Model-checking Distributed Components: The Vercors Platform

Tomás Barros[1], Antonio Cansado[2], Eric Madelaine[2] and Marcela Rivera[2]

[1] *Univ. Diego Portales. Ejército 441, Santiago, Chile*
[2] *INRIA Sophia Antipolis, CNRS - I3S - Univ. Nice Sophia Antipolis 2004, Route des Lucioles, BP 93, F-06902 Sophia-Antipolis Cedex - France*
Email :First.Last@sophia.inria.fr

Abstract

This article presents a component verification platform called Vercors providing means to analyse the behaviour properties of applications built from distributed components. From the behavioural specification of primitive components, and from the architectural description of the composite components, our tools build models encoding the interactions between the components, suitable for analysis by model-checking tools. The models are hierarchical and parameterized, expressing in a compact way the system behaviour. Then we have tools for instantiating those parameterized models using finite abstractions, and producing input for state-of-the-art verification tools. Our current work also targets the generation of models that include controllers modelling the dynamic management of architectural transformation of an application, such as changes in bindings or replacement of sub-components. We describe the existing tools, give tracks for further developments and show how realistic case-studies can be model-checked using our platform.

Keywords: Hierarchical components, distributed asynchronous components, formal verification, behavioural specification, model-checking.

1 Introduction

Component programming has gained attention these last years due to the need of more reliable software. Through the reuse of software, designers are able to speed up their development process and avoid falling into unnecessary bugs.

Component programming proposes to split the system into smaller pieces of software that interact through well defined frontiers, called interfaces. In case of the Fractal [6] component model, new components (called composites) can be created by composing existing (and smaller) components in a hierarchical fashion, enhancing the reusability of component libraries.

[0] This research work is carried out under the ACI Securité FIACRE funded by the french government, and under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265) and under the associated team OSCAR funded by INRIA and University of Chile.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Components may be bound through their interfaces if typing requirements are met, ensuring a basic compatibility. Although Fractal clearly exposes both provided and required services (method signatures) of each component through its interfaces, it is well-known that static typing of bound interfaces is not enough for guaranteeing a correct assembly of components: even if they statically match their interfaces, off-the-shelf components may not work together due to the lack of a dynamic behavioural compatibility, resulting in mismatch between their protocols, and often deadlocks.

Expressing a precise behaviour requires a solid mathematical background. Formal methods aim at defining a system without ambiguity, so that formal tools, implementing either theorem proving or model-checking, can be used to bring strong guaranties on the program properties. For being usable in real-life programming environments, these methods require support from automatic software tools which hide the complexity of the underlying logics from the developer. Although we can find success cases in the hardware industry, the usage of formal methods and of verification tools is still limited when it comes to software.

Compared to classical (sequential) software development, the context of distributed components is more complex because of the intricate interaction between the distributed parts of an application. This requires specific care in handling asynchrony, remote failures, communication delays, etc. On the other hand, component frameworks provide both a programming model that helps the developer to abstract away from the details of the underlying execution platform, and tools for supporting these abstractions (description languages, middle-ware layers providing strong behavioural guaranties, etc.).

Amongst the research work being done on the formalisation of distributed component behaviour, the closest to our motivations certainly is the one developed by the Sofa team [17], similar to the Fractal component model. In Sofa, components have a *frame* (specification) and an *architecture* (implementation) protocols, and verification is done through a trace language inclusion of the *architecture* within the target *frame*. In a different flavor, the work of Carrez et al on behavioural typing of components [7] defines a *sound assembly* and compatibility concepts which ensure correctness of the composition, but in a framework based on the Corba component model CCM [15], where they have no hierarchy of components.

There exist a number of verification platforms for process algebras. The only one supporting behaviours of hierarchical components that we are aware of is Sofa; it includes a static analysis module based on the Java Path Finder tool, and a home-made model-checker implementing the Sofa compliance relation. The main difference with our approach is that being based on a trace language semantics, they have no support for (congruent) minimization of state space.

We made slightly different choices: our models are based on bisimulation theories, and we take advantage of their congruence properties, together with the structuring capability of hierarchical components, to keep the state-space complexity of our models manageable. In the last years our work has been concentrated in the automatic building of behavioural models for distributed component systems. In [3] we introduced a new semantic model named pNet extending the networks of communicating automata [1], by adding parameters to their communication events

and processes in the spirit of symbolic transition graphs [13]. In [4] we used pNets to model the behaviour of Fractal (synchronous) components, including the representation of the Fractal "non-functional operations" for the dynamic management of the component assemblies. Then in [5] we extended this work to the distributed implementation of Fractal using the ProActive library, which features asynchronous communication between distributed components.

This paper presents the implementation of those ideas, using the principles described in the previous papers. We are building a platform called Vercors for semi-automatic verification of distributed hierarchical components, integrating our model generation algorithms with the verification toolbox `CADP` [10].

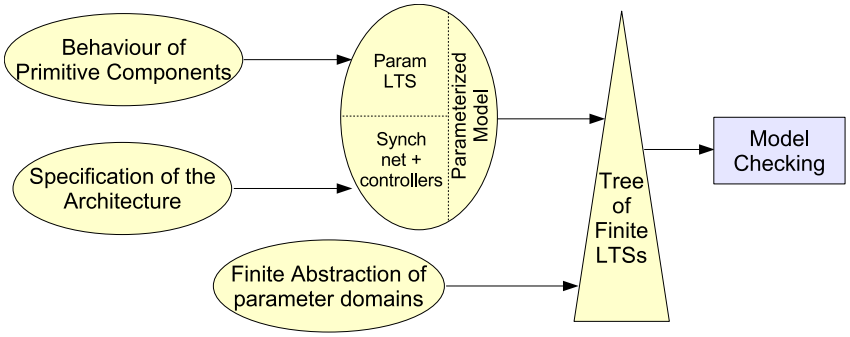

Fig. 1. Principles of the Vercors platform

Fig. 1 presents the principles of our platform: on the left side are the user inputs, consisting of the specification of the functional behaviour for the primitive components, and the architecture description. From these files, the tools build a parameterized model in pNet format. The next step consists in instantiating this model, using finite partitions of the parameter domains, currently provided by the user (though this information could possibly be inferred from the syntax of the property to be proved). This gives us a finite hierarchical network of LTSs, that is finaly translated into the input format suitable for the tools of the CADP verification toolbox.

Note on abstractions: there are two different notions of abstractions in the method supported by our platform. The first one is that the user-provided specifications are based on "simple types" (essentialy enumerated types, integers, and record types), abstracting away from the user-defined classes that occurs in the implementation; providing tool assistance for this level is not addressed in this paper, but it is assumed that they are proper abstract interpretations of the user types. The second abstraction level goes from parameterized to finite systems: a finite instantiation of a parameterized model is an abstraction in the sense of [8]. Starting with first order (countable) data domains, we define abstractions as partitions in which the abstract domain has values corresponding to a finite number of distinguished concrete values, plus one or more extra values representing the rest of the concrete domain. Abstract domains are equiped with abstract operators, and these abstractions define Galois insertions. We apply these kind of abstractions for the domains of value-passing variables, keeping enough abstract values in the abstract domains to represent each distinguished value of the parameter in the formula. In [8] the authors prove that this preserves safety and liveness properties in branching

time logics. We also apply the same abstraction scheme for parameters specifying the topology of our systems; in this case, the [8] theorem is not applicable (the result cannot be generalised to arbitrary values of the parameters), and the diagnostics should be considered with a lot of care, but can still be useful as "debugging hints".

We illustrate the platform functionality through the verification of a realistic case-study provided by France Telecom, previously analysed by the Sofa team in Prague [16]. This example has several qualities that motivated its choice for this paper: first it is extracted from a realistic case-study, that will hopefully be part of a standard set of examples provided to the whole Fractal community for comparisons. Secondly it is small enough for a description in a 15 pages paper, but big enough to illustrate some model size issues. Last it is suitable to illustrate some of the features we wanted to show, including the treatment of parameters, and the modelling of multicast interfaces.

The next section is a short explanation of the case-study, that will be used to illustrate each functionality of the tools in the sequel of the paper. Then we present in details the various tools in the verification chain, and the input and intermediate formats used. In section 4 we sketch the work in progress and the new functionalities planned for the next version, that will most probably be available before the workshop. In section 5 we describe more directions for future work, that will come in complement to the functionalities already in the platform.

2 Case Study Description

The Charles University in Prague, in collaboration with France Telecom, has developed a prototype implementation of a WiFi Internet access system using the Fractal component model [16]. We demonstrate how our Vercors platform can be used to verify parts of the specification and to provide diagnostics for the designer.

For simplicity we focus only on a subset of the system. It consists in a client (identified by an IP address), who tries to connect to a wireless network providing Internet access. Any client accessing a web page should be at first authenticated; if not, the client end ups in a login web page. At the login page, the user is asked either a valid ticket id, or a frequent flyer id (if the user is registered in a frequent flyer programme). Once authenticated, the user is granted access to any web page he desires until his time-lease expires.

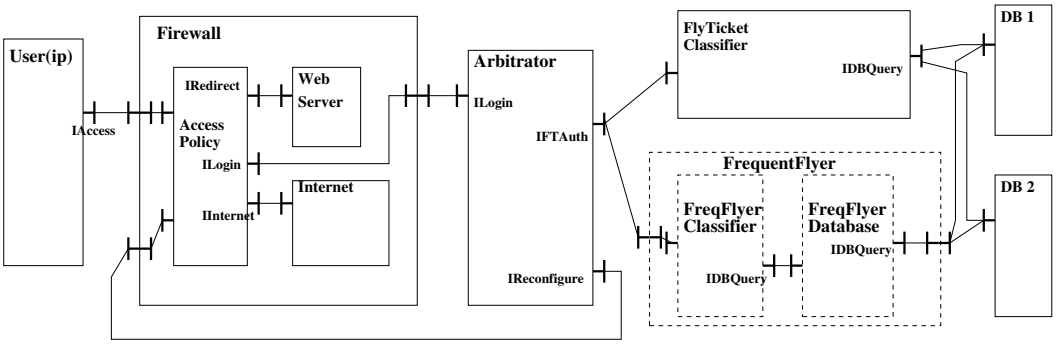

Fig. 2. The airport basic architecture

For the authentication process, the system must query either: 1) the appropriate

airline database for a given ticket id, or 2) all airline databases for a given frequent flyer id. In the first case it returns the requested ticket, otherwise it returns the longest time-lease from the tickets collected during the query.

The architecture of interest can be seen in Fig. 2. The main component inside the Firewall is AccessPolicy, which routes the traffic between the WebServer and the Internet based on the firewall rules.

The Arbitrator is responsible for the authentication process. It queries FlyTicketClassifier for 1) and FrequentFlyer for 2), and then sends back the new access policies to the Firewall.

The ticket id has an airline identifier which is used by the FlyTicketClassifier to query an appropriate airline database, whereas the frequent flyer id does not. Therefore, the FrequentFlyer must broadcast to all databases the given id, gather all the results and select one. The broadcast is handled by the FrequentFlyerDatabase, while the selection of the longer-lasting lease is done by FrequentFlyerClassifier.

3 Platform Overview

Our verification platform comprises several tools for assisting the whole process of verification. Rather than creating a new model-checker, we implement our model-generation methods in a way that will be efficiently integrated with existing state-of-art tools for checking component specifications.

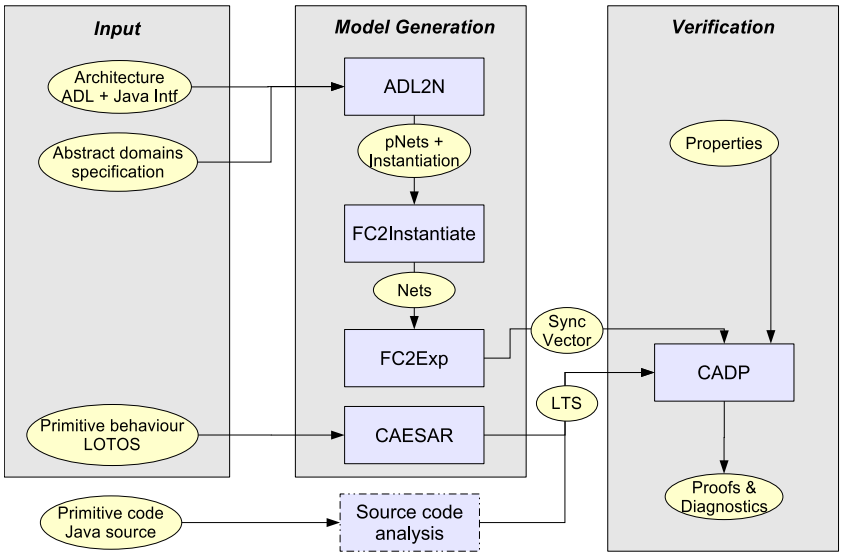

Fig. 3. The VERCORS architecture

Fig. 3 gives a snapshot of the platform. In the next sections we shall describe in details its three parts: the input from the user (3.1), the behavioural model generation (3.2) and the verification of properties (3.3). We illustrate our platform through the formal verification of the previously outlined case study.

3.1 User Input

The architecture shown in Fig. 2 is specified in a XML file using the Architecture Description Language (Fractal ADL). This file specifies the deployment topology of the component system. It defines: the component nature (primitive or composite), its content (primitive code or subcomponents); client and server interfaces of each component; bindings between interfaces at deployment. A small extract of the ADL can be seen in Listing 1.

Listing 1: Part of the ADL file for the Firewall component

```
<component name="Firewall">
 <interface signature="IAccess" role="server" name="IAccess"/>
 <interface signature="IReconfiguration" role="server" name="IReconfiguration"/>
 <interface signature="ILogin" role="client" name="ILogin"/>
 <component name="WebServer">
  <interface signature="IRedirect" role="server" name="IRedirect"/>
  <controller desc="primitive"/>
 </component>
...
 <binding client="AccessPolicy.IRedirect" server="WebServer.IRedirect"/>
 <controller desc="composite"/>
</component>
```

The user should also provide the signatures of the component interfaces. Fractal defines an Interface definition language (IDL), but in the implementations we consider, we use Java interfaces, describing the signatures of the methods of each component Interface.

The last input requested from the user is a description of the functional behaviour of primitive components specified in an automata based language (currently LOTOS).

For instance, Listing 2 shows how the main method encoding the activity of the component (runActivity for ProActive) is modelled by a process (ACCESSPOLICYBODY), which offers synchronisations for every method of its server interfaces. Parameters are given by Sorts ranging over their abstract domain and interfaces are encoded by gates where the synchronisation offers take place.

Listing 2: AccessPolicy.lotos

```
behaviour ACCESSPOLICYBODY[IACCESS,IRECONF,IREDIRECT,...]({} of Rules)
where
process METHOD_GET[IREDIRECT,IACCESS,...](rules: Rules, ip:IP, m:MethodGet) : exit(Rules) :=
    let url:URL = getURL(m) in
        [(url eq url(0)) or (ip NotIn rules)] ->
            (   IREDIRECT !C(redirectPage(ip)) of IRedirect;
                IREDIRECT !ip !url(0);
                IACCESS !ip !url(0);
                exit(rules)    )
        [] [(ip IsIn rules)] ->  ...
endproc
process ACCESSPOLICYBODY[IACCESS,IRECONF,IREDIRECT,...](rules: Rules): noexit :=
     (( choice ip:IP [] choice url:URL []
          ( IACCESS !ip !C(get(url)) of IAccess;
            METHOD_GET[IREDIRECT,IINTERNET,IACCESS](rules,ip,get(url)) )
     ) []
      ( choice ip:IP []
          ( IRECONF !C(DisablePortBlock(ip)) of IReconfiguration;
            METHOD_DISABLEPORTBLOCK(rules,DisablePortBlock(ip)) ))
...
```

LOTOS enables us to express transition systems with parameters, and is a natural choice for interfacing with the CADP toolset. Yet many other automaton-based language could be used here and we are envisaging to provide a notation that would be easier for non-specialists.

3.2 Building the Model

We take a two-fold approach: 1) the architecture and hierarchy information is extracted from the ADL analysis, and 2) each of the primitive's functional behaviour is specified by the Vercors user in an automata based language.

We start the analysis with the description of a composite component, specified in an ADL file. The user can select a node in the component tree, and the tool will analyse from this layer down through the hierarchy.

3.2.1 Parameterized Model

The construction begins by giving the ADL file to `ADL2N`, which is a tool written in Java for the ADL analysis of Fractal components. `ADL2N` obtains the possible actions on the component's interfaces by introspection of the Java interfaces provided as the signature of the component's interfaces in the ADL file. `ADL2N` searches the classes of such Java interfaces in its classpath.

The output of the ADL analysis is a hierarchical synchronisation network with parameterized actions [3]. A synchronisation network is a form of generalised parallel operator, where each of its arguments is typed by a Sort that is the set of its possible observable actions. The allowed interactions between the sub-components are given by a synchronisation constraint, namely a finite set of synchronisation vectors, which is a subset of the Cartesian product of the sub-component's interface actions. The sub-components behaviours are the actual automata filling the network's arguments.

Our *parameterized actions* have a rich structure, and they can encode value passing in the communication actions, assignment of state variables, and indexes denoting process families.

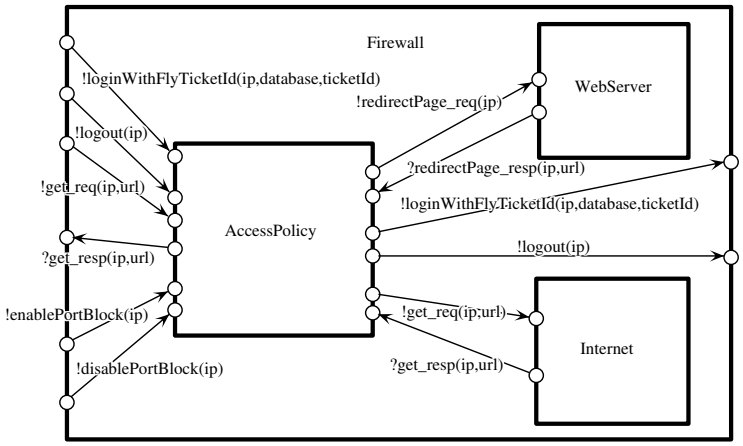

Fig. 4. Firewall pNet obtained by ADL analysis

For instance, the network resulting of the ADL analysis for the component Fire-

wall in Listing 1 is shown in Fig. 4. The arrows herein represent synchronizations between pNets, and you can observe that methods invocations that return results are encoded by two separate edges with corresponding events (e.g. `get_req(ip,url)` and `get_resp(ip,url)`).

For addressing as early as possible the well-known issue of state-explosion, we include in `ADL2N` a number of hiding and abstraction techniques. Hiding is used by individually selecting which method calls the user wants to observe, while the others become unobservable. Data domain abstraction has been discussed in the introduction.

In practice, the user of `ADL2N` will use the tool GUI to specify at the same time the methods that will be visible, the parameters that are significant, and the finite instantiations of those parameters. `ADL2N` then creates two files: a pNet in hierarchical Parameterized FC2 format expressing the synchronisations between components and an instantiation file defining the domain abstractions.

The FC2 format was created in the nineties as a mean of communication between several software tools in the process algebra area [14]. It allows for description of labelled transition systems and networks of such. Automata are tables of states, each state being in turn a table of outgoing transitions with target indexes; networks are vectors of references to subcomponents (i.e., to other tables), together with synchronisation vectors (legible combinations of subcomponent behaviours acting in synchronised fashion). Subcomponents can be networks themselves, allowing hierarchical description.

We have defined Parameterized FC2 as an instance of FC2, using the extension mechanism of the FC2 syntax and adding elements for parameters and operators (reference manual in [2]). It is a textual format for compatibility reasons, but not intended to be written by humans. It has the advantages of being structured and compact, and that one given parameterized model could later be transformed using different instantiations.

3.2.2 Finite model, and choice of an instantiation

The next step is to instantiate the parameterized model using finite abstract domains for its parameters. This abstraction is defined by the user during the interaction with `ADL2N`.

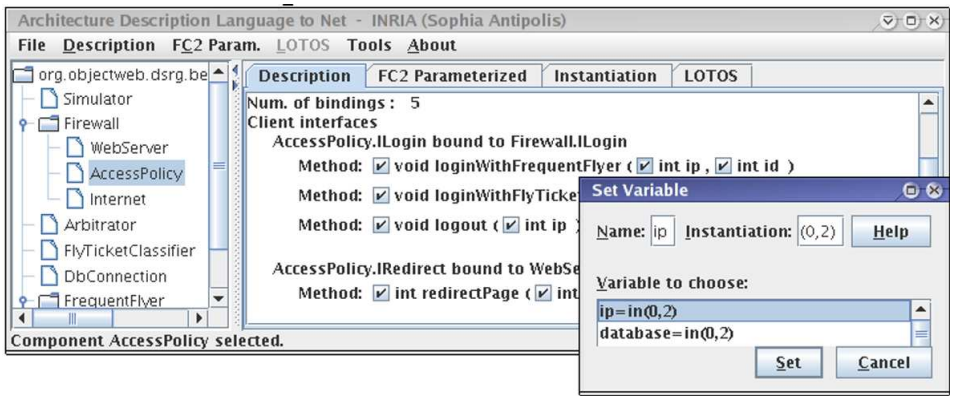

Fig. 5. The ADL2N tool

For simplifying the presentation in this paper, we consider a system with only one user (this is _not_ an abstraction of a multi-user system). For the other parameters, we follow the finite abstraction principle described in the introduction, for which abstract domains represent full partitions of the domains. There are 3 distinguished values for the possible web pages `url`: one for the login page, one for the web page requested by the user in the Internet and a third one for every other possible web page. We shall consider only 2 ticket `ids`, one encoding a distinguished ticket given by the user and the other for remaining tickets. Since only the distinguished ticket is valid, then the authentication is granted only if the system successes in finding this ticket. For the validity of the tickets, a boolean representing a valid/invalid ticket is enough as we do not take into consideration the time control. Similarly, the system has two databases, one representing the database with the ticket needed.

The parameterized network is the input for the tool `FC2Instantiate`. Given a system of communicating LTSs with parameters and the domain of its unbound parameters, `FC2Instantiate` is a Java tool that generates a finite system of communicating automata by translating each of the parameters to all the values in its domain. The output is in FC2 format.

3.2.3 Interoperability with Verification Tools

Up to now we have described several transformations and tools, from the user-level specifications to the behavioural semantics, and to a finite instantiation, using our pNets formal model. Now we connect to the input format of various verification tools.

We use several engines from the `CADP` toolbox in our verification process. Being based on process algebras theory, the `CADP` toolbox provides among many others: a compiler for high-level protocol descriptions written in the ISO language LOTOS, on-the-fly capabilities, distributed space-generation and several diagnostics. All these features, although not originally intended at component verification, fit nicely within our platform for such purposes.

We provide a Java tool called `FC2EXP`, that translates our sets of synchronisation vectors from FC2 format to hierarchical EXP format [12], the format for synchronisation vectors defined by `CADP`.

The LOTOS specifications of primitive components are compiled into LTSs using the `CAESAR` tool of `CADP`. In addition, we need to map the names of the LOTOS gates and offers with the actions specified in the pNets. Currently this is not done automatically by our tools and must be provided by the user.

Finally, the set of LTSs containing the functional behaviour of primitives is synchronised with the synchronisation vectors created earlier, and the behaviour of the full system is built and verified using `CADP`.

The full functional behaviour is easily generated in a single desktop machine (Pentium 4 3GHz, 1GB RAM). This model was built with all remote method calls kept visible, Ignoring the components drawn in dash lines of the Fig. 2, the LTS of the system has 2152 states with 6553 transitions (non-minimised) and 57 states with 114 transitions (reduced), both with 17 visible labels, while the biggest primitive component has 5266 states with 27300 transitions. One could also reduce the size of the various parts of the models, taking into account the set of actions that

occur in the formulas to be proved, and using environment interfaces to reduce the intermediate construction sizes. Finally, for bigger systems, it is possible to use the on-the-fly facilities of CADP for the state space generation, or even to compute the state space in a distributed manner on a computation grid.

3.3 Verification

We give here two examples of verification, the first one being a straight-forward deadlock detection, and the other a functional verification of the firewall. For the first one there is no need of specifying a formula, whereas in the second case a custom property must be supplied.

3.3.1 Property 1 : Deadlock Detection

Listing 3: Diagnostics

```
<initial state>
""loginWithFlyTicketId(IAccess)(0,1,1)""
""loginWithFlyTicketId(ILogin)(0,1,1)""
""loginWithFlyTicketId(IAccess)(0,1,1)""
""CreateToken_req(IFTAuth)(1,1)""
""GetFlyTicketValidity_req(IFTAuth)(1,1)""
""GetFlyTicketValidity_resp(IFTAuth)(1,1)""
""CreateToken_resp(IFTAuth)(1)""
<deadlock>
```

In this context, our User component may login, logout or request a web page in any order at any time. We check whether a client, before being authenticated, is indeed routed to the WebServer no matter which page is requested, and granted access to the Internet otherwise. We were able to find a deadlock in our model, and the corresponding diagnostic is shown in Listing 3. Breadth-first-search is used to find the shortest trace. We believe our platform is able to exhibit more compact traces than the ones found by Sofa [16], that contain interleaved actions not related to the deadlock.

3.3.2 Diagnostic Explanation

We modelled the login method as being void, which means no confirmation is sent back to the user with his new status, thus it is natural that the client logs in again. Supposing he does log in twice, the Arbitrator, still processing the first login call, queries the database for the ticket information, then tries to send back the new access policy to the Firewall. The problem arises because the latter is blocked trying to make a second login call to the Arbitrator. It is a clear deadlock, yet realistic and difficult to detect without exhaustive tools, that can be model-checked with this methodology.

Remark that this interesting result was detected in a synchronous call context. A ProActive implementation of the system would not deadlock due to asynchronous services provided by the library. As in our current version of ADL2N asynchronous mechanisms are not modelled, the analysis of this scenario is not yet possible. We shall relieve this limitation as soon as the new version of ADL2N (see section 4) is released.

3.3.3 Property 2 : Inevitability
Here we use a μ-calculus formula to specify that a user, requesting access to a given web page (encoded "1" in the formula), will inevitably get it:

```
[true*.'"get_req(IAccess)(0,1)"']
      mu X . (< true > true and [ not '"get_resp(IAccess)(0,1)"' ] X)
```

The model-checker answers "false", and the diagnostics is shown in Fig. 6 by a LTS encoding a counterexample.

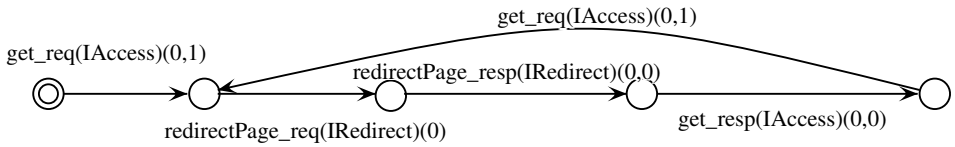

Fig. 6. Counterexample where the user does not receive the requested web page.

3.3.4 Diagnostic Explanation
The counterexample found reflects the user requesting a distinguished web page before logging in. The user is blocked and redirected to the local web server, who sends back the login page (given by the number 0).

4 Work in progress

Despite being capable of building the full functional behaviour, two strong limitations can be seen in `ADL2N v0.8`: it does not include non-functional aspects, thus deployment and reconfiguration errors cannot be checked; and asynchronous semantics of the ProActive library are not considered. Handling these requires the addition of specific controllers as described in [4] and [5], which can be derived from the analysis of the ADL and Java interfaces.

The new version of the `ADL2N` tool includes a two-phases compilation of ADL files. First, we create an abstract model encoding the common semantics between different Fractal implementations. Then, all implementation-specific controllers are added depending on the component model the user selects.

In the tool back-end, the pNets models are compiled into Parameterized FC2. Higher-level communication primitives are translated by creating intermediate controllers, similar to our non-functional controllers. As a generic pNets compiler will be used, adding support for new controllers does not require any change in the compiler, and extending the model is straight-forward.

Through these new design considerations, we believe our tool is easily extensible to new Fractal implementations and future model-checkers. We will provide the user with a tool for verifying synchronous and asynchronous implementations of the Fractal model, and keep the user free to define other implementation-specific controllers of interest.

5 Future work

Various tools for static analysis, model checking, and equivalence checking can operate on the models we generate. One long term goal of this work is to integrate the

various techniques and tools involved in this software platform, so that the platform can be integrated in a development environment, and used by non-specialists. At the same time, the platform must be flexible and open enough to serve as a basis for easy prototyping of new techniques and tools on real Java/ProActive code.

Of particular interest are tools that deal directly with some classes of parameterized or infinite models. We expect to study the connection with tools such as `FAST` or `TReX`, from the Persee Project [11]. Integrating these techniques into our models is not trivial, but they could allow infinite (yet regular) systems to be checked directly from the parameterized models, avoiding the state-explosion that occurs during the instantiation.

In the domain of Grid computing where ProActive is positioned, multicast interfaces take a major role and we expect that most large-scale applications exploit these features. Therefore, they must be included in the model, and hopefully into a higher-level specification language where architectural and communication primitives are provided, so that real scenarios of distributed component verification are possible and feasible for the users. In our case-study, this mechanism is illustrated by the FrequentFlyer scenario (the dashed components in Fig. 2): the FrequentFlyer component uses broadcast and gathercast primitives to ask information to several databases in parallel.

It is important to note that pNets can be used to model higher level communication primitives such as multicast and gathercast: it will be easy to add these functionalities to the `ADL2N` tool, when these primitives will be more stable in the Grid component model currently being defined (in the CoreGRID NoE).

Last we need more user-friendly methods for the specification of logical properties. `CADP` performs on-the-fly model-checking of regular alternation-free μ-calculus formulas. Although powerful, defining these formulas is error-prone and not suitable for non-experts (see property in 3.3.3). Therefore, following the work done by Dwyer et al. [9], we intend to define natural language-like patterns for component-based specification such as `AfterDeployment` for meaning the temporal scope after a successful deployment, `NoErrors`, `ControlActions` and `FutureUpdate` for grouping specific actions.

6 Conclusion

Throughout this article, we have introduced our distributed component verification platform, with hope of bringing formal verification of system's designs to non-specialists users. We provide models and tools to automate and hide the complexity of formal methods, supporting the designer from user specifications up to the diagnostics.

The main contributions of this work are:

- An architectural presentation of the Vercors platform.
- A description of how we have implemented the model generation methods described in previous papers in semi-automatic tools, including the input and intermediate formats based on our pNets formal model, and the connections we made with the `CADP` verification toolset.

- A step-by-step example using the current version of the tools, for the model-checking of a realistic case study.
- A discussion on what still needs to be implemented before being able to check properties about deployment and transformation of a component-based application.
- A discussion of research directions for enhancing the scope of the platform, including: new types of verification engines, the addition of high level primitives specific to distributed grid computation, and the use of user-friendly languages for expressing user requirements.

Although our platform is not yet complete, we believe it is a step towards a realistic automatic component verification framework. Many intermediate formats are used through the process, but the platform is evolving to relieve the overhead of manipulating these complex languages through the use of automatic or quasi-automatic tools that we provide. We expect to attract ProActive users by providing useful diagnostics of non-trivial design flaws.

References

[1] A. Arnold. Nivat's processes and their synchronization. *Theor. Comput. Sci.*, 281(1-2):31–36, 2002.

[2] T. Barros. *Formal specification and verification of distributed component systems.* PhD thesis, Université de Nice - INRIA Sophia Antipolis, November 2005.

[3] T. Barros, R. Boulifa, and E. Madelaine. Parameterized models for distributed java objects. In *Forte'04 conference*, volume LNCS 3235, Madrid, September 2004. Spinger Verlag.

[4] T. Barros, L. Henrio, and E. Madelaine. Behavioural models for hierarchical components. In Patrice Godefroid, editor, *Model Checking Software, 12th International SPIN Workshop*, volume LNCS 3639, pages 154–168, San Francisco, CA, USA, August 2005. Springer.

[5] T. Barros, L. Henrio, and E. Madelaine. Verification of distributed hierarchical components. In *International Workshop on Formal Aspects of Component Software (FACS'05)*, Macao, October 2005. Electronic Notes in Theoretical Computer Science (ENTCS).

[6] E. Bruneton, T. Coupaye, M. Leclercp, V. Quema, and J. Stefani. An open component model and its support in java. In *7th Int. Symp. on Component-Based Software Engineering (CBSE-7)*, LNCS 3054, may 2004.

[7] Cyril Carrez, Alexandro Fantechi, and Elie Najm. Behavioural contracts for a sound assembly of components. In *Forte'03 conference*, number 2767 in LNCS, Berlin, 2003.

[8] R. Cleaveland and J. Riely. Testing-based abstractions for value-passing systems. In *Int. Conference on Concurrency Theory (CONCUR)*, volume 836 of *Lecture Notes in Computer Science*, pages 417–432. Springer, 1994.

[9] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. 21st International Conference on Software Engineering*, pages 411–420. IEEE Computer Society Press, ACM Press, 1999.

[10] 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.

[11] A. Griffault, F. Herbreteau, G. Point, G. Sutre, A. Vincent, M. Sighireanu, S. Bardin, A. Finkel, and D. Nowak. Intégration des outils PERSÉE. Technical report, Livrable RC1, Projet PERSÉE de l'ACI Sécurité Informatique, April 2005.

[12] F. Lang. Exp.open 2.0: A flexible tool integrating partial order, compositional, and on-the-fly verification methods. In *Proceedings of the 5th International Conference on Integrated Formal Methods IFM'2005*, Eindhoven, The Netherlands, 2005. LNCS 3771.

[13] H. Lin. Symbolic transition graph with assignment. In U. Montanari and V. Sassone, editors, *CONCUR '96*, Pisa, Italy, August 1996. LNCS 1119.

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

[15] OMG. Corba components, version 3. Document formal/02-06-65, June 2002.

[16] F. Plasil P. Jezek, J. Kofron. Model checking of component behavior specification: A real life experience. In *International Workshop on Formal Aspects of Component Software (FACS'05)*, Macao, October 2005. Electronic Notes in Theoretical Computer Science (ENTCS).

[17] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11), nov 2002.