bank, nat] → bool]

account: [[Bank, nat] → nat]

one_newBank: AXIOM one?(newBank(la))

one_order_R: AXIOM ¬ one?(order_R(self, id, p1))

one_order_E: AXIOM ¬ one?(order_E(self))

one_success: AXIOM
  one?(success(self)) = IF two?(self) THEN ok?(self) ELSE FALSE ENDIF

one_fail: AXIOM
  one?(fail(self)) = IF two?(self) THEN ¬ ok?(self) ELSE FALSE ENDIF

oneOrder_newBank: AXIOM ¬ oneOrder?(newBank(la))

oneOrder_order_R: AXIOM oneOrder?(order_R(self, id, p1)) = one?(self)

oneOrder_order_E: AXIOM ¬ oneOrder?(order_E(self))

oneOrder_success: AXIOM ¬ oneOrder?(success(self))

oneOrder_fail: AXIOM ¬ oneOrder?(fail(self))

two_newBank: AXIOM ¬ two?(newBank(la))

```

two_order_R: AXIOM \neg two?(order_R(self, id, p₁))

two_order_E: AXIOM two?(order_E(self)) = oneOrder?(self)

two_success: AXIOM \neg two?(success(self))

two_fail: AXIOM \neg two?(fail(self))

Pdef_newBank: AXIOM Pdef(newBank(la))

Pdef_order_R: AXIOM
Pdef(order_R(self, id, p₁)) = (Pdef(self) \wedge P_order_R(self, id, p₁))

Pdef_order_E: AXIOM Pdef(order_E(self)) = (Pdef(self) \wedge P_order_E(self))

Pdef_success: AXIOM Pdef(success(self)) = (Pdef(self) \wedge P_success(self))

Pdef_fail: AXIOM Pdef(fail(self)) = (Pdef(self) \wedge P_fail(self))

P_init_newBank: AXIOM P_init(newBank(la))

P_init_order_R: AXIOM \neg P_init(order_R(self, id, p₁))

P_init_order_E: AXIOM \neg P_init(order_E(self))

P_init_success: AXIOM \neg P_init(success(self))

P_init_fail: AXIOM \neg P_init(fail(self))

P_order_R: AXIOM P_order_R(self, id, p₁) = one?(self)

P_order_E: AXIOM P_order_E(self) = oneOrder?(self)

P_success: AXIOM P_success(self) = (two?(self) \wedge ok?(self))

P_fail: AXIOM P_fail(self) = (two?(self) \wedge \neg ok?(self))

member_newBank: AXIOM member(newBank(la), id) = search(la, id)

member_order_R: AXIOM
one?(self) \supset member(order_R(self, id₁, p₁), id) = member(self, id)

member_order_E: AXIOM
oneOrder?(self) \supset member(order_E(self), id) = member(self, id)

member_success: AXIOM

$(two?(self) \wedge ok?(self)) \supset$
 $member(success(self), id) = member(self, id)$

member_fail: AXIOM

$(two?(self) \wedge \neg ok?(self)) \supset member(fail(self), id) = member(self, id)$

account_newBank: AXIOM $account(newBank(la), id) = find(la, id)$

account_order_R: AXIOM

$one?(self) \supset account(order_R(self, id1, p1), id) = account(self, id)$

account_order_E: AXIOM

$oneOrder?(self) \supset account(order_E(self), id) = account(self, id)$

account_success: AXIOM

$(two?(self) \wedge ok?(self)) \supset$
 $account(success(self), id) = account(self, id)$

account_fail: AXIOM

$(two?(self) \wedge \neg ok?(self)) \supset$
 $account(fail(self), id) = account(self, id)$

ok1: AXIOM

$one?(self) \supset$
 $ok?(order_E(order_R(self, id, p1))) =$
 $IF member(self, id) THEN (p1 \leq account(self, id)) ELSE FALSE ENDIF$

received: [Bank \rightarrow Bank]

received: AXIOM

$one?(self) \supset$
 $received(order_E(order_R(self, id, p1))) = order_R(self, id, p1)$

exclu: THEOREM

$\neg (one?(self) \wedge oneOrder?(self)) \wedge$
 $\neg (one?(self) \wedge two?(self)) \wedge \neg (oneOrder?(self) \wedge two?(self))$

comp: THEOREM $Pdef(self) = (one?(self) \vee oneOrder?(self) \vee two?(self))$

impl: THEOREM

$(one?(self) \supset Pdef(self)) \wedge$
 $(oneOrder?(self) \supset Pdef(self)) \wedge (two?(self) \supset Pdef(self))$

noblk: THEOREM

```


$$\forall (self: Bank, id: nat, p_1: nat):$$

  (Pdef(self)  $\supset$ 
    (P_order_R(self, id, p_1)  $\vee$ 
      P_order_E(self)  $\vee$  P_fail(self)  $\vee$  P_success(self)))

P_init1: THEOREM P_init(self)  $\supset$  one?(self)

P_init2: THEOREM P_init(self)  $\supset$  Pdef(self)

th_order: THEOREM
  oneOrder?(self)  $\supset$ 
    ( $\exists$  (avant: Bank, ia: nat, pa: nat):
      (one?(avant)  $\wedge$  (order_R(avant, ia, pa) = self)))

END theoryBankSpec

theoryTime: THEORY
BEGIN

  IMPORTING theoryBankSpec

  self, self2: VAR Bank

  la: VAR list[[nat, nat]]

  id, p_1: VAR nat

  time: [Bank  $\rightarrow$  nat]

  duration: [[Bank, Bank]  $\rightarrow$  nat]

  time_newBank: AXIOM time(newBank(la)) = 3

  time_order_R: AXIOM time(order_R(self, id, p_1)) = 1 + time(self)

  time_order_E: AXIOM time(order_E(self)) = 2 + time(self)

  time_success: AXIOM time(success(self)) = 3 + time(self)

  time_fail: AXIOM time(fail(self2)) = 4 + time(self)

  th: THEOREM
    Pdef(order_E(self))  $\supset$ 
      (time(order_E(self)) - time(received(order_E(self))))  $\leq$  3

END theoryTime

```

D The Simplification Using PVS

In the following proof we start from a general definition of `ok?` and we simplify it according to the new bank STS definition. The context of the proof is the `theoryBank2` (the version with an explicit buffer computation) and some theorems coming from the representation.

```
abs_pdef: AXIOM theoryBankSpec.Pdef(self)  $\supset$  theoryBank2.Pdef(self)
```

```
abs_one: AXIOM
  theoryBankSpec.Pdef(self)  $\supset$ 
  one?(self) = (idle?(self)  $\wedge$  null?(buffer(self)))
```

```
abs_oneOrder: AXIOM
  theoryBankSpec.Pdef(self)  $\supset$ 
  oneOrder?(self) = (idle?(self)  $\wedge$  length(buffer(self)) = 1)
```

```
abs_two: AXIOM
  theoryBankSpec.Pdef(self)  $\supset$ 
  two?(self) = (process?(self)  $\wedge$  null?(buffer(self)))
```

END essai

The main rules are shown, the other steps are either trivial or simple rewriting.

simpok :

```
|-----
{1}  FORALL (self: Bank):
      theoryBankSpec.two?(self) IMPLIES
      (theoryBankSpec.ok?(self) = theoryBank2.ok?(self))
```

Skolemizing and flattening, then
Applying `th_two` and then
Instantiating quantified variables,
this simplifies to:

simpok :

```
{-1} two?(self!1) IMPLIES
      (EXISTS (avant: Bank):
        (oneOrder?(avant) AND (order_E(avant) = self!1)))
{-2} theoryBankSpec.two?(self!1)
|-----
{1}  (theoryBankSpec.ok?(self!1) = theoryBank2.ok?(self!1))
```

Splitting conjunctions,

we get 2 subgoals:

simpok.1 :

```
{-1} EXISTS (avant: Bank): (oneOrder?(avant) AND (order_E(avant) = self!1))
[-2] theoryBankSpec.two?(self!1)
    |-----
[1]  (theoryBankSpec.ok?(self!1) = theoryBank2.ok?(self!1))
```

Skolemizing and flattening, then

Replacing using formula -2, and then

Applying theoryBankSpec.impl where

self gets order_E(avant!1),

this simplifies to:

simpok.1 :

```
{-1} (one?(order_E(avant!1)) IMPLIES Pdef(order_E(avant!1))) AND
      (oneOrder?(order_E(avant!1)) IMPLIES Pdef(order_E(avant!1))) AND
      (two?(order_E(avant!1)) IMPLIES Pdef(order_E(avant!1)))
[-2] oneOrder?(avant!1)
[-3] (order_E(avant!1) = self!1)
[-4] theoryBankSpec.two?(order_E(avant!1))
    |-----
[1]  (theoryBankSpec.ok?(order_E(avant!1)) =
      theoryBank2.ok?(order_E(avant!1)))
```

Applying disjunctive simplification to flatten sequent, then

Hiding formulas: -2, -1,

this simplifies to:

simpok.1 :

```
{-1} (two?(order_E(avant!1)) IMPLIES Pdef(order_E(avant!1)))
[-2] oneOrder?(avant!1)
[-3] (order_E(avant!1) = self!1)
[-4] theoryBankSpec.two?(order_E(avant!1))
    |-----
[1]  (theoryBankSpec.ok?(order_E(avant!1)) =
      theoryBank2.ok?(order_E(avant!1)))
```

Splitting conjunctions,

we get 2 subgoals:

simpok.1.1 :

```
{-1} Pdef(order_E(avant!1))
[-2] oneOrder?(avant!1)
[-3] (order_E(avant!1) = self!1)
```

```

[-4] theoryBankSpec.two?(order_E(avant!1))
    |-----
[1]  (theoryBankSpec.ok?(order_E(avant!1)) =
      theoryBank2.ok?(order_E(avant!1)))

```

Rewriting using `abs_two`, matching in `*`, then
 Rewriting using `theoryBank2.ok2`, matching in `*`, and then
 Applying `th_order`
 this simplifies to:
`simpok.1.1` :

```

{-1} FORALL (self: Bank):
      oneOrder?(self) IMPLIES
        (EXISTS (avant: Bank, ia: nat, pa: nat):
          (one?(avant) AND (order_R(avant, ia, pa) = self)))
[-2] Pdef(order_E(avant!1))
[-3] oneOrder?(avant!1)
[-4] (order_E(avant!1) = self!1)
{-5} (process?(order_E(avant!1)) AND null?(buffer(order_E(avant!1))))
    |-----
[1]  (theoryBankSpec.ok?(order_E(avant!1)) =
      IF member(avant!1, PROJ_1(first(buffer(avant!1))))
      THEN (PROJ_2(first(buffer(avant!1))) <=
            account(avant!1, PROJ_1(first(buffer(avant!1))))
            ELSE FALSE
            ENDIF)

```

Instantiating quantified variables,
 this simplifies to:
`simpok.1.1` :

```

{-1} oneOrder?(avant!1) IMPLIES
      (EXISTS (avant: Bank, ia: nat, pa: nat):
        (one?(avant) AND (order_R(avant, ia, pa) = avant!1)))
[-2] Pdef(order_E(avant!1))
[-3] oneOrder?(avant!1)
[-4] (order_E(avant!1) = self!1)
[-5] (process?(order_E(avant!1)) AND null?(buffer(order_E(avant!1))))
    |-----
[1]  (theoryBankSpec.ok?(order_E(avant!1)) =
      IF member(avant!1, PROJ_1(first(buffer(avant!1))))
      THEN (PROJ_2(first(buffer(avant!1))) <=
            account(avant!1, PROJ_1(first(buffer(avant!1))))
            ELSE FALSE
            ENDIF)

```

Splitting conjunctions,

we get 2 subgoals:

simpok.1.1.1 :

```
{-1} EXISTS (avant: Bank, ia: nat, pa: nat):
      (one?(avant) AND (order_R(avant, ia, pa) = avant!1))
[-2] Pdef(order_E(avant!1))
[-3] oneOrder?(avant!1)
[-4] (order_E(avant!1) = self!1)
[-5] (process?(order_E(avant!1)) AND null?(buffer(order_E(avant!1))))
|-----
[1] (theoryBankSpec.ok?(order_E(avant!1)) =
      IF member(avant!1, PROJ_1(first(buffer(avant!1))))
      THEN (PROJ_2(first(buffer(avant!1))) <=
            account(avant!1, PROJ_1(first(buffer(avant!1))))
            ELSE FALSE
      ENDIF)
```

Skolemizing and flattening, then

Replacing using formula -2, and then

Applying theoryBankSpec.impl where

self gets order_R(avant!2, ia!1, pa!1),

this simplifies to:

simpok.1.1.1 :

```
{-1} (one?(order_R(avant!2, ia!1, pa!1)) IMPLIES
      Pdef(order_R(avant!2, ia!1, pa!1)))
AND
      (oneOrder?(order_R(avant!2, ia!1, pa!1)) IMPLIES
      Pdef(order_R(avant!2, ia!1, pa!1)))
AND
      (two?(order_R(avant!2, ia!1, pa!1)) IMPLIES
      Pdef(order_R(avant!2, ia!1, pa!1)))
{-2} one?(avant!2)
{-3} (order_R(avant!2, ia!1, pa!1) = avant!1)
[-4] Pdef(order_E(avant!1))
{-5} oneOrder?(order_R(avant!2, ia!1, pa!1))
[-6] (order_E(avant!1) = self!1)
{-7} process?(order_E(avant!1))
{-8} null?(buffer(order_E(avant!1)))
|-----
[1] (theoryBankSpec.ok?(order_E(order_R(avant!2, ia!1, pa!1))) =
      IF member(order_R(avant!2, ia!1, pa!1),
            PROJ_1(first(buffer(order_R(avant!2, ia!1, pa!1))))
```

```

THEN (PROJ_2(first(buffer(order_R(avant!2, ia!1, pa!1)))) <=
      account(order_R(avant!2, ia!1, pa!1),
                PROJ_1(first(buffer
                          (order_R(avant!2, ia!1, pa!1)))))
ELSE FALSE
ENDIF)

```

Applying disjunctive simplification to flatten sequent, then

Hiding formulas: -3, -1,

this simplifies to:

simpok.1.1.1 :

```

{-1} (oneOrder?(order_R(avant!2, ia!1, pa!1)) IMPLIES
      Pdef(order_R(avant!2, ia!1, pa!1)))
[-2] one?(avant!2)
[-3] (order_R(avant!2, ia!1, pa!1) = avant!1)
[-4] Pdef(order_E(avant!1))
[-5] oneOrder?(order_R(avant!2, ia!1, pa!1))
[-6] (order_E(avant!1) = self!1)
[-7] process?(order_E(avant!1))
[-8] null?(buffer(order_E(avant!1)))
|-----
[1] (theoryBankSpec.ok?(order_E(order_R(avant!2, ia!1, pa!1))) =
      IF member(order_R(avant!2, ia!1, pa!1),
                PROJ_1(first(buffer(order_R(avant!2, ia!1, pa!1)))))
      THEN (PROJ_2(first(buffer(order_R(avant!2, ia!1, pa!1)))) <=
            account(order_R(avant!2, ia!1, pa!1),
                      PROJ_1(first(buffer
                                (order_R(avant!2, ia!1, pa!1)))))
            ELSE FALSE
            ENDIF)

```

Splitting conjunctions,

we get 2 subgoals:

simpok.1.1.1.1 :

```

{-1} Pdef(order_R(avant!2, ia!1, pa!1))
[-2] one?(avant!2)
[-3] (order_R(avant!2, ia!1, pa!1) = avant!1)
[-4] Pdef(order_E(avant!1))
[-5] oneOrder?(order_R(avant!2, ia!1, pa!1))
[-6] (order_E(avant!1) = self!1)
[-7] process?(order_E(avant!1))
[-8] null?(buffer(order_E(avant!1)))
|-----

```

```

[1] (theoryBankSpec.ok?(order_E(order_R(avant!2, ia!1, pa!1))) =
    IF member(order_R(avant!2, ia!1, pa!1),
        PROJ_1(first(buffer(order_R(avant!2, ia!1, pa!1))))
    THEN (PROJ_2(first(buffer(order_R(avant!2, ia!1, pa!1)))) <=
        account(order_R(avant!2, ia!1, pa!1),
            PROJ_1(first(buffer
                (order_R(avant!2, ia!1, pa!1)))))
    ELSE FALSE
    ENDIF)

```

Rewriting using `abs_oneOrder`, matching in `*`, then
Rewriting using `idle_order_R`, matching in `*`, and then
Rewriting using `buffer_order_R`, matching in `*`,
this simplifies to:
simpok.1.1.1.1 :

```

[-1] Pdef(order_R(avant!2, ia!1, pa!1))
[-2] one?(avant!2)
[-3] (order_R(avant!2, ia!1, pa!1) = avant!1)
[-4] Pdef(order_E(avant!1))
{-5} (idle?(avant!2) AND length(cons((ia!1, pa!1), buffer(avant!2)))) = 1)
[-6] (order_E(avant!1) = self!1)
[-7] process?(order_E(avant!1))
[-8] null?(buffer(order_E(avant!1)))
|-----
{1} (theoryBankSpec.ok?(order_E(order_R(avant!2, ia!1, pa!1))) =
    IF member(order_R(avant!2, ia!1, pa!1),
        PROJ_1(first(cons((ia!1, pa!1), buffer(avant!2))))
    THEN (PROJ_2(first(cons((ia!1, pa!1), buffer(avant!2)))) <=
        account(order_R(avant!2, ia!1, pa!1),
            PROJ_1(first(cons
                ((ia!1, pa!1), buffer(avant!2)))))
    ELSE FALSE
    ENDIF)

```

Applying disjunctive simplification to flatten sequent, then
Rewriting using `length`, matching in `-6`, and then
Simplifying with decision procedures,
this simplifies to:
simpok.1.1.1.1 :

```

[-1] Pdef(order_R(avant!2, ia!1, pa!1))
[-2] one?(avant!2)
[-3] (order_R(avant!2, ia!1, pa!1) = avant!1)
[-4] Pdef(order_E(avant!1))

```

```

{-5} idle?(avant!2)
{-6} length(buffer(avant!2)) = 0
[-7] (order_E(avant!1) = self!1)
[-8] process?(order_E(avant!1))
[-9] null?(buffer(order_E(avant!1)))
|-----
[1] (theoryBankSpec.ok?(order_E(order_R(avant!2, ia!1, pa!1))) =
    IF member(order_R(avant!2, ia!1, pa!1),
        PROJ_1(first(cons((ia!1, pa!1), buffer(avant!2)))))
    THEN (PROJ_2(first(cons((ia!1, pa!1), buffer(avant!2))))) <=
        account(order_R(avant!2, ia!1, pa!1),
            PROJ_1(first(cons
                ((ia!1, pa!1), buffer(avant!2)))))
        ELSE FALSE
    ENDIF)

```

Applying th_list4 where

```

    la gets buffer(avant!2),
    this simplifies to:
simpok.1.1.1.1 :

```

```

{-1} (length(buffer(avant!2)) = 0) IMPLIES (buffer(avant!2) = null)
[-2] Pdef(order_R(avant!2, ia!1, pa!1))
[-3] one?(avant!2)
[-4] (order_R(avant!2, ia!1, pa!1) = avant!1)
[-5] Pdef(order_E(avant!1))
[-6] idle?(avant!2)
[-7] length(buffer(avant!2)) = 0
[-8] (order_E(avant!1) = self!1)
[-9] process?(order_E(avant!1))
[-10] null?(buffer(order_E(avant!1)))
|-----
[1] (theoryBankSpec.ok?(order_E(order_R(avant!2, ia!1, pa!1))) =
    IF member(order_R(avant!2, ia!1, pa!1),
        PROJ_1(first(cons((ia!1, pa!1), buffer(avant!2)))))
    THEN (PROJ_2(first(cons((ia!1, pa!1), buffer(avant!2))))) <=
        account(order_R(avant!2, ia!1, pa!1),
            PROJ_1(first(cons
                ((ia!1, pa!1), buffer(avant!2)))))
        ELSE FALSE
    ENDIF)

```

Splitting conjunctions,

```

    we get 2 subgoals:
simpok.1.1.1.1.1 :

```

```

{-1} (buffer(avant!2) = null)
[-2] Pdef(order_R(avant!2, ia!1, pa!1))
[-3] one?(avant!2)
[-4] (order_R(avant!2, ia!1, pa!1) = avant!1)
[-5] Pdef(order_E(avant!1))
[-6] idle?(avant!2)
[-7] length(buffer(avant!2)) = 0
[-8] (order_E(avant!1) = self!1)
[-9] process?(order_E(avant!1))
[-10] null?(buffer(order_E(avant!1)))
    |-----
[1] (theoryBankSpec.ok?(order_E(order_R(avant!2, ia!1, pa!1))) =
    IF member(order_R(avant!2, ia!1, pa!1),
        PROJ_1(first(cons((ia!1, pa!1), buffer(avant!2)))))
    THEN (PROJ_2(first(cons((ia!1, pa!1), buffer(avant!2))))) <=
        account(order_R(avant!2, ia!1, pa!1),
            PROJ_1(first(cons
                ((ia!1, pa!1), buffer(avant!2)))))
    ELSE FALSE
    ENDIF)

```

Replacing using formula -1, then
Rewriting using first_cons_null, matching in 1, and then
Simplifying with decision procedures,
this simplifies to:
simpok.1.1.1.1.1 :

```

[-1] (buffer(avant!2) = null)
[-2] Pdef(order_R(avant!2, ia!1, pa!1))
[-3] one?(avant!2)
[-4] (order_R(avant!2, ia!1, pa!1) = avant!1)
[-5] Pdef(order_E(avant!1))
[-6] idle?(avant!2)
[-7] length(buffer(avant!2)) = 0
[-8] (order_E(avant!1) = self!1)
[-9] process?(order_E(avant!1))
[-10] null?(buffer(order_E(avant!1)))
    |-----
{1} (theoryBankSpec.ok?(order_E(order_R(avant!2, ia!1, pa!1))) =
    IF member(order_R(avant!2, ia!1, pa!1), ia!1)
    THEN (pa!1 <= account(order_R(avant!2, ia!1, pa!1), ia!1))
    ELSE FALSE
    ENDIF)

```

Rewriting using `member_order_R`, matching in `*`, then
 Rewriting using `account_order_R`, matching in `*`, and then
 Hiding formulas: `-1, -2, -4, -5, -6, -7, -8, -9, -10`,
 this simplifies to:
`simpok.1.1.1.1.1 :`

```
[-1] one?(avant!2)
    |-----
{1}  (theoryBankSpec.ok?(order_E(order_R(avant!2, ia!1, pa!1))) =
      IF member(avant!2, ia!1) THEN (pa!1 <= account(avant!2, ia!1))
      ELSE FALSE
      ENDIF)
```

Merging and generalizing,
 we get 3 subgoals:
`simpok.1.1.1.1.1.1 :`

```
|-----
{1}  FORALL (avant_1: Bank, ia_1, pa_1: nat):
      one?(avant_1) IMPLIES
      (theoryBankSpec.ok?(order_E(order_R(avant_1, ia_1, pa_1))) =
        IF member(avant_1, ia_1) THEN (pa_1 <= account(avant_1, ia_1))
        ELSE FALSE
        ENDIF)
```

References

1. <http://www.irisa.fr/lande/jensen/dispo.html>.
2. Franz Achermann and Oscar Nierstrasz. Applications = Components + Scripts – A Tour of Piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.
3. Michel Allemand and Jean-Claude Royer. Mixed Formal Specification with PVS. In *Proceedings of the 15th IPDPS 2002 Symposium, FMPPTA*. IEEE Computer Society, 2002.
4. Robert Allen, Remi Douence, and David Garlan. Specifying and Analyzing Dynamic Software Architectures. In *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)*, volume 1382 of *Lecture Notes in Computer Science*, pages 21–37. Springer-Verlag, 1998.
5. Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.
6. André Arnold. Systèmes de transitions finis et sémantique des processus communicants. *Technique et Science Informatiques*, 9(3):193–216, April 1989.
7. André Arnold, G. Point, Alain Griffault, and Antoine Rauzy. The altarica formalism for describing concurrent systems. *FUNDINF: Fundamenta Informatica*, 34:109–124, 2000.
8. André Arnold. *Finite Transition Systems*. International Series in Computer Science. Prentice-Hall, 1994. ISBN 0-13-092990-5.

9. E. Astesiano, B. Krieg-Bruckner, and H.-J. Kreowski Eds., editors. *Algebraic Foundations of System Specification*. IFIP State-of-the-Art Reports. Springer Verlag, 1999. ISBN 3-540-63772-9.
10. Egidio Astesiano, Manfred Broy, and Gianna Reggio. *Algebraic Specification of Concurrent Systems*, pages 467–520. IFIP State-of-the-Art Reports. Springer Verlag, 1999. ISBN 3-540-63772-9.
11. Manfred Broy, Martin Wirsing, and Claude Pair. A Systematic Study of Models of Abstract Data Types. *Theoretical Computer Science*, 33:139–174, 1984.
12. M. Cerioli, T. Mossakowski, and H. Reichel. From Total Equational to Partial Conditional. In H.J. Kreowski, B. Krieg-Brueckner, and E. Astesiano, editors, *Algebraic Foundation of Information Systems Specification*, chapter 3, pages 31–104. Springer Verlag, 1999.
13. Christine Choppy, Pascal Poizat, and Jean-Claude Royer. Formal Specification of Mixed Components with Korrigan. In *Proceedings of the 8th Asia-Pacific Software Engineering Conference, APSEC'2001*, pages 169–176. IEEE, 2001.
14. Christine Choppy, Pascal Poizat, and Jean-Claude Royer. Specification of Mixed Systems in KORRIGAN with the Support of a UML-Inspired Graphical Notation. In Heinrich Hussmann, editor, *Fundamental Approaches to Software Engineering. 4th International Conference, FASE 2001*, volume 2029 of *LNCS*, pages 124–139. Springer, 2001.
15. Christine Choppy, Pascal Poizat, and Jean-Claude Royer. The Korrigan Environment. *Journal of Universal Computer Science*, 7(1):19–36, 2001. Special issue: Tools for System Design and Verification, ISSN: 0948-6968.
16. Gerardo Costa and Gianna Reggio. Specification of abstract dynamic-data types: A temporal logic approach. *Theoretical Computer Science*, 173(2):513–554, 1997.
17. F. Cuppens and C. Saurel. Towards a formalization of availability and denial of service. In *Information Systems Technology Panel Symposium on Protecting Nato Information Systems in the 21st Century*, Washington, 1999.
18. Grit Denker, José Meseguer, and Carolyn L. Talcott. Protocol specification and analysis in Maude. In N. Heintze and J. Wing, editors, *Proceedings of Workshop on Formal Methods and Security Protocols, June 25, 1998, Indianapolis, Indiana*, 1998. <http://www.cs.bell-labs.com/who/nch/fmsp/index.html>.
19. Wolfgang Emmerich. Distributed component technologies and their software engineering implications. In *Proceedings of the 24th Conference on Software Engineering (ICSE 02)*, pages 537–546. ACM Press, 2002.
20. Wolfgang Emmerich and Nima Kaveh. F2: Component technologies: Java beans, COM, CORBA, RMI, EJB and the CORBA component model. In Volker Gruhn, editor, *Proceedings of the Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engineering (ESEC/FSE-01)*, volume 26, 5 of *SOFTWARE ENGINEERING NOTES*, pages 311–312. ACM Press, 2001.
21. W. O. D. Griffioen and H. P. Korver. The bakery protocol: A comparative case-study in formal verification. Report CS-R9569, CWI, Amsterdam, November 1995.
22. W. O. D. Griffioen and H. P. Korver. The bakery protocol: A comparative case-study in formal verification. In J. C. van Vliet, editor, *CSN'95 (Computer Science in the Netherlands)*, pages 109–121. Stichting Mathematisch Centrum, 1995.
23. M. Grosse-Rhode. From Algebra Transformation to Labelled Transition Systems. In E. Astesiano, editor, *Recent Trends in Algebraic Development Techniques, Selected Papers of the 12th Workshop on Algebraic Development Techniques, WADT'97*, volume 1376 of *Lecture Notes in Computer Science*, pages 269–284. Springer-Verlag, 1998. ISSN 0302-9743.
24. David Harel. Statecharts: A visual formulation for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
25. M. Hennessy and H. Lin. Symbolic Bisimulations. *Theoretical Computer Science*, 138(2):353–389, 1995.
26. Paola Inverardi, Alexander L. Wolf, and Daniel Yankelevich. Static checking of system behaviors using derived component assumptions. *ACM Transactions on Software Engineering and Methodology*, 9(3):239–272, July 2000.

27. Michael Jackson and Pamela Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, 24(10):831–847, October 1998.
28. Teodor Knapik. Concurrency and Real-Time Specifications with Many-Sorted Logic and Abstract Data Type: an Example. In *Proceedings of the CARI Conference*. INRIA, october 1994. Ouagadougou.
29. Leslie Lamport and Nancy Lynch. *Distributed Computing: Models and Methods*, pages 1156–1199. Elsevier Science Publishers, 1990.
30. David C. Luckham, John L. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.
31. Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.
32. T. Murata. Petri nets: properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.
33. OMG. CORBA Component Model Specification, v3.0. Technical report, Object Management Group, 2002. www.omg.org/technology/documents/.
34. F. Orejas, M. Navarro, and A. Sanchez. Implementation and behavioural equivalence: a survey. In M. Bidoit and C. Choppy (Eds.), editors, *Recent Trends in data Type Specification*, volume 655 of *LNCS*, pages 93–125. Springer Verlag, august 1993.
35. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.
36. Liang Peng, Annya Romanczuk, and Jean-Claude Royer. A Translation of UML Components into Formal Specifications. In Theo D'Hondt, editor, *TOOLS East Europe 2002*, pages 60–75. Kluwer Academic Publishers, 2003. ISBN: 1-4020-7428-X.
37. 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.
38. Jean-Claude Royer. The GAT Approach to Specify Mixed Systems. *Informatica*, 27(1):89–103, 2003.
39. Jean-Claude Royer and Michael Xu. Analysing Mailboxes of Asynchronous Communicating Components. In D. C. Schmidt R. Meersman, Z. Tari and al., editors, *On The Move to Meaningful Internet Systems 2003: Coopis, DOA, and ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, pages 1421–1438. Springer Verlag, 2003.
40. Carron Shankland, Muffy Thomas, and Ed Brinksma. Symbolic Bisimulation for Full LOTOS. In *Algebraic Methodology and Software Technology AMAST'97*, volume 1349 of *Lecture Notes in Computer Science*, pages 479–493. Springer-Verlag, 1997.
41. Martin Wirsing. *Algebraic Specification*, volume B of *Handbook of Theoretical Computer Science*, chapter 13, pages 675–788. Elsevier, 1990. J. Van Leeuwen, Editor.
42. Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, March 1997.
43. C. Yu and V. Gligor. A specification and verification method for preventing denial of service. *IEEE Transactions on Software Engineering*, 16(6):581–592, June 1990.
44. Pamela Zave and Michael Jackson. A component-based approach to telecommunication software. *IEEE Software*, 15(5):70–78, 1998.

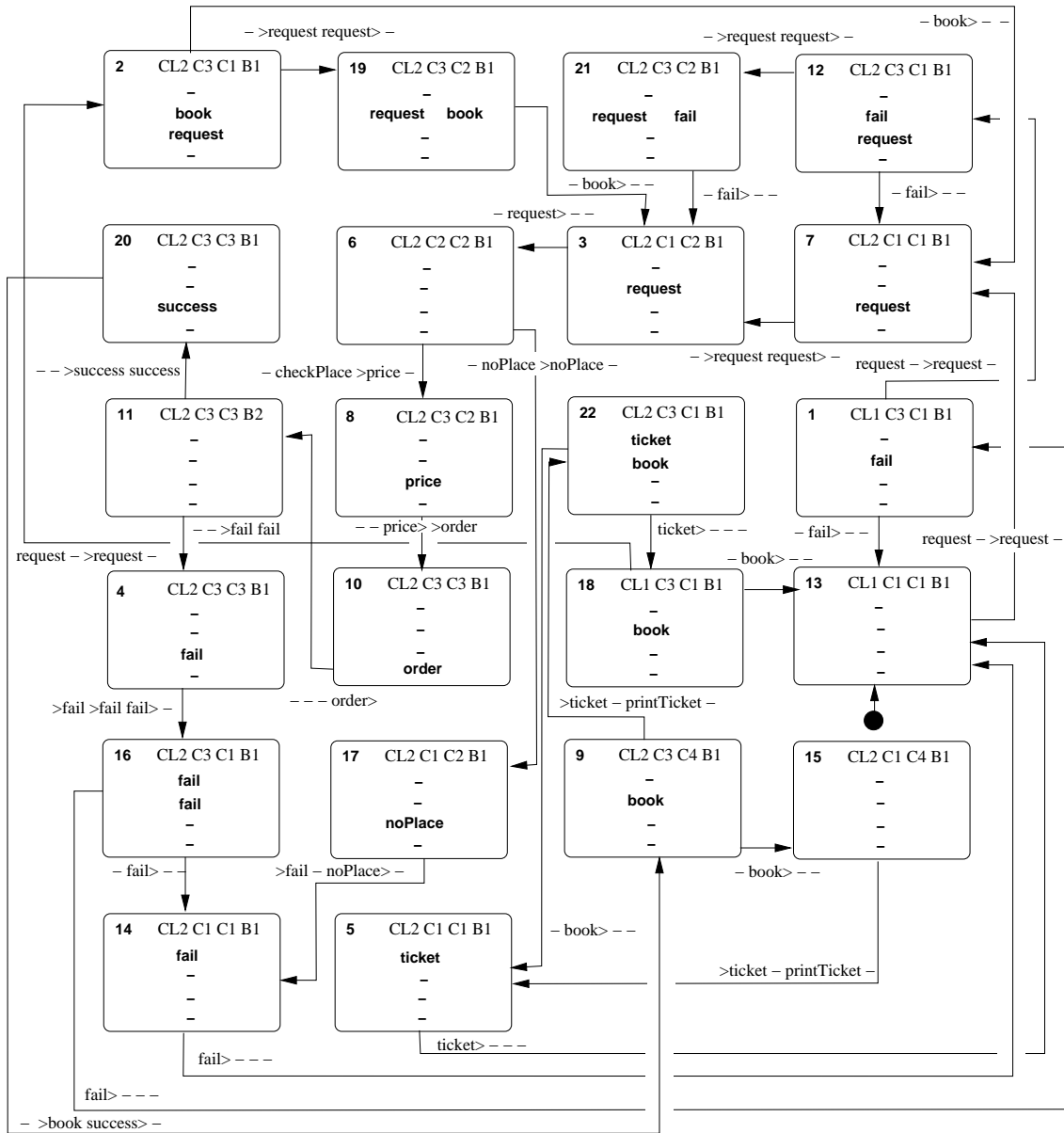


Fig. 8. The Dynamic Behaviour: Client x Company x Counter x Bank

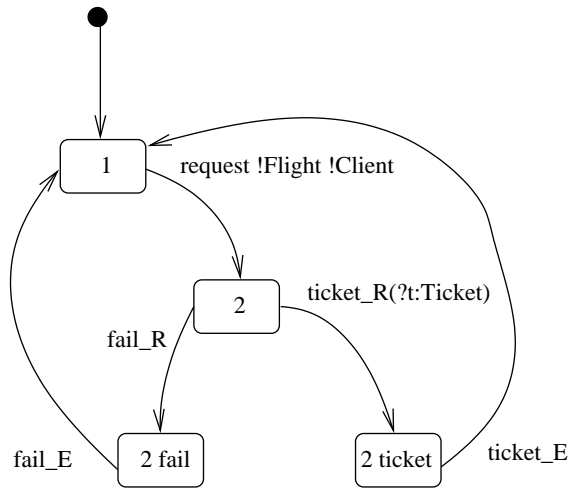


Fig. 9. The Specialised Behaviour for Client

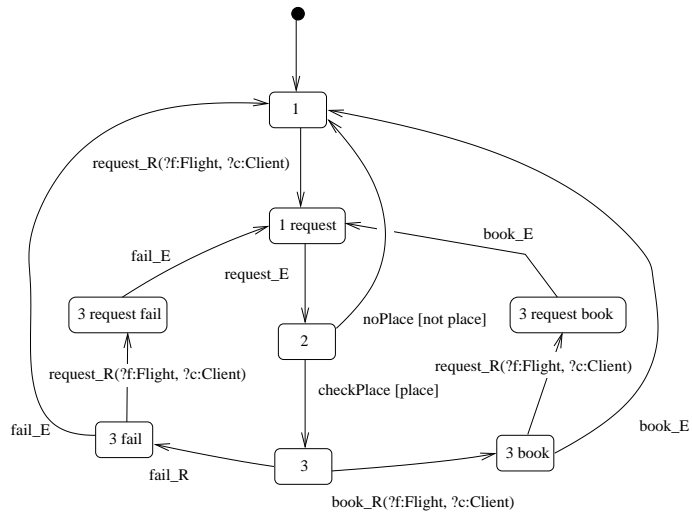


Fig. 10. The Specialised Behaviour for Company

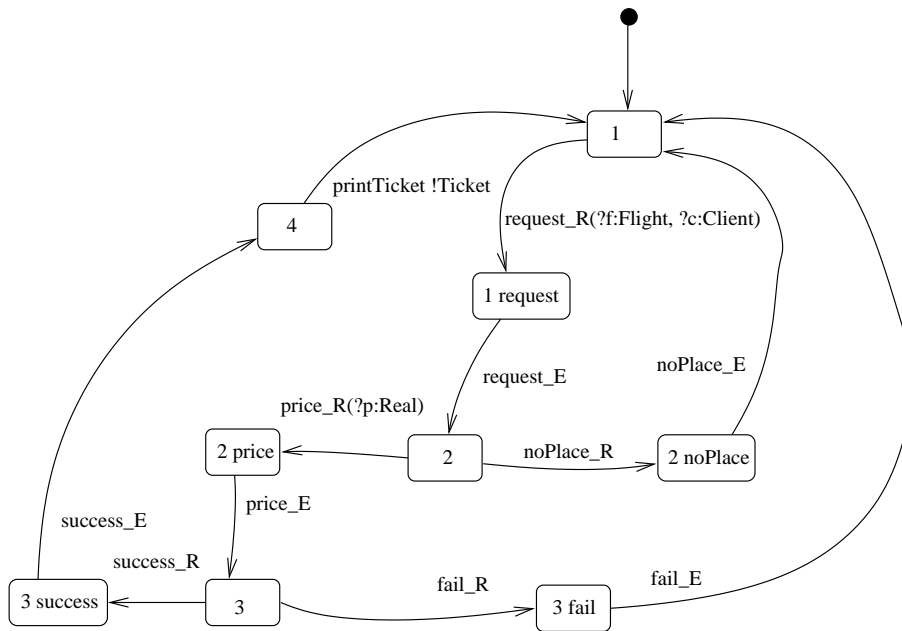


Fig. 11. The Specialised Behaviour for Counter