Formal description and analysis for distributed systems

Tomás Barros and Eric Madelaine

INRIA Sophia-Antipolis
(tomas.barros, eric.madelaine)@sophia.inria.fr

Abstract. We introduce a graphical syntax to model distributed systems with asynchronous communications. We extend the general notion of labelled transition systems and hierarchical networks of communicating systems (synchronisation networks) to add parameters to the communication events. Our agents can also be parameterized to encode sets of equivalent agents running in parallel. Our model is suitable for compositional description of distributed system behaviours as for models resulting from static analysis of source code.
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 verification tools we use are based on Process Algebra theories (Fc2 tools and CADP). Once instantiated, we check properties by reachability analysis (when possible) or by on-the-fly model checking.
We have validated our approach by modeling the recently started Chilean system for electronic invoices and by checking formal properties (requirements) on it.

1 Introduction

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, to local or wide area networks, to pervasive and mobile computing.

Each one of those applications have specific requirements. Methods and tools to specify their behaviour (requirements) and to check the correctness of their implementations, become necessary. These methods should have a sound formal basis to be used by tools, but also should be simple enough to be used by non-specialists. They also should be as automatic as possible, hiding the complexity in their logics and algorithms.

There exists a number of semi-formal or formal frameworks for the description of software systems. In the first family we cannot avoid the most recent versions of UML, in which statecharts diagrams have received some sort of formal semantics; however these formalisms are not well adapted for use in connection with automatic verification tools, and it is unclear whether they will ever be suitable for the modelisation of distributed objects. More formal, and closer to our concerns are various trends of process algebras, including Value-passing CCS [Mil89], μCRL [GP94] and LOTOS. But what we seek is a formalism that will naturally guaranty that we can automatically derive finite representations; all these algebras are much too expressive for this goal, and would require some elaborate restrictions, or compilation phase. A number of Intermediate Formats can also be compared to our formalism, including NTIF [GL02] (but this one has no parallel composition), or PROMELA [spi03] (but it would be difficult to use in our action-based framework).

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 specification framework is based on process algebra theories [BPS01], and makes use of software tools [Mad92,RdSBR94,GLM02] to perform automatic proof of behavioural properties. We extend the general notion of labelled transition systems and hierarchical networks of communicating systems (synchronisation networks [Arn94]) to add parameters to the communication events. Our agents can also be parameterized to encode sets of equivalent agents running in parallel.

We want both, to represent infinite systems (encoding realistic applications) in a finite manner by using parameters, but also to get finite models (instantiations) from a parameterized one. Thereby, the parameters in our systems are typed variables of simple enumerable types: booleans, integers, intervals, finite enumerations or structured objects. Our model is suitable to compositional description of distributed system behaviours as to extraction of models from static analysis of source code. Our team is already working in model generation from static analysis of the source code for PROACTIVE [CKV98] applications [BM03]. PROACTIVE is a Java implementation of distributed active components with asynchronous communications and replies by means of future references, developed by our team.

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. Once instantiated, we use the tools to compose the system hierarchically and to verify properties by reachability analysis or on-the-fly model checking.

In the next section we introduce our parameterized models. Section 3 explains how we instantiate and verify properties in our system. In the last section we conclude about our work and we introduce the perspectives.

2 Parameterized Models

Our models are communicating automata which are composed hierarchically through synchronisation networks [Arn94]. We shall use Parameterized Labelled Transition Systems (*pLTS*) and Parameterized Networks (*pNet*) for compact and more natural representation; and for better understanding and usability by non specialist.

We extend the general notion of *labelled transition systems* [Mil89] to add parameters in the states and in the transition labels. The parameters are typed variables of simple countable types: booleans, integers, intervals, finite enumerations or structured objects.

The variables are inside expression vectors. Each expression may include prefixes, unitary and infixes operators; as well as constants of type string and integer.

In our *pLTS* the states are labelled by an expression. The transitions are labelled by a guarded action with an explicit resulting expression vector ($[guard]$ $action \rightarrow result$), such that the arity of the resulting vector is the same than the arity of the arrival state.

Every *pLTS* is inside a box with ports. The ports represents the observable actions of the process which can be synchronised with other process actions. They are represented by bullets in the edge of the box, and they are labelled with actions in transition labels of the *pLTS*.

A *pNet* is a finite set of boxes with ports for synchronisation. Each box is related to a *pLTS* or another *pNet* (hierarchy), which defines its behaviour. In between two or more boxes, the ports can be interconnected through links (edges) for synchronisation. The links are labelled with τ or an optional parameterized action. Inside a network, we will refer to each box as a process.

We use a graphical approach based in the graphical syntax of Autograph [RdSBR94]. Autograph is a graphical display system for both labelled transition graphs and networks of communicating systems.

Figure 1 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 a time. Each consumer requests a single element from the buffer ($!B.Q\_get()$) and waits for the response ($?B.R\_get()$).

We have introduced in Figure 1 the notation to encode sets of processes. In the figure, $\mathbf{Consumer^c}$ encodes a set, whose elements are **Consumer** processes (running in parallel) for each element 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 identifier ($\in domain(c)$).

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 1 when the variable N is evaluated to 0 in the **Buffer** automaton, then the state is to be considered as the initial state.

When new variables are introduced, their type is defined. The type of a variable is propagated in communication events from the sending process to the receiving one. A typed variable in a reception action, restricts the reception to this type of variable. In the figure, the action $?P.Q\_put(x)$ may receive only integers since x is typed as int in the communication port.

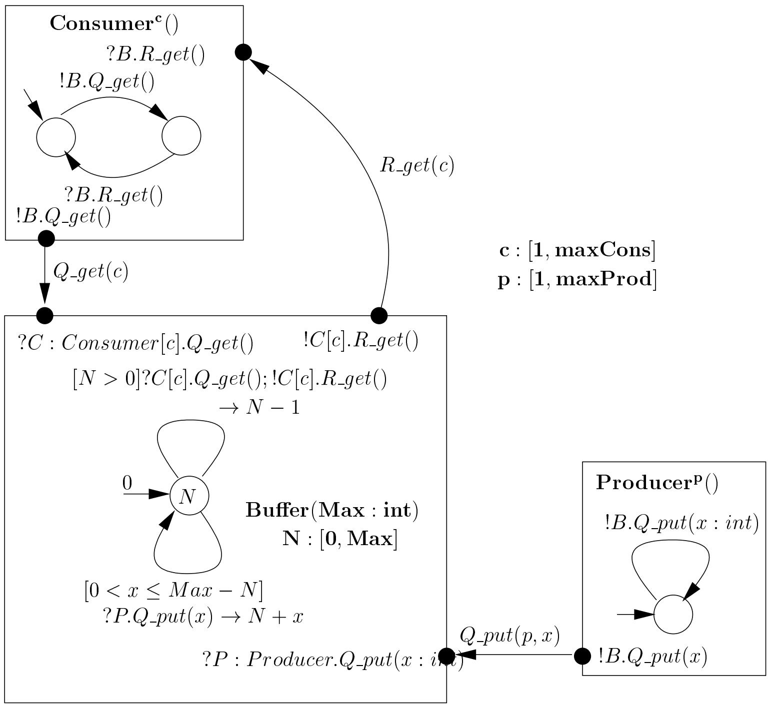

Fig. 1. Parameterized consumer-producer system

3 Finite Models and Verification

We distinguish three kinds of variables in our systems: constructor parameters, such as Max in Figure 1; bounded variables (finite domain) and unbounded variables (infinite domain). In Figure 1, the variable N in the **Buffer** component is bounded. In the same component, the variable x seems to be unbounded, but since its domain is restricted at the guard of the transition edge, its domain become finite and the variable x is bounded. For the **Producer** component, the variable x is unbounded.

By fixing the value of the constructor parameters and fixing the unbounded variables to a finite domain, a finite Labelled Transition System (*LTS*) can be generated from our parameterized systems. In Figure 1, if we fix the value of Max to 3, $maxCons$ and $maxProd$ to 2; and the domain of variable x in the **Producer** component to $[1, 3]$, we obtain the *LTS* shown in Figure 2.

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 variables domain values. In both, parameterized and instantiated systems, the automatas and networks are described in FC2 format [RdSBR94] files. The FC2 format encodes in a compositional way finite transition systems: basic systems are described by explicit enumeration of states and edges, decorated by labels representing structures (names) and behaviours (actions); the composition of systems is made by synchronisation networks [Arn94], expressing how behaviours of components are combined at the next level of the structure.

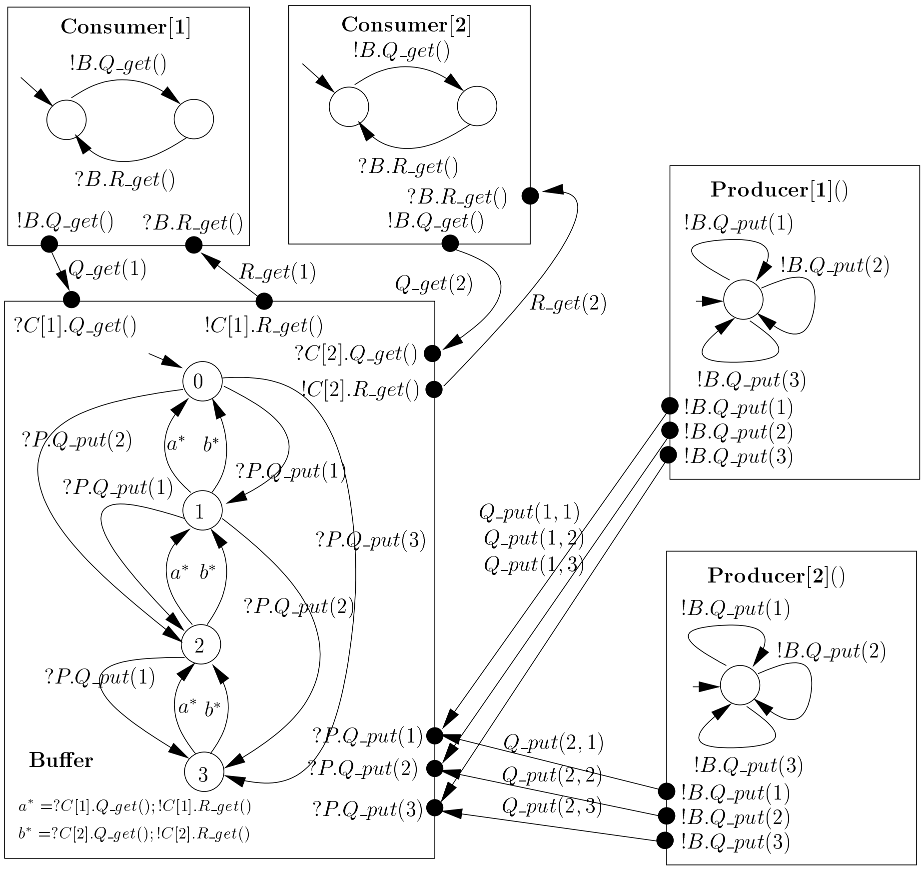

Fig. 2. Instantiated consumer-producer system

Once instantiated, we compose the system hierarchically using the FC2Tools [RdSBR94], applying as much minimisation as possible at each level. One technique to avoid the *state explosion* problem, which profits from the parameterized structure of our models, consists in grouping the basic components whose occurrence is shared in a set of variables, then applying hiding and minimisation on these groups before instantiating to the domain of the variable set. This technique, as well as hiding and minimisation in our models is illustrated in [BM04].

After we instantiate the components and compose the system, we want to prove several properties of it. Those properties can be provided at early stage of the design as informal requirements, or as scenarios. From the informal requirements, it should be extracted and formally expressed the set of properties to check in the system.

For reachability properties, we use a graphical approach, in which the designer will use an *abstraction automaton* (similar to a transition labelled automaton but with logical operators in the labels and ending or acceptance states) defining a family of desired or non-desired *abstract actions*, each of them being a set of traces (a regular language) of the actions of the original LTS. From the original (concrete) system and the abstraction automaton (expressing the property), the tool FC2tools [RdSBR94] builds an abstract LTS, whose actions are the abstract actions. If an abstract action is present in the abstract system, then one of the corresponding concrete sequence is possible in the concrete system. Note that this method cannot deal with "inevitability" properties, as it relies only on the existence of a path. The same method applies to reachability of non-desired abstract actions, so we can prove that some sequence cannot occur in the concrete system by showing that a sequence is not possible.

For inevitability-like properties the abstract action method is not sufficient. In those cases, we rely on the tool EVALUATOR [MS00], which uses a form of regular μ-calculus logics. EVALUATOR performs on-the-fly verification of the temporal property on a given Labelled Transition System (LTS). The temporal logic used to express properties in EVALUATOR is called regular alternation-free μ-calculus. It allows direct encodings of "pure" branching-time logics like CTL and ACTL. We express our desired properties in ACTL, and we use the macros provided with EVALUATOR to transform them to regular μ-calculus. Then we use EVALUATOR to verify the formula. The result of this verification is *true* or *false* for the given formula and LTS.

For instance, we would like to prove the following two properties in the system of Figure 1:

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

For the property 1, we use reachability analysis. The abstraction automaton for this property is shown in Figure 3. In addition, Figure 3 express the property that once the buffer fed, a consumer will be able to get an element (**OK** state).

Since we work with parameterized system, we want to express the properties parameterized too. This is the case in Figure 3, where $\bigvee_{domains}(action)$ means the or-exclusive composition of the *action* for each of the elements in *domains*. The *otherwise* action means any other action different from the actions in the outgoing edges of the same state.

However, the tools that we use support neither parameterized system nor parameterized properties. We need to get instantiations from the abstraction automaton for the same variable domains than to the instantiated system where we will check the property.

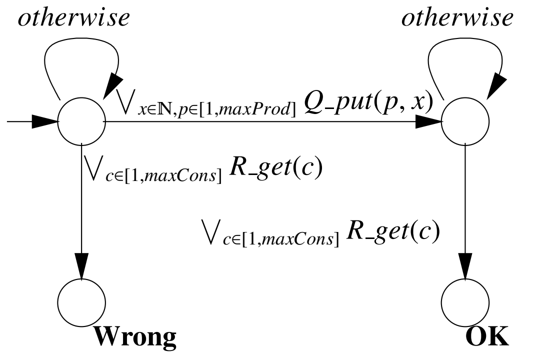

Fig. 3. Property: can not get elements from the buffer before feeding it

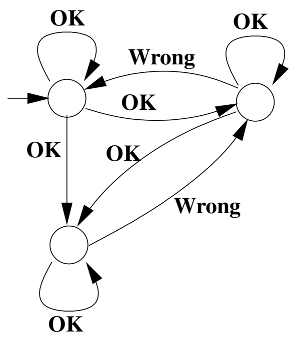

Fig. 4. Property verification result

In Figure 4 is shown the abstract automaton (minimised by weak bisimulation) resulting from verifying the property to the instantiated (finite) system shown in Figure 2. In the automaton the action **OK** is possible from the initial state, which means that the state labelled as **OK** in the abstraction automaton (Figure 3) is reachable from the initial state, and so, we proved that is it possible to get an element from the buffer once the buffer has been fed. Likewise, since there are not **Wrong** actions from the initial state in the result, we conclude that the state labelled as **Wrong** in the abstraction automaton is not reachable from the initial state, and so, we have proved that is not possible to get an element from the buffer if it has not been fed before. Because we are interested in to verify the property in the initial state of the system, we centre our attention only in the outgoing edges from the initial state in the resulting abstract automaton.

The property 2 is an inevitability-like property, which we can not express by an abstraction automaton. We express the property by the ACTL formula: $AG(Q\_get(c) \Rightarrow AF\ R\_get(c))$. Using the macros from EVALUATOR for any integer value of c, the formula is translated to the regular μ-calculus formula shown in equation 1.

$$[true * .Q\_get(c)]\mu X.(< true > true \wedge [\neg R\_get(c)]X) \tag{1}$$

As it was the case for the abstraction automaton, the formula should be instantiated for the variables domains of the instantiated system where we will verify the property. The property was successfully verified for the instantiated system shown in Figure 2.

4 Realistic Study Case

We have validated our approach in a system borrowed from the current effort made by the Chilean administration to provide electronic procedures for their sales and tax system: "DTE: *Documentos Tributarios Electrónicos*" [DTE]. This gave us a realistic case study, from which we have done a formal abstract specification and we have proved some correctness properties at the level of the specification. Our results are shown in [BM04].

In DTE the global behaviour is composed by three parameterized agents: an unique tax agency, a bounded quantity of vendors and a bounded quantity of buyers; each of them is described by an hierarchical composition of communicating automata. In total, the system includes fifteen parameterized automata that are composed up to four levels of hierarchy.

The instantiations are done by fixing the domain of seven variables. We made several instantiations and composition without minimisation for different variable domains to debug and to see the impact of the variable domains in the system size. We did not scale up much in that way.

Using the grouping, hiding and minimisation techniques describe in [BM04], we gained from 0% up to 99.9% in the size of intermediary compositions for different variable domains. Therefore, those techniques enables us to work with larger variable domains.

By verifying the properties, we found some mistakes and misunderstanding in the formal specification. After correcting the specification, seven properties (extracted from the informal requirements [DTE]) where successfully verified.

5 Conclusion and Perspectives

We have introduced a graphical syntax based on Labelled Transition System (LTS) and Synchronisation Networks which include variables. We named them *pLTS* and *pNet* respectively.

We developed a tool to generate a finite system from a parameterized one by fixing the variables domains. This capacity to instantiate finite *LTS* from a *pLTS* enables us, in between others: make debugging analysis (for a small instantiation), compare different instantiations, instantiate based on per-formula criteria, search for better minimisations, etc.

Once the specification is validated we want to use it to check the correctness of implementations. This check will need a refinement pre-order, that allows the implementation to make some choices amongst the possibilities left by the specification. It should be compatible with the composition by synchronisation networks. Amongst the many refinement relations that have been defined in the literature, to our knowledge, few have been implemented in LTS-based tools. 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. Our team is already working in this subject for ProActive [CKV98]. ProActive is a Java implementation of distributed active objects with asynchronous communications and replies by means of future references. The model generation is done by a behavioural semantics for Java/ProActive applications, given in the form of SOS rules working on the method call graph.

Those rules give a procedure to build finite LTS and synchronisation networks representing the application [BM03]

A work should be done to formalise our lose of information because of the abstractions we do. We should find the right approximations in the *Abstract Interpretation* [Cou01] theories that enables us to reason about properties in our models from the properties in their instantiations. We also need to extend the language to express the properties as parameterized abstraction automatas and regular μ-calculus formulas.

The graphical syntax we have introduced was intuitively developed. We are working to formalise the syntax and to give a semantics which allow us to explore new capabilities of our representation such us parameterized composition of components. We have also started to analyse the *pLTS* to integrate it with OPEN/CAESAR [Gar98] tools to make "on-the-fly" model-checking.

References

[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.

[BM04] T. Barros and E. Madelaine. Formalisation and proofs of the chilean electronic invoices system. Technical Report RR-????, INRIA, mar 2004.

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

[CKV98] D. Caromel, W. Klauser, and J. Vayssière. Towards seamless computing and metacomputing in Java. *Concurrency Practice and Experience*, 10(11–13):1043–1061, November 1998.

[Cou01] P. Cousot. Abstract interpretation based formal methods and future challenges, invited paper. In R. Wilhelm, editor, *Informatics — 10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer-Verlag, 2001.

[DTE] Gobierno de chile, servicio de impuestos internos, documento tributario electrónico (dte). http://www.sii.cl.

[Gar98] Hubert Garavel. Open/caesar: An open software architecture for verification, simulation, and testing. Technical Report RR-3352, INRIA, Jan 1998.

[GL02] Hubert Garavel and Frédéric Lang. Ntif: A general symbolic model for communicating sequential processes with data. Technical Report RR-4666, INRIA, Dec 2002.

[GLM02] Hubert Garavel, Frédéric Lang, and Radu 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 μCRL: a language for processes with data. In Andrews et al., editors, *Proceedings of the International Workshop on Semantics of Specification Languages*, Workshops in Computing Series, pages 231–250, 1994.

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

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

[MS00] R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. In *Proceedings of FMICS'2000*, Berlin, April 2000.

[RdSBR94] A. Ressouche, R. de Simone, A. Bouali, and V. Roy. The fctool user manuel. http://www-sop.inria.fr/meije/verification/, 1994.

[spi03] *The SPIN Model Checker, Primer and Reference Manual*. Addison-Wesley, 2003. ISBN 0-321-22862-6.