Behavioural Models for Hierarchical Components

Tomás Barros, Ludovic Henrio, Eric Madelaine

Behavioural Models for Hierarchical Components

Tomás Barros, Ludovic Henrio[2], and Eric Madelaine

INRIA Sophia Antipolis, CNRS - I3S - Univ. Nice Sophia Antipolis
2004, Route des Lucioles, BP 93, F-06902 Sophia-Antipolis Cedex - France
[2] Univ. of Wesminster, Watford Rd Northwick park, Harrow, HA1 3TP, UK
First.Last@sophia.inria.fr

Abstract. We describe a method for the specification and verification of the dynamic behaviour of component systems. Building applications using a component framework allows the developers to specify the architecture, the deployment, the life-cycle of the system with well-defined formalisms, and to gain productivity by reusing existing components.
But then one wants to make sure that the application built from existing components is safe, in the sense that its parts fit together appropriately and behave together smoothly. Each component must be adequate to its assigned role within the system, and the update or replacement of a component should not cause deadlock or failure of the rest of the system. The usual notion of type compatibility of interfaces is not sufficient; we need to capture the dynamic interaction between components, and typically to avoid deadlocks or unexpected behaviours in the system.
In this work, we focus on hierarchical component systems. We describe both the functional behaviour and the non-functional features (life-cycle management) of components in terms of synchronised transition systems; we define a notion of correct component composition; then we show how we can prove, using (compositional) model-checking techniques, temporal properties of a component system. Transformations of a system, for example replacement of a sub-component, are expressed as transformations of its behavioural semantics, allowing to prove preservation of some properties, or the validity of new properties after transformation.

1 Introduction

Components have emerged as a new programming paradigm in software development. Beyond structuring concepts inherited from modules and objects, component frameworks provide means for architecture and deployment description. Some frameworks define a number of non-functional features for controlling the life-cycle of the components and the application, or allow for construction of distributed components. In general words, a component is a self contained entity that interacts with its environment through well-defined interfaces (provided services and required functionalities to be provided by other components). Besides these interactions, a component does not reveal its internal structure.

In hierarchical component frameworks like Fractal [6], different components can be assembled together creating a new self contained component which can be itself assembled to other components in a upper level of hierarchy. Hierarchical components make visible the hierarchy of the system and hide, at each level, the complexity of the sub-entities. The compositional aspect together with the separation between functional and non-functional aspects helps the implementation and maintenance of complex software systems.

The challenge that we want to address in this work is to build a formal framework to ensure that compositions are correct. Standard components systems have typed interfaces, that ensure some level of static compatibility between the components: interfaces are bound only if their operations have compatible types in the classical sense (OO method typing). This does not prevent assembled components from having non compatible behaviours, that could lead to deadlocks, live-locks, or other kinds of safety problems. A number of recent works do try to address better dynamic guaranties, e.g. research on behavioural typing or contracts [7], as well as frameworks like Wright [1] or Sofa [12].

Our approach is to give behavioural specifications of the components in the form of hierarchical synchronised transition systems. The semantics of a composite is then computed as a product of the LTSs of its sub-components with the controller of the composite. This system can be checked against requirements expressed as a set of temporal logic formulas, or again as an LTS. We aim to provide the final user with tools to verify the behaviour at the design phase (definition), the assembly phase (implementation), as well as the dynamic reconfiguration (maintenance) of the component system. Therefore the intended user of our framework is the application developer in charge of those tasks. In this work, we choose to rely on the Fractal hierarchical component model.

The models for the functional behaviour of basic components may be derived from automatic analysis of source code (involving adequate data abstraction), as we have described in [4], or expressed by the developer in a dedicated specification language, e.g. the graphical language for synchronised automata used in this paper.

Our main contributions in this paper are:

- a methodology for building behavioural models of hierarchical components, including non-structural reconfiguration operations,
- the modelling of the full behaviour of the application as a hierarchy of parameterized LTSs,
- a specification of structural reconfigurations as transformations of the LTS expressing the component behaviour,
- a classification of correctness properties for a component system together with tools allowing and easing their verification.

Our final target is distributed component systems communicating asynchronously. We have shown in [3] how we build models for distributed objects and verify their properties; in this paper we concentrate on the modelisation of the control and transformation operations of hierarchical components, and

we leave for further work the integration with the asynchronous communication semantics. One example of distributed implementation of Fractal is given in [5].

In Section 2 we present the features of Fractal that will be useful for the understanding of this paper, and we introduce a small example that will serve as an illustration for the rest of the paper. Section 3 discusses the notion of correct behaviour. In section 4 we recall the main features of the formal models that we defined in [4]. Section 5 develops, step by step, the formalisation and the behaviour computation of this example, starting with the specification of base components, then building the composite controllers, computing the composite behaviour, specifying errors, dealing with deployment and transformation phases, and finally proving in Section 6 some properties of the assembly.

2 The Fractal Component Model

The Fractal component model provides an homogeneous vision of software systems architecture with a few but well defined concepts such as component, controller, content, interface, binding. It is recursive – components structure is auto-similar at any arbitrary level (hence the name 'Fractal'); it is completely reflexive, i.e., it provides introspection and inter-cession capabilities on components structure.

2.1 Guidelines to Fractal Components

A Fractal component is formed out of two parts: a *controller* and a *content*. The content of a component is composed of (a finite number of) other components, called *sub-components*, which are under the control of the controller. This allows for hierarchic components, in the sense that components may be nested at any arbitrary level. A component that exposes its content is called a *composite* component. A component that does not expose its content, but at least one control interface, is called a *primitive* component.

The controller of a component can have *external* and *internal* interfaces. A component can interact with its environment through *operations* at its external interfaces, while internal interfaces are accessible only from the component's sub-components.

Interfaces can be of two sorts: *client* and *server*. A server interface can receive methods invocations while a client interface emits methods call. A *functional* interface provides or requires functionalities of a component, while a *control* interface is a server interface that corresponds to a "non functional aspect", such as introspection, configuration or reconfiguration.

A component controller encodes the control behaviour associated with a particular component. Fractal defines three basic (optional) levels of control capabilities for a component: no control at all, introspection, and configuration. Only the latter is of interest to us. At the configuration control level, Fractal proposes four control interfaces:

– Attribute control: provides operations to get and set attribute values of the component.
– Binding control: provides operations to bind and unbind the component client interfaces.
– Content control: provides operations to add and remove sub-components into/from the component.
– Life cycle control: provides operations to stop and start the component, as well as to get its current status (stopped/started).

The Fractal specification defines a number of constraints on the interplay between functional and non-functional operations. In particular :

– Content and binding control operations are only possible when the component is stopped.
– When started, a component can emit or accept invocations. Note that this does not prevent control operations to throw an error (exception) because of an unstable state.
– When stopped, a component do not emit invocations and must accept invocations through control interfaces ; whether or not an invocation to a functional interface is possible is undefined.

Other features are left unspecified in the Fractal definition, and may be set by a particular Fractal implementation, or left to be specified at user level. For this paper, we make the following choices: (1) the start/stop operations are recursive, i.e. they affect the component and each one of its sub-components; (2) functional operations cannot fire control operations. (3) the controller (membrane) of composites is only a forwarder between external and internal functional interfaces without any other control capability; The last feature implies that there is exactly one internal interface for each external interface of a composite.

2.2 Component System Example

In this section we introduce a particular component system as an example, which we will use later to better explain our work. Fig. 1 is a graphical view for it.

The example is built from three *primitive* components (**A**, **B** and **Logger**), which are composed in two levels of hierarchy defined by two *composite* components (**C** and **System**). Each component exposes the interfaces for the control operations they support (in our example all the components support life-cycle control operation through the interface I_{lf} and binding control operations through the interface I_{bc}).

All the functional interfaces in the example are typed either by the type **I**, the type **L**, or the type **R**. We define the type **I** having the operation foo(), the type **L** having the operation log() and the type **R** having the operation reset().

The system is deployed in a bottom-up fashion from the innermost components to the outer component (**System** in our example). At each level of hierarchy a specific deployment is applied. For instance, at the **C** level of hierarchy in Fig. 1 the deployment includes, among others, the binding between the interface I_{c} of **A** and the interface I_{p} of **B**.

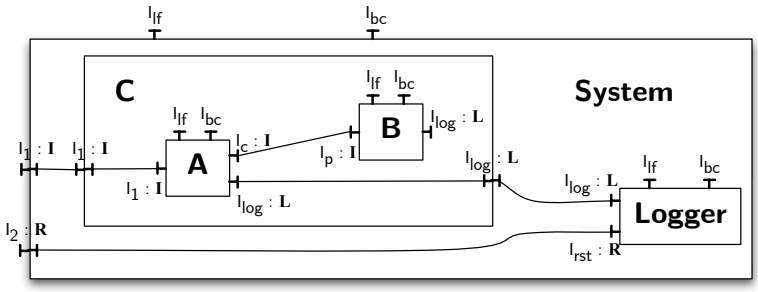

Fig. 1. A simple component system

3 Defining Correct Behaviour

Control (i.e., non-functional) operations can introduce changes on the component behaviour. For instance, adding or replacing a sub-component may add features (new actions) to the system. A set of control operations is called a *transformation phase*.

We make the assumption (this is a restriction with respect to the Fractal specification) that no functional operation can fire control operations. Then we are interested in three phases in the components behaviour:

1. Deployment: this is the building phase of a component. In this phase the component's content (its sub-components) is defined as well as the initial transformation phase (sequence of control operations), as described usually in the application ADL. The application deployment typically ends with a recursive start operation.
2. Running phase: only functional operations occur here.
3. Reconfiguration: we distinguish between non-structural reconfigurations (life cycle and binding controls) and structural transformations (adding, removing or updating components).

From these definitions, we discuss the *correctness* of the component system:

1. *Initial composition*: "Is the deployed system behaving correctly?". The concept of "correct behaviour" covers the absence of dead-locks and in general safety and liveness properties (common sense properties like not using an unbound required interface, or any user-requirement expressed as a temporal logic property). Ultimately, it could be "Does this implementation respect a pre-defined specification? (with respect to some implementation pre-order)".
2. *Reconfiguration*: "After a transformation phase, does the system behave correctly?". This covers both preservation of some properties valid before the transformation, and the satisfaction of a new set of properties, corresponding to features added by the transformation. These proofs must take into account the intricate interplay between functional and non-functional actions during transformation, like the management of the internal state of subcomponents.

For example, one can expect to be able to prove the safety and transparency (from the user point of view) of the replacement of a components by another one.

We want to provide the user with tools that help answer those questions **before** deploying the application or applying a transformation, so he can be confident about the reconfigurations he will apply and therefore, have a reliable system.

4 Formalism

In [4] we have defined a parameterized and hierarchical model for synchronised networks of labelled transition systems. We have shown how this model can be used as an intermediate format to represent the behaviour of distributed Java applications, and check their temporal properties.

Our model is an adaptation of the *symbolic transition graphs with assignment* of [11] into the *synchronisation networks* of [2]: we extend the general notion of Labelled Transition Systems (LTS) and hierarchical networks of communicating systems (synchronisation networks) by adding parameters to the communication events in the spirit of [11].

We model the behaviour of a process as a parameterized Labelled Transition System (pLTS). We use parameters both for encoding data in value passing messages and for manipulating indexed families of processes.

Then we use a parameterized Net to synchronise a finite number of processes. A parameterized Synchronisation Network (pNet) 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 actions to be synchronised between the arguments of the Net are encoded in a *transducer* automaton. A state in the transducer defines a particular set of synchronisations, then a state change in the transducer introduces a new set of synchronisations, i.e. it models a dynamic reconfiguration. A Net with a unique state is called a *static* Net.

Given a finite instantiation of the parameters in the model, we have introduced in [4] an automatic procedure producing a (hierarchical) finite instantiation of the parameterized LTSs and Nets. Having done the instantiation, we can generate the synchronisation product, which is an LTS encoding the full behaviour of the Net when synchronising the actions of its processes as defined in the transducer. Since the synchronisation product is an LTS, it can be used as an argument in a upper Net definition. In other words, we do support hierarchical composition of processes.

Our formalism fits nicely in the components model. The behaviour of a primitive component is a LTS, that can be specified by the developer, or derived from code analysis. For a given composite, its content is the arguments of the Net and its initial bindings are encoded in the initial state of the transducer. The LTS of a composite encodes the functional behaviour of the component but also the control operations that do not change the geometry of the composite, namely

start/stop, and bind/unbind operations. In the sequel, we define our transducers using a set of small automata, that we call *controllers*. On this model, we can check all properties during and after the "initial composition", and involving reconfigurations only relying on start, stop, bind, and unbind.

We deal with transformations that change the arity of the Net or the structure of the application (add/remove/update of components) as transformers of the model: starting with a hierarchical model in a given state, we build a new model after a sequence of transformations, in which we maintain the state of the components that were not changed by the transformation. We can then check for the properties (preserved or new) of the reconfigured system.

5 Building the Behaviour for the Example

We start building an automaton for each component encoding both its functional and non-functional (control) behaviours. In this section, we show how we build the controller automata, in a bottom-up fashion, for primitives and for composite components.

To benefit from the compositional properties of our models, we define this construction in the context of a given temporal logic formula, or more generally for a given set of actions that the user wants to observe. Then we shall consider *abstract* automata for a given family of *hidden actions* (renamed as τ actions), or conversely for a given family of *visible actions* (all others are hidden), minimised by weak bisimulation at each step of the construction.

In particular, specific models can be constructed to exhibit to check the correct detection of some classes of errors.

5.1 Computing Controller Automata

We use a general purpose *Controller* as a scheme from which we get instantiations to specific components.

In the general purpose Controller shown in Fig. 2, we have a finite number k of sub-component automata ($\mathbf{SubC^k}$), a life-cycle automaton ($\mathbf{LF}$), a finite number np of external ($\mathbf{E\_PI^{np}}$) and internal ($\mathbf{I\_PI^{np}}$) provides interface automata, and a finite number nr of external ($\mathbf{E\_RI^{nr}}$) and internal ($\mathbf{I\_RI^{nr}}$) requires interface automata . Synchronised actions are encoded by links between processes (in the graphics we use an ellipse when more than two actions are synchronised).

To obtain the *Controller* for a component we instantiate the general controller, using the sub-components and interfaces that the component defines. For instance, for the component $\mathbf{C}$, the set $\{\mathbf{SubC^k}\}$ becomes $\{\mathbf{A}, \mathbf{B}\}$. Since we build the controller automata in a bottom-up fashion, the controllers of the sub-components have been already built when we compose them. The automaton encoding the sub-component behaviour is the controller of this sub-components, hiding the internal functional actions that are not specified as visible.

Please remark that this instantiation fixes the set of sub-components and internal/external interfaces. The resulting pNet is still parameterized, and its actions contain variables for value-passing and for reference-passing.

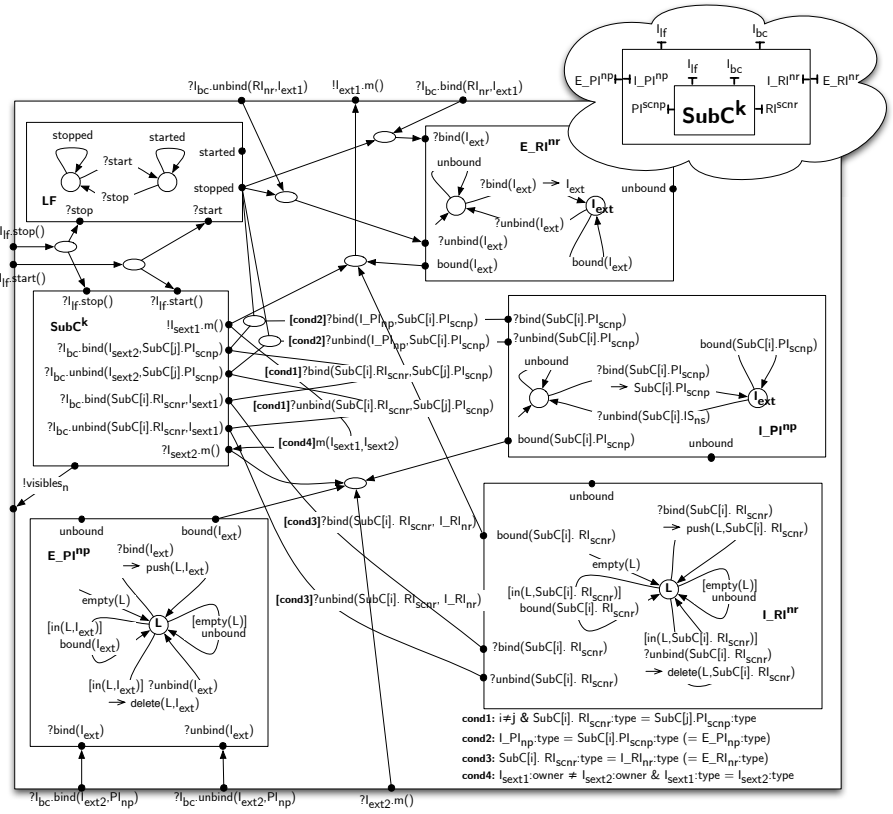

Fig. 2. General purpose Controller

For a primitive component, the set $\{\mathbf{SubC^k}\}$ is reduced to a single automaton which encodes its functional behaviour; the set of internal interfaces ($\{\mathbf{I\_PI^{np}}\}$ and $\{\mathbf{I\_RI^{nr}}\}$) is empty. The functional behaviour automaton encodes calls and receptions of methods on the component interfaces (in addition to internal actions). Whether it is obtained by source analysis or given by the user is outside the scope of this paper.

Because of space restrictions we only present the controller for component **A** which is the product of the 5 LTSs composing the pNet shown in Fig. 3. The controller for the composite component **C** is displayed in annexe A. The full version of this paper is available at http://www-sop.inria.fr/oasis/Vercors/.

In Fig. 3, I_{ext*} are variables encoding the set of external interfaces to which the interfaces of **A** can potentially be bound. This set is instantiated at the next level of hierarchy by type matching analysis (conditions **cond1**, **cond2** and **cond3** in the general controller). For instance when building the controller of **C**, the variable I_{ext1} in the figure becomes the set $\{\mathsf{B.I_P}\}$.

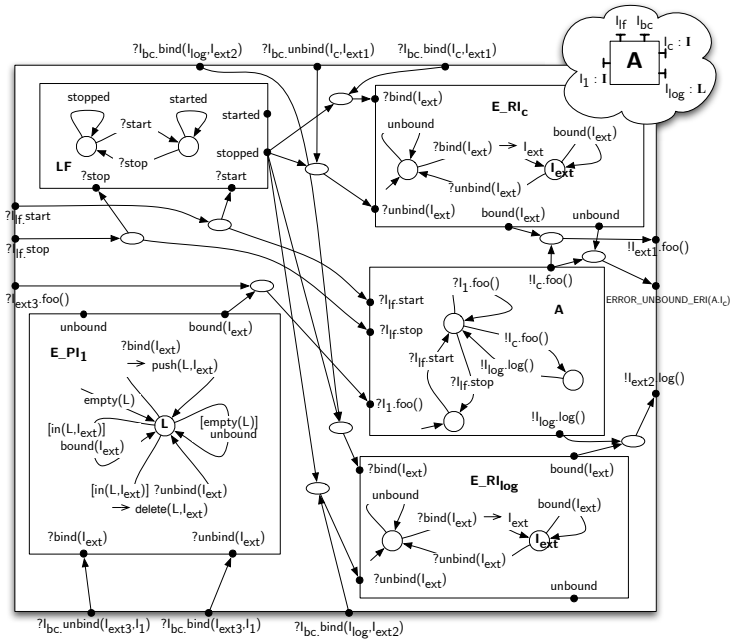

Fig. 3. Controller for **A**

In addition, the controller pieces in Fig. 3 include some of the constraints introduced in Section 2, e.g. that the bindings of requires interfaces are only possible when the component is stopped or that calls to requires interfaces are only possible when these interfaces are bound.

5.2 Detecting Errors

We can introduce in our model the detection of common sense errors (undesired behaviours) introduced in Section 3. For instance, by triggering an ERROR_UNBOUND message upon a call to the operations of the interface $\mathsf{I_{log}}$ when it is unbound, we can detect the erroneous uses of the $\mathsf{I_{log}}$ interface. This is shown in Fig. 4.

In addition to common sense errors, others undesired behaviours are directly or intrinsically defined in the Fractal specification. In order to keep simplicity and clarity during our guided example, we will consider only the error consisting in calling an operation on an unbound interface.

5.3 Species of Temporal Properties

All the temporal properties (that do not involve a structural transformation) can be expressed and verified directly on the controller automaton of a component, or of the whole application. Yet, it is possible to define classes of properties

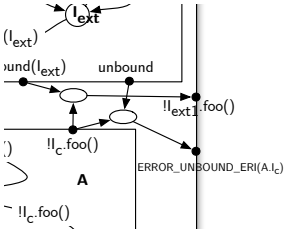

Fig. 4. Zoom into the **A** controller detecting errors

that can be checked on smaller systems, avoiding to build the global state-space. This section identifies abstractions and tools allowing to verify some specific categories of properties.

For a component **C** (including the full application itself), let us call $\mathbf{O}_F$ the set of external functional operations, $\mathbf{O}_E$ the set of observable errors, $\mathbf{O}_I$ the set of internal actions chosen to be observable. Then we define the set of deployment action as $\mathbf{O}_D = \neg(\mathbf{O}_F \cup \mathbf{O}_E \cup \mathbf{O}_I)$.

Deployment As we mention in Section 2.2, a system is deployed in a bottom-up fashion in the component hierarchy. At each node (composite component) of the system a specific deployment is applied.

This deployment is defined by the user; e.g. in Fractal, the bindings for the sub-components of a composite, can be given using its ADL. The deployment is a sequence of internal control operations of the composite, possibly interleaved with functional operations, and terminating with a distinguished successful state $\surd$. In our example (Fig. 1):

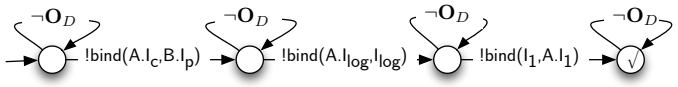

Fig. 5. Deployment automaton for **C**

The interplay between the building of all components of the application, and their start operations (that are usually applied recursively after building) may be quite complex and error-prone. So it may be useful for the developer to check, independently, that the deployment (possibly without start) of each component succeeds, and that the global deployment, including start operations, is also successful. This will be checked on the synchronisation of the component controllers with their respective deployment automata.

Functional behaviour A functional property is a property concerning only functional actions, or more precisely properties of a system after correct deploy-

ment, on a system in which we forbid any subsequent control action. This kind of formulas can be model-checked on a controller automaton for which we already have proved correct deployment, and in which we build only the relevant part of the behaviour, either by an had-hoc construction algorithm, or using on-the-fly techniques.

Functional behaviour properties are useful for component systems that do not perform any reconfiguration or for which non-functional actions have a transparent behaviour regarding functional aspects, i.e. non-functional actions commute with functional ones.

Non-structural Reconfiguration Non-structural reconfiguration, i.e. involving only bind, unbind, start and stop operations, can be dealt with directly on the controller automaton. Indeed, the interleaving between functional and non-functional actions may have consequences on the state of the system ; we cannot provide any general abstraction fitting with this case that could reduce the complexity of the model construction for this class of properties.

Structural Transformations *Remove*, *add* and *update* are the main control operations that modify the content of a composite. The first remark is that there is no hope to encode all possible future transformations in the model. Then, technically, *add* and *remove* operations change the arity of the enclosing Net, so they cannot be modelled as transducer transitions. Instead we model the structural transformation operations as functions transforming the whole hierarchical model of the application ; each elementary structural change affects a single Net or LTS in the model.

Update could be expressed as a sequence *unbind*;remove;add;bind**, but this would lead both to less efficient implementations and to more complex model constructions and proofs: we are interested in expressing full sequences of transformations, that preserve properties of the system, while elementary transformations usually don't.

The main difficulty with structural reconfigurations is that one wants to keep the rest of the system in the same state. A large application should not be stopped when updating or adding a specific sub-component, and the state of a replaced component itself should be preserved whenever possible. The framework ensures minimum conditions before replacements (in terms of stopped/unbound state), but we have to assume that the developer will specify which data from the replaced components are to be saved, and how this data will be mapped in the new component.

In our formalism, this tree transformation and state transfer is expressed on the hierarchical pNets, as the following sequence of steps :

- build the new hierarchical pNet, by replacement of the transformed part; call $\mathcal{S}'$ the semantics of this new system;
- define a mapping between actions in the original and the new systems, based on a user-defined mapping between the action names and parameters in the replaced component;

– identify the set $\mathcal{T}$ of states on the initial system where the transformation is possible;
– build a synchronised product of the old and new system, using the mapping of old to new actions, and adding in each state of $\mathcal{T}$ a transition $\xrightarrow{t}$ encoding the transformation; we call $\mathcal{T}'$ the image of $\xrightarrow{t}$ in this product;
– finally obtain the controller automaton of the transformed system, $\mathcal{A}'$ defined by: the set of initial states of $\mathcal{A}'$ is $\mathcal{T}'$, the states and the transitions of $\mathcal{A}'$ are those of $\mathcal{S}'$ reachable from $\mathcal{T}'$.

The actions mapping will eventually be defined in terms of the source language of the application, but this is out of the scope of this paper.

6 Proving Properties

In our tools, we use the modal μ-calculus as a powerful internal language for logic formulas. For this paper, we prefer to use an Action-based Computation Tree Logics (ACTL, see e.g. [8]), that may be more suitable for a human reader.

1. **Deployment**: We want at first to verify that the deployment for a component is always successful. This is done by proving the ACTL formula
$$[true]\surd \qquad (1) \text{ (all paths lead to success)}$$
in the synchronisation product between the component controller and its deployment. This formula is true for the deployment in Fig. 5.
A second property we would like to verify is the absence of error during the deployment. This is done by proving the formula
$$\mathbf{EF}_{\neg \mathbf{O}_E} < \mathbf{O}_E > false \qquad (2)$$
in the synchronisation product between the component controller and its deployment. This property is also true for the deployment in Fig. 5. However, in a scenario very reasonable, let's suppose the user starts the component **C** at the end of the deploy (which means to add a !start transition before the state $\surd$). Under this scenario the property is not true anymore (even though the deployment is possible), and the model-checking tool give us the counter-example shown in Fig. 6

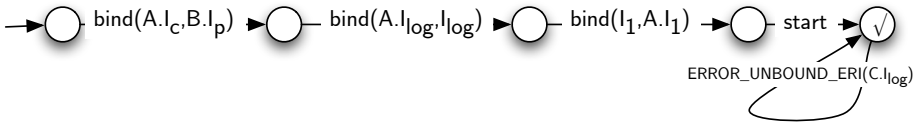

Fig. 6. Diagnostic path

The error is because the required interface $\mathsf{C.I_{log}}$ may be used before it is bound, which in fact is true since the interface $\mathsf{I_{log}}$ of **C** will be bound at the next level of hierarchy (when deploying **System**). This example also shows us the importance of the hierarchical behaviour of start and stop.

2. **Functional behaviour**: We would like to verify the absence of errors during a running phase, i.e. the absence of errors after the deploy until a new reconfiguration phase. If $\mathbf{O}_E$ is the set of observable errors and $\mathbf{O}_D$ is set of observable control operations, then we can verify the property in **System** by proving the ACTL formula:
$$\mathbf{A}[true]\sqrt{\mathbf{AG}_{\neg \mathbf{O}_D}[\mathbf{O}_E]false} \qquad (3)$$
is true in its controller. The proof is successful for **System**.
3. **Transformation**: Suppose we do, during the application running-phase, an update of the sub-component **B** in **C** by a component **B2**. **B2** has a similar behaviour than **B**, but in addition it logs the calls to its $\mathsf{I_p}$ interface using its $\mathsf{I_{log}}$ interface. When we prove formula (3), it becomes false in this updated system, and the tool gives us a path containing the action ERROR_UNBOUND_ERI(B2.log). This is because in the initial deployment of the system, we did not bind the interface $\mathsf{I_{log}}$ of **B**. Since **B** did not use its interface $\mathsf{I_{log}}$, the composition did not produced an undesired behaviour. However, the new **B2** uses its $\mathsf{I_{log}}$ interface, and so it produces the error. So the update of **B** by **B2** should be followed by a binding of its $\mathsf{I_{log}}$ interface. This example, likely to happen in real systems, shows the necessity of formal tools of verification for checking reconfiguration requirements.
If after the update and before starting the system, we bind the interface $\mathsf{I_{log}}$ of **B2** to the internal interface $\mathsf{I_{log}}$ of **C**, then the property is preserved.

7 Related Work

Most component frameworks available today only have tools for checking the static type compatibility of interfaces. Work on behaviour compatibility is quite recent, and not yet available on industrial plate-forms. We mention here research works and tools that may be the closest to our approach.

Wright [1] was an early proposal for specifying the behaviours of components in an Architecture Description Language (ADL). They use connectors similar to our Nets, that define the possible interactions between a set of roles (specification of sub-components). The behaviours of roles is specified in CSP, and they have a notion of compatibility based on a variant of CSP's refinement pre-order that ensures the absence of deadlock.

Darwin [10] is an Architecture Description Language, in which a distributed program is represented as a hierarchical composition of subsystems, with interacting processes at the leaves of the hierarchy. The behaviours are specified and computed in a way similar to ours, including weak bisimulation minimisation during the bottom-up construction. Verification of safety and liveness properties, specified in terms of finite-state automata, is done by the Tracta tool. The main difference with our approach is that Darwin expresses only the functional operations of the components, and does not support system reconfiguration.

Sofa [12] defines a hierarchical component system. At each level of hierarchy, a *frame protocol* specifies the external behaviour of the component, while a *architecture protocol* describes an implementation capturing also the internal

synchronisation between its sub-components. The architecture protocol can be automatically generated from the frame protocols of its sub-components. The behaviours are expressed as regular expressions, and the substitutability of components is based on trace language inclusion, though it is yet unclear how to compare with our bisimulation-based semantics. One specificity of Sofa is its sophisticated mechanism for detecting and reporting errors. They also have a syntax for expressing asynchronous operations, for which the emission of a method call and the return of its result may be interleaved with other events.

A quite different approach is advocated by Carrez, Fantechi and Najm in [7]. They propose a (non-hierarchical) component model in which interfaces are given a behavioural type expressed in a kind of modal process algebra. Then, they define the *sound assembly* of components as the conjunction of compliance of components to their interface (contracts), and compatibility between interfaces. The type language definition ensures that the compatibility is decidable and can be computed efficiently. Unfortunately, the compliance relation is more complex, and may even require theorem-proving techniques, but only needs to be guarantied once for a given component.

8 Discussion and Conclusion

This paper provides methods and tools allowing the user to prove the correctness of the behaviour of hierarchical components. One of our main contributions is the specification of the behaviour of non-functional aspects, and the hierarchical building of LTSs modelling the behaviour of the system of components. Our approach rely on the definition of a generic controller allowing (once instantiated) to encode the whole behaviour of any component except non-structural reconfiguration. Then a component behaviour is obtained by synchronisation product of the LTSs expressing the behaviour of its content and the control behaviour associated to its interfaces. Structural (dynamic) reconfiguration is handled by a LTS transformation. The tools provided to the user include:

- a controller automaton allowing to prove general properties on the behaviour of a component provided no structural reconfiguration is considered;
- an error detection: firing of error messages upon common sense errors can automatically be added; then, for example, the user may prove the absence of such messages in order to assert the correctness of the application;
- a set of hiding mechanisms in order to facilitate the proof of usual species of temporal properties;
- modelling of structural reconfigurations as transformations of the application model, thus allowing to reason about the most general components reconfigurations.

We have developed a tool in Java that automatically and incrementally generates the synchronisation files for a component system from its description, and we use the CADP[9] tool set to calculate the synchronisation product, minimise the systems, and model-check the formulas.

A promising perspective is to extend this framework in order to specify and verify the behaviour of asynchronous distributed components. Such a work would benefit from our previous experience in specification of asynchronous communicating objects [3], and consist in extending and adapting the notion of asynchronous method calls, request queues, etc.

Finally, many approaches are being developed to cover the right composition of components considering their functional aspects. One of the strongest advantage of using components is the separation of concerns from the user point of view. However, when coming to behavioural verification, one still needs to take into account the inter-play between functional and non-functional aspects, at least for existing component models. The main contribution of this paper is to encode the deployment and reconfigurations as part of the behaviour of the system, and thus verify the behaviour of the whole component system.

References

1. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.
2. A. Arnold. *Finite transition systems. Semantics of communicating sytems.* Prentice-Hall, 1994.
3. I. Attali, T. Barros, and E. Madelaine. Formalisation and proofs of the chilean electronic invoices system. In *XXIV International Conference of the Chilean Computer Science Society (SCCC 2004)*, pages 14–25, Arica, Chili, November 2004. IEEE Computer Society.
4. T. Barros, R. Boulifa, and E. Madelaine. Parameterized models for distributed java objects. In *Forte'04 conference*, Madrid, 2004. LNCS 3235, Spinger Verlag.
5. Francoise Baude, Denis Caromel, and Matthieu Morel. From distributed objects to hierarchical grid components. In *International Symposium on Distributed Objects and Applications (DOA), Catania, Sicily, Italy, 3-7 November*. LNCS, 2003.
6. E. Bruneton, T. Coupaye, and J. Stefani. Recursive and dynamic software composition with sharing. Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02), June 2002.
7. A. Fantechi C. Carrez and E. Najm. Behavioural contracts for a sound assembly of components. In Springer-Verlag, editor, *in proceedings of FORTE'03*, volume LNCS 2767, November 2003.
8. R. De Nicola and F.W. Vaandrager. Action versus state based logics for transition systems. In I. Guessarian, editor, *Semantics of Systems of Concurrent Processes, LITP Spring School on Theoretical Computer Science*, volume 469 of *LNCS*, La Roche Posay, France, 1990. Springer.
9. 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.
10. D. Giannakopoulou, J. Kramer, and S. Chi Cheung. Behaviour analysis of distributed systems using the tracta approach. *Automated Software Engg.*, 6(1):7–35, 1999.
11. H. Lin. Symbolic transition graph with assignment. In U. Montanari and V. Sassone, editors, *CONCUR '96*, Pisa, Italy, 26–29 August 1996. LNCS 1119.
12. F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11), nov 2002.

A Appendix: C Controller

The controller for the composite **C** is the synchronisation product of the 7 LTLs composing the Net in Fig. 7. We distinguish in the figure the internal control operations, which are labelled inside the controller Net (e.g. $?\mathsf{bind}(\mathsf{A.I_{log}}, \mathsf{I_{log}})$), from the external control operations, which are in the edge of the Net (e.g. $?\mathsf{I_{bc}}.\mathsf{bind}(\mathsf{I_{ext2}}, \mathsf{I_1})$). The internal control operations are those used during the deployment of the component, while the external control operation are used during the deployment of the next level of hierarchy. Since the start/stop operations are hierarchical, they appear twice, both as internal and as external control operations.

At this level of hierarchy, the variables $\mathsf{I_{ext*}}$ in the controller of **A** (Fig. 3) have been instantiated: $\mathsf{I_{ext1}}$ becomes $\{\mathsf{B.I_p}\}$, $\mathsf{I_{ext2}}$ becomes $\{\mathsf{C.I_{log}}\}$ and $\mathsf{I_{ext3}}$ becomes $\{\mathsf{C.I_1}\}$.

Similarly to the primitive components, we can see in the figure some constraints in the control operations, such as that the binding between the internal interface $\mathsf{I_1}$ of **C** and the external provides interface $\mathsf{I_1}$ of **A**, encoded by $?\mathsf{bind}(\mathsf{I_1}, \mathsf{A.I_1})$ is possible only when the composite **C** is stopped. We also see in the figure an edge for the functional calls between the sub-components **A** and **B** named as $\mathsf{foo}(\mathsf{A.I_c}, \mathsf{B.I_p})$; by default this call is hidden to the upper levels of hierarchy since its an internal action of **C**, but we chose to keep it visible. Recall the final user can specify the internal actions he wants to observe, which will remain visible to the upper levels of hierarchy. Thus allowing the user to prove temporal properties involving those actions.

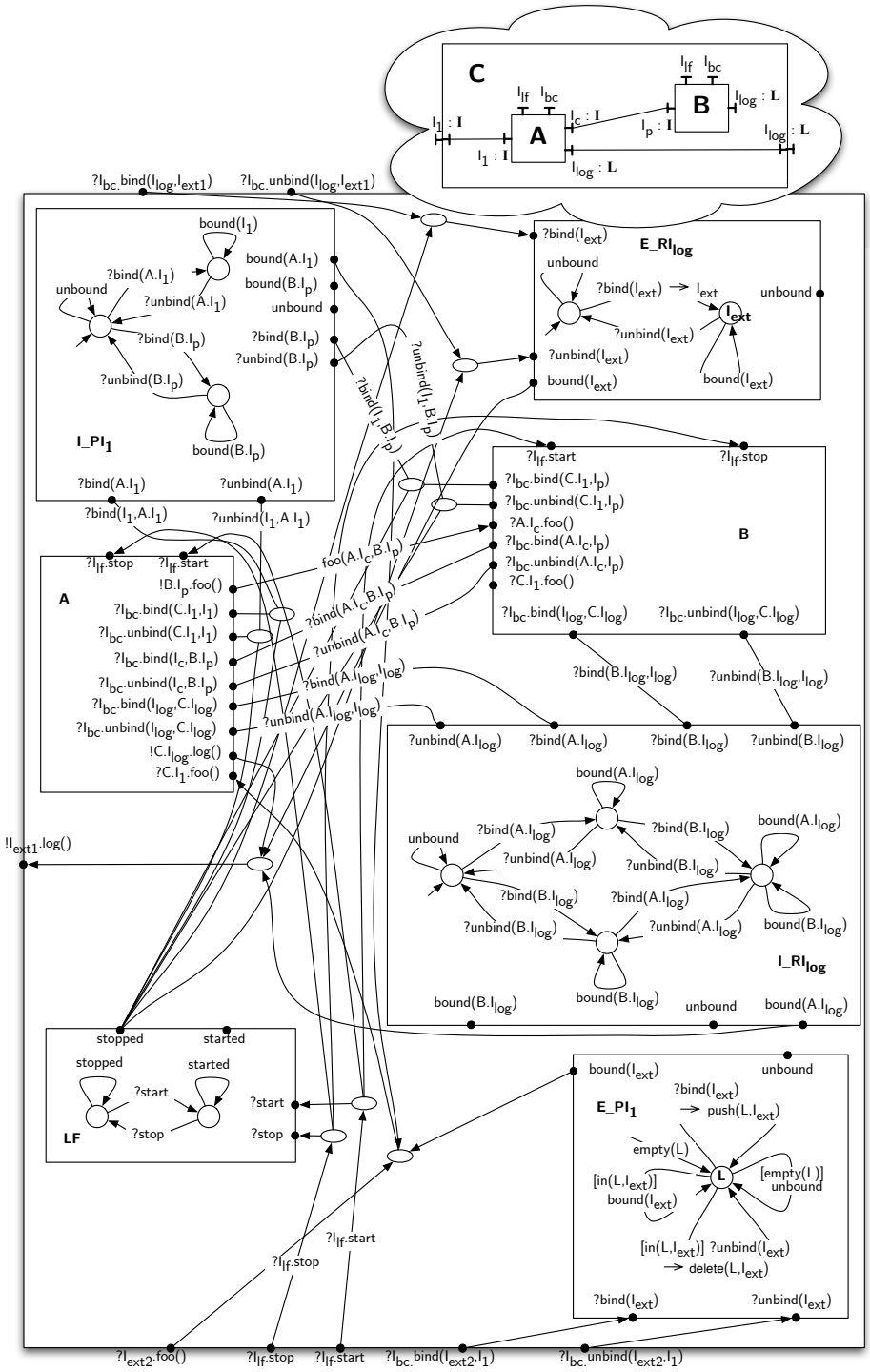

Fig. 7. Controller of **C**