# Formalisation and verification of the Chilean electronic invoice system

Tomás Barros, Eric Madelaine

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *Formalisation and verification of the Chilean electronic invoice system*

Tomás Barros — Eric Madelaine

N° 5217

Juin 2004

Thème COM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--5233--FR+ENG

# Formalisation and verification of the Chilean electronic invoice system

Tomás Barros , Eric Madelaine

Thème COM — Systèmes communicants

Projet OASIS

Rapport de recherche n° 5217 — Juin 2004 — 55 pages

**Abstract:** We present a case study describing the formal specification and verification of the Chilean electronic invoice system, which has been defined by the Chilean taxes administration. The system is described by graphical specifications consisting of labelled transition systems, composed using synchronisation networks. Both, transition systems and networks, are parameterized. We use verification tools based on Process Algebra theories to check the requirements on those graphical specifications. We introduce a method and a tool to obtain finite systems from these parameterized ones by fixing the parameters domains, so we can use standard tools for verifying properties in finite systems. We also analyse different methods to avoid the state explosion problem by taking advantage of the parameterized structure and instantiations.

**Key-words:** formal specification, program properties, verification, model-checking, process algebra, components, minimisation

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

# Formalisation et preuves du système de factures électroniques au Chili

**Résumé :** Nous présentons un cas d'étude de spécification formelle et de vérification : le système de facturation électronique défini récemment par l'administration chilienne des taxes. Nous décrivons ce système à l'aide d'une spécification graphique, constituée par des systèmes de transitions étiquettés, composés au moyen de réseaux de synchronisation. Les systèmes de transitions et les réseaux de synchronisation sont paramétrés. Nous utilisons des outils basés sur la théorie des Algèbres de Processus pour vérifier les propriétés de ces spécifications graphiques. Nous définissons une méthode et un outil permettant d'obtenir des systèmes finis à partir de ces systèmes paramétrés, en fixant le domaine des paramètres, de manière à pouvoir utiliser des outils standard de vérification de modèle pour la vérification des propriétés. Nous analysons différentes méthodes pour limiter le phénomène d'explosion de l'espace d'états, en prenant en compte la structure paramétrée du système et des instanciations des paramètres.

**Mots-clés :** spécification formelle, propriétés de programme, modélisation, vérification, vérification de modèle, algèbre de processus, composants, minimisation

# 1 Introduction

This report presents a case study in formal specification and verification. It is 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). This system will progressively replace the current classical invoice system on paper. It 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.

The definition of the system started in December 2000 inside the tax agency (SII) by the creation of a study team. In August 2002 the initial specs were defined and 8 companies (taxpayers) were invited to participate in a first prototype. In January 2003, all these enterprises were emitting fictitious electronic invoices for testing. From April 24th, 2003, the electronic invoices emitted by these enterprises are recognised as valid legal invoices. On September 2nd 2003, the system has been opened to any enterprise who would like to use electronic invoices.

However, the tax 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 a certification process that the new enterprises should accomplish to be incorporated. This certification process includes testing, simulation and checking steps.

Thus, the question arises whether specific implementations of the system parts, eventually developed by different companies, obey their specifications and behave correctly inside the whole system.

In this work, we address this question at a design level, by developing a formal behavioural specification of the invoice system. Then we select a small number of requirements from those published by the SII, and use model-checkers to prove that they are satisfied by our specification.

We are interested here in the safety 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 specifically want to address two questions: how do we specify formally the behaviour of a component of the system, 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 parts, behave correctly.

There exist nowadays a number of software environments, or middlewares, for facilitating the development of applications distributed over networks. These tools can be used in a variety of contexts, ranging from multiprocessors or clusters of machines, via local or wide area networks, to pervasive and mobile computing. Each of these application fields have specific requirements. Therefore methods and tools to specify their behaviours (requirements) and to check these specifications against their implementations are necessary. These methods should be formal enough to be used by the tools, but simple enough to be used by non-specialists. They also should be as automatic as possible, hiding the complexity in their logics and algorithms.

We propose a pragmatic approach based on graphical specifications for communicating and synchronised distributed objects, in which both events (messages) and agents (distributed objects) can be parameterized. Our example gives us a realistic case study; its structure is small enough to be presented in length in this report, but complex enough to show interesting analysis questions. We give a formal abstract specification of the full system, in terms of a hierarchy of communicating transition systems (the vendor subsystem, for example, is itself composed of 11 smaller components, organised in 4 hierarchical levels). Further we show how to compute a global transition system, limiting as much as possible the state space explosion, and prove some correctness properties at the level of the specification.

Our specification framework is based on process algebra theories [Mil89, BPS01], and we use classical software tools, in particular action-based model-checkers [Mad92, ARBR94, GLM02], to automatically prove behavioural properties. In the area of process algebras, there have been numerous developments to integrate value-passing features into the original "pure" calculi; this was indeed strongly required both for theoretical reasons (study of expressiveness of proof systems) and for practical goals (realistic specification formalisms, semantics of languages). We mention the seminal article on value-passing CCS [Hen91], that started the work on symbolic proof systems and bisimulations; the $\mu$CRL process algebra [GP94]; and the specification language LOTOS [ISO98]. Our goals here are somewhat different. We are seeking a compromise between expressiveness of a specification language, intuitiveness of the graphical version of the language (best supported by a notion of communicating and synchronised components), and the possibility to submit the models to automatic analysis through classical (finite-state) model-checkers or equivalence checkers.

Our model, defined in [MBB04], is an adaptation of the *symbolic transition graphs with assignment* of [Lin96] and of the *synchronisation networks* of [Arn94]: we extend the general notion of labelled transition systems (LTS) and hierarchical networks of communicating systems (synchronisation networks) to add parameters to the communication events. This communication events can be guarded with conditions on their parameters. Our agents can also be parameterized to encode sets of equivalent agents running in parallel. The parameters are typed variables of simple enumerable types: booleans, integers, intervals, finite enumerations or structured objects. Our model is suitable both for compositional description of distributed system behaviours and for models resulting from static analysis of source code. Our team is already working on model generation from static analysis of the source code for ProActive [CKV98] applications. ProActive is a Java implementation of distributed active objects with asynchronous communications and replies by means of future references, developed in our team, in the context of the ObjectWeb consortium [Obj]. We have first developed a behavioural semantics for ProActive [BM03], and an algorithm based on the method call graph of the application, that builds a network of finite transition systems, from abstractions of the application source having only finite data domains. Then we have extended this approach in

[MBB04], working now with countable abstractions of the data domains; the new semantics builds parameterized networks of parameterized labelled transition systems.

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.

The main contributions of this work are:

- The definition of parameterized Labelled Transition Systems (pLTS) and Networks (pNets) which enables us to have a finite definition of an infinite system. We also provide a graphical syntax to represent those parameterized systems.
- The development of a tool to get finite instantiations from the *pLTS* and *pNets* by bounding their parameter domains.
- The proposal of methods to avoid the *state explosion problem* by hierarchical composition, hiding and grouping by variables.
- The validity of our approach on a realistic case study (The Chilean electronic invoice system), in terms of a full graphical specification of the system, and the verification of a small set of properties.

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 give the main definitions of our model, taken from [MBB04], and of its graphical language. Section 4 introduces the tools and methodology to verifying properties. Section 5 gives the full specification of the Chilean electronic invoices system in the form of parameterized labelled transition systems and synchronisation networks. In section 6 we explain how to formalise the requirements, how we construct the proofs for those requirements, and the results obtained on our specification. Section 7 explains how to derive finite instantiations and introduces methods to limit as much as possible the *state explosion problem*. Finally section 8 introduce some related work and section 9 discusses conclusions and perspectives of this work.

# 2 Electronic invoices in Chile

In this section, we informally describe the electronic invoice system recently realized in Chile, which official, though informal, specification was published in September 2002. For a detailed explanation, please look at [DTE].

## 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 **SII**). 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.

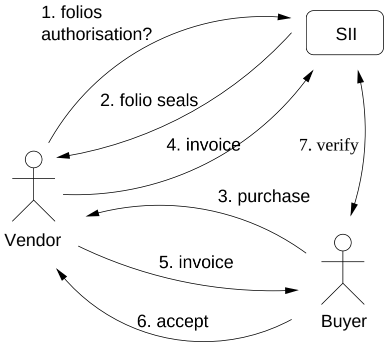

Figure 1: Normal Scenario

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.

## 2.2 System properties

Some of the behavioural properties that the system should respect are listed below. We have extracted those from the informal requirements in [DTE], where they appear either explicitly or implicitly.

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 either accepted by the buyer or cancelled in SII.

These properties are formalised and verified in section 4

# 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 first define finite LTSs and Nets. For a hierarchy of Nets (with LTSs at the leaves), the semantics is given by a synchronisation product, that allows to compute the global LTS for the system. Then we give the definition of the parameterized LTSs and Nets. Those have no direct synchronisation product : their semantics is obtained by instantiations to finite structures.

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 [Mil89]. 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* $s \overset{\alpha}{\rightarrow} s'$ *for* $(s, \alpha, s') \in \ \rightarrow$.

Then we define **Nets** in a form inspired by [Arn94], 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 (operators, or transformers, on transition systems), in a sense similar to the open Lotos expressions of [Lak96]. 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* $< A_G, I, T >$ *where* $A_G$ *is a set of global actions,* $I$ *is a finite set of Sorts* $I = \{I_i\}_{i=1,\ldots,n}$*, and* $T$ *(the transducer) is a LTS* $T = (T_T, s_{0_t}, L_T, \rightarrow_T)$*, such that* $\forall \overrightarrow{v} \in L_T, \overrightarrow{v} =< l_t, \alpha_1, \ldots, \alpha_n >$ *where* $l_t \in A_G$ *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* $\{LTS_i = (S_i,\ s_{0_i},\ L_i, \rightarrow_i)\}_{i=1\ldots n}$ *and a Net* $< A_G, \{I_i\}_{i=1\ldots n}, (S_T, s_{0_T}, L_T, \rightarrow_T) >$*, such that* $\forall i \in [1,n], L_i \subseteq I_i$*, we construct the product LTS* $(S,\ s_0,\ L,\ \rightarrow)$ *where* $S = S_T \times \bigtimes_{i=1}^{n}(S_i)$*,* $s_0 = s_{0_T} \times \bigtimes_{i=1}^{n}(s_{0_i})$*,* $L = A_G$*, and the transition relation is defined as:*

$$\rightarrow \triangleq \{s \xrightarrow{l_t} s' \mid s =< s_t, s_1, \ldots, s_n >, s' =< s'_t, s'_1, \ldots, s'_n >,$$

$$\exists\ s_t \xrightarrow{\overrightarrow{v}} s'_t \in \rightarrow_T, \overrightarrow{v} =< l_t, \alpha_1, \ldots, \alpha_n >,\ \forall i \in [1,n],\ (\alpha_i \neq idle \wedge s_i \xrightarrow{\alpha_i} s'_i \in \rightarrow_i) \vee (\alpha_i = idle \wedge s_i = s'_i)$$

Note that the result of the product is a LTS, which in turn can be synchronised with other LTSs in a Net. This property enables us to have different levels of synchronisations, i.e. a hierarchical definition for a system.

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) [Lin96].

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 [CR94]), 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,* $\mathcal{P}\nabla\wr\}$ *encoding an observable local sequential program (with assignment of variables),* $?P.m(\overline{x})$ *encoding the reception of a call to the method m from the process P (*$\overline{x}$ *will be affected by the arguments of the call) and* $!P.m(\overline{e})$ *encoding a call to the method m of a remote process P with arguments* $\overline{e}$*.*

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* $\overrightarrow{v_s}$*,*

$s_0 \in S$ *is the initial state,*

$L = (b, \alpha(\overrightarrow{x}), \overrightarrow{e})$ *is the set of labels (parameterized actions), where b is a boolean expression,* $\alpha(\overrightarrow{x})$ *is a parameterized action, and* $\overrightarrow{e}$ *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* $< pA_G, H, T >$ *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* $\forall \overrightarrow{v} \in L_T, \overrightarrow{v} =< l_t, \alpha_1^{k_1}, \ldots, \alpha_n^{k_n} >$ *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 $\overrightarrow{v} =< l_t, \alpha_1^{k_1}, \ldots, \alpha_n^{k_n} >$, each $\alpha_i^{k_i}$ corresponds to the $\alpha_i$ action of the $k_i$-nth corresponding argument LTS.

In the framework of this report, 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 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 [Mil89, MPW92].

## 3.1 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 [BRRdS94], 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 ($\overrightarrow{v_s}$) and the edges are labelled by $[b]\ \alpha(\overrightarrow{x}) \rightarrow \overrightarrow{e}$ (see Definition 6).

An *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 a finite number of *ports* on its border, represented as labelled bullets, each one encoding a particular parameterized action of the Sort.

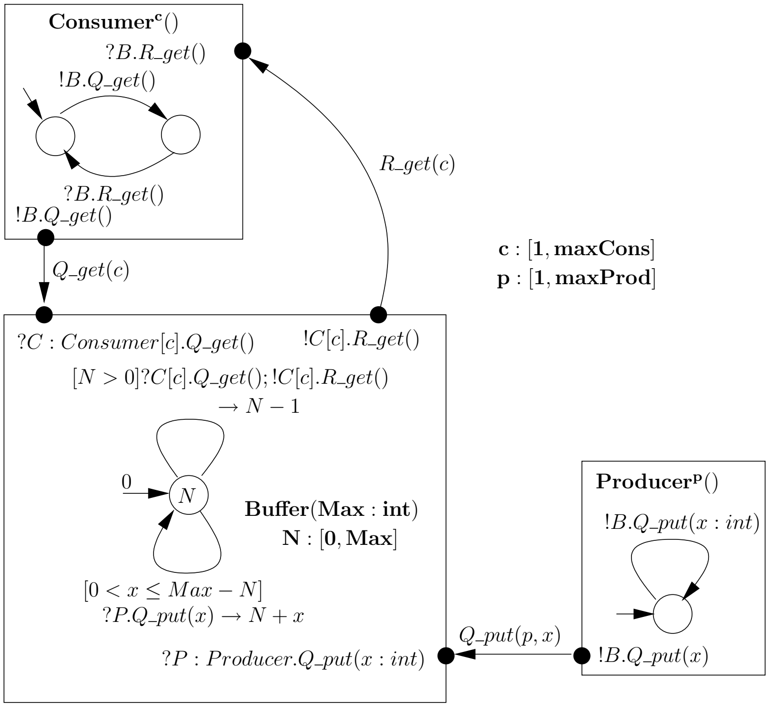
$\mathbf{Consumer^c}()$
$?B.R\_get()$
$!B.Q\_get()$
$?B.R\_get()$
$!B.Q\_get()$
$R\_get(c)$
$Q\_get(c)$
$\mathbf{c} : [\mathbf{1}, \mathbf{maxCons}]$
$\mathbf{p} : [\mathbf{1}, \mathbf{maxProd}]$
$?C : Consumer[c].Q\_get()$
$!C[c].R\_get()$
$[N > 0]?C[c].Q\_get(); !C[c].R\_get() \rightarrow N - 1$
$0$
$N$
$\mathbf{Buffer(Max : int)}$
$\mathbf{N} : [\mathbf{0}, \mathbf{Max}]$
$[0 < x \leq Max - N]$
$?P.Q\_put(x) \rightarrow N + x$
$?P : Producer.Q\_put(x : int)$
$Q\_put(p, x)$
$\mathbf{Producer^p}()$
$!B.Q\_put(x : int)$
$!B.Q\_put(x)$

Figure 2: Parameterized consumer-producer system

Figure 2 shows an example of such a parameterized system. It is composed of a single buffer and a bounded quantity of consumers (*maxCons*) and producers (*maxProd*). Each producer feeds the buffer with a quantity ($x$) of elements at once. Each consumer requests a single element from the buffer (!*B.Q_get*()) and waits for the response (?*B.R_get*()).

Figure 2 also introduces the notation to encode sets of processes; for example, $\mathbf{Consumer^c}()$ encodes the set of **Consumer**() processes for each value in the domain of $c$. Therefore, each element in the domain of $c$ is related (identifies) to an individual process of the set. Each process knows its own identity.

The edges between ports in Figure 2 are called links. Links express synchronisation between internal boxes or to external processes. They also can be between ports of different instantiation of the same box. Each link encodes a transition in the Transducer LTS of the *pNet*.

When the initial state is parameterized with an expression, it can be indicated which evaluation of the expression (for which value of the variables) is to be considered as the initial state. In Figure 2 the initial state is defined as the state where $N = 0$.

The various elements of the graphical language described here are naturally translated into pLTSs and pNets. A *drawing* in our language may contain an arbitrary composition of pNets and pLTSs. A single pNet would have an outside box, its ports representing the global actions, and containing as one box inside for each hole in the pNet, with inner ports defining the sort of each hole. Each link encodes a synchronisation vector. All pNets drawn in this report are *static*: their transducers have only one state. If we had to represent dynamic pNets, we would have to add the transducer LTS in the drawing of the Net.

An instantiation of the system described in Figure 2 is shown in Figure 3 for better understanding. This instantiation is done when considering 2 consumers, 2 producers and a buffer capacity of 3.

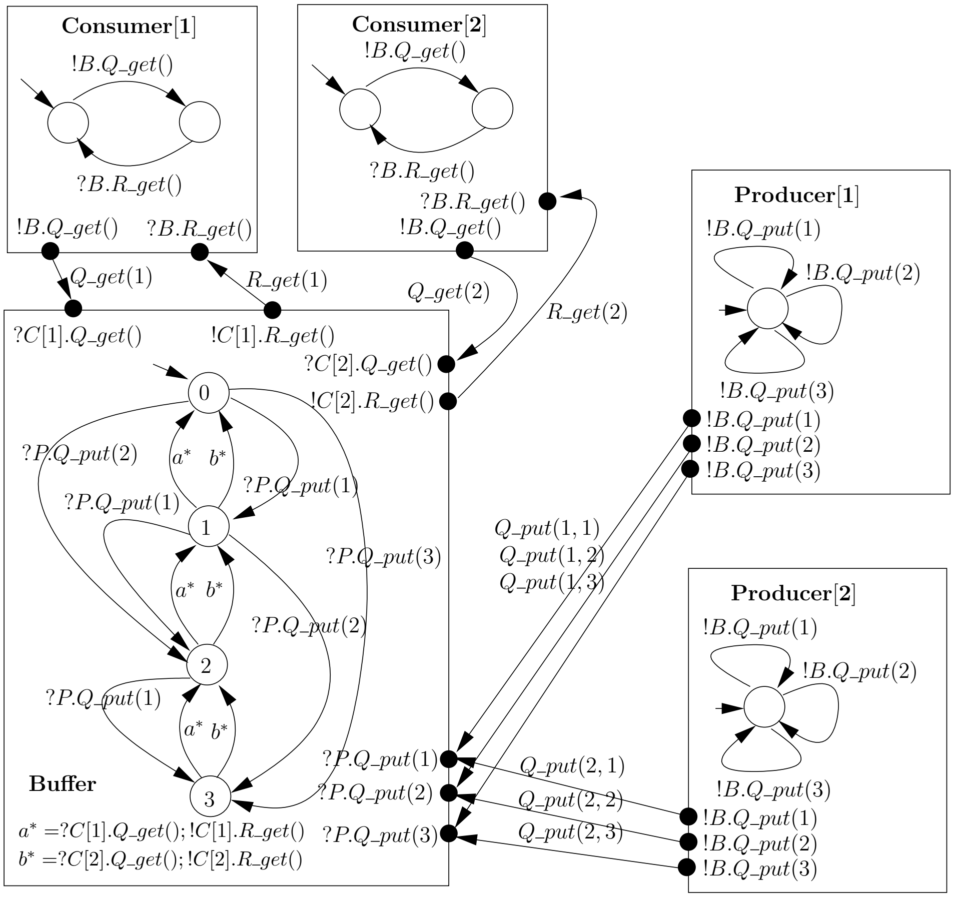

Figure 3: Parameterized consumer-producer system

# 4 Verification methodology

The checking tools we use allow for checking properties in a very expressive logics: the regular $\mu$-calculus [MS00], 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 [DNV].

Let us give two simple properties to demonstrate those two approaches, that should hold on the system introduced in section 3.1 (Figure 2):

1. No consumer can get any element from the buffer before it is fed (reachability property).
2. Once a consumer has requested an element to the buffer, it will eventually obtain it (inevitability property).

### 4.0.1 Reachability properties

The use of *abstraction automata* for expressing and verifying reachability properties was advocated in the framework of the FC2Tools [BRRdS94]. 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 product system naturally proves the satisfiability of the corresponding formula, while its absence proves the negation of this formula.

Property 1 is a (negative) reachability property since it describes a non desirable scenario.

This property is expressed as the abstraction automaton in Figure 4, after instantiated with two consumers ({c1,c2}), two producers ({p1,p2}) and a buffer capacity of 3. In Figure 4 we use the $\bigvee$ operator just as a shortcut to expresses or-exclusive actions. The *otherwise* action means any other action different from the actions in the outgoing edges of the same state. In addition, Figure 4 express the property that once the buffer fed, a consumer will be able to get an element (**OK** state).

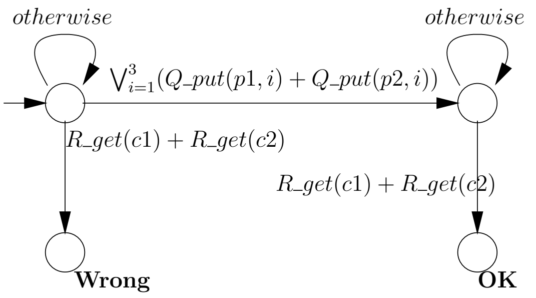

Figure 4: Property: can not get elements from the buffer before feeding it

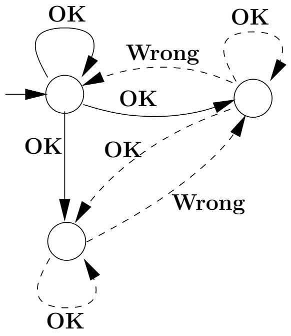

Figure 5: Property verification result

In Figure 5 is shown the LTS (minimised by weak bisimulation) resulting from the verification of Property 1 in the instantiated (finite) system. In the LTS, the action **OK** is possible from the initial state, which means that the paths from the initial state to the **OK** acceptance state in the abstraction automaton (see Figure 19) are possible from the initial state in the instantiated system. Then we have proved that is it possible to get an element from the buffer once the buffer has been fed. On the contrary, 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 the **Wrong** actions occurring in Figure 19 is: a non-desired behaviour can happen if, in the system, we start from a state different that the initial one. Since we want to verify the property in the initial state, we have proved that is not possible to get an element from the buffer if it has not been fed before.

#### 4.0.2 $\mu$-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 [GLM02]. 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. Evaluator also includes preprocessors that translate formulas from various temporal logic languages into regular alternation-free $\mu$-calculus, including the action-based version of CTL, called ACTL [DNV]. We express our desired properties in ACTL and we use EVALUATOR to verify the formula. The result of this verification is a *true* or *false* answer, and a diagnostics.

Property 2 is an inevitability property since it requests a scenario that must happen in finite time, in all possible futures, under a condition. We reformulate it in a more precise way, first in english :

*If a consumer requests an element from the buffer, it will eventually obtain it.*

We express this property using the following ACTL formula:

$$AG(Q\_get(c1) \Rightarrow AF\ R\_get(c1))$$

This formula was successfully proved to be true in the consumer-producer system.

# 5 Formalisation

We have used this graphical language to build pLTSs and pNets for the formal specification of the Chilean invoices system. The goal in this work 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.
- We simplify the data part of the system. In our abstraction, 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*).

## 5.1 The Vendor system

Figure 6 shows the network that defines the (parameterized) 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 from 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 (Base Vendor) 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 3.1 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. Note that in the action !*SII.sendCancellation*(*inv*), the process identifier of **CI** (*inv*) becomes a value-passing variable for the external observer.

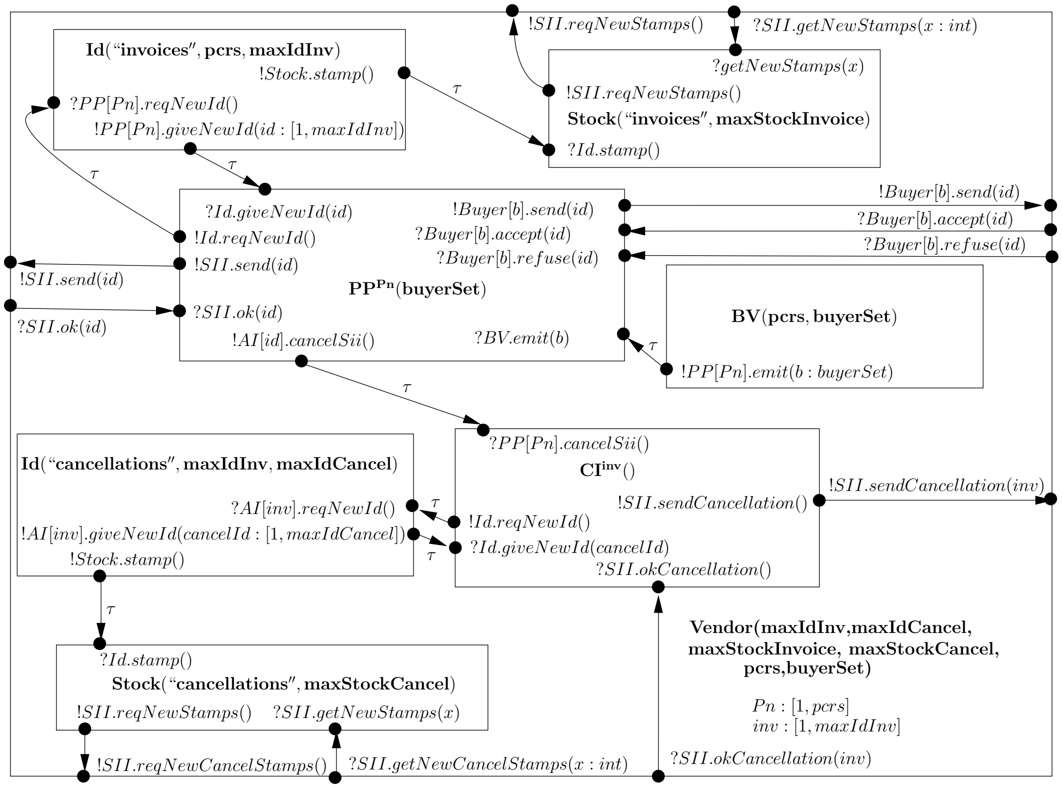

Figure 6: The Vendor system

### 5.1.1 Stamp stock

SII specifies that before emitting an invoice, a vendor should have (at least) one stamp available. The Stamp stock component, named **Stock** manipulates a stock of stamps for this purpose. It also requests new stamps from SII and it manages replies to these requests. The **Stock** behaviour is described by the automaton in Figure 7.

The stamps provided by SII for invoices are different from those for cancellations. Therefore, they are managed by separated **Stock** processes as we have seen in the vendor synchronisation network (Figure 6).

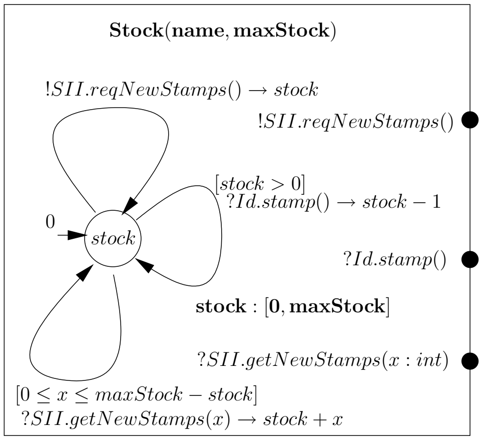

Figure 7: Stock process

Any process that requires a stamp should request it from the **Stock** process, using the ?*Id.stamp* call. The **Stock** process will serve this request only if the number of stamps in stock is more than zero. In the automaton, the state has a *stock* variable coding the number of stamps available. When a stamp request is served, the stock variable is decreased.

The request for new stamps to SII can be done at any moment and its response is asynchronous. The automaton must be ready to receive the response at any moment. In other words, the request for new stamps is a non-blocking request. Once the response arrives, a transition to the state with a stamp stock equal to the current stock plus the newly arrived stamps ($x$ in ?*SII.getNewStamps*($x$)) is done. The constraint in the reception of new stamps ($0 \leq x \leq maxStock - stock$) avoids to have a response that can exceed the capacity of the stock.

### 5.1.2 Id provider

According to the published specifications from SII, a unique and correlative number, named *folio*, should be assigned to the new documents. Once assigned, a *folio* can not be reused at all. The behaviour of the component in charge of this task, named **Id**, is shown in Figure 8. The *id* variable in the figure represents our abstraction of the document's *folio*.

The process **Id** provides an interface to processes requesting a unique id (?*P*[*i*].*reqNewId*()) which is restricted to serve up to *pcrs* processes ($i : [1, pcrs]$). Notice that the variable *pcrs* is local in this process. In the vendor network (Figure 6) it takes the value *pcrs* to the **Id** process for invoices, and the value *maxIdInv* to the **Id** process for cancellations.

The response to a request for ids is the action *x*!.*giveNewId*(*id*). As described in [DTE], for every digital document, a "digital seal" should be generated using a stamp previously given by SII. We do not specify the "digital seal" generation process, but we do express the fact that for every new invoice a stamp is consumed. This is done by synchronising the !*Stock.stamp*() action with the **Stock** process before giving the response.

The **Id** process, in contrast to the **Stock** process, can handle several request simultaneously. It may also receive requests when there are no available stamps (could not synchronise !*Stock.stamp*()

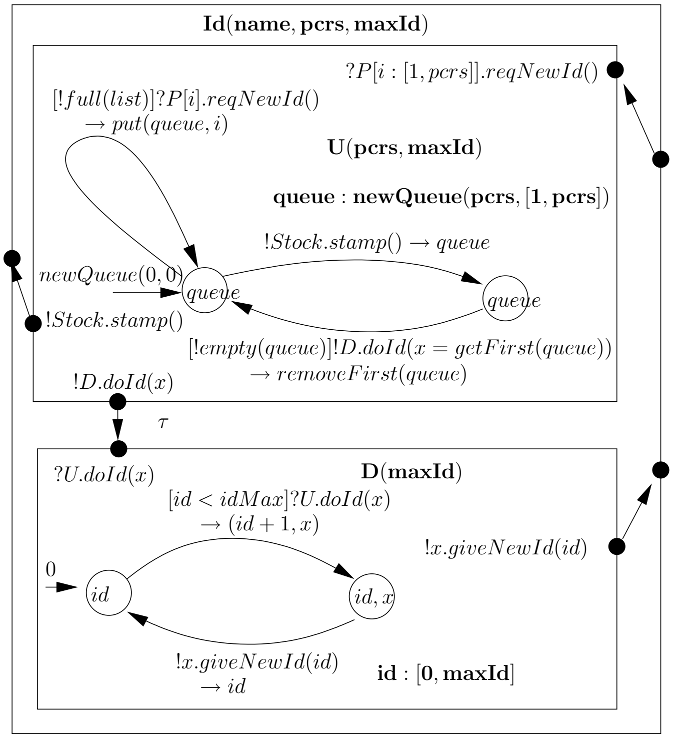

Figure 8: Id process

with the **Stock** process). In this case, the responses will be delayed until stamps become available, and then will be served in the same order as the requests came in (FIFO).

In Figure 8, in the upper automaton **U**, the states are labelled by the variable *queue* which represents the possibles states of a FIFO queue (the state of a queue is defined by its contents and its order). We have introduced six operators for queue manipulation:

- *newQueue*(*x* : *int*, *s* : *finite set*): generates all the possible states for a queue of size *x* that accepts elements from the set *s*.
- *full*(*q* : *queue*): checks whether the queue *q* has reached its capacity.
- *empty*(*q* : *queue*): checks whether the queue *q* contains no elements.
- *put*(*q* : *queue*, *e* ∈ *s*): generates the state representing the current queue *q* with the element *e* added at the end.

- *getFirst*($q$ : *queue*): returns the first element in the queue $q$
- *removeFirst*($q$ : *queue*): generates the state representing the current queue $q$ with its first element removed.

The initial state is when the queue is empty (*newQueue*$(0, 0)$). As the automaton can handle requests from several processes, the identification of the calling process is assigned to the variable $i$. When a new id is requested, a transition to the state identified by the current queue, with $i$ added at the end, is made. As soon as a stamp becomes available, the automaton sends the message !*D*.*doId*($x$) ($x$ is the first element in the queue) to the lower automaton and a transition to the state identified by the current queue with the first element removed is made.

The lower automaton is a counter: every time the message ?*U*.*doId*($x$) is received, a transition to the state identified by the next id value and the process identification ($x$) is made. Finally the response to the process requesting an id is made by the transition labelled !$x$.*giveNewId*(*id*), where *id* is the new id value (the assigned id). This transition goes to the state equivalent to the initial state but representing the last assigned id value.

Actually, under the assumption that the processes that request a new id will never make a second request before getting a response to the first one, the queue length is equivalent to the number of processes ($[1, pcrs]$ for invoices and $[1, maxIdInv]$ for cancellations, see Figure 6).

### 5.1.3 Purchase lifetime

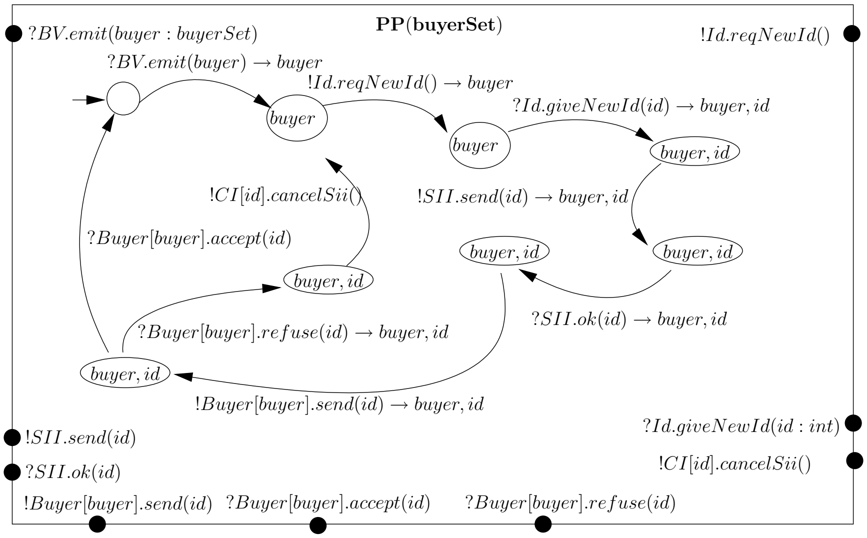

Figure 9: Purchase lifetime process

The purchase process behaviour is described by the automaton in Figure 9. This process (**PP**) takes care of the main actions during the cycle of a purchase. It is started by a request to emit a new invoice to a specific buyer (*?BV.emit*(*buyer*)). Once started, it asks for a new id number (*folio*) to assign to the invoice. Then the invoice is sent to SII and only once it has been received by SII, it is sent to the buyer as described by the specifications [DTE]. If the buyer refuses the invoice, this process activates the cancellation process for the id number and the cycle is restarted (requests new id, sends invoice to SII and then to the buyer). If the buyer accepts the invoice, the process becomes ready to process a new purchase.

### 5.1.4 Firing purchases

Figure 10 shows the automaton (**BV**) in charge of starting new purchase processes. A purchase process is started when a commercial transaction, which requests the emission of an invoice, is done. In an implementation, this action is fired by an interaction with a user or an external system. The automaton reflects this through a transition (!*PP*[*p*].*emit*(*buyer*)) to the state labelled by the next available purchase process, or to the initial state when the last one has been reached.

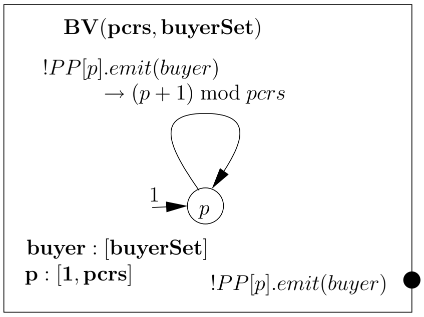

Figure 10: Base vendor process

### 5.1.5 Cancellation of invoices

According to SII, every emitted invoice can potentially be cancelled. For each possible emitted invoice id, there is a process in charge of the cancellation task (**CI** process) shown in Figure 11. In an implementation, this process will probably be created dynamically. We express this by creating statically all the possible processes which may possibly be fired in the life time of the system (through an initial action that synchronises with its context wherever it is the case). Once started, the **AI** process requests an id for the cancellation document and sends it to SII.

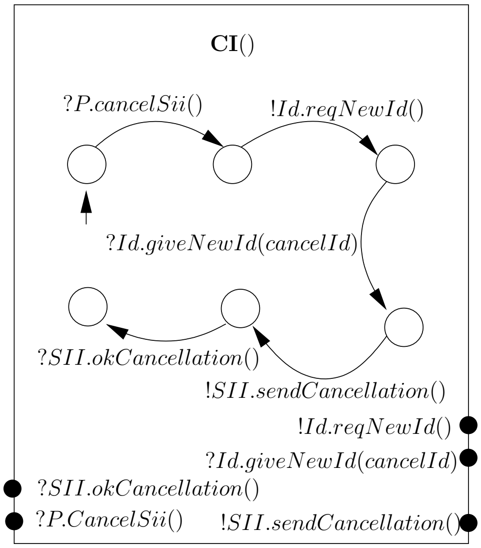

Figure 11: Cancel of an invoice process

## 5.2 The SII system

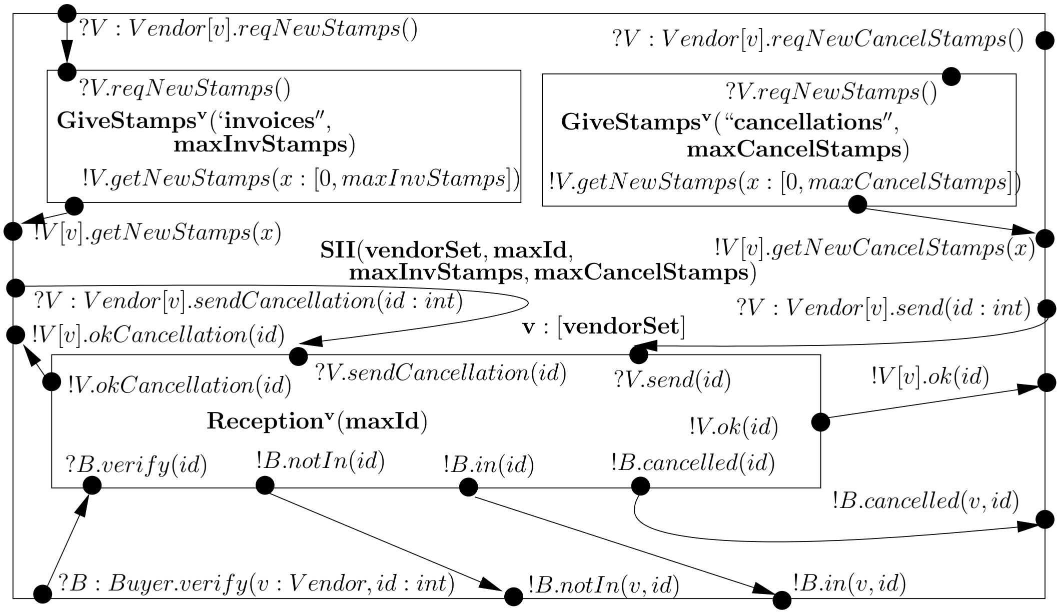

Figure 12: The SII system

Figure 12 shows the network that defines the behaviour of SII. It has two processes **GiveStamps** to provide stamps when requested: one for invoice stamps and one for cancellation stamps. The third process, **Reception**, is in charge of receiving the invoices and the possible cancellations. It also returns the status of an invoice whenever some other process asks for it. Notice that the three processes that form the SII network are parameterized by the variable *v*, which encodes the vendor set of the system. When instantiated, each of the processes will be instantiated once for each vendor. There are no synchronisations between processes concerning different vendors.

### 5.2.1 Documents reception and status checking

According to the specifications, SII provides on-line services to answer new stamps requests, to receive documents (invoices and cancellations), and to verify the status of an invoice. The behaviour of the SII process providing the last two services (**Reception**) is defined by the synchronisation network shown in Figure 13. It is composed of three automata sets whose elements are related to one specific document id (*id*). The top right automaton (**Recp2**) takes care of receiving an invoice,

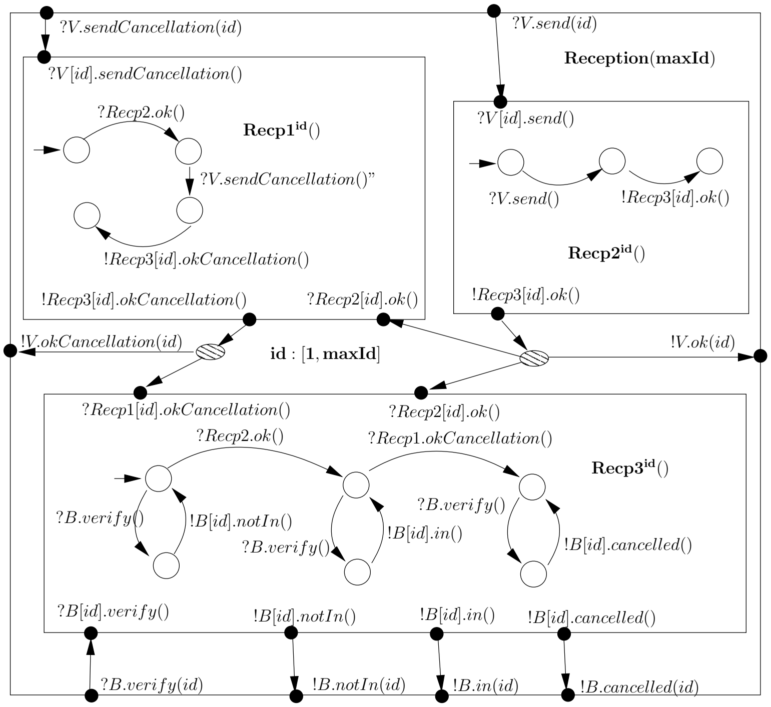

Figure 13: The reception and verification process

the top left automaton (**Recp1**) takes care of receiving a cancellation document (when relevant) and the bottom automaton (**Recp3**) takes care of returning the status for an invoice.

The responses to an invoice status request for a given id (*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*)).

Figure 13 introduces the syntax to indicate a synchronisation between more than two actions. A multiple synchronisation is represented by an ellipse with multiple arriving and/or outgoing edges from/to the ports of the processes which actions must be done simultaneously. All three processes 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).

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

### 5.2.2 Stamps provider

The process in charge of responding to requests for new stamps (**GiveStamps**) is shown at Figure 14. Its structure is simple: every time new stamps are requested, it answers with a certain number of them. The number of stamps to be given is non deterministic (encoded by *x* in Figure 14) and ranges between zero and some upper bound.

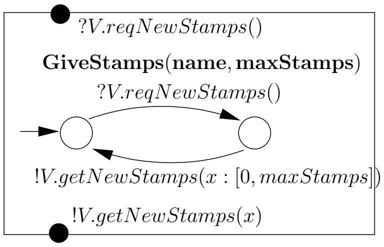
$?V.reqNewStamps()$
$\mathbf{GiveStamps(name, maxStamps)}$
$?V.reqNewStamps()$
$!V.getNewStamps(x : [0, maxStamps])$
$!V.getNewStamps(x)$

Figure 14: The give stamps process

## 5.3 The Buyer system

The buyer automaton is shown in Figure 15. The buyer consists of one internal automaton set whose elements are related with specific vendors and invoices ids (*v*, *id*).

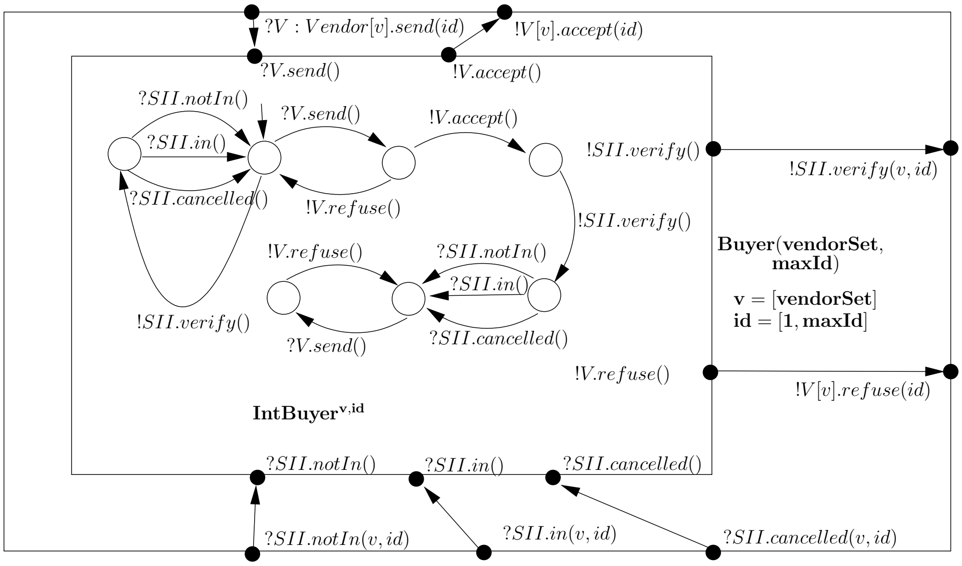

Figure 15: The Buyer system

The internal automaton in the figure represents the expected behaviour of a buyer (for an specific vendor and invoice id). Upon reception of an invoice (?*V*[*v*].*send*(*id*)), the buyer will accept (!*V*[*v*].*accept*(*id*)) or refuse it (!*V*[*v*].*refuse*(*id*)). Some reasons to refuse an invoice are: the buyer is not the correct addressee, the commercial transaction has never been realized or the buyer has already received an invoice with the same id from the same vendor. We do not express the reasons to refuse (except in the latter case), but only the fact that the invoice has been refused. If refused, the buyer automaton returns to the initial state (ready to receive another invoice with the same id from the vendor). If accepted, the automaton proceeds to check the status of the invoice with SII. Upon the answer to the status request, the automaton makes a transition to a state where any new reception of the same invoice id and vendor will always be refused.

## 5.4 The Global System

The global behaviour of the electronic invoices system is obtained by integrating the vendors, the buyers and the SII as shown in Figure 16.

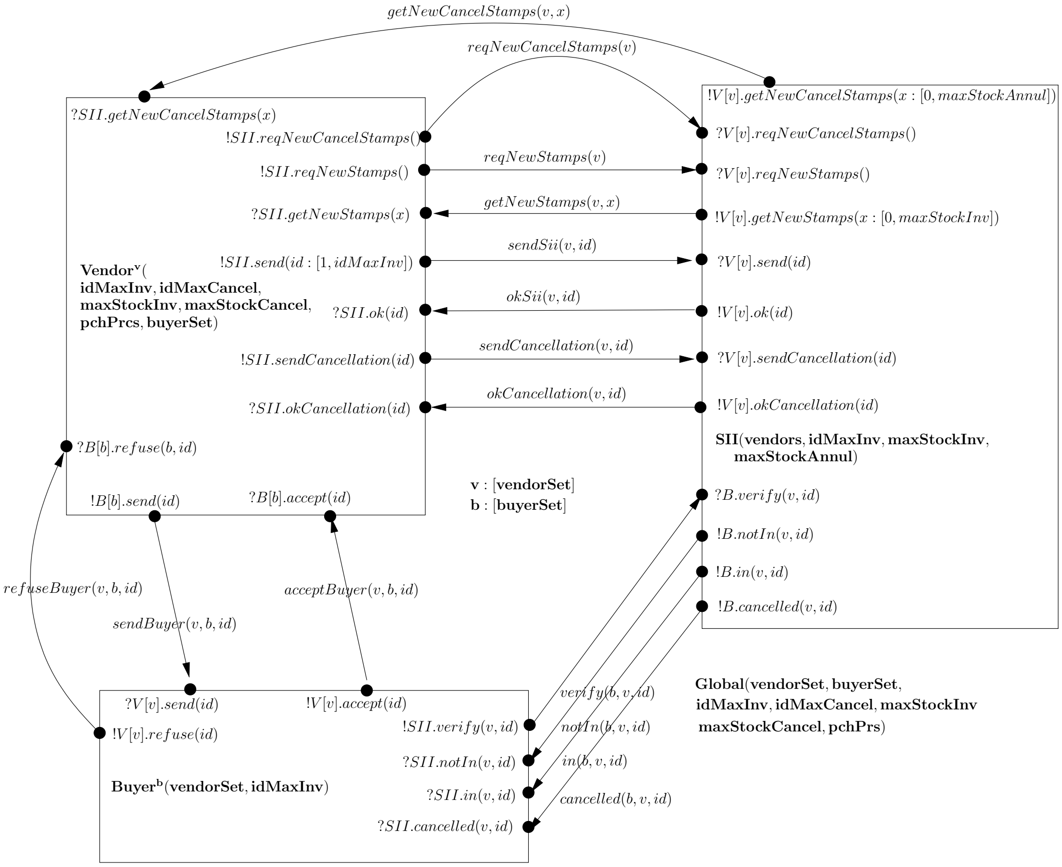

Figure 16: The Global system

There is an arbitrary number of vendors, buyers and a single SII. The synchronisation links are labelled in order to make their contents visible. Those links reflect the possible communications between the processes such as: requesting new stamps (*reqNewStamps*(*v*)), sending an invoice to SII (*sendSii*(*v*, *id*)), refusing an invoice by a Buyer (*refuseBuyer*(*v*, *b*, *id*)), getting the status of an invoice (*in*(*b*, *v*, *id*), *notIn*(*b*, *v*, *id*) or *cancelled*(*b*, *v*, *id*)).

In order to generate instantiations from the model, we have to define the value of the following variables:

- the maximal number of invoices id ($idMaxInv$)
- the maximal number of cancellations id ($idMaxCancel$)
- the capacity of the stamps stock for invoices ($maxStockInv$)
- the capacity of the stamps stock for cancellations ($maxStockCancel$)
- the maximal number of purchase processes for a vendor ($pchPrcs$)
- the set of vendors ($vendorSet$)
- the set of buyers ($buyerSet$)

# 6 Running proofs

In section 2.2 we have presented several examples of properties of interest for this system, they are:

1. A taxpayer could not 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 emitter.
4. An invoice id (folio), can be used only once. There will not be two invoices of the same type and taxpayer whose ids are the same.
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 ends to be accepted by the buyer, or to be cancelled in SII.

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. Rather than generating directly the global LTS, we benefit from the compositional structure of the system. We shall go deeply on this subject in section 7.

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

A finite instantiation of a parameterized model is an abstraction in the sense of [CR94]. Such an abstraction will preserve a given formula if it has enough abstract values in the (finite) abstract domain of each parameter in the formula, specifically one for each distinguished value of the parameter in the formula, plus an extra value representing the rest of the concrete domain.

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. With a minimal stock capacity of 1 and providing SII gives infinitely often authorisation stamps, the system can work. 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.

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 the individual process we are interested in 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 we instantiate the system with the variables values shown in Table 6.1:

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

Table 1: Instantiation of data domains

## 6.2 Verifying properties

For each property we show its formalisation and verification result for the instantiation to the variable domains described above. Since we have defined in the variable domains a representative, as an individual element, for each possible value of the variable; a property valid for this representative is valid for all the possible values of the variable.

Reachability properties are expressed as abstraction automata (see section 4.0.1) and the others as ACTL formulas (see section 4.0.2).

### 6.2.1 A taxpayer could not 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

Figure 17 shows the abstraction automaton that encodes this property. The automaton expresses both conditions in the statement of the property: *The taxpayer (vendor) can not emit invoices if it has not received stamps* (a safety property) and *once it has received stamps, the taxpayer can emit as many invoices as the quantity of stamps received*.

The state labelled **OK1** is reached when the vendor has received one stamp and emitted one invoice. It is analogous for the states labelled **OK2** and **OK3** for two and three received stamps and emitted invoices respectively. The state labelled **Wrong** is reached when the vendor has emitted more invoices than the number of stamps it possesses (a non-desired behaviour). The *otherwise* action in Figure 17 allows any other action different from the actions in the outgoing edges of the same state.

As explained in section 4, from the original system and the abstraction automaton, FC2tools builds an abstract (product) LTS, whose actions are the labels of the accepting states of the abstraction automaton encoding the property.

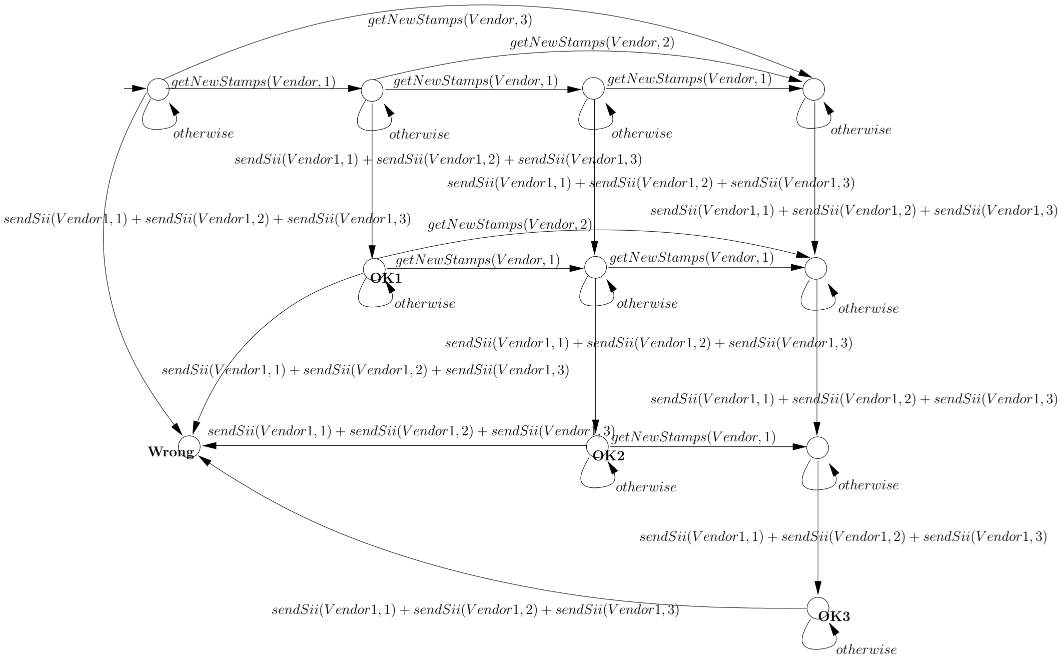
getNewStamps(Vendor, 3)
getNewStamps(Vendor, 2)
getNewStamps(Vendor, 1)
getNewStamps(Vendor, 1)
getNewStamps(Vendor, 1)
otherwise
otherwise
otherwise
otherwise
sendSii(Vendor1, 1) + sendSii(Vendor1, 2) + sendSii(Vendor1, 3)
sendSii(Vendor1, 1) + sendSii(Vendor1, 2) + sendSii(Vendor1, 3)
sendSii(Vendor1, 1) + sendSii(Vendor1, 2) + sendSii(Vendor1, 3)
sendSii(Vendor1, 1) + sendSii(Vendor1, 2) + sendSii(Vendor1, 3)
getNewStamps(Vendor, 2)
getNewStamps(Vendor, 1)
getNewStamps(Vendor, 1)
OK1
otherwise
otherwise
otherwise
sendSii(Vendor1, 1) + sendSii(Vendor1, 2) + sendSii(Vendor1, 3)
sendSii(Vendor1, 1) + sendSii(Vendor1, 2) + sendSii(Vendor1, 3)
sendSii(Vendor1, 1) + sendSii(Vendor1, 2) + sendSii(Vendor1, 3)
sendSii(Vendor1, 1) + sendSii(Vendor1, 2) + sendSii(Vendor1, 3)
getNewStamps(Vendor, 1)
Wrong
OK2
otherwise
otherwise
sendSii(Vendor1, 1) + sendSii(Vendor1, 2) + sendSii(Vendor1, 3)
sendSii(Vendor1, 1) + sendSii(Vendor1, 2) + sendSii(Vendor1, 3)
OK3
otherwise

Figure 17: Abstraction automaton encoding Property 1

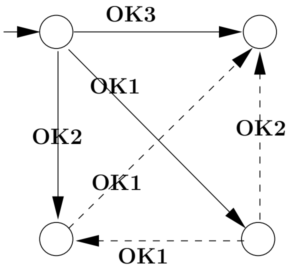

Figure 18: Verification Result of Property 1

Figure 18 shows the abstract automaton (reduced by weak bisimulation) resulting from verifying the property. In the automaton the actions **OK1**, **OK2** and **OK3** are reachable from the initial state, which means that the states labelled accordingly in the abstraction automaton (Figure 17) are reachable from the initial state in the concrete system. Since the resulting abstract automaton does not have any action labelled **Wrong**, we conclude that the state labelled **Wrong** in the abstraction automaton is not reachable. Thus, the property holds in the system.

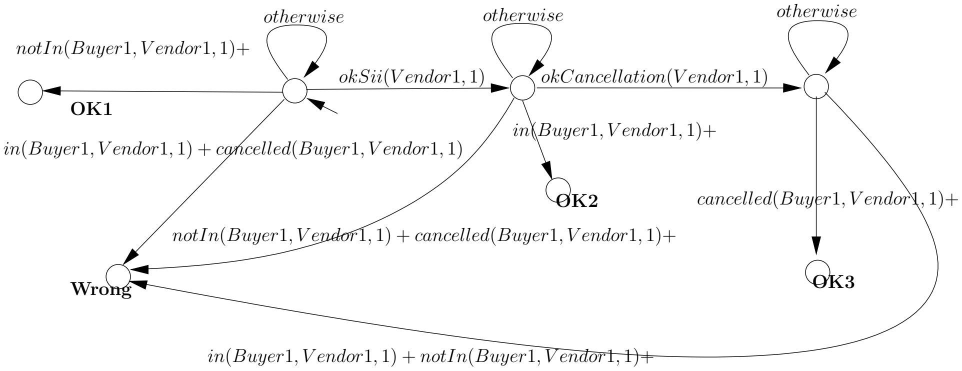
otherwise
otherwise
otherwise
notIn(Buyer1, Vendor1, 1)+
okSii(Vendor1, 1)
okCancellation(Vendor1, 1)
OK1
in(Buyer1, Vendor1, 1)+
in(Buyer1, Vendor1, 1) + cancelled(Buyer1, Vendor1, 1)
OK2
cancelled(Buyer1, Vendor1, 1)+
notIn(Buyer1, Vendor1, 1) + cancelled(Buyer1, Vendor1, 1)+
Wrong
OK3
in(Buyer1, Vendor1, 1) + notIn(Buyer1, Vendor1, 1)+

Figure 19: Abstraction automaton encoding Property 2

### 6.2.2 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

The abstraction automaton encoding this property is shown in Figure 19. The automaton not only expresses that the responses are right (otherwise the state **Wrong** is reached) but also that they are possibles (the states **OK1**, **OK2** and **OK3** are reachable).

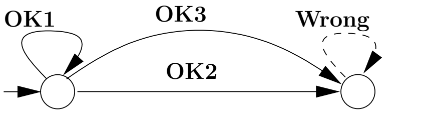

Figure 20: Property 2 verification result

Figure 20 shows the abstract automaton (reduced by weak bisimulation) resulting from verifying the property using the tools.

We observe in Figure 20 a transition labelled as **Wrong**. However, notice that this **Wrong** action is not leaving from the initial state.

The property we are checking here, as well as all the other properties listed in this section, express conditions that should be valid in the initial state of the system (but are not invariants, valid in all states). Following the reasoning introduced in section 4.0.1, we conclude that the property holds.

### 6.2.3 Every invoice refused by a buyer must be cancelled by the emitter

This is an inevitability property, i.e. it not only specifies something that can or can not happen, but something that must happen in a finite time.

The property requests an obligation under a certain condition, it can be reformulated as: *if an invoice is refused by the buyer (the receptor), the invoice must be cancelled by the vendor (the emitter)*

As we said before, we can not express this kind of property using an abstraction automaton. In these cases we use temporal logic formulas to express the property, and we use the model checker Evaluator [GLM02] to verify their satisfiability.

This property is expressed using action-based ACTL by Formulae 1

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

Formulae 1 was successfully proved to be true in the system. Note that this not only shows that the vendor subsystem correctly reacts when it receives a cancellation, but also that the composition of the global system effectively let this behaviour happen (no deadlock in the protocols).

### 6.2.4 An invoice id (folio), can be used only once. There will not be two invoices with the same type and and the same taxpayer whose ids are the same

Figure 21 shows the automaton encoding this property. It expresses the safety property: *an invoice id can not be used twice*, but also the reachability property: *an invoice id can be used once*.

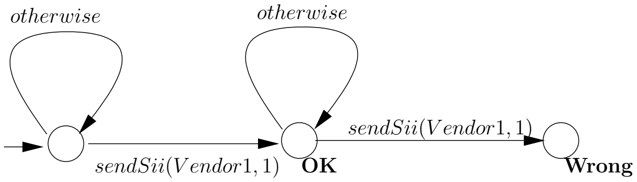

Figure 21: Abstraction automaton encoding Property 4

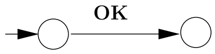

Figure 22: Property 4 verification result

Figure 22 shows the result of the verification process, from which we conclude that the property holds in the system.

### 6.2.5 It is not possible to cancel an invoice which has not been emitted

Figure 23 shows the automaton encoding this property. In addition to the safety property: *It is not possible to cancel an invoice which has not been emitted*, it also expresses the reachability property: *once emitted, an invoice can be cancelled*.

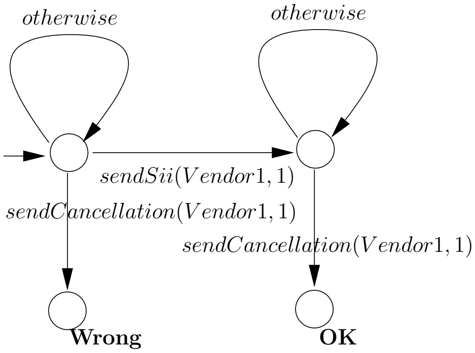

Figure 23: Abstraction automaton encoding Property 5

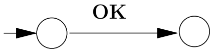

Figure 24: Property 5 verification result

Figure 24 shows the result of the verification process, from which we conclude that the property holds in the system.

### 6.2.6 Every invoice sent to a buyer, should be sent to SII at first

This property can be reformulated as: *there can not be invoices sent to a Buyer which have not been sent before to the SII before*.

Figure 25 shows the automaton encoding this property. It expresses the safety property: *there can not be invoices sent to a Buyer which have not been sent before to the SII*, and the reachability property: *once sent to SII, an invoice can be sent to the Buyer*.

Figure 26 shows the result of the verification process, from which we conclude that the property holds in the system.

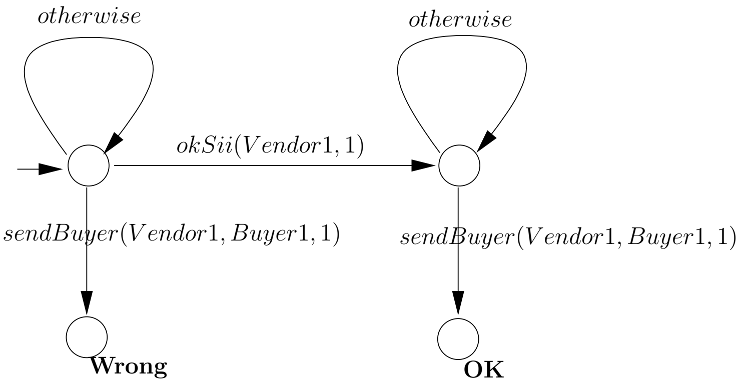
$otherwise$
$otherwise$
$okSii(Vendor1, 1)$
$sendBuyer(Vendor1, Buyer1, 1)$
$sendBuyer(Vendor1, Buyer1, 1)$
Wrong
OK

Figure 25: Abstraction automaton encoding Property 6

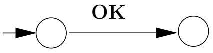

Figure 26: Property 6 verification result

### 6.2.7 Every emitted invoice ends to be accepted by the buyer, or to be cancelled in SII

As in section 6.2.3, this property is an inevitability property, i.e. something that must happen in all possible paths. We express it by Formula 2.

$$\begin{aligned} &AG(\mathit{sendSii}(\mathit{Vendor1}, \mathit{Buyer1}, 1) \Rightarrow \\ &\quad AF(\mathit{okCancellation}(\mathit{Vendor1}, \mathit{Buyer1}) \vee \mathit{acceptBuyer}(\mathit{Vendor1}, \mathit{Buyer1}, 1)). \end{aligned} \tag{2}$$

Formula 2 was successfully proved to be true in the system.

## 6.3 Time-line: How do the proofs improve the model ?

The model presented in section 5, is an improvement of our initial model, based upon the verification of the properties described above. In this subsection we introduce an overview of this improving process from the lessons learned while verifying the property 2 (described in 6.2.2).

Property 2: *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*.

### 6.3.1 Adjusting the Buyer model

In the initial model, the buyer behaviour was described by the process in Figure 27.

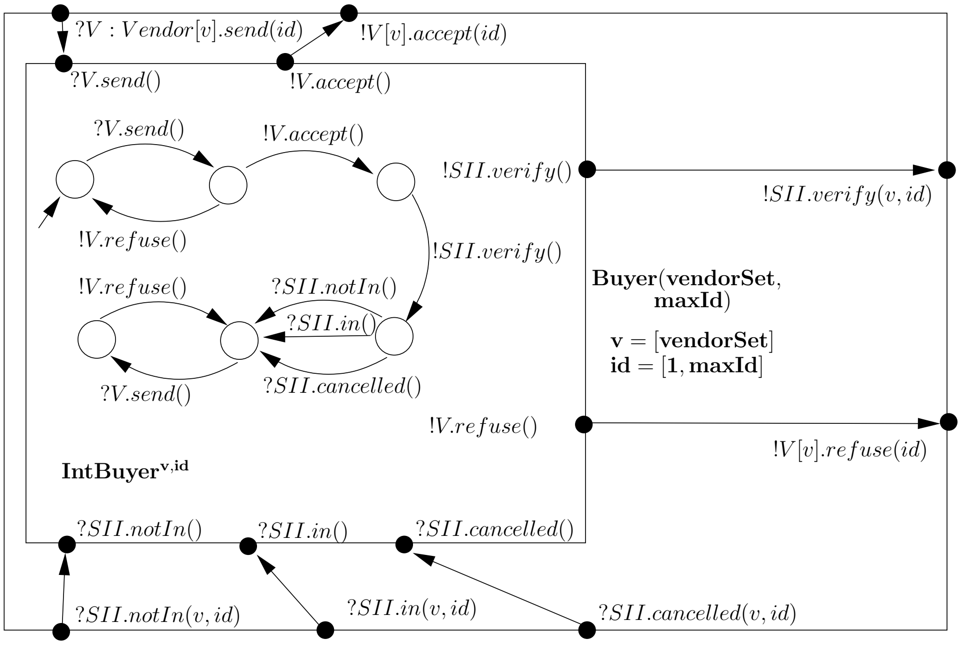

Figure 27: The Buyer system

In comparison with Figure 15, in Figure 27 the behaviour of asking for the invoice status at the initial state ($!SII.verify()$) is missing.

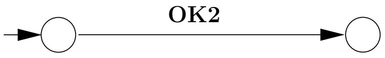

Figure 28: Property check result

The abstract automaton resulting from verifying the property in the initial model is shown in Figure 28. We can conclude from Figure 28, that the state labelled as **OK1** and **OK3** (Figure 19) were not reachable, but only the state labelled as **OK2**. This unreachability situation was not a direct lack in the model, and also it was not a non-desired behaviour. The responses from SII to a request for invoice status are only fired after someone (in our case a buyer) has requested for it via the action *verify*($b, v, id$) (Figure 16).

In the initial model, the only moment the action *verify*($b, v, id$) was made, was after the invoice was accepted by the buyer (Figure 27). The expected correct response to this request at that moment was *in*($b, v, id$), and in fact, it was the only reachable action as proved by the abstract automaton in Figure 28. However, besides that case, no one was requesting the invoice status (action *verify*($b, v, id$)), and this is the reason why the other possible responses never occurred (*notIn*($b, v, id$) and *cancelled*($b, v, id$)).

The other responses are in the system, because SII can not directly control the correct behaviours of Vendors, so this verification step is a guaranty to the Buyer about the validity of the received invoices.

To fire the other possibles responses (and so, to verify that they are correct), we had to match a request for invoice status in a moment when the invoice was not received yet or it was already cancelled.

There were two options to produce this match in the model: by introducing a wrong behaviour Vendor (a Vendor sending invoices to Buyers before sending them to SII, and sending invoices to Buyers after cancelling them); or by adding a behaviour to the Buyer, so that it would ask for the invoice status even when it had not received it. For simplicity we have chosen the second option, since it required fewer modifications in the model. The result of applying this modification is part of the final model, as shown in Figure 15.

### 6.3.2 Adjusting the Reception process in the SII model

Once we corrected the Buyer model as described above, we verified Property 2 again. The abstract automaton resulting from this verification is shown in Figure 29. In Figure 29 there are **Wrong** actions coming out of the initial state, which means that there was a non-desired behaviour in the original system, i.e. SII was giving wrong answers to the invoice status requests.

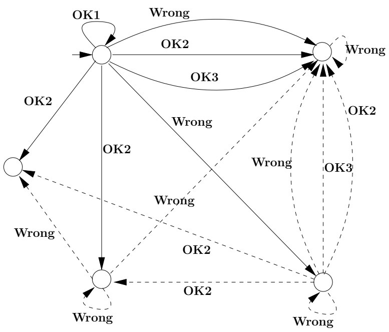

Figure 29: Property check result

After reviewing the model, we realized that the SII was accepting a cancellation before receiving the corresponding invoice (the invoice to be cancelled by the cancellation). This was a fallacy in the original Reception process in the SII model, which is shown in Figure 30.

To construct the final model (Figure 13), we have removed from Figure 30 the intermediary actions *okCancelled*() and *okIn*(). Those actions were used to fire changes in the lower automaton, which gives the answers to invoice status requests. Instead of using these intermediary actions, we used the synchronisation between more than three actions (introduced in 5.2.1) to fire the changes in the lower automaton. Finally, we have added to the **Recp1** automaton the action encoding the reception of the invoice, before receiving the cancellation.

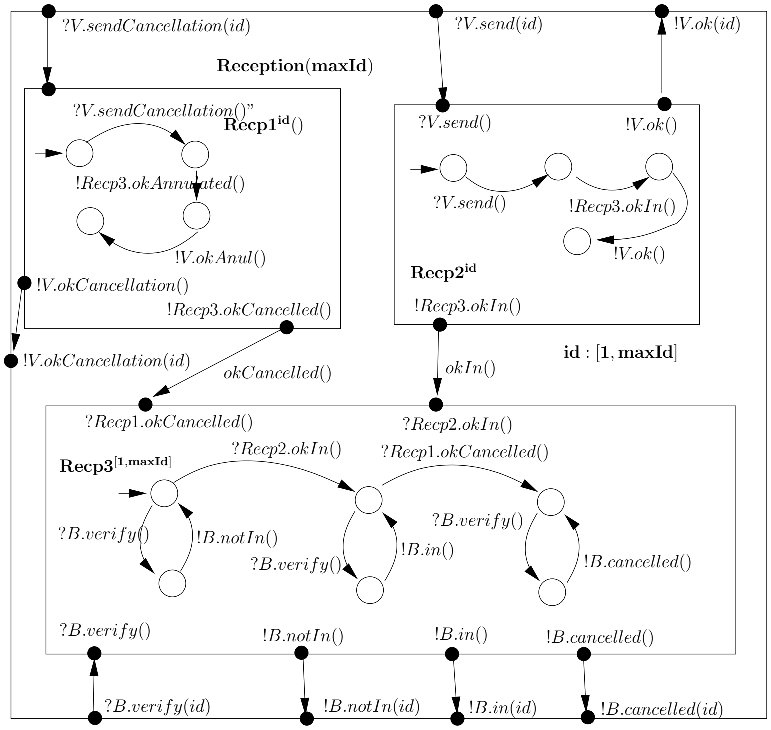

Figure 30: The reception and verification process

After correcting the Reception process, we verified Property 2 again as described in 6.2.2. This time the property was successfully verified.

Note that property 5, verified in 6.2.5, aims to avoid explicitly the non-desired behaviour described here. This is not a property explicitly mentioned in the informal requirements, but we have added it to the properties list after our experience in correcting the Reception process, in order to have a more reliable system description.

# 7 Building finite automata

As explained in section 4, 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) for 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 the instantiation of the variable domains. The user gives these domains for the unbounded variables. In both cases (parameterized and instantiated), the automata and networks are described in files in FC2 format [ARBR94].

## 7.1 Generating the global synchronisation product by brute force

Once the system is instantiated, we should avoid to generate directly 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-know *state explosion* problem. Some techniques that we exploit are the following: compositional hierarchy, parameterized representation and per-formula basis are shown during this chapter to limit the state explosion.

Nevertheless, we have produced small instantiations of the system by brute force. The idea is to have experiments showing how the variable domains affect the system size, to make an initial analysis, and to compare these results later with various techniques. The results of this brute force instantiations are shown in Table 2

An analysis of the values in Table 2 allows us to verify some aspects of the specification and potentially discover errors in it. In fact, there are some suspicious values observed in the instantiations 1 and 3: as we can observe, they share the same states/transitions numbers even when they are instantiated by different variable domains for the cancellation ids. However, as we explain in section 5, for each emitted invoice id, there is a process, and only one, in charge of the potential cancellation of it. Since this process, named **CI**, is the only one that requests ids for cancellation documents, and because this process is instantiated only once (invoices id = 1), there will be only one request for each cancellation id (independent of the number of cancellation ids available) and so the first and third instantiations are equivalent. Following the same reasoning, it is natural to observe different values for the instantiation number 9.

We also see in Table 2 how the variables impact the size of the instantiated processes. For example, the number of purchase processes strongly affect the size of the Vendor process (and so the global product), which is expected since it defines concurrent processes; but it does not affect at all the other process sizes since the purchase processes parameter is only relative to the Vendor. We

| $N^r$ | $idMaxInv$ | $idMaxCancel$ | $maxStockInv$ | $maxStockCancel$ | $pchPrcs$ | $buyerSet$ | $vendorSet$ | **Vendor** | **Buyer** | **SII** | **Global** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 140/860 | 6/9 | 64/348 | 752/2,816 |
| **2** | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 5,404/37,162 | 216/972 | 16,384/168,960 | 58,960/290,208 |
| **3** | 1 | 3 | 1 | 1 | 1 | 1 | 1 | 140/860 | 6/9 | 64/348 | 752/2,816 |
| **4** | 1 | 1 | 5 | 1 | 1 | 1 | 1 | 420/3,456 | 6/9 | 64/476 | 2,256/10,896 |
| **5** | 1 | 1 | 1 | 5 | 1 | 1 | 1 | 420/3,432 | 6/9 | 64/476 | 2,256/10,752 |
| **6** | 1 | 1 | 1 | 1 | 5 | 1 | 1 | 51,428/347,944 | 6/9 | 64/348 | 278,256/1,199,648 |
| **7** | 1 | 1 | 1 | 1 | 1 | 3 | 1 | 404/2,500 | 6/9 | 64/348 | 3,088/11,824 |
| **8** | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 140/860 | 36/108 | 4,096/44,544 | 565,504/4,235,264 |
| **9** | 3 | 3 | 1 | 1 | 1 | 1 | 1 | 8,812/63,710 | 216/972 | 16,384/168,960 | 90,064/462,208 |
| **10** | 1 | 1 | 1 | 1 | 1 | 1 | 3 | 140/860 | 216/972 | 262,144/4,276,224 | *unknown* |
| **11** | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 4,950/38,697 | 1,296/7,776 | *unknown* | *unknown* |

Table 2: Brute force instantiations

also observe that the number of Vendors is the parameter that affects the most the size of the global system.

Note, that up to this point of the discussion, the tool to instantiate parameterized systems can be useful as an early debugging tool. The **unknown** values are due to memory constraints in the production machine, that did not enable us to generate the brute force product.

## 7.2 Structural actions hiding

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 upper levels of the system. This technique, in conjunction with minimisation, gives promising results.

We propose, on a per-property basis, to hide all the actions that are not explicitly in the property we want to prove.

For instance, let us recall Property 1: *A taxpayer could not 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.* This property is shown formalised as an abstraction automaton in Figure 17.

When minimising a system with hidden actions, the more actions you hide, the stronger reductions you obtain. For a given domain of variables, we can know how many links will exist in between two agents when instantiated.

For instance, Figure 17 in page 33 has two groups of explicit actions: $getNewStamps(v, id)$, with $v = \{Vendor1\}$ and $id = \{1, 2, 3\}$; and $sendSii(v, id)$, with $v = \{Vendor1\}$ and $id = \{1, 2, 3\}$. The idea is to hide any other action that is not concerned by the property in the system. To hide the other action means to consider any other action as the non-observable action $\tau$ (see e.g. figure 31).

Together with hiding, we successively generate the synchronisation product and we minimise it using bisimulation equivalence. We make this product by incrementally choosing at each level the pair of processes that share the most actions to be synchronised.

For a given domain of variables, we can know how many links will exist in between two agents when instantiated. For instance in the Vendor the communication $!PP[Pn].giveNewId(id)$ from the **Id** process to the **PP** process will be instantiated to a number of $domain(id) \times domain(p)$ communication links. Given the domain of variables, we propose to synchronise first, at each level, the pair of processes whose synchronisation product will have more hidden actions (i.e. there is no other pair where we can hide more communications than in this one).

Figure 31 graphically shows the composition using this technique for the Vendor and for the variable domains in Table 3. The result of minimisation is shown in the same table.

| $N^r$ | $idMaxInv$ | $idMaxCancel$ | $maxStockInv$ | $maxStockCancel$ | $pchPrcs$ | $buyerSet$ | **Sub1** | | | | **Sub2** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | Strong | | Branch | | Strong | | Weak | |
| | | | | | | | before | after | before | after | before | after | before | after |
| **1** | 3 | 3 | 3 | 3 | 2 | 2 | 1,832/4,036 | 492/1,139 | 1,652/3,688 | 189/484 | 494/1,119 | 140/302 | 383/900 | 20/36 |
| **2** | 3 | 3 | 5 | 5 | 2 | 5 | 10,754/24,415 | 492/1,433 | 9,674/22,285 | 189/664 | 494/1,119 | 140/302 | 383/900 | 20/36 |
| **3** | 3 | 3 | 5 | 5 | 3 | 5 | 518,704/1,712,810 | 4,810/21,414 | 469,009/1,572,215 | 1,165/6,319 | 494/1,119 | 140/302 | 383/900 | 20/36 |
| | | | | | | | **Sub3** | | | | **Sub4** | | | |
| | | | | | | | Strong | | Weak | | Strong | | Weak | |
| | | | | | | | before | after | before | after | before | after | before | after |
| **1** | 3 | 3 | 3 | 3 | 2 | 2 | 34/40 | 34/36 | 21/27 | 14/16 | 560/3,130 | 8/20 | 80/412 | 8/12 |
| **2** | 3 | 3 | 5 | 5 | 2 | 5 | 34/52 | 34/36 | 21/39 | 14/16 | 840/5,554 | 8/20 | 120/744 | 8/12 |
| **3** | 3 | 3 | 5 | 5 | 3 | 5 | 6,709/22,976 | 3,962/11,748 | 1,614/6,144 | 274/836 | 840/5,554 | 8/20 | 120/744 | 8/12 |
| | | | | | | | **Sub5** | | | | **Vendor** | | | |
| | | | | | | | Strong | | Weak | | Strong | | Weak | |
| | | | | | | | before | after | before | after | before | after | before | after |
| **1** | 3 | 3 | 3 | 3 | 2 | 2 | 95/189 | 31/61 | 44/43 | 8/7 | 124/674 | 124/550 | 32/136 | 20/68 |
| **2** | 3 | 3 | 5 | 5 | 2 | 5 | 95/189 | 31/61 | 44/43 | 8/7 | 186/1,119 | 186/1,013 | 48/254 | 28/126 |
| **3** | 3 | 3 | 5 | 5 | 3 | 5 | 6,178/23,771 | 605/2,166 | 406/1,131 | 67/156 | 3,630/29,187 | 3,630/25,557 | 402/2,710 | 227/1,388 |

Table 3: Vendor minimisation with structural hiding

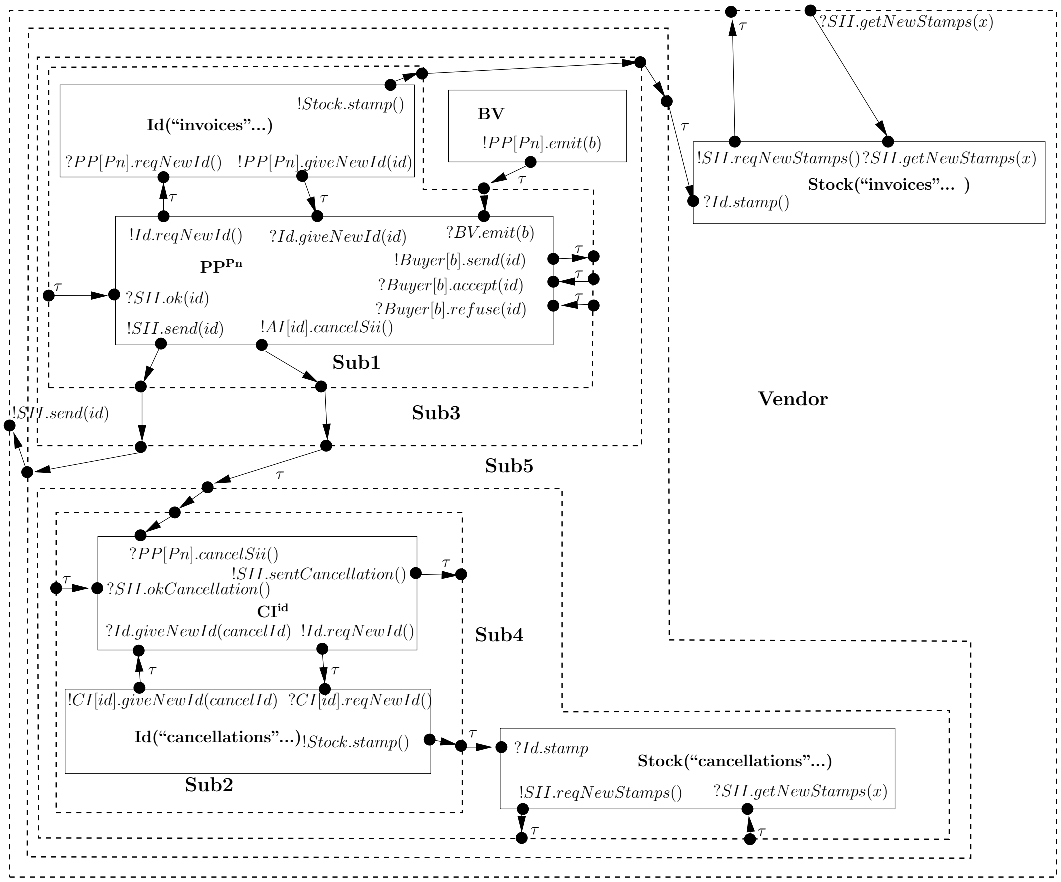

Figure 31: Vendor with structural hiding

Note that the synchronisation product order depends on the variable domains. A different variable domain, such as in Table 2, will require a different order. We observe that this method enables us to scale up in the size of variable domains, compared to brute force instantiations (Table 2).

### 7.3 Grouping by variables

The technique of structural actions hiding, described above, looks very promising when applied to the Vendor, as shown by the results in Table 3. If we try to apply the same reasoning to the global system as a whole, the first synchronisation product that we should make is the one shown in Figure 32.

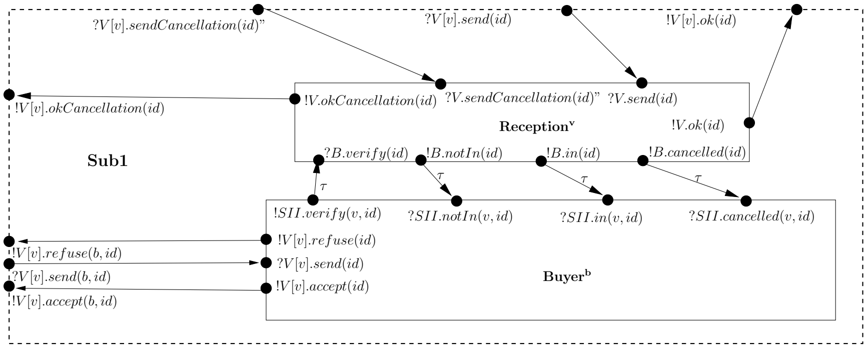

Figure 32: First composition for the global system

However, for any of the variable domains in Table 3, we run out of memory when generating the synchronisation product in Figure 32.

We propose a new method that benefits from the parameterized structure of the system. The idea is to group processes that share a common parameter. For instance, in Figure 13 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, when instantiated, those synchronisations are made only between the three processes with the same value of *id*. So an instantiation of the **Reception** process is the interleaving of the synchronisation product of the three processes for each value of *id* in the instantiation. Therefore, for any instantiation we have the following equivalence:

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

Thus we apply hiding and minimisation to $(Recp1|Recp2|Recp3)$ before instantiating the *id* parameter. Naming the synchronisation product $Recp1|Recp2|Recp3$ as $SimpleReception$, and following the same reasoning, we can conclude the following strong equivalence:

$$
\begin{aligned}
System \sim (&\\
&(IntBuyer^{b}|SimpleReception|CI)^{id}\\
&|PPPn|BaseVendor|Id(invoices)|Stock(cancellations)\\
&|GiveStamps(cancellations)|Id(cancellations)|Stock(invoices)|GiveStamps(invoices)\\
&)^{v}
\end{aligned}
$$

## 7.4 Mixing methods

Remember the property we are using to show our approach to generate the global LTS limiting as much as possible the *state explosion* problem: *A taxpayer could not 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.*

Applying first a grouping by variables and then the structural actions hiding (for this property and the variable domains in Table 4), the global system is arranged as in Figure 33. The sizes of the intermediary synchronisation product and the global LTS, are shown in Table 4.

| $N^r$ | $idMaxInv$ | $idMaxCancel$ | $maxStockInv$ | $maxStockCancel$ | $pchPrcs$ | $buyerSet$ | $vendorSet$ | **Sub1** | | | | **Sub2** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | Strong | | Weak | | Strong | | Weak | |
| | | | | | | | | before | after | before | after | before | after | before | after |
| **1** | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 280/1,065 | 280/1,065 | 280/1,065 | 80/328 | 595/2,490 | 595/2,490 | 176/792 | 128/576 |
| **2** | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 280/1,065 | 280/1,065 | 280/1,065 | 80/328 | 595/2,595 | 595/2,595 | 176/824 | 128/608 |
| | | | | | | | | **Sub3** | | | | **Sub4** | | | |
| | | | | | | | | Strong | | Weak | | Strong | | Weak | |
| | | | | | | | | before | after | before | after | before | after | before | after |
| **1** | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 18,521/59,992 | 1,360/4,947 | 4,757/13,392 | 375/1,448 | 228/573 | 216/536 | 71/178 | 57/150 |
| **2** | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 384,835/1,630,516 | 10,545/51,650 | 47,385/178,932 | 1,861/10,246 | 1,488/5,669 | 1,389/5,305 | 325/1,328 | 251/1,048 |
| | | | | | | | | **Sub5** | | | | **Sub6** | | | |
| | | | | | | | | Strong | | Weak | | Strong | | Weak | |
| | | | | | | | | before | after | before | after | before | after | before | after |
| **1** | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 167/423 | 167/389 | 47/124 | 30/70 | 6/13 | 6/13 | 6/13 | 1/1 |
| **2** | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 725/3,035 | 725/2,751 | 146/676 | 93/379 | 8/20 | 8/20 | 8/20 | 1/1 |
| | | | | | | | | **Sub7** | | | | **Sub8** | | | |
| | | | | | | | | Strong | | Weak | | Strong | | Weak | |
| | | | | | | | | before | after | before | after | before | after | before | after |
| **1** | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 312/996 | 204/661 | 42/86 | 25/55 | 2,061/7,253 | 24/47 | 51/84 | 6/5 |
| **2** | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 2,776/10,564 | 1,977/7,544 | 278/687 | 169/449 | 19,304/81,931 | 34/67 | 253/538 | 8/7 |
| | | | | | | | | **Sub9** | | | | **Sub10** | | | |
| | | | | | | | | Strong | | Weak | | Strong | | Weak | |
| | | | | | | | | before | after | before | after | before | after | before | after |
| **1** | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 72/354 | 72/354 | 18/66 | 12/45 | 144/492 | 144/492 | 24/51 | 12/33 |
| **2** | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 136/740 | 136/740 | 32/136 | 20/88 | 272/1,004 | 272/1,004 | 40/100 | 20/68 |
| | | | | | | | | **Global** | | | | | | | |
| | | | | | | | | Strong | | Weak | | | | | |
| | | | | | | | | before | after | before | after | | | | |
| **1** | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 20,736/141,696 | 20,736/120,960 | 144/792 | 144/792 | | | | |
| **2** | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 73,984/546,176 | 73,984/472,192 | 400/2,720 | 400/2,720 | | | | |

Table 4: Global system grouped by variables and structural hiding

This combination of techniques has enabled us to scale up to a variable domains size that we could not handle before due to the *state explosion* problem. All the verification of properties, described in section 6, were done in the global LTS generated using this methodology.

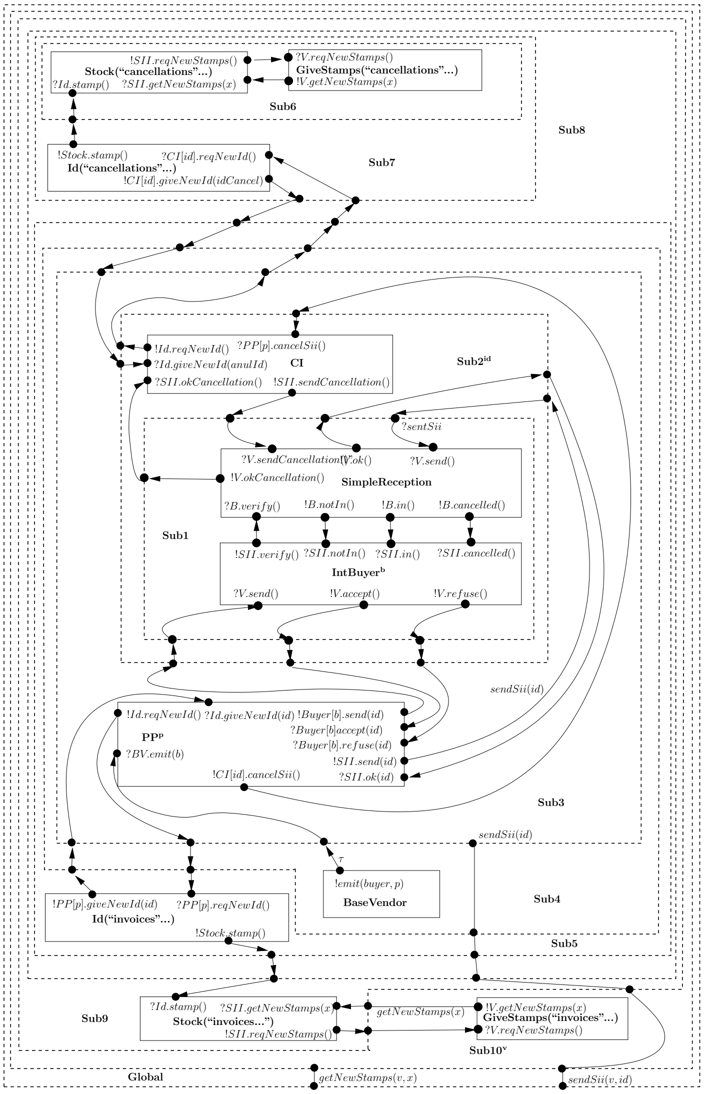

Figure 33: Global system results when grouping by variables and using structural hiding

# 8 Related Work

### 8.0.1 Case study

A similar case study is done by Tronel and all in [TLG03] for the ScalAgent deployment protocol. ScalAgent is a platform for embedded systems, written in Java, to configure, deploy, and reconfigure distributed software. In [TLG03] they make a fully automatic verification for a Ups (*Uninterruptible Power Supply*) management system for large scale sites, deployed in ScalAgent. As us, they model the system as networks of communicating LTSs, which exchange messages by rendez-vous communications. In contrast with us, they start from a formal description of the system thanks to the infrastructure of ScalAgent, which describes its configuration in XML. 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 [TLG03] they have chosen to make an automatic translator from the XML description to LOTOS [ISO98]; the proofs are done by reachability analysis of *ERROR* states, which are defined into those XML descriptions.

The main advantages of [TLG03] are: they use a fully automatic approach, and they directly use the tools from CADP which already includes hiding mechanisms and the use of interfaces constraints [CK96]. They also use parameters but included in the translation to LOTOS and not directly in the formal models as we do. This does not allow them to profit from the parameterized structure of the system to get better minimisations. They determine the variables domains by static analysis. Similar to us, they make finite instantiations for different parameters domains, and they use this instantiation capacity to do debugging and analysis. They find the minimal required instantiations to check the properties by empirical analysis.

Finally, even when we use the same theories and methods to check properties, our aims are different. In [TLG03] they have developed a fully automatic verification methodology specific to ScalAgent, targeted to this specific higher level design language rather than the an implementation language. For us, our study case was analysed with the aim to address any kind of distributed application with asynchronous communications, and also to include the verification of implementations. As we said before, our models are suitable also as models generated from source code.

### 8.0.2 Specification languages

There is a large literature about languages to formally describe concurrent and/or distributed systems at different levels of abstraction. Two of them that could be well suitable to our aims are: NTIF [GL02] and Promela [Hol03].

NTIF is designed to become an intermediate language for E-LOTOS and suitable for efficient model checking and theorem proving. Similar to us, NTIF defines LTS where the transitions encodes 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 requires other tools and formalisms, even when each particular process could be defined in NTIF. Because it has not process composition operators, it does not have hierarchical composition as our synchronisation networks, so it can not profit from modular composition and minimisation; and it does not have a graphical approach to describe systems as we do.

PROMELA [Hol03] is a language designed to describe distributed system. Its way to formalise properties for model checking (Buchi acceptance automata) and its checking results are difficult to master. It does not have process hierarchy, so it can not benefit from modular composition and minimisation. They support simple type parameters as well as guards in the communications actions, but they do not support parameterized processes (set of processes). Even when its models can be graphically visualised, they do not have a graphical approach to describe the systems as we do. One advantages of PROMELA over our model is that it supports dynamic creation of processes (though it treats it, as we do, in a bounded and static way within the Spin model-checker).

As we said, the literature of languages is too rich and to analyse every one of them is outside the scope of this report.

# 9 Conclusion and Perspectives

We have introduced a method and a formalism to formally describe distributed systems and verify their properties, and we have validated our approach through a case study of a real system, the Chilean electronic invoices. We argue that this method is suitable to a developer, not necessarily with expertise in formal methods, by following the methodology used on this case study.

We focus in the behaviour properties. Other analysis such as the data flow or data security requires other specialised analysis and/or tools. The contributions of our work relies in the following points:

- We have defined a language to describe in a natural manner the behaviour of distributed systems (with parameters) via network of processes. This language is a combination and an extension of works from [Arn94] and [Lin96]. 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. The system is fully described using 11 pLTS that synchronise in 7 pNets through 4 levels of hierarchy. In total, the model contains 27 parameterized synchronisations.
- We have developed a tool to obtain finite non-parameterized systems from our language given the variable domains.
- Once instantiated, we generate the LTS describing the whole system behaviour by incremental synchronisation of processes. We group by parameters and we use hiding with minimisation at each level of synchronisation to limit as much as possible the *state explosion* problem. Before, we were limited to generate a global LTS with around $5,6 \times 10^5$ states (see Table 2), using this methodology, we were able to produce a global LTS equivalent to one having around $1,2 \times 10^{12}$ states if generated by brute force to the variable values in Table 6.1.
- Finally, we have shown how to verify the properties of the system, using our instantiation tool and classical finite-state model-checking tools.

Additionally, the instantiation tool is a good tool (for a small instantiation), for comparing different instantiations, instantiating based on per-formula criteria and searching for better minimisations. Especially this debugging capacity provides early detection of errors or backtrack analysis.

Finally, 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)

## 9.1 Perspectives

In the medium term, we plan to integrate our parameterized models to OPEN/CAESAR [GLM02] to do "on-the-fly" model checking. Then, the instantiations and synchronisations product will be generated as needed by the verification tool; and for some reachability properties, the generation of the whole state space will not be necessary.

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 implementation 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. It includes the generation of a parameterized model from the source code. We do generate models for systems implemented in ProActive [CKV98], this generation is described in [MBB04].

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-specialist (for instance designer 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

[ARBR94] R. de Simone A. Ressouche, A. Bouali, and V. Roy. The fc2tool user manuel. http://www-sop.inria.fr/meije/verification/, 1994.

[Arn94] A. Arnold. *Finite transition systems. Semantics of communicating sytems*. Prentice-Hall, 1994.

[BM03] R. Boulifa and E. Madelaine. Model generation for distributed Java programs. In E. Astesiano N. Guelfi and G. Reggio, editors, *Workshop on scientiFic engIneering of Distributed Java applIcations*, Luxembourg, nov 2003. Springer-Verlag, LNCS 2952.

[BPS01] J.A. Bergstra, A. Pose, and S.A. Smolka. *Handbook of Process Algebra*. North-Holland, 2001.

[BRRdS94] A. Bouali, A. Ressouche, V. Roy, and R. de Simone. The fc2tools set. In D. Dill, editor, *Computer Aided Verification (CAV'94)*, Standford, june 1994. Springer-Verlag, LNCS.

[CK96] Shing Chi Cheung and Jeff Kramer. Context constraints for compositional reachability an alysis. *ACM Transactions on Software Engineering and Methodol ogy*, 5:334–377, Oct 1996.

[CKV98] 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.

[CR94] Rance Cleaveland and James Riely. Testing-based abstractions for value-passing systems. In *International Conference on Concurrency Theory*, pages 417–432, 1994.

[DNV] R. De Nicola and F.W. Vaandrager. Action versus state based logics for transition systems. pages 407–419.

[DTE] Gobierno de chile, servicio de impuestos internos, factura electrónica. http://www.sii.cl/cvc/dte/menu.html.

[GL02] 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.

[GLM02] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4:13–24, August 2002.

[GP94] J.F. Groote and A. Ponse. Proof theory for $\mu$CRL: a language for proce sses with data. In Andrews et al., editors, *Proceedings of the International Workshop on Semant ics of Specification Languages*, Workshops in Computing Series, pages 231–250. Springer Verlag, 1994.

[Hen91] M. Hennessy. A proof system for communicating processes with value-passing. *Formal Asp. Comput.*, 3(4):346–366, 1991.

[Hol03] G. Holzmann. *The SPIN Model Checker, Primer and Reference Manual*. Addison-Wesley, 2003. ISBN 0-321-22862-6.

[ISO98] ISO: Information Processing Systems - Open Systems Interconection. Lotos - a formal description technique based on the temporal ordering of observational behaviour. ISO 8807, Aug 1998.

[Lak96] A. Lakas. *Les Transformations Lotomaton : une contribution à la pré-implémentation des systèmes Lotos*. PhD thesis, juin 1996.

[Lin96] Huimin Lin. Symbolic transition graph with assignment. In U. Montanari and V. Sassone, editors, *CONCUR '96*, Pisa, Italy, 26–29 August 1996. LNCS 1119.

[Mad92] E. Madelaine. Verification tools from the concur project. *EATCS Bull.*, 47, 1992.

[MBB04] E. Madelaine, R. Boulifa, and T. Barros. Parameterized models for distributed java objects. In *Forte'04 conference*, Madrid, 2004. Spinger Verlag.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989. ISBN 0-13-114984-9.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100(1), 1992.

[MS00] 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, April 2000.

[Obj] Objectweb consortium. http://www.objectweb.org.

[TLG03] Frédéric Tronel, Frédéric Lang, and Hubert 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.

# Contents

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399