

# Parameterized Specification and Verification of the Chilean Electronic Invoices System

Isabelle Attali, Tomás Barros, Eric Madelaine  
*INRIA Sophia-Antipolis, France*  
(Isabelle.Attali, Tomas.Barros, Eric.Madelaine)@sophia.inria.fr

## Abstract

*We present the complete process of a formal specification and verification of the Chilean electronic invoice system which has been defined by the tax agency. We use this case study as a real-world and real-size example to illustrate our methodology for specification and verification of distributed applications. Our approach is based on a new hierarchical and parameterized model for synchronised networks of labelled transition systems. In this case study, we use a subset of the model as a graphical specification language. We check this formal specification of the invoice system against its informal requirements, described in terms of parameterized temporal logic formulas. Their satisfiability cannot be checked directly on the parameterized model: we introduce a method and a tool to instantiate the parameterized models and properties, allowing to use standard (finite-state, bisimulation-based) model-checkers for the verification. We also illustrate the use of different methods to avoid the state explosion problem by taking advantage of the parameterized structure and instantiations.*

## 1. Introduction

This work shows a case study in formal specification and verification. It was inspired by the recent definition by the Chilean tax administration of a new system for the exchange of electronic documents between sales actors (vendors, buyers, and the tax administration), that will progressively replace the current classical invoice system on paper. This system is an example of a massively distributed application, with strong constraints on security (authentication, integrity, non-repudiation), and a number of exchange protocols between the actors, that we shall detail later. Furthermore, the administration is only defining a specification of the application components that must be run by each actor (e.g. a vendor), and not a formally certified implementation of these components. There is an incorporation process that the new companies should accomplish. This incor-

poration process includes testing, simulation and checking steps. Thus, the question will arise whether specific implementations of the system components, possibly developed by different software companies, will obey their specification and behave correctly inside the whole system.

We use this case study to illustrate our approach to the specification and verification of distributed applications. In this respect, this example has a number of interesting features: first it is a real-world application, whose (informal) requirements have been published on the tax agency web server. It is a large example, that fits well with the idea of parameterized models: it has a complex component structure (17 automata in 4 levels of hierarchy), it will run with thousands of instances of the components and the smallest instantiation, with two actors of each type, already has a pretty large state space (over  $10^{12}$ ). Last, even if not a standard “critical system”, it is naturally the kind of distributed applications for which both security and safety have a very strong economical impact.

We are interested here in the behavioural properties of this distributed system (the sequences of communication events within the system, and the progress of the protocols between the actors), and we suppose that the security aspects are addressed at a different level. We mainly want to address two questions: how do we formally specify the behaviour of a system component, so that an implementation can be compared to this specification? And how do we verify that the global system, composed from a given number of those components, behaves correctly?

In [2] we have defined a parameterized and hierarchical model for synchronised networks of labelled transition systems. We have shown how this model can be used as an intermediate format to represent the behaviour of distributed applications, and check their temporal properties.

In this paper, we illustrate another usage of the same model: we define a graphical language that corresponds to a subset of our parameterized model. We use it to build a formal specification of the electronic invoice system and to check its properties. The graphical language retains the main features of the formal model: labelled transition sys-

tems and synchronisation networks, in which both events (messages) and agents (distributed objects) can be parameterized. The main restriction is that the network topology is static in the current version of the graphical language; this restriction could be lifted, as done in the case of Lotos by Lakas [14], but this was not required for our example.

The design of the model and of the graphical language was guided by several ideas. Having a framework based on process algebras and bisimulation semantics made possible to benefit from compositional methods for specification and for verification [17, 4]. The value-passing and parameterized features were strongly required, both for expressiveness of the model, and for better usability of the language by application developers. For convenience, we limit data domains to be of simple first order types; this allows us to define methods and tools for instantiating the parameterized models into finite hierarchical models, and to use standard finite-state checking tools [5, 11] for verifying the system properties. We also use a variant of the graphical language to express a large family of safety properties, and use temporal logics only for more complex properties.

We believe that a graphical notation for Labelled Transition Systems and Synchronisation Networks (in which both events and agents can be parameterized), associated with methods and tools for an instantiation of the parameterized models into finite hierarchical models, is a mandatory first step to bring formal methods into practical use. On one hand, our graphical notation with parameterized features is both intuitive and expressive enough for application developers when formally specifying the model and its intended properties; on the other hand, parameterized techniques master the complexity and enhance the applicability of these algorithms (usually limited in practice on relatively small size systems) for specification and verification.

Our model is an adaptation of the *symbolic transition graphs with assignment* of [15] into the *synchronisation networks* of [1]: we extend the general notion of Labelled Transition Systems (LTS) and hierarchical networks of communicating systems (synchronisation networks) from [1] adding parameters to the communication events in the spirit of [15]. Events can be guarded with conditions on their parameters. Our processes can also be parameterized to encode sets of equivalent processes running in parallel. The parameters are typed variables of simple enumerable types: booleans, integers, intervals, finite enumerations or structured objects.

Our approach is suitable both for compositional description of distributed system behaviours and for generating models from static analysis of some source code. We have shown in [2] how to generate models by static analysis of the source code for ProActive applications. ProActive [6] is a Java implementation of distributed active objects with asynchronous communications and replies by means of fu-

ture references.

We have developed a tool which, given a finite domain for the parameters, can generate finite labelled transition systems and synchronisation networks from the parameterized models.

Our main contributions in this work are:

- the complete process of formalisation and verification of the Chilean electronic invoices system (the full version is available in [3]);
- the use of this case study as a real-world and real-size example to illustrate our methodology for specification and verification of distributed system defined in [2];
- the development of a method and a tool to instantiate the parameterized models, allowing to use standard (finite-state, bisimulation based) model-checkers for the analysis;
- the proposal of methods to avoid the *state explosion problem*, in a parameterized setting, by hierarchical composition, hiding and grouping by variables.

In the next section we give an informal presentation of the electronic tax system, as specified by the Chilean administration in September 2002 and we list the informal requirements that we shall verify on our specification. In section 3, we introduce our theoretical models. Section 4 presents our graphical syntax to describe system behaviours. Section 5 overviews the formalisation of the Chilean electronic invoices using the graphical syntax. In section 6, we introduce our verification methodology and give two examples of property verification. Section 7 explains how we derive finite instantiations and we introduce methods to limit as much as possible the *state explosion problem*. Finally section 8 introduces some related work and section 9 draws the conclusions and the perspectives of this work.

## 2. Electronic invoices in Chile

In this section, we informally describe the electronic invoice system recently realized in Chile, as published in September 2002; for a detailed explanation, please look at [9].

### 2.1. System description

The Chilean law requires any commercial transaction done in Chile to be supported by a legal document previously authorised by the tax agency (Servicio de Impuestos Internos, from now on **SI**). There are several types of documents depending on the transaction such as the invoice for sales, or the forms for the transportation of goods. For a specific taxpayer and document type, each emitted document is

assigned a unique number named *id*. Before emitting a document, it must be authorised by SII: this is done through an authorisation stamp specific to a set of documents, a document type and a taxpayer. The taxpayer obtains authorisation stamps via the SII Web site. We call the emitter of an invoice a “vendor” and its receptor a “buyer”, even if those may be simply two different *roles* of the same taxpayer.

Every generated document must be sent to SII before sending it to the buyer and before the transport of goods (if relevant). All documents must include a digital seal, generated from the document data and the authorisation stamp.

SII has created a Web site where the buyer can verify if an invoice has been authorised and verify whether the emitter has sent the same invoice to SII than the buyer has received.

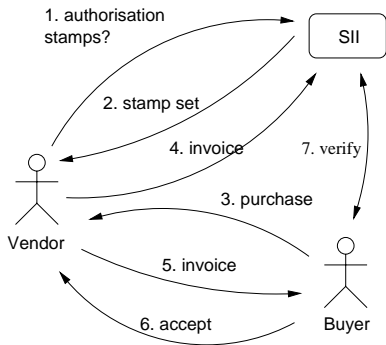


Figure 1. Normal Scenario

The most common scenario is shown in Figure 1. In step 1 the vendor asks for authorisation stamps. SII responds with a stamp set (step 2). Once a buyer has made a purchase (step 3), the vendor generates an invoice, sends it first to SII (step 4), then sends it to the buyer (step 5). In this scenario, the buyer will accept the invoice (step 6) and later it will verify the validity of the invoice with SII (step 7).

An electronic invoice is *well emitted* if it respects the format specifications defined by SII; if this is not the case, SII will refuse it and the invoice will be considered as never emitted. On the buyer’s side, if the transaction has never been realized or if there are errors in the invoice information, the buyer may refuse the invoice and consider it as never received. Then it is the duty of the emitter to send a cancellation of the invoice to SII.

Note that Figure 1 is just a drawing meant to explain the application, not a part of the specification. In fact the available specification is informal, and consists in a natural language (spanish) description of the protocols and of document formats. Our first task then was to identify within this informal specification the parts relative to the communication between the various subsystems, and to extract a list of semi-formal requirements.

## 2.2. System properties

Some of the behavioural properties that the system should respect are listed below. The published requirements [9] focus more on the format and contents of the electronic invoices than on the behaviour of the system. Properties 1, 3, 4, and 6 appear explicitly in the requirements. Properties 2, 5 and 7 do not, but they appeared as useful extensions of the specification.

1. A taxpayer cannot emit invoices if it has not received stamps from SII. More specifically, a taxpayer can emit as many invoices as the quantity of stamps received from SII.
2. SII gives the right answers to the invoice status request: not present when it has not been sent to SII, present when it has been sent, and cancelled when it has been cancelled by the vendor.
3. Every invoice refused by a buyer must be cancelled by the vendor.
4. An invoice id can be used only once.
5. It is not possible to cancel an invoice which has not been emitted before.
6. Every invoice sent to a buyer, should be sent to SII first.
7. Every emitted invoice finishes being accepted by the buyer or cancelled in SII.

In section 6.2 we formalise and verify two of those properties. All properties are formalized and fully verified in the full version of this paper [3].

## 3. Definitions

In this section we introduce the theoretical model that supports our approach. Our systems are distributed, communicating, asynchronous processes organised in hierarchical synchronisation networks.

We start with an unspecified set of communications **Actions** *Act*, that will be refined later.

We model the behaviour of a process as a Labelled Transition System (LTS) in a classical way [17]. The LTS transitions encode the actions that a process can perform in a given state.

**Definition 1 LTS.** A labelled transition system is a tuple  $LTS = (S, s_0, L, \rightarrow)$  where  $S$  is the set of states,  $s_0 \in S$  is the initial state,  $L \subseteq Act$  is the set of labels,  $\rightarrow$  is the set of transitions:  $\rightarrow \subseteq S \times L \times S$ . We write  $s \xrightarrow{\alpha} s'$  for  $(s, \alpha, s') \in \rightarrow$ .

Then we define **Nets** in a form inspired by [1], that are used to synchronise a finite number of processes. A Net is a

form of generalised parallel operator, and each of its arguments are typed by a **Sort** that is the set of its possible observable actions.

**Definition 2 Sort.** A Sort is a set  $I \subseteq Act$  of actions.

A LTS  $(S, s_0, L, \rightarrow)$  can be used as an argument in a Net if it agrees with the corresponding Sort ( $L \subseteq I_i$ ). Then a Sort characterises a family of LTSs which satisfy this inclusion condition.

Nets describe dynamic configurations of processes, in which the possible synchronisations change with the state of the Net. They are Transducers, in a sense similar to the open Lotos expressions of [14]. They are encoded as LTSs which labels are synchronisation vectors, each describing one particular synchronisation of the process actions:

**Definition 3 Net.** A Net is a tuple  $\langle A_G, I, T \rangle$  where  $A_G$  is a set of global actions,  $I$  is a finite set of Sorts  $I = \{I_i\}_{i=1..n}$ , and  $T$  (the transducer) is a LTS  $T = (T_T, s_0, L_T, \rightarrow_T)$ , such that  $\forall \vec{v} \in L_T, \vec{v} = \langle l_i, \alpha_1, \dots, \alpha_n \rangle$  where  $l_i \in A_G$  and  $\forall i \in [1..n], \alpha_i \in I_i \cup \{idle\}$ .

We say that a Net is *static* when its transducer vector contains only one state. Note that a synchronisation vector can define a synchronisation between one, two or more actions from different arguments of the Net. When the synchronisation vector involves only one argument, its action can occur freely.

The semantics of the Net construct is given by the synchronisation product:

**Definition 4 Synchronisation Product.** Given a set of LTS  $\{LTS_i = (S_i, s_{0_i}, L_i, \rightarrow_i)\}_{i=1..n}$  and a Net  $\langle A_G, \{I_i\}_{i=1..n}, (S_T, s_{0_T}, L_T, \rightarrow_T) \rangle$ , such that  $\forall i \in [1..n], L_i \subseteq I_i$ , we construct the product LTS  $(S, s_0, L, \rightarrow)$  where  $S = S_T \times \prod_{i=1}^n (S_i)$ ,  $s_0 = s_{0_T} \times \prod_{i=1}^n (s_{0_i})$ ,  $L = A_G$ , and the transition relation is defined as:

$$\begin{aligned} & \rightarrow \triangleq \{s \xrightarrow{l_i} s' \mid s = \langle s_t, s_1, \dots, s_n \rangle, s' = \langle s'_t, s'_1, \dots, s'_n \rangle \\ & \exists s_t \xrightarrow{\vec{v}} s'_t \in \rightarrow_T, \vec{v} = \langle l_t, \alpha_1, \dots, \alpha_n \rangle, \forall i \in [1..n], (\alpha_i \neq idle \wedge s_i \xrightarrow{\alpha_i} s'_i \in \rightarrow_i) \vee (\alpha_i = idle \wedge s_i = s'_i) \} \end{aligned}$$

Note that the result of the product is a LTS, which in turn can be synchronised with other LTSs in a Net. This scheme enables us to have different levels of synchronisations, i.e. a hierarchical definition for a system. It also allows us to consider Nets as generic parallel operators, in a process algebra whose constants are LTSs. In this algebra, strong and weak bisimulations are congruences for Nets operators, in a way that extends naturally CCS laws.

Next, we introduce our parameterized systems which are an extension from the above definitions to include parameters. These definitions are connected to the semantics of Symbolic Transition Graph with Assignment (STGA) [15].

Parameterized Actions have a rich structure, for they take care of value passing in the communication actions, of assignment of state variables, and of process parameters. In order to be able to define variable instantiation as an *abstraction* of the data domains (in the style of [7]), we restrict these domains to be **simple types**, namely: booleans, countable sets, integers or intervals over integers and finite structured objects. This should also include arrays of simple types, but this is not part of this paper.

**Definition 5 Parameterized Actions** are:  $\tau$  the non-observable action,  $\mathcal{M}$  encoding an observable local sequential program (with assignment of variables),  $?P.m(\vec{x})$  encoding the reception of a call to the method  $m$  from the process  $P$  ( $\vec{x}$  will be affected by the arguments of the call) and  $!P.m(\vec{e})$  encoding a call to the method  $m$  of a remote process  $P$  with arguments  $\vec{e}$ .

A parameterized LTS is a LTS with parameterized actions, with a set of parameters (defining a family of similar LTSs) and variables attached to each state. Parameters and variables have a simple type. Additionally, the transitions can be guarded and have a resulting expression which assigns the variables associated to the arriving state:

**Definition 6 pLTS.** A parameterized labelled transition system is a tuple  $pLTS = (K, S, s_0, L, \rightarrow)$  where:

- $K = \{k_i\}$  is a finite set of parameters,
  - $S$  is the set of states, and each state  $s \in S$  is associated with a finite set of variables  $\vec{v}_s$ ,
  - $s_0 \in S$  is the initial state,
  - $L = (b, \alpha(\vec{x}), \vec{e})$  is the set of labels (parameterized actions), where  $b$  is a boolean expression,  $\alpha(\vec{x})$  is a parameterized action, and  $\vec{e}$  is a finite set of expressions.
- $\rightarrow \subseteq S \times L \times S$  is the set of transitions:

**Definition 7 Parameterized Sort.** A Parameterized Sort is a set  $pI$  of parameterized actions.

**Definition 8 A pNet** is a tuple  $\langle pA_G, H, T \rangle$  where:  $pA_G$  is the set of global parameterized actions,  $H = \{pI_i, K_i\}_{i=1..n}$  is a finite set of holes (arguments). The transducer  $T$  is a pLTS  $T = (K_G, S_T, s_{0_T}, L_T, \rightarrow_T)$ , such that  $\forall \vec{v} \in L_T, \vec{v} = \langle l_t, \alpha_1^{k_1}, \dots, \alpha_n^{k_n} \rangle$  where  $l_t \in pA_G$ ,  $\alpha_i \in pI_i \cup \{idle\}$  and  $k_i \in K_i$ .

The  $K_G$  of the transducer is the set of global parameters of the pNet. Each hole in the pNet has a sort constraint  $pI_i$  and a parameter set  $K_i$ , expressing that this "parameterized hole" corresponds to as many actual arguments as necessary in a given instantiation. In a synchronisation vector  $\vec{v} = \langle l_t, \alpha_1^{k_1}, \dots, \alpha_n^{k_n} \rangle$ , each  $\alpha_i^{k_i}$  corresponds to the  $\alpha_i$  action of the  $k_i$ -nth corresponding argument LTS.

In the framework of this paper, we do not want to give a more precise definition of the language of parameterized actions, and we shall not try to give a direct definition of the

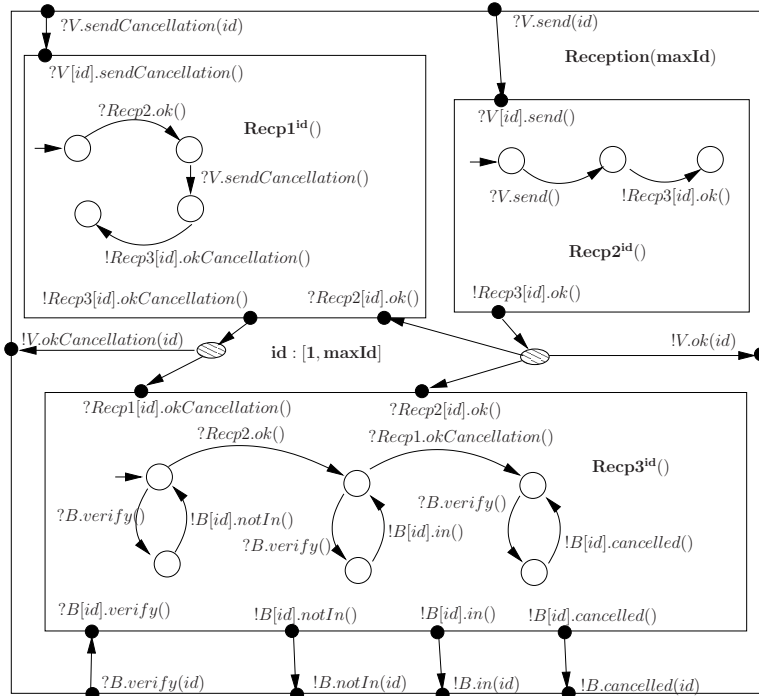


Figure 2. The reception and verification process

synchronisation product of pNets/pLTSs. Instead, we shall instantiate separately a pNet and its argument pLTSs (abstracting the domains of their parameters and variables to finite domains, before instantiating for all possible values of those abstract domains), then use the non-parameterized synchronisation product (Definition 4). This is known as the early approach to value-passing systems [17, 18].

#### 4. Graphical Language

We provide a graphical syntax for representing *static* Parameterized Networks, that is a compromise between expressiveness and user-friendliness. We use a graphical syntax similar to the Autograph editor [5], augmented by elements for parameters and variables: a *pLTS* is drawn as a set of circles representing states and edges representing transitions, where the states are labelled with the set of variables associated with it ( $\vec{v}_s$ ) and the edges are labelled by  $[b] \alpha(\vec{x}) \rightarrow \vec{e}$  (see Definition 6).

A *static pNet* is represented by a set of boxes, each one encoding a particular Sort of the pNet. These boxes can be filled with a pLTS satisfying the Sort inclusion condition. Each box has labelled bullets on the border, each one encoding a particular parameterized action of the Sort, we name those bullets as ports.

Figure 2 shows an example of such a parameterized system taken from the SII formalisation. It also introduces the

notation to encode sets of processes; for example,  $\mathbf{Recp1}^{id}$  encodes the set of  $\mathbf{Recp1}$  processes for each value in the domain of  $id$ . In the rest of this section, we use this example to introduce the main graphical features of our language.

The network **Reception** in Figure 2 specifies the part of the SII process in charge of receiving the documents (invoices and cancellations), and answering requests about the status of an invoice. It is parameterized by the  $id$  of the invoice and is composed by three automata sets whose elements take care of one specific document  $id$  (of a given vendor). The top right automaton (**Recp2**) takes care of receiving an invoice, the top left automaton (**Recp1**) takes care of receiving a cancellation document and the bottom automaton (**Recp3**) gives the status of an invoice when requested.

The responses to an invoice status for a given  $id$  are or-exclusive: the invoice is not present at SII ( $!B.notIn(id)$ ), the invoice has been sent to SII ( $!B.in(id)$ ), or the invoice has been cancelled by the vendor ( $!B.cancelled(id)$ ).

The edges between ports in Figure 2 are called links. Links express synchronisation between internal boxes or to external processes. Each link encodes a transition in the Transducer LTS of the *pNet*.

Figure 2 introduces as well the syntax to indicate a synchronisation in between more than two actions. A multiple synchronisation is represented by an ellipse with multiple arriving/outgoing edges from/to the ports of the processes whose actions must be done simultaneously. All three pro-

cesses are parameterized by *id*. In the reception ports, the *id* variable encodes the restriction that the receiving call is effectively addressed to the corresponding process (must be a match between the identity and the *id* in the call). Note that this restriction is expressed only in the port and not in the transitions inside the automaton, for instance the reception of the  $?Recp2[id].ok()$  call in **Recp1** will fire the transition  $?Recp2.ok()$  in the automaton.

In **Recp3**, initially an invoice is considered as not received. Upon reception, its status is changed to be present through a message sent by the **Recp2** ( $!Recp3[id].okIn()$ ). Then, if a cancellation arrives for an invoice, its status is changed to be cancelled through a message sent by **Recp1** ( $!Recp3[id].okCancellation()$ ). Note that the reception of a cancellation is only possible after the reception of the invoice to be cancelled (only after the transition  $?Recp2.ok()$  is made).

The full specification of this case study has shown that this graphical language is both expressive enough to represent a large and complex system and its interactions, and natural enough both for the system developers to draw the specification and for the reader to understand very intuitively the drawings. We are in the process of building a graphical editor that will help the developers to create the drawings, as well as translate them into the formal model and display graphically the results of the verification tools.

## 5. Formalisation

We have used this graphical language to build pLTSs and pNets for the formal specification of the Chilean invoices system. The intention here is not to describe all aspects of the system specification. We rather concentrate on the behaviour of the system, the communications between the distributed processes and their temporal properties.

- We assume that the communication channels are reliable.
- Security aspects (authentication, integrity) and document format verification are supposed to be treated elsewhere. All the processes in the system are trusted.
- There are only two types of documents, invoices and cancellations and only two types of authorisation stamps, one for invoices and another for cancellations. The only specific value to be considered for a document is its identification number (*id*).

The reception process of SII was introduced in section 4. In this section we concentrate on the synchronisation network of the Vendor and the global system. The interested reader will find the complete system formalisation in [3].

### 5.1. The Vendor system

Figure 3 shows the network that defines the behaviour of the Vendor. It has two pairs of **Stock** and **Id** processes: one pair for invoices and the other for cancellations. The **Stock** process manipulates a stock of stamps. It provides stamps for the generation of documents and requests new stamps to SII. The **Id** process assigns a unique sequential number to each new document (once a stamp has been provided by the **Stock** process). There is one single **BV** process that initiates new purchases. The purchase process (**PP**) takes care of the main life's cycle of a purchase. It is parameterized with the variable *pcrs*, which encodes the number of purchases that can be treated simultaneously (Section 4 explains the notation  $P^n$  for processes). There is a cancellation process (**CI**) for each invoice *id* (which can possibly be cancelled). The **PP** process sends requests to the **Id** invoices process for new invoices *ids* while the **CI** process does so with the **Id** cancellations process.

The actions to synchronise with external processes are represented by labelled bullets in the surrounding frame box. Note that even when the **Id** and **Stock** processes seem to be the same for invoices and cancellations, they could be instantiated with different domain of variables, resulting in different finite non-parameterized processes.

### 5.2. The Global System

The global behaviour of the system is formed by an arbitrary number of vendors, buyers and a single SII. The synchronisation links are labelled so they are visible. Those links reflect the possible communications events, such as: to request new stamps from the vendor *v* ( $reqNewStamps(v)$ ), to send the invoice *id* from the vendor *v* to SII ( $sendSii(v, id)$ ) or to refuse the invoice *id* sent from the vendor *v* to the buyer *b*. The global system is fully described using 15 pLTSs structured by 7 pNets in 4 levels of hierarchy.

## 6. Properties Verification

Given a system and a set of properties, we want to prove that these properties hold in the system. In this section we introduce our methodology to verify the properties listed in section 2.2 and we explain the verification results for two of them.

The verification tools we use work over finite LTSs. To use them in the invoices system, we instantiate the processes and networks and we generate the synchronisation product (global LTS) of those instantiations. Instead of generating directly the global LTS, we benefit from the compositional structure of the system. We go deeply on this subject in section 7.

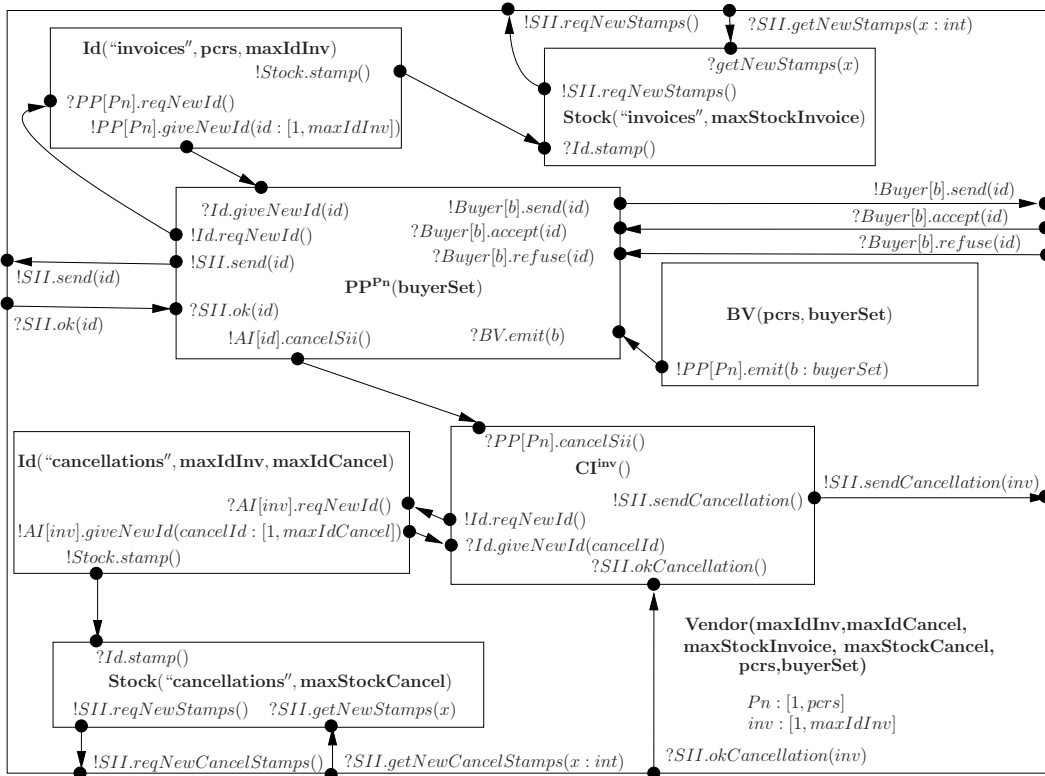


Figure 3. The Vendor system

The verification was done over the global synchronisation product of the instantiated processes and networks which form the system. The instantiation is made with the variable domains described below.

### 6.1. Data domains

An finite instantiation of a parameterized model is an abstraction in the sense of [7, 19]. Starting with first order (countable) data domains, we define abstractions in which the abstract domain has values corresponding to a finite number of distinguished concrete values, plus one or more extra values representing the rest of the concrete domain. These abstractions define Galois insertions [7]. Such an abstraction will preserve a given formula if it has enough abstract values in the abstract domain of each parameter in the formula, to represent each distinguished value of the parameter in the formula.

We observe that all the properties listed in section 2.2 involve at most one buyer and/or one vendor. This does not mean that the property should be valid for only one specific vendor/buyer in the set of all the possible vendors/buyers, but for every possible combination of vendors and buyers as individual entities. Therefore, to verify the properties, it is sufficient to instantiate the system with two vendors and

two buyers. In both cases, one encodes every vendor/buyer as an individual entity, and the second encodes the remaining vendors/buyers.

To have *many* invoices, as Property 1 states, we instantiate the maximal number of invoices to three (invoice  $id \in [1..3]$ ): two encode two particular invoices and the third encodes the rest of them. The stamps for invoices in the model are unbounded, only the stamp's stock capacity needs to be bounded to get an instantiation. Since SII gives infinitely often authorisation stamps, the system can work with a minimal stock capacity of 1. However, we choose to set its capacity to 3 (the vendor can get as much as 3 stamps from SII at once) to have the scenario, between others, in which the vendor spends all the *ids* it received from a single request for authorisation stamps.

Since all the invoices can be potentially cancelled, we need at least the same quantity of cancellation *ids* as the quantity of invoices, therefore we instantiate the maximum number of cancellations to three. Following the same reasoning than the stamps for invoices, we also set the capacity of the cancellation stamp stock to 3.

Finally, we instantiate the purchase processes that a vendor can manipulate simultaneously to two: one encoding an individual process and the other encoding all the remaining processes that may be running during the life's cycle of

the system.

Summarising, to verify our 7 properties it is sufficient to instantiate the system with the variables values shown in Figure 4.

Max. Invoices	Max. Cancellations	Invoice stamps stock	Cancellation stamps stock	Purchase processes	Buyers	Vendors
3	3	3	3	2	{Vendor1, Vendor2}	{Buyer1, Buyer2}

Figure 4. Instantiation of data domains

## 6.2. Verification methodology

The checking tools we use allow for checking properties in a very expressive logics: the regular  $\mu$ -calculus [16], and in a number of more classical temporal logics that translate into this one. However, writing properties directly in a temporal logic language is difficult and error-prone, and we prefer, whenever this is possible, to express the properties as automata, written in a variant of our graphical language.

More precisely, reachability properties, expressing scenarios that are desirable or not, are specified by *abstraction automata*, a form of pLTSs with terminal states in which labels are predicates over parameterized actions. This is clearly simpler, for non-specialists, than having different formalisms for models and for properties. Alas this is not enough, and there are properties that cannot be checked this way, typically fairness or inevitability properties. For those we use directly a temporal logics, being either  $\mu$ -calculus or a variant of a higher-level action-based logic, like ACTL [8].

**6.2.1. Reachability properties** The use of *abstraction automata* for expressing and verifying reachability properties was advocated in the framework of the FC2Tools [5]. They are labelled transition systems with logical predicates in their labels, and with acceptance states. Each acceptance state defines one *abstract action*, representing a set of traces (a regular language) from the actions of the model we want to check.

From the original (concrete) system and the abstraction automaton (expressing the property), FC2tools builds a product LTS, whose actions are the labels in the acceptance states of the abstraction automaton encoding the property. If an action is present in the product LTS, then one of the corresponding concrete sequence is possible in the concrete system. The presence of an abstract action in the prod-

uct system naturally proves the satisfiability of the corresponding formula, while its absence proves the negation of this formula.

For instance, Property 2 says: “SII gives the right answers to the invoice status request, i.e.: not present when it has not been sent to SII, present when it has been sent, and cancelled when it has been cancelled by the vendor”.

This is a reachability property since it can be reformulated as *SII does not give wrong answers*, i.e. a scenario that should not be possible.

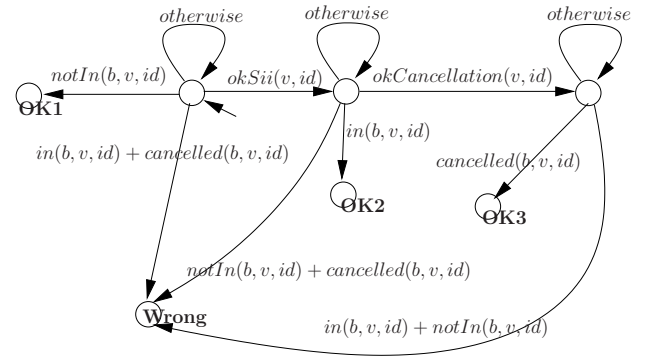


Figure 5. Abstraction automaton encoding Property 2

The abstraction automaton expressing this property is shown in Figure 5. In this automaton, the *otherwise* action means any other action different from the actions in the outgoing edges of the same state. In addition, the automaton not only expresses that the responses are right (otherwise the state **Wrong** is reached) but also that they are possible (states **OK1**, **OK2** and **OK3** are reachable).

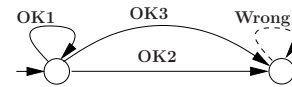


Figure 6. Property 2 verification result

We have used the FC2Tools to check this property: from the instantiated Net of the system, the tools build a global system minimized by weak bisimulation, then build its product with the property automaton, resulting in the LTS in Figure 6. In the LTS, the actions **OK1**, **OK2** and **OK3** are possible from the initial state, which means that the paths from the initial state to those acceptance states in the abstraction automaton (see Figure 5) are possible from the initial state in the instantiated system. Then we have

proved that all the responses from SII to an invoice status request are possible. Likewise, since there are no **Wrong** actions possible in the initial state in the result, we conclude that the path from the initial state to the state labelled as **Wrong** in the abstraction automaton is not possible from the initial state of the instantiated system. The accurate reading of this **Wrong** action in Figure 5 is: a non-desired behaviour can happen if, in the system, we start from a state different than the initial one. Since we want to verify the property in the initial state, we have proved that SII does not give wrong responses to invoice status requests.

**6.2.2.  $\mu$ -calculus formulas** The *abstract automaton* method of the FC2Tools is only usable for reachability properties. For other kinds of formulas, including fairness and inevitability properties, we use the EVALUATOR tool from the CADP tool-set [11]. EVALUATOR performs an on-the-fly verification of properties expressed as temporal logic formulas on a given Labelled Transition System (LTS). The temporal logic it used is called regular alternation-free  $\mu$ -calculus. It allows direct encodings of "pure" branching-time logics including the action-based version of CTL, called ACTL [8]. We express our desired properties in ACTL (it is much easier to write and read than pure  $\mu$ -calculus) and we use EVALUATOR to verify the formula. The result of this verification is a *true* or *false* answer, and a diagnostics.

For instance, Property 3 is an inevitability property since it requests a scenario that must happen in a finite time, in all possible futures, under a condition. We reformulate it in a more precise way :

"If an invoice *id*, emitted by a vendor *v* to a buyer *b* is refused by the buyer it will eventually be cancelled by the vendor "

in which the boxed actions correspond respectively to the *refuseBuyer(v, b, id)* and *sendCancellation(v, id)* actions in the model.

We express this property using the following ACTL formula:

$$AG(\text{refuseBuyer}(\text{Vendor1}, \text{Buyer1}, 1) \Rightarrow AF \text{ sendCancellation}(\text{Vendor1}, 1))$$

To check this property, we have used our instantiation tool to produce a hierarchical Net, instantiated with the values of Figure 4, then the FC2Tools to compute a flat, minimized LTS for this system (in the FC2 format), using the optimisations described in section 7. This system was passed to the EVALUATOR model-checker, together with the formula. The result was positive: this formula holds from the initial state of the system.

The seven properties listed in section 2.2 have been successfully verified using this methodology. Two of them were specified using ACTL formulas. For a complete description see [3].

### 6.3. Improving the model through properties verification

The model we have introduced in Section 5 was not the initial model we designed, but was improved after the verification of the properties listed in section 2.2. Some of the properties were not valid in this initial model and we had to review our formalisation to correct some parts. This verification and review not only improve the model but also the informal requirements defined by SII. For instance Property 5 says "It is not possible to cancel an invoice which has not been emitted before". Though it sounds obvious, we had not included this condition in the initial model since it is not explicitly written in the informal requirements. When verifying Property 2, we got undesired behaviours which exposed this lack in the initial model and so in the informal requirements. In fact, because of this experience, Property 5 was added to the verification list to have a more reliable formalisation. Without a formal verification as described in this paper, a programmer can easily overlook this condition during the implementation phase resulting in an application with potential and difficult to discover errors.

During this reviewing process, the instantiation tool proved very useful as a debugging tool of the system specification. We have done instantiations for small domains of variables to search the reasons why the properties were not valid in the system. Due to the size of the global LTS, those smaller instantiations were much easier to analyse than the complete instantiations.

## 7. Building finite automata

As explained in Section 6, the verification tools we use can only work over finite systems. To use them, we instantiate the processes and networks that form the system and we generate the synchronisation product (global LTS) of those instantiations.

We have developed a tool that automatically generates a finite automaton (LTS) and/or synchronisation network from a *pLTS* or from a *pNet*, given a finite abstract domain for each unbounded variable in the system. In both cases (parameterized and instantiated), the automata and networks are described in the FC2 format [5].

Once the system is instantiated, we avoid the direct generation of the global LTS by brute force, i.e. without any pre-processing before calculating the global synchronisation product. This would lead us directly to the well-known *state explosion* problem.

When verifying properties, usually we do not need to observe all the events in the system. At each synchronisation product, we can hide the actions that are not involved in a specific property and which are not required to synchronise at a upper level. This technique, in conjunction with minimisation, gives promising results as shown in [3].

Additionally, we try to benefit from the parameterized structure of the system: whenever possible, we group communicating processes that share a common parameter. Then we apply hiding and minimisation to the synchronisation product of those groups before instantiating to the common parameter domain. In that way, we avoid the intermediate instantiation of each member of the group by each value in the parameter domain, and hence a potential state explosion. For instance, in Figure 2 is shown the structure of the **Reception** process. It is defined by a pNet that synchronises three processes (**Recp1**, **Recp2** and **Recp3**), each one parameterized by *id*. However, an instantiation of the **Recp1** process to a particular value of *id* will synchronise actions only with an instantiation of the **Recp2** and **Recp3** processes to the same particular value of *id*, and vice-versa. In fact, an instantiation of the **Reception** process is the free interleaving of the three processes for each particular value of *id*. Therefore, for any instantiation we have the following equivalence, where  $P1|P2$  is the synchronisation product of  $P1$  and  $P2$  and  $P^i = \underbrace{P|P|\dots|P}_i P$ :

$$Recp1^{id}|Recp2^{id}|Recp3^{id} = (Recp1|Recp2|Recp3)^{id}$$

Thus we apply hiding and minimisation to  $(Recp1|Recp2|Recp3)$  before instantiating the *id* parameter.

On our machines, a brute force approach was limited to a global LTS with approximately  $5,6 \times 10^5$  states. The combination of the techniques above has enabled us to scale up to analysing a system corresponding to a brute force LTS that would have had around  $1,2 \times 10^{12}$  states (with the parameters values in Table 4). Using these techniques, the global LTS (reduced by weak bisimulation) generated when verifying Property 1, which requires to observe two parameterized actions, contains 400 states and its bigger intermediary structure generated contains only 1,861 states. The complete verification of the properties listed in section 2.2, were done on the global LTS generated using this methodology.

## 8. Related Work

### 8.1. Case study

A similar case study is done by Tronel et al. in [20] for the SCALAGENT deployment protocol. SCALAGENT is a platform for embedded systems, written in Java, to configure,

deploy, and reconfigure distributed software. In [20], they make a full automatic verification for a Ups (*Uninterruptible Power Supply*) management system to large scale sites, deployed in SCALAGENT.

We use a graphical approach to formalise the system from the informal description, and we translate the informal requirements to formal properties to be checked. In [20], they have chosen to make an automatic translator from the XML configuration description of SCALEAGENT to LOTOS [13], and the verification is done by reachability analysis of *ERROR* states, which are included into those XML descriptions.

They also use parameters but they are included in the translation to LOTOS and not directly in the formal models as we do. This does not allow them to benefit from the parameterized structure of the system and then get better minimisations. Like we do, they make finite instantiations to different parameters domains, and they use this instantiation capacity to do debugging and analysis. In that sense, they find the minimal required instantiations to check the properties by empirical analysis.

Finally, even if we use similar theories and methods to check properties, our aims are different. In [20], they have developed a full automatic verification methodology specific to SCALAGENT; their approach is focused on a high level design language rather than the implementation code. Our framework has been designed in order to address any distributed application with asynchronous communications, which includes the verification of implementations against requirements.

### 8.2. Specification languages

There is a large literature about languages to formally describe concurrent and/or distributed systems at different levels of abstraction. We focus on two of them, that could be well suitable to our aims: NTIF [10], PROMELA [12].

NTIF was designed to become an intermediate language for E-Lotos. Similar to us, NTIF defines LTSs where the transitions encode communication events. It supports data exchange in the communications and guarded actions (including conditions on input values) as we do. The main difference with our language is that NTIF is designed to describe sequential processes whereas we do so for asynchronous concurrent communicating processes. For process composition, NTIF relies on other formalisms, even when each particular process could be defined in NTIF. Because of this, NTIF can not profit from modular composition and minimisation by itself. Finally, NTIF does not have a graphical language to describe systems as we do.

PROMELA [12] is a language designed to describe distributed systems. It does not have process hierarchy, so it can not benefit from modular composition and minimisation.

tion. It supports simple type parameters as well as guards in the communications actions, but it does not support parameterized processes (set of processes). Even when its models can be graphically visualised, it does not have a graphical input syntax to describe the systems as we do.

## 9. Conclusion and Perspectives

We have introduced a methodology to formally describe a system and verify its properties, and we have validated our approach through a real system, the Chilean electronic invoices. We claim that this method is suitable to a developer, not necessarily with expertise in formal methods, by following the methodology presented in this case study.

We focus in the behaviour properties. Other analysis such as the data flow or data security requires other specialised methods and tools. The originality of our work can be summarised as follows:

- We have defined a model to describe in a natural manner the behaviour of distributed systems (with parameters) via networks of processes. This model is an extension of previous work done in [1] and [15]. We have introduced as well a graphical syntax to describe those networks.
- Using this graphical syntax, we have shown how to model the Chilean electronic invoices system from its informal specifications.
- We have developed a tool to obtain finite non-parameterized systems from our language given the variable domains. This generation allows us to use off-the-shelf model-checkers.
- We have introduced a methodology to limit as much as possible the *state explosion* problem during this generation.
- Finally, we have shown how to formalise and verify the properties of the system.

Additionally, the instantiation tool provides a debugging analysis (for a small instantiation), and allows to compare different instantiations and search for better minimisations. This debugging capacity provides early detection of errors or backtrack analysis.

Our parameterized models achieve three different roles: they describe in a natural and finite manner infinite systems (when considering unbounded variable domains), they describe a family of systems (when considering various variable domains) and they describe in a compact way large systems (when considering large variable domains).

In the medium term, we plan to integrate our parameterized models with the OPEN/CAESAR facility of the CADP tool set [11] to perform “on-the-fly” model checking. Then, both the instantiations and the synchronisation products will

be generated as needed by the verification tool. For some reachability properties, the generation of the whole state space will possibly be avoided.

Once the specification is validated, we want to use it to check the correctness of implementations. This check requires a refinement pre-order that allows the developer to make some choices amongst the possibilities left by the specification. This is a work we plan to do in a tool that will benefit from the compositional structure of our models. This tool includes the generation of a parameterized model from the source code. We do generate models for systems implemented in ProActive [6], this generation is described in [2].

The final goal of our team is to develop a full set of methods and tools for the description, analysis and verification of distributed systems. These methods and tools should be as much automatic as possible and naturally usable by non-specialists (for instance software engineers). They should include not only a methodology to describe a system and verify the specifications, but also to verify the correctness of implementations.

## References

- [1] A. Arnold. *Finite transition systems. Semantics of communicating systems*. Prentice-Hall, 1994.
- [2] T. Barros, R. Boulifa, and E. Madelaine. Parameterized models for distributed java objects. In *Forte'04 conference*, Madrid, 2004. Springer Verlag.
- [3] T. Barros and E. Madelaine. Formalisation and proofs of the chilean electronic invoices system. Technical Report RR-5217, INRIA, june 2004.
- [4] J. Bergstra, A. Pose, and S. Smolka. *Handbook of Process Algebra*. North-Holland, 2001.
- [5] A. Bouali, A. Ressouche, V. Roy, and R. de Simone. The fc2tools set. In D. Dill, editor, *Computer Aided Verification (CAV'94)*, Stanford, june 1994. Springer-Verlag, LNCS.
- [6] D. Caromel, W. Klauser, and J. Vayssière. Towards seamless computing and metacomputing in Java. *Concurrency Practice and Experience*, 10(11–13):1043–1061, Nov. 1998.
- [7] R. Cleaveland and J. Riely. Testing-based abstractions for value-passing systems. In *International Conference on Concurrency Theory (CONCUR)*, volume 836 of *Lecture Notes in Computer Science*, pages 417–432. Springer, 1994.
- [8] R. De Nicola and F. Vaandrager. Action versus state based logics for transition systems. In I. Guessarian, editor, *Semantics of Systems of Concurrent Processes, LITP Spring School on Theoretical Computer Science*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419, La Roche Posay, France, 1990. Springer.
- [9] Gobierno de chile, servicio de impuestos internos, factura electrónica. <https://palena.sii.cl/cvc/dte/menu.html>.
- [10] H. Garavel and F. Lang. NTIF: A general symbolic model for communicating sequential processes with data. In *Proceedings of FORTE'02 (Houston)*. LNCS 2529, nov 2002.

- [11] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4:13–24, Aug. 2002.
- [12] G. Holzmann. *The SPIN Model Checker, Primer and Reference Manual*. Addison-Wesley, 2003. ISBN 0-321-22862-6.
- [13] ISO: Information Processing Systems - Open Systems Interconnection. Lotos - a formal description technique based on the temporal ordering of observational behaviour. ISO 8807, Aug 1989.
- [14] A. Lakas. *Les Transformations Lotomaton : une contribution à la pré-implémentation des systèmes Lotos*. PhD thesis, Univ. Paris VI, june 1996.
- [15] H. Lin. Symbolic transition graph with assignment. In U. Montanari and V. Sassone, editors, *CONCUR '96*, Pisa, Italy, 26–29 Aug. 1996. LNCS 1119.
- [16] R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. In S. Gnesi, I. Schieferdecker, and A. Rennoch, editors, *Proceedings of the 5th Int. Workshop on Formal Methods for Industrial Critical Systems FMICS'2000 (Berlin, Germany)*, GMD Report 91, pages 65–86, Berlin, Apr. 2000.
- [17] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989. ISBN 0-13-114984-9.
- [18] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100(1), 1992.
- [19] J. Riely. *Applications of Abstraction for Concurrent Programs*. PhD thesis, University of North Carolina at Chapel Hill, 1999.
- [20] F. Tronel, F. Lang, and H. Garavel. Compositional verification using CADP of the ScalAgent deployment protocol for software components. In *6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems FMOODS'2003*, Paris, France, Nov 2003.