Finite Model Generation for Distributed Java Programs

R. Boulifa and E. Madelaine
INRIA Sophia-Antipolis, BP 93, 06902 Sophia-Antipolis Cedex, France
{rabea.boulifa eric.madelaine}@sophia.inria.fr

***Abstract*— We present techniques for analyzing the source code of distributed Java applications, and building finite models of their behaviour. The models are labelled transition systems, representing the communication events between the distributed objects. When combined with techniques for abstracting the data values used by the programs, and especially values used in the creation of distributed objects, to bounded domains, our construction terminates. We provide models suitable for automatic verification, and typically for model checking. Moreover our models are structured in a compositionnal way, so that we can use verification techniques that scale up to applications of realistic size.**

I. Introduction

The use of formal verification techniques to enhance the reliability of software systems is not yet widely accepted, though there are a number of methods and tools available. We are specifically interested here in the behavioural properties of applications distributed over local networks or on the Internet. Examples are grid computing applications, entreprise applications over heterogeneous and mobile networks and terminals, web-services and e-commerce, etc. These applications make heavy usage of various kinds of asynchronous protocols for communication between remote components of the application.

Ideally, one should be provided with a development environment including high level languages for the specification of the requirements of the system, and automatic tools for checking the implementation against the specification. At the heart of this is the ability to generate, from the code of an implementation, a model that is both precise enough to encompass the property we want to prove, but small enough to be manageable by the tools (and at least have a finite representation).

A number of tool developments have been done recently, for example in the Slam project [1] for analysis and verification of C programs, or in the Bandera project [2] for sequential or multi-threaded Java programs. Typically this kind of tools use static analysis techniques, coupled with abstraction and slicing, to produce a finite model that is fed to various model-checking tools. The whole process cannot be fully automatic, because the data abstractions must be provided by the user, and also because other forms of approximations are needed to deal with the control flow of the program (and potential dynamic creation of objects).

We are developing a similar framework, dedicated to distributed Java applications in which communications between remote objects are done by asynchronous method calls. We are interested in proving global properties of distributed applications, namely temporal properties capturing the significant events of the lifecycle of distributed objects: sending of remote method calls, receiving results from remote computations, selecting and serving requests from the local request queue. Distributed applications fit naturally with compositionnal models, and we take advantage of this for structuring the models, thus keeping them much smaller, and we rely on verification tools that make use of this structure.

We rely on a middleware called ProActive [3] that provides the developer with a high-level programming API for building distributed Java applications, ranging from Grid computing to pervasive and mobile applications. ProActive has a formal semantics, that garanties that programs behave independantly of their distribution on the physical network; *active objects* are the basic units of activity, distribution and mobility used for building these applications.

A distributed or concurrent application built using ProActive is composed of a number of medium-grained entities called *activities*, distributed on various nodes. An activity is composed of a set of objects, one of them being *active*, and of a queue of pending *requests* (that have been received and should be served later). The active object has a specific *service* method that specifies the order of requests that the object should serve.

Method calls to active objects are always asynchronous with transparent *future objects* and synchronization is handled by a mechanism called *wait-by-necessity*. Note that while asynchronous, there is a guaranty of delivery, and a conservation of order for remote method calls ; this is achieved thanks to a short *rendez-vous* phase at request sending.

The verification tools we use to check our models (e.g.

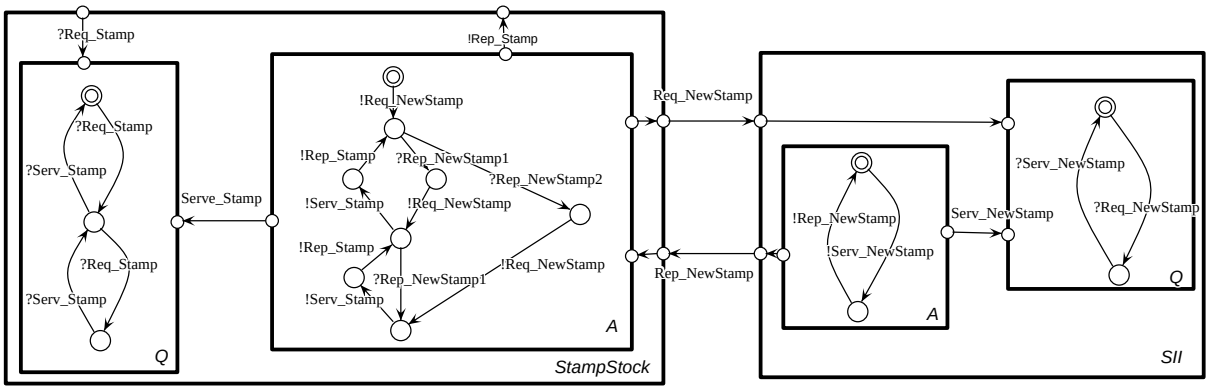


Fig. 1. Constructed models for StampStock and SII

Evaluator [4]) are based on Process Algebras theories : models are (finite) communicating labelled transition systems, their semantics is considered modulo bisimulation congruences, and their emphasis is on properties related with bisimulation semantics, including safety and liveness properties in modal branching time logics, and more generally equivalence of models with different level of refinement. These tools take advantage of congruence properties to avoid state explosion of the models. This approach allows us to build the models of our applications on a per-object basis, and even to specify their desired behaviour in a component-wise manner.

Our contributions in this paper are:

- Finite, hierarchical models for distributed applications communicating through asynchronous method calls, suitable for automatic verification of temporal properties of the applications.
- A behavioural semantics for distributed Java/ProActive applications, in the form of SOS rules expressed on the method call graph of the application, and a procedure for executing these rules, that is garantied to terminate producing a finite labelled transition system (LTS) for each active object class in the application.

II. Compositional Models for Distributed Objects

Due to lack of space, we shall not give here a full and formal presentation of the model and of their construction, but rather try to stress the most important points in the approach. The full paper is available in [5].

a) Synchronized Labelled Transition Systems: As usual in the setting of process algebras and distributed applications, we give the behavioural semantics of programs in terms of LTSs, which labels represent the significant events we want to observe (communication between activities, plus some local method calls). We use hierarchical composition of LTSs, synchronized by synchronization networks [6]; these can be viewed as generic parallel composition operators, and we also use them to express graphical networks [7] like those in figure 1.

b) Decomposition: We want to build a (finite) LTS for each activity in the system, and compose them by synchronization networks, based on the messages they exchange, namely requests to remote method calls, and their responses. In the next section, we shall see how we build a finite LTS for each active object class of an application. In our graphical networks formalism, each activity is enclosed in a box, and has ports corresponding to the messages. A request queue is associated to each activity, that we abstract also as a finite LTS. Figure 1 (taken from the specification of the chilean electronic tax system) gives an example of two such activities, where StampStock accepts request `Stamp`, and in turn sends requests `NewStamps` to SII.

Inside each activity box, we find two LTSs, one for the object behaviour model, and one for the queue model. The ProActive library has primitives allowing the programmer to control very precisely the inspection of the queue and the service of requests; this is reflected in the synchronization between the queue and the behaviour LTSs.

c) Analysis of the topology: There is no hope to enumerate exactly by static analysis all the activities potentially created during the lifetime of an application, because this set may be infinite, and because activities may be created dynamically. The best we can do by static analysis is to identify the (finite) set of activity creation points in the source code, and to define an approximation of the possible values of the creation arguments. This will gives us a set of potentially created activities.

We currently have two approaches for this enumeration. In the first approach, we compute a fully enumerated (finite) set of activities: in simple cases where the topology is fixed and clearly identified in the source

code, static analysis will give us a full enumeration of the activities. In cases where this is not possible, we rely on a user-specified enumeration (e.g. as annotations in the source code). In the second approach, we request that the user provides an abstraction of all arguments types into integers (or finite domains), and we keep these integer parameters in our models. Parameterized models will later be instanciated to finite models, if necessary, before using standard verification tools. Moreover, this abstraction can be tuned depending on the properties we want to prove, keeping the model size as small as possible; whenever possible, arguments that are not significant for the analysis are abstracted away.

The second part of the topology information required for building the networks is the set of links between the activities. For each activity we compute those links: the set of incoming requests is the set of public methods of its active object, qualified by their arguments (abstracted as in the previous paragraph). The set of outgoing requests is computed from the program points where a method of a remote active object is called; here the identification of the remote object, and the arguments passed to the method, are obtained symbolically, and depends on the parameters of the current activity. In the parameterized approach, we keep them in their symbolic form. The set of service synchronization message, obtained from the service primitives used in the service method of the active object.

III. Behaviour of activities

Now we switch to the behaviour of one activity, and we describe the procedure for computing its LTS.

The behavioural semantics of an activity is described as a set of SOS rules, based on the Method Call Graph (MCG) of the activity, rather than its original source code. The MCG is generated by usual static analysis techniques, retaining the additional information useful for the specific treatement of the ProActive primitives. This makes the semantics (and the model production procedure) much more compact than if we had to deal with the full set of language constructions.

Before building the MCG, we rely on abstract interpretation techniques to abstract away from the infinite data domains manipulated by the Java source code (including the data passed as arguments to activity creation and method calls already mentionned). We use also slicing techniques [8] to eliminate portions of the system irrelevant to the proof of some specific properties. This is similar to what is done e.g. in the Bandera system [9] (and in practice we use tools from the Bandera toolset).

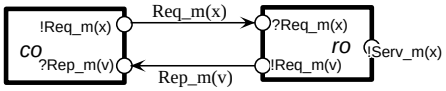
The SOS rules give us a procedure for constructing the LTS modeling the behaviour of an activity, traversing its MCG, starting with its service method. The rules have the following form:

$$\frac{\{Condition\}*}{< v = pattern,\ n,\ \mathcal{A},\ \mathcal{M},\ \mathcal{S}_n,\ \mathcal{S}_m > \Longrightarrow_A \quad < v',\ n',\ \mathcal{A}',\ \mathcal{M}',\ \mathcal{S}'_n,\ \mathcal{S}'_m >} \ (\ )$$

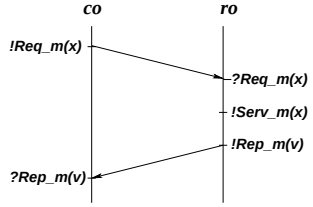
where $v = pattern$ is the current analysed MCG node, n is the last LTS node created, $\mathcal{A}$ the currently constructed LTS, $\mathcal{M}$ a mapping from MCG nodes already visited to corresponding LTS nodes, $\mathcal{S}_n$ a stack of (not yet visited) MCG nodes, and $\mathcal{S}_m$ (a finite abstraction of) the current method call stack.

The rules are organized in the following three groups, corresponding to the various types of the MCG nodes, and to the specific ProActive primitives:

- The rules describing the intra-activity constructions of the MCG, dealing with intra-procedural control flow and with local method invocation. Our treatment of recursivity is guarantied to terminate: we detect recursion whenever the name of the current method is included in the method call stack (so we never need to repeat a method name in our representation of the call stack, which stays finite). In the produced LTS, we only keep trace of (local) method calls if they are useful for the properties we want to prove, otherwise, the corresponding states are merged.
- The rules describing the communication between objects; the events of our models are the requests and responses of remote methods, and the local service primitives.



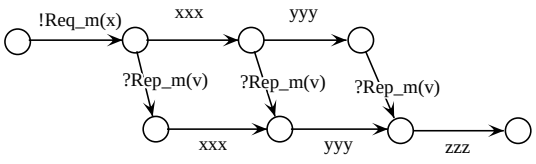
Consider an object co (the current active object), calling method m (with a non-void return type) of a remote active object ro.



We model these calls by message exchanges between the 2 processes: co emits a request $!Req\_m(x)$, that is instantaneously received by ro's queue as $?Req\_m(x)$. The rule for Remote Method Call constructs a transition $(n \xrightarrow{!Req\_m(x)} n')$ in co's behaviour model, corresponding to

$(s \xrightarrow{?Req\_m(x)} s')$ transitions in ro's queue model.
The remote object ro may eventually decide to serve this request. Starting the execution of the method call within object ro is modeled by a message $Serv\_m(x)$ local to ro (a transition labelled $!Serv\_m(x)$ in ro's behaviour model and some transitions labelled $?Serv\_m(x)$ in ro's queue model).
And finally, when method m returns a result, we build a transition $(n \xrightarrow{!Rep\_m(v)} n')$ in ro's LTS, that will be received as $?Rep\_m(v)$ in *co*.

- Finally the rules dealing with the *wait-by-necessity* semantics for future values: computation in distributed objects being asynchronous, the result of a remote method call may be received with some delay. ProActive allows the calling activity to continue its local computation, until this value is really needed, at which point it will eventually wait for the value. This is represented in our model by interleaving the return of the value (action $?Rep\_m(v)$) with the sub-LTS representing the possible computations of *co* between the method call and all possible first utilisation points of the return value. For example, the interleave of the LTS having the transitions labelled *xxx* and *yyy* between these two points, afterwards the transition labelled *zzz* modeling the explicit using of the result, produces the following LTS :



IV. Models for Queues

Each activity has a request queue, dealing with arrival and service of requests. The queue is always accepting requests (i.e. public methods of the active object). The service method of the active object is in charge of managing the order in which requests should be served, depending on its own internal state. If unspecified, requests will be served in FIFO order.

In most cases the queue is unbounded, and we cannot give a complete model in terms of LTSs. We shall artificially set a bound to the size of the queues. There are many practical cases in which we need only a bounded queue; however this is a global property of the system, and we shall have to prove it whenever possible [10].

A general model would be a finite queue of length n, in which each cell can host all possible values for the method name and (abstracted) arguments. This is still a huge state-space, exponential in the number of distinct requests. Fortunately, it is often possible to factorize this structure into separated queues dealing with independant request types. We use information from static analysis to perform this factorization.

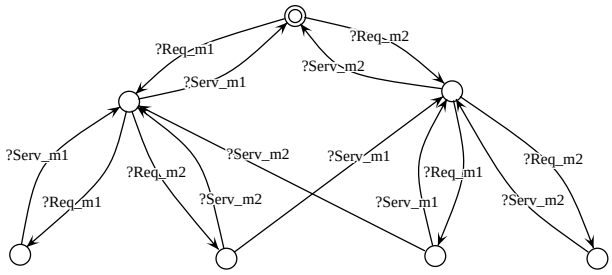


Fig. 2. An example of queue with size 2 receiving 2 requests

V. Conclusion

We have presented techniques for producing finite models representing the behaviour of distributed Java applications. These models encode the message exchanges between remote objects (method calls and responses), and are suitable for using with automatic model checkers or equivalence checkers.

The model generated for each distributed object is a *process*, built as the composition of (communicating) finite labelled transition systems. The key points here are to use compositional models to keep individual LTSs as small as possible, and to use the compositionality features of the semantics (bisimulation congruences) and of the checking tools to master the state explosion.

The model generation procedure is based on the Method Call Graph of the application components. In this sense, it is not restricted to Java code, and could be applied to other language constructs as well. However we rely heavily on the semantics of communication of the Java/ProActive middleware, that provides the primitives for distributed object creation and transparent migration, guaranties the atomicity of remote requests sendings, etc. Application to other frameworks would require the adaptation of the semantics at a similar level.

We also make heavy usage of abstraction and slicing techniques, for keeping our models finite and as small as possible. This is similar to the techniques used in the Bandera system [2] in the case of sequential or multi-threaded Java programs. Yet our models are very different in nature from those of Bandera, since we are interested in the communication between objects, rather than the internal states of programs. In practice, we could not generate our models from the internal representation (BIR) of the Bandera toolset, that is a fully flattened (though symbolic) representation of the state space.

We are currently extending the intermediate formats representing hierarchical labelled transition systems to take into account fully parameterized systems. This gives us models that are closer to the source code (so we have less static analysis work to perform). It also leaves us with the choice of computing afterwards a finite instantiation of the parameterized system before using "standard" model checking tools, or to use directly the new generation of tools that deal directly with parameterized systems.

The models we have presented are hierarchical at the level of activities (plus some limited structuration within each activity). The compositionality of the approach will allow us much higher benefits with the next version of the ProActive middleware, in which activities will become distributed components, with the ability of building bigger components from smaller ones. In this setting, it will be essential to have behavioural specifications of the components, and methods to ensure that compositions of distributed components are correct with respect to those specifications.

We are currently working on the implementation of the analysis platform, and shall report on experimental results on realistic ProActive applications. Future directions of this work include extensions of the *ProActive* communication models, e.g. taking into account group communication and security policies.

REFERENCES

[1] S. K. R. Thomas Ball, "Automatically validating temporal safety properties of interfaces," in *Workshop on Model Checking of Software (SPIN 2001)*, ser. LNCS, no. 2057. Springer-Verlag, 2001.

[2] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, S. Laubach, and H. Zheng, "Bandera: Extracting finite-state models from java source code," *Int. Conference on Software Engineering (ICSE 2000)*, 2000.

[3] D. Caromel, W. Klauser, and J. Vayssière, "Towards seamless computing and metacomputing in Java," *Concurrency Practice and Experience*, vol. 10, no. 11–13, pp. 1043–1061, November 1998.

[4] R. Mateescu and M. Sighireanu, "Efficient on-the-fly model-checking for regular alternation-free mu-calculus," in *5th International Workshop on Formal Methods for Industrial Critical Systems FMICS'2000*, Berlin, Apr. 2000.

[5] R. Boulifa and E. Madelaine, "Model generation for distributed java programs," full version under submission, see our web page at: http://www-sop.inria.fr/oasis/vercors/.

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

[7] V. Roy and R. de Simone, "Auto and autograph," in *Workshop on Computer Aided Verification, New-Brunswick*. LNCS 531, Spring-Verlag, June 1990, pp. 65–75, also available as RR-4460.

[8] D. Binkley and K. B. Gallagher, "Program slicing," *Advances in Computers*, vol. 43:1-50, 1996.

[9] H. Z. Matthew Dwyer, John Hatcliff, "Slicing software for model construction," *Journal of High-order and Symbolic Computation*, 2000.

[10] R. Boulifa and E. Madelaine, "Preuve de propriétés de comportement de programmes proactive," INRIA, Tech. Rep. RR-4460, May 2002, in french.