Behavioural Models for Hierarchical Components

Tomás Barros, Ludovic Henrio, Eric Madelaine

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Behavioural Models for Hierarchical Components

Tomás Barros — Ludovic Henrio — Eric Madelaine

N° 5591

June 2005

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--5591--FR+ENG

Behavioural Models for Hierarchical Components

Tomás Barros , Ludovic Henrio * , Eric Madelaine

Thème COM — Systèmes communicants
Projet OASIS

Rapport de recherche n° 5591 — June 2005 — 33 pages

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. Reconfigurations 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.

Key-words: Behaviour, Model generation, Components, Reconfiguration, Hierarchy, Model-Checking, Process Algebra, Transition Systems

* Univ. of Wesminster, Watford Rd Northwick park, Harrow, HA1 3TP, UK

Modèles comportementaux pour composants hiérarchiques

Résumé : Nous décrivons une méthode pour spécifier et vérifier le comportement dynamique des systèmes à base de composants. L'utilisation d'un système de composants permet aux développeurs de spécifier l'architecture, les procédeures de déploiement, le cycle de vie d'un système complexe en utilisant des formalismes bien définis, et de gagner en productivité en réutilisant des composants existants.

Mais il faut alors être sùr que l'application ainsi créée est sure, au sens où ses parties s'assemblent correctement, et fonctionnent en harmonie. Chacun des composants doit correspondre au rôle que le reste du système attend de lui, et la mise à jour ou le remplacement d'un composant ne doit pas causer de blocage ou d'échec pour les autres.

Les notions habituelles de compatibilité de type des interfaces n'est pas suffisante pour cela; nous devons exprimer l'interaction dynamique entre les composants, pour éviter, typiquement, les blocages ou les comportements incorrects du système.

Dans cet article, nous nous concentrons sur le cas des systèmes de composants hiérarchiques. Nous décrivons aussi bien leur comportement fonctionnel que leurs mécanismes non-fonctionnels (controle du cycle de vie) en termes de systèmes de transitions. Nous définissons une notion de composition correcte des composants; puis nous montrons comment prouver, en utilisant des techniques de model-checking compositionnel, les propriétés de logique temporelle d'un système. Les reconfigurations d'un système, comme le remplacement d'un sous-composant, sont représentées par des transformations de sa sémantique comportementale, ce qui nous permet de prouver la préservation de certaines propriétés, ou la validité de nouvelles propriétés après la transformation.

Mots-clés : Comportement, génération de modèles, Composants, Reconfiguration, Vérification de modèles, Algèbres de processus, Systèmes de Transitions

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 [BCS02], 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 [CCN03], as well as frameworks like Wright [AG97a] or Sofa [PV02].

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 [BBM04], 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 [ABM04] 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 [BCM03].

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 notions of the formal models that we defined in [BBM04]; this includes the definitions of *parameterized synchronised networks of labelled transition systems*, a graphical language for a subclass of those systems, and the definition of (finite) instantiation of those parameterized systems. 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, specifying errors, computing the composite behaviour, building specific abstract models useful for representing deployment and reconfiguration phases. In Section 7 we give examples of proofs of some properties of the assembly. Then we compare with a number of related works.

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 *binding* is a connection path between component interfaces. The Fractal model distinguish between *primitive bindings* and *composite bindings*. We focus only in primitive bindings. A primitive binding is a binding between one client interface and one server interface. A primitive binding between a client interface c and a server interface s of two components C and S must verify one of the following constrains:

- c and s are external interfaces, and C and S have a direct common enclosing component. Such bindings are called *normal bindings*.
- c is an internal interface, s is an external interface, and S is a sub component of C. Such bindings are called *export bindings*.
- c is an external interface, s is an internal interface, and C is a sub component of S. Such bindings are called *import bindings*

Additionally, a primitive binding can be established only if the server interface accepts at least all the operations invocations that the client interface can emit, and a client interface can be bound to at most one server interface, while several client interfaces can be bound to the same server interface.

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.

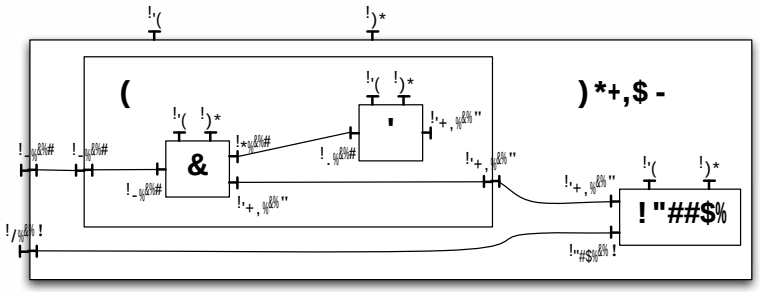

Figure 1: A simple component system

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**.

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 [BBM04] 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.

4.1 Theoretical Model

Our model is an adaptation of the *symbolic transition graphs with assignment* of [Lin96] into the *synchronisation networks* of [Arn94]: 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 [Lin96].

We start with an unspecified set of communications **Actions** Act, that will be refined later.

We model the behaviour of a process as a Labelled Transition System (LTS) in a classical way [Mil89]. The LTS transitions encode the actions that a process can perform in a given state.

Definition 1 LTS. *A labelled transition system is a tuple* $(S, s_0, L, \rightarrow)$ *where* S *is the set of states,* $s_0 \in S$ *is the initial state,* $L \subseteq Act$ *is the set of labels,* $\rightarrow$ *is the set of transitions* $: \rightarrow \ \subseteq S \times L \times S$. *We write* $s \xrightarrow{\alpha} s'$ *for* $(s, \alpha, s') \in \ \rightarrow$.

Then we define **Nets** in a form inspired by [Arn94], that are used to synchronise a finite number of processes. A Net is a form of generalised parallel operator, and each of its arguments are typed by a **Sort** that is the set of its possible observable actions.

Definition 2 Sort. *A Sort is a set* $I \subseteq Act$ *of actions.*

A LTS $(S, s_0, L, \rightarrow)$ can be used as an argument in a Net only if it agrees with the corresponding Sort $(L \subseteq I_i)$. In this respect, a Sort characterises a family of LTSs which satisfy this inclusion condition.

Nets describe dynamic configurations of processes, in which the possible synchronisations change with the state of the Net. They are Transducers, in a sense similar to the open Lotos expressions of [Lak96]. Each state of the transducer corresponds to a given configuration of the network in which a given set of synchronisations is possible; some of those synchronised actions can trigger a change of state of the transducer. Transducers are encoded as LTSs which labels are synchronisation vectors, each describing one particular synchronisation of the process actions:

Definition 3 Net. *A Net is a tuple* $< A_G, I, T >$ *where* A_G *is a set of global actions,* I *is a finite set of Sorts* $I = \{I_i\}_{i=1,\ldots,n}$, *and* T *(the transducer) is a LTS* $(T_T, s_{0_t}, L_T, \rightarrow_T)$, *such that* $\forall \overrightarrow{v} \in L_T, \overrightarrow{v} =< l_t, \alpha_1, \ldots, \alpha_n >$ *where* $l_t \in A_G$ *and* $\forall i \in [1, n], \alpha_i \in I_i \cup \{idle\}$.

We say that a Net is *static* when its transducer vector contains only one state. Note that a synchronisation vector can define a synchronisation between one, two or more actions from different arguments of the Net. When the synchronisation vector involves only one argument, its action can occur freely.

The semantics of the Net construct is given by the synchronisation product:

Definition 4 Synchronisation Product. *Given a set of LTS* $\{LTS_i = (S_i,\ s_{0_i},\ L_i, \rightarrow_i)\}_{i=1...n}$ *and a Net* $< A_G, \{I_i\}_{i=1...n}, (S_T, s_{0_T}, L_T, \rightarrow_T) >$, *such that* $\forall i \in [1,n], L_i \subseteq I_i$, *we construct the product LTS* $(S,\ s_0,\ L,\ \rightarrow)$ *where* $S = S_T \times \prod_{i=1}^{n}(S_i)$, $s_0 = s_{0_T} \times \prod_{i=1}^{n}(s_{0_i})$, $L = A_G$, *and the transition relation is defined as:*

$\rightarrow \triangleq \{s \xrightarrow{l_t} s' |\ s =< s_t, s_1, \ldots, s_n >, s' =< s'_t, s'_1, \ldots, s'_n >,$
$\exists\ s_t \xrightarrow{\overrightarrow{v}} s'_t \in \rightarrow_T, \overrightarrow{v} =< l_t, \alpha_1, \ldots, \alpha_n >,\ \forall i \in [1,n],\ (\alpha_i \neq idle \wedge s_i \xrightarrow{\alpha_i} s'_i \in \rightarrow_i) \vee (\alpha_i = idle \wedge s_i = s'_i)$

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

Next, we introduce our parameterized systems which are an extension from the above definitions to include parameters. These definitions are connected to the semantics of Symbolic Transition Graph with Assignment (STGA) [Lin96].

Parameterized Actions have a rich structure, for they take care of value passing in the communication actions, of assignment of state variables, and of process parameters. In order to be able to define variable instantiation as an *abstraction* of the data domains (in the style of [CR94]), we restrict these domains to be **simple (countable) types**, namely: booleans, enumerated sets, integers or intervals over integers and finite records, arrays of simple types.

Definition 5 Parameterized Actions *are:* τ *the non-observable action,* $\mathcal{M}$ *encoding an observable local sequential program (with assignment of variables),* $?m(P, \overline{x})$ *encoding the reception of a call to the method* m *from the process* P *(*$\overline{x}$ *will be affected by the arguments of the call) and* $!P.m(\overline{e})$ *encoding a call to the method* m *of a remote process* P *with arguments* $\overline{e}$.

A parameterized LTS is a LTS with parameterized actions, with a set of parameters (defining a family of similar LTSs) and variables attached to each state. Parameters and variables types are simple. Additionally, the transitions can be guarded and have a resulting expression which assigns the variables associated to the target state:

Definition 6 pLTS. *A parameterized labelled transition system is a tuple* $pLTS = (K, S, s_0, L, \rightarrow)$ *where:*

$K = \{k_i\}$ *is a finite set of parameters,*
S *is the set of states, and each state* $s \in S$ *is associated with a finite set of variables* $\overrightarrow{v_s}$,
$s_0 \in S$ *is the initial state,*

$L = (b, \alpha(\overrightarrow{x}), \overrightarrow{e})$ is the set of labels (parameterized actions), where b is a boolean expression, $\alpha(\overrightarrow{x})$ is a parameterized action, and $\overrightarrow{e}$ is a finite set of expressions.
$\rightarrow \subseteq S \times L \times S$ is the set of transitions:

Definition 7 Parameterized Sort. *A Parameterized Sort is a set pI of parameterized actions.*

Definition 8 *A* **pNet** *is a tuple $< pA_G, H, T >$ where: pA_G is the set of global parameterized actions, $H = \{pI_i, K_i\}_{i=1..n}$ is a finite set of holes (arguments). The transducer T is a pLTS $(K_G, S_T, s_{0_T}, L_T, \rightarrow_T)$, such that $\forall \overrightarrow{v} \in L_T$, $\overrightarrow{v} =< l_t, \alpha_1^{k_1}, \ldots, \alpha_n^{k_n} >$ where $l_t \in pA_G$, $\alpha_i \in pI_i \cup \{idle\}$ and $k_i \in K_i$.*

The K_G of the transducer is the set of global parameters of the pNet. Each hole in the pNet has a sort constraint pI_i and a parameter set K_i, expressing that this "parameterized hole" corresponds to as many actual arguments as necessary in a given instantiation. In a synchronisation vector $\overrightarrow{v} =< l_t, \alpha_1^{k_1}, \ldots, \alpha_n^{k_n} >$, each $\alpha_i^{k_i}$ corresponds to the α_i action of the k_i-nth corresponding argument LTS.

4.2 Graphical Language

We provide a graphical syntax for representing *static* Parameterized Networks, that is a compromise between expressiveness and user-friendliness. We use a graphical syntax similar to the Autograph editor [BRRdS94], augmented by elements for parameters and variables: a *pLTS* is drawn as a set of circles representing states and edges representing transitions, where the states are labelled with the set of variables associated with it ($\overrightarrow{v_s}$) and the edges are labelled by $[b]\ \alpha(\overrightarrow{x}) \rightarrow \overrightarrow{e}$ (see Definition 6).

An *static pNet* is represented by a set of boxes, each one encoding a particular Sort of the pNet. These boxes can be filled with a pLTS satisfying the Sort inclusion condition. Each box has a finite number of *ports* on its border, represented as labelled bullets, each one encoding a particular parameterized action of the Sort.

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

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

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

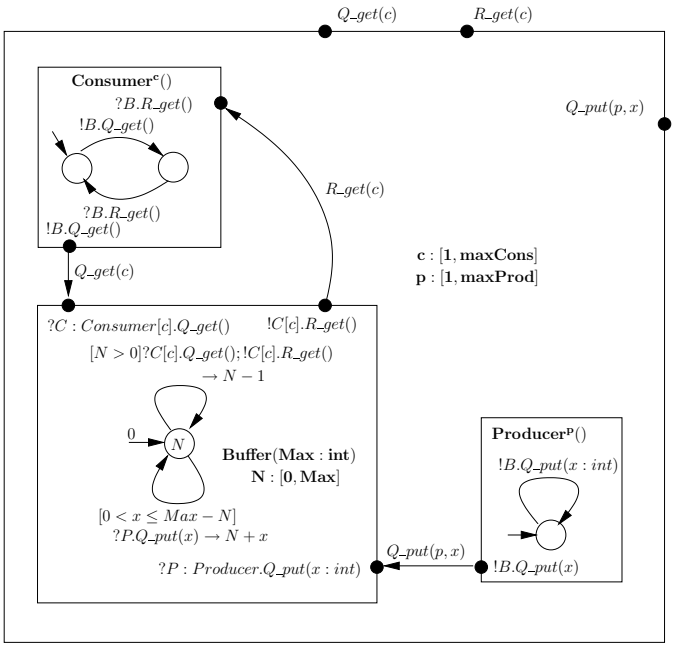

Figure 2: Parameterized consumer-producer system

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

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

4.3 Instantiation

In the framework of this paper, we do not want to give a more precise definition of the language of parameterized actions, and we shall not try to give a direct definition of the synchronisation product of pNets/pLTSs. Instead, we shall instantiate separately a pNet and its argument pLTSs (abstracting the domains of their parameters and variables to finite domains, before instantiating for all possible values of those abstract domains), then use

the non-parameterized synchronisation product (Definition 4). This is known as the early approach to value-passing systems [Mil89, MPW92].

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

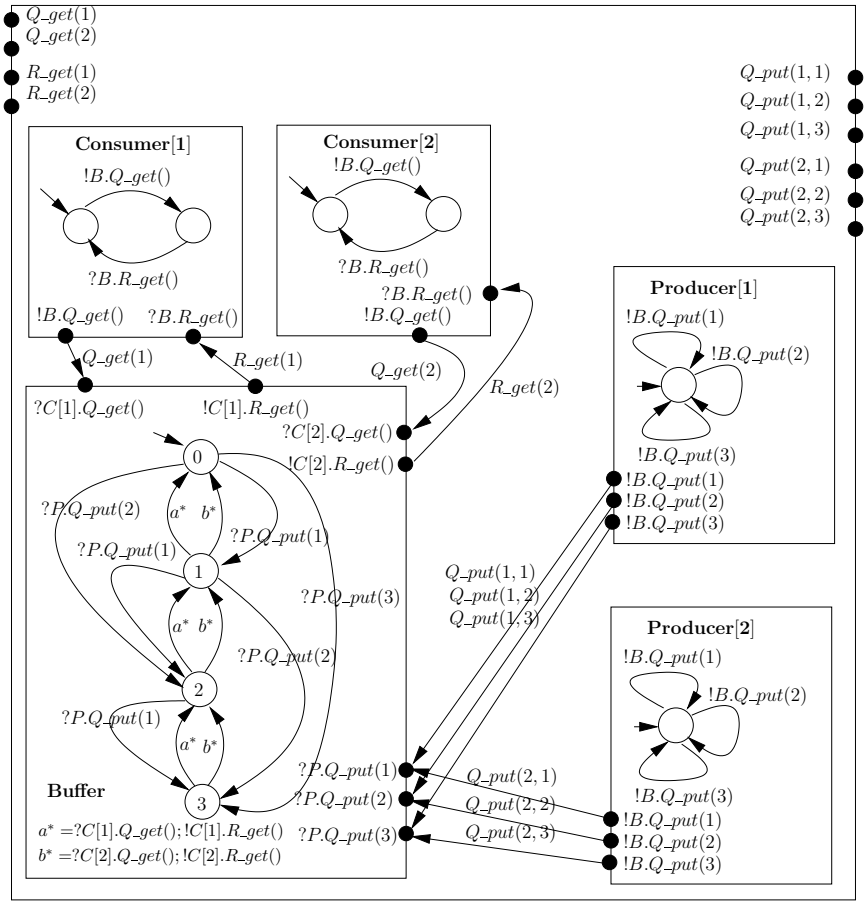

Figure 3: Parameterized consumer-producer system

4.4 Application to components

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. 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 reconfigurations 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 basic reconfigurations, in which we maintain the state of the components that were unchanged. We can then check for the properties (preserved or new) of the reconfigured system.

5 Building the Behaviour for the Example

In this section we introduce our way to build the behaviour of a component system for the example introduced in 2.2. Later in the paper we extend our work for any component system.

Since reconfiguration phases change the behaviour of a component, we need to build the set of all the behaviours after applying those transformation phases. This means in our formalism to build the component's transducer, where the transition between different states are fired by control operations. For instance, in the composite **C** the communication between **A** and **B** is not possible until the interface I_c in **A** is bound to the interface I_p in **B**. Then the control operation that binds those interfaces corresponds to a transition to a state where the communication becomes possible.

We build the transducers of our models using controller pieces that are composed with the sub-components using a synchronous parallel operator (no event is possible if it is not possible in the current state of the controller parts). The result of this building is an automaton which we named as *Controller*.

5.1 Primitive components

We suppose the functional behaviours of the primitive components are known, whether they are obtained by source analysis or given by the user is outside the scope of this paper. The functional behaviour of a primitive component is expressed as an automaton with labels encoding methods calls and receptions in its interfaces as well as internal actions.

The primitive components can be implemented as basic runtime entities (such as Java objects) to which control operation capabilities are added (for instance by the developer in the source code, by reflection, using a Fractal Factory or such in FracTalk [fra] by a variable names convention).

The functional behaviours of the primitive components **A**, **B** and **Logger** are shown in Fig. 4.

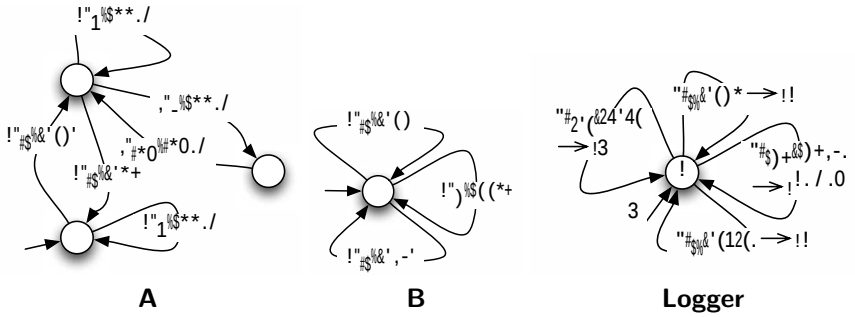

Figure 4: Behaviour of the base components of **A**, **B** and **Logger**

Logger provides a logging functionality through its provided interface $\mathsf{I}_{\mathsf{log}}$ up to n calls, unless a call is done to the method reset() of its interface $\mathsf{I}_{\mathsf{rst}}$, the method reset() resets the counter to 0. We intentionally do not include value passing in the communications to keep simplicity.

We start by adding the control capabilities (life-cycle and binding) to **A**, i.e. building its controller. **A** provides one interface (I_1) and requires two interfaces (I_c and $\mathsf{I}_{\mathsf{log}}$). Since **A** is a primitive component, there are only the external views of its interfaces (there is no internal interface since there is no internal binding in **A**). Then we build the controller for **A** as the synchronisation product of the 5 LTLs composing the Net in Fig. 5. The orientation of the links in the figure are used to help the visual view and understanding of the system, but they do not have any semantic meaning.

In Fig. 5 we can see the automata encoding the control operations for the external requires interfaces I_c (**E_RI$_\mathsf{c}$**) and $\mathsf{I}_{\mathsf{log}}$ (**E_RI$_{\mathsf{log}}$**), and for the external provides interface I_1 (**E_PI$_1$**). We also see the automaton encoding the life-cycle control operation (**LF**) and the functional behaviour of **A**. Synchronised actions are encoded by links between processes (in the graphics we use an ellipse when more than two actions are synchronised).

Fig. 5 includes some constrains (of common sense or from the Fractal SPEC) 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.

In the functional behaviour of **A** (Fig. 4) we observe the presence of the non-functional actions $?\mathsf{I}_{\mathsf{lf}}.\mathsf{start}$ and $?\mathsf{I}_{\mathsf{lf}}.\mathsf{stop}$. We do not want to break the separation of concerns reached by Fractal but only to keep both simplicity and some generality during this paper. The Fractal specification does not define all details of the controller constraints and semantics; some features will only be defined by specific implementations. For instance, in the Fractal implementation Julia [jul], the primitive components are enriched with *interceptors* whose role is to suspend the incoming method calls while the component is stopped. If we are modelling for the Julia implementation, we can express this constraint in our approach by linking the started port of the **LF** controller part to the reception of methods calls.

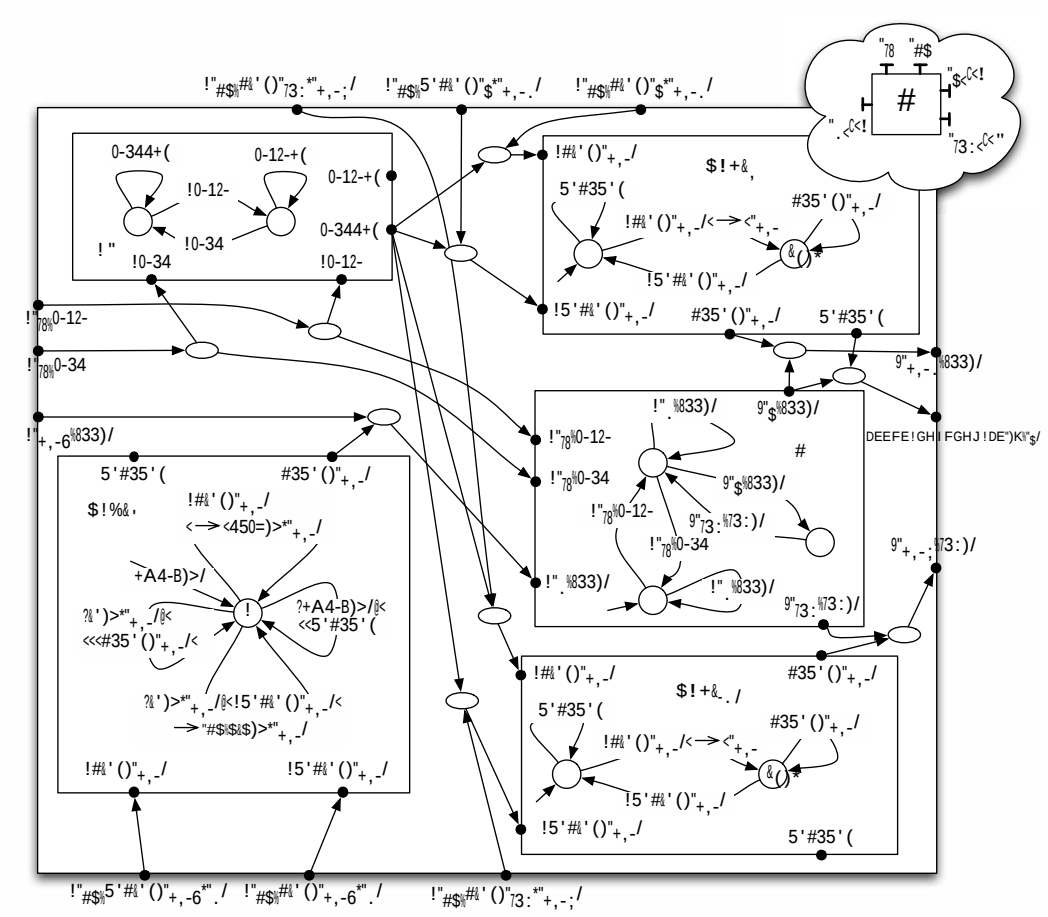

Figure 5: Controller for **A**

As we mention in the introduction, our main target is distributed component systems communicating asynchronously. Specifically we target the ProActive implementation of Fractal [BCM03]. However, at the time of this paper the behaviour of the life-cycle interface in ProActive components is not yet fully fixed. Momentarily we assume for our example that the functional part of the primitives are conscious of the life-cycle control interface.

In Fig. 5, 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 (i.e. once its environment is defined). For instance when building the controller of **C**, the variable I_{ext1} in the figure becomes the set $\{\mathsf{B.I_P}\}$.

The *Controller* of **A** is the automaton resulting from the synchronisation product of Fig. 5. This controller encodes both, the functional and non-functional behaviours of **A**. We use the controller to calculate the behaviour of the component after deployment as shown later.

Using the same methodology, we build the controllers for the other primitive components **B** and **Logger**.

5.2 Composites

A finite set of sub-components forms a composite. As we introduced in section 2.2, a component system is deployed in a bottom-up fashion from the innermost components to the outer component. At each level of hierarchy (defined by a particular composite) a specific deployment is applied.

This section describes the method we use to build the *Controller* of a given composite. The model of the composite is a parameterized Network, which arguments are the models of its subcomponents. This model includes both the functional aspects of the behaviour (coming eventually from the user specifications of basic components), and the non-functional management aspects (that we automatically generate from the ADL specification of the composite).

The global behaviour of the application will be computed later, building the synchronous product for each composite component in a bottom-up fashion, after instantiation of the parameters. At each level, only a selected set of actions (functional or non-functional) will be observable. This allows for a *grey-box* construct, in which an reduced model can be constructed for proving a given formula or set of formulas.

The controller for the composite **C** is the synchronisation product of the 7 LTLs composing the Net in Fig. 6. We use a syntax C.I to designate an interface I belonging to a component C. When C is absent, the interface belongs to the component itself, i.e. to the component of the controller. The arguments for the bind and unbind operations are always a *client* interface in the first argument and a *server* interface in the second.

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. $?\mathsf{I_{bc}.bind}(\mathsf{I_{ext2}}, \mathsf{I_1})$). 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. In Fig. 6 the boxes **A** and **B** correspond to the *grey box* behaviour of **A** and **B** respectively.

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 it is an internal action of **C**, but we chose to keep it visible. Recall that the final user can specify the internal actions he wants to observe, which will

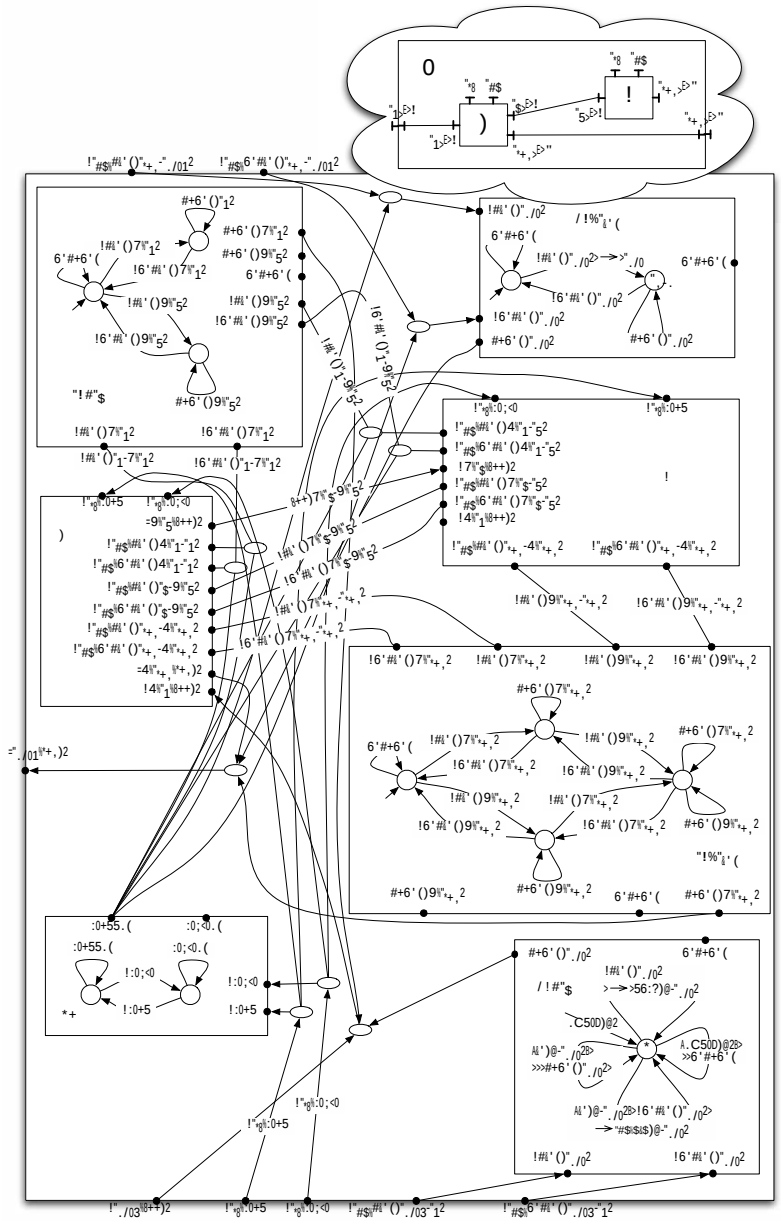

Figure 6: Controller of **C**

remain visible to the upper levels of hierarchy. Thus allowing the user to prove temporal properties involving those actions.

5.3 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 I_{log} when it is unbound, we can detect the erroneous uses of the I_{log} interface. This is shown in Fig. 7.

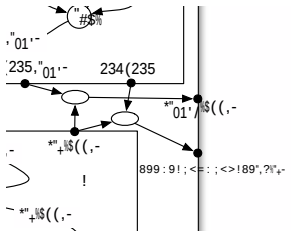

Figure 7: Zoom into the **A** controller detecting errors

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.

5.4 General purpose Controller

The principles exposed for the example in the previous section are applied here in a systematic way: we have defined a *general purpose Controller*, that will be instantiated for each component in each level of hierarchy in the system, using the information available in the components ADL specification. Then the LTSs will be computed in a bottom-up fashion. The general purpose controller is shown in Fig. 8.

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. Note that the size of the system at a given level only depends on the complexity of this level of hierarchy (and of the actions the user wants to observe), not on the complexity of the lower levels.

In particular, specific models can be constructed to focus on the detection of some classes of errors.

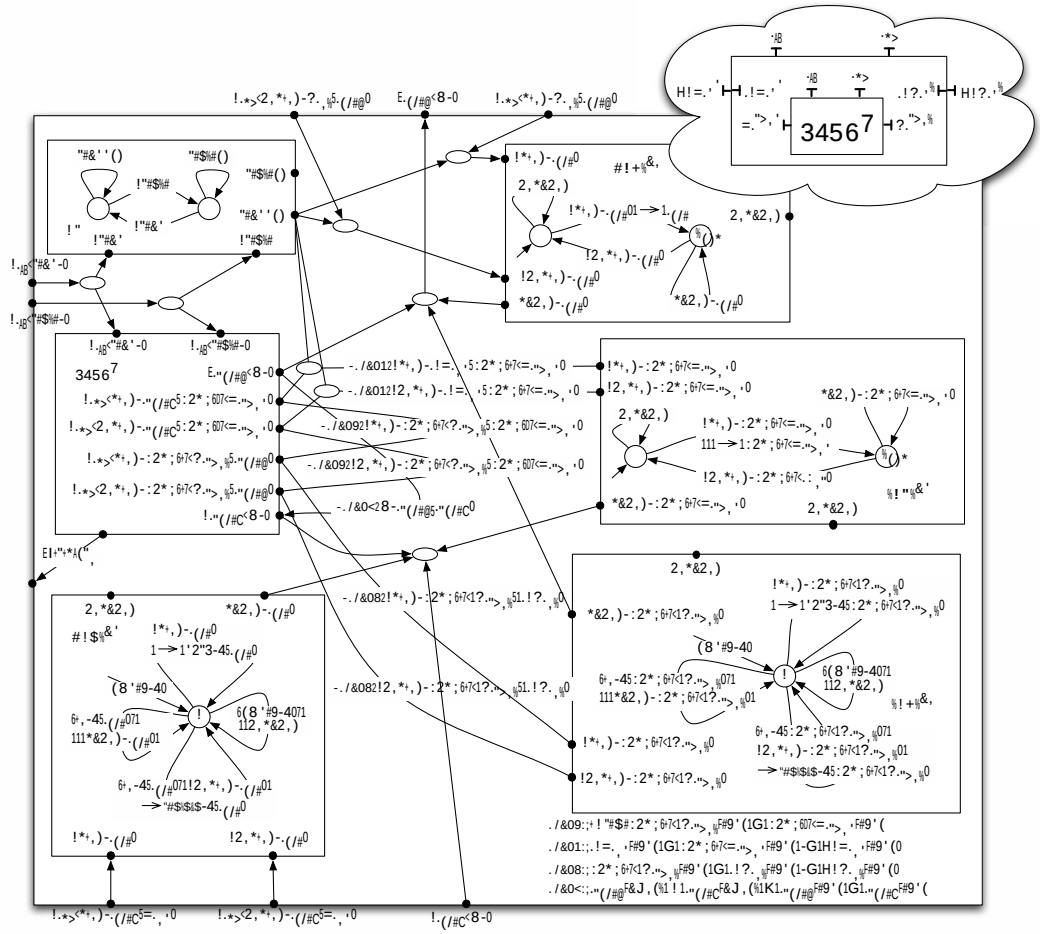

Figure 8: General purpose Controller

In the general purpose Controller shown in Fig. 8, we have a finite number k of sub-component automata (**SubC^k**), a life-cycle automaton (**LF**), a finite number np of external (**E_PI^{np}**) and internal (**I_PI^{np}**) provides interface automata, and a finite number nr of external (**E_RI^{nr}**) and internal (**I_RI^{nr}**) requires interface automata.

To obtain the *Controller* for a component (primitive or composite), we instantiate the general controller, using the sub-components and interfaces that the component ADL defines. For instance, for the composite component **C**, the set {**SubC^k**} will be replaced by the networks representing the sub-components **A** and **B**. 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.

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).

5.5 Deployment and Static Automaton

Now we want to compute, in a bottom-up manner, the full behavioural model at each level, but taking into account the *deployment phase* defined earlier.

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 $\surd$. For instance, the deployment of **C** in our example (Fig. 1) is shown in Fig. 9.

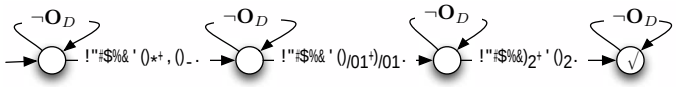

Figure 9: Deployment automaton for **C**

We define the *static automaton* of a component, corresponding intuitively to its black-box behaviour after deployment, as being the part of its (instantiated) controller automaton after successful deployment, hiding the internal operations (except those chosen to be visible) and forbidding any further reconfiguration actions.

For a component $\mathbf{C}$ (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 and $\mathbf{O}_C$ the set of internal control operations. Then we define the set of external control operations $\mathbf{O}_D = \neg(\mathbf{O}_F \cup \mathbf{O}_E \cup \mathbf{O}_I \cup \mathbf{O}_C)$.

Let $\mathbf{C_t}$ be the controller of an automaton and $\mathbf{D}$ its deployment, then we define its static automaton $\mathbf{S}$ using *CCS*[Mil89] operators as:

$$\mathbf{S} = (\mathbf{C_t}|\mathbf{D})\backslash\mathbf{O}_D$$

modulo minimisation by weak bisimulation.

Since the deployment automata of the primitive components are reduced to a single ($\surd$) state, their static automata are equivalent to their controller automata as we suggested before.

As an example, Fig. 10 is the Static automaton of component **C**. It includes only external binding operations (between **C**, **System**, and **Logger**) such as $?\mathsf{I_{bc}}.\mathsf{bind}(\mathsf{I_{log}}, \mathsf{Logger}.\mathsf{I_{log}})$, functional actions of **C** such as $!\mathsf{Logger}.\mathsf{I_{log}}.\mathsf{log}()$, and errors relative to the external bindings of **C** ($\mathsf{ERROR\_UNBOUND\_ERI}(\mathsf{C}.\mathsf{I_{log}})$).

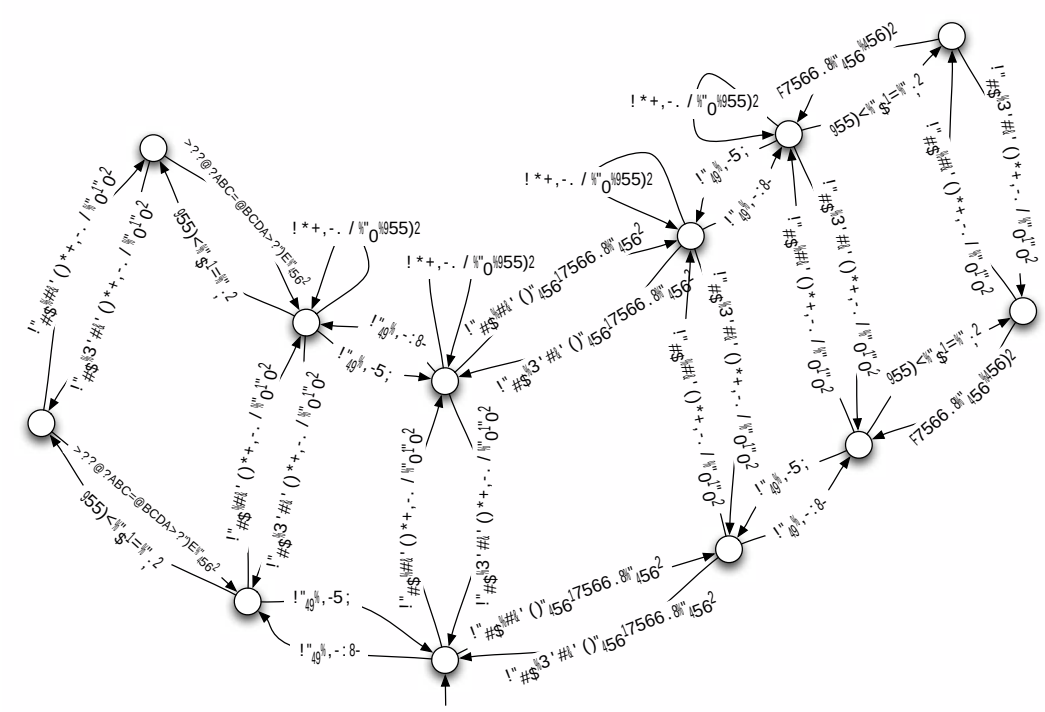

Figure 10: Static automaton for **C**

6 Species of Temporal Properties

All the temporal properties (that do not involve a structural reconfiguration) 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 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.

6.0.1 Deployment

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 such as in Fig. 9) 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.

6.0.2 Functional behaviour

A functional property is a property concerning only functional actions, or more precisely properties of a system after correct deployment, 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 (this is the Static automaton defined in the previous section), 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.

6.0.3 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. However, 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.

6.0.4 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 reconfiguration 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 reconfigurations, that preserve properties of the system, while elementary reconfigurations 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 reconfiguration 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.

7 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. [DNV90]), 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

   $$[true]\surd \qquad (1)$$ (all paths lead to success)

   in the synchronisation product between the component controller and its deployment. This formula is true for the deployment of **C** (Fig. 9).

   A second property we would like to verify is the absence of error during the deployment. This is done by proving the formula

   $$\mathbf{EF}_{true} < \mathbf{O}_E > false \qquad (2)$$

   in the synchronisation product between the component controller and its deployment. This property is also true for the deployment of **C**. However, in a scenario very reasonable, let's suppose the user starts the component **C** at the end of the deployment (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. 11 (Note that we label successful synchronisations between the actions $?\alpha$ and $!\alpha$ as α)

   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.

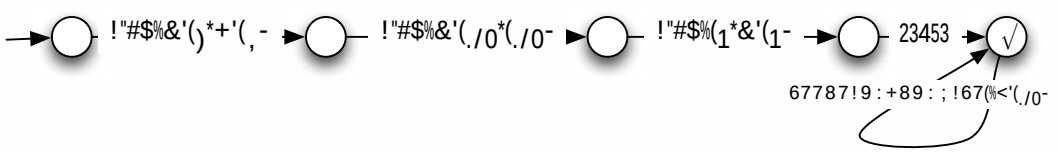

Figure 11: Diagnostic path

2. **Functional behaviour**: We would like to verify the absence of errors during a running phase, i.e. the absence of errors after the deployment until a new reconfiguration phase. We can verify the property in a component by proving the ACTL formula:

$$\mathbf{EF}_{\neg \mathbf{O}_D} < \mathbf{O}_E > false \qquad (3)$$

is true in its static automaton. For instance, the proof is successful for **System**.

Another property we would like to prove (extracted for example from the user requirements) can be that every call to the function foo() in the interface $\mathsf{I_c}$ of **A** to the interface $\mathsf{I_p}$ of **B** is eventually logged in **Logger**.

This inevitability property is checked by verifying in the static automaton of **System** the ACTL formula:

$$\mathbf{AG}_{true}[\mathsf{foo(A.I_c, B.I_p)}]\ \mathbf{A}(true_{\neg \mathbf{O}_D}\ \mathbf{U}_{\mathsf{log(C.I_{log}, Logger.I_{log})}}\ true) \qquad (4)$$

Since we are proving the property for a functional phase, Formula (4) intrinsically forbids external control actions ($\neg \mathbf{O}_D$) to happen. In Figure 12 we show the static automaton for **System** when forbidding external control operations. We consider an instantiation where **Logger** has a logging capacity of 2 ($n = 2$ in Fig. 4).

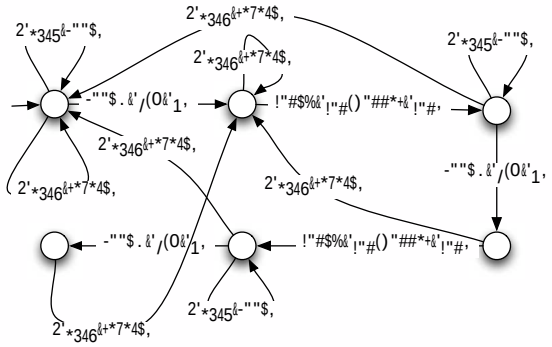

Figure 12: $\mathbf{S}\backslash\mathbf{O}_C$ for **System**

Formula (4) is false in **System** and the model-checking tool give us the diagnostic shown in Fig. 13.

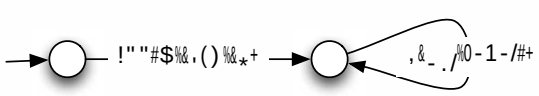

Figure 13: Property (4) diagnostic for **System**

This diagnostic is showing us one case where the Formula (4) is false because the behaviour of **System** can fall in an infinite loop of the action $?\mathsf{I_{ext}}.\mathsf{reset}()$ after the action $\mathsf{foo}(\mathsf{A.I_c}, \mathsf{B.I_p})$. Then the computation tree contains traces where the action $\mathsf{log}(\mathsf{C.I_{log}}, \mathsf{Logger.I_{log}})$ never happen, this is known as an unfair path.

We need to add a constraint to avoid the infinitely loop shown in Fig. 13 by considering only fair paths (i.e where the constraint happen infinitely often) when proving the property. In practise, this is expressed in the logical formula itself. Then Formula (4) becomes the formula:

$$\mathbf{AG}_{true}[\mathsf{foo}(\mathsf{A.I_c}, \mathsf{B.I_p})]\ \mathbf{AG}_{\neg \mathsf{log}(\mathsf{C.I_{log}}, \mathsf{Logger.I_{log}})}\mathbf{EF}_{\neg \mathbf{O}_D} < \mathsf{log}(\mathsf{C.I_{log}}, \mathsf{Logger.I_{log}}) > true \quad (5)$$

This formula express that after a $\mathsf{foo}(\mathsf{A.I_c}, \mathsf{B.I_p})$ and while $\mathsf{log}(\mathsf{C.I_{log}}, \mathsf{Logger.I_{log}})$ is not reached, $\mathsf{log}(\mathsf{C.I_{log}}, \mathsf{Logger.I_{log}})$ is reachable in a finite number of transitions (which avoid the unfair paths).

However, we are considering the assumption that **System** is in an environment where the method $\mathsf{reset}()$ of its provided interface I_1 is infinitely often invoked. We should be careful about those assumption, for instance suppose that this interface is binding to a button of a graphical interface for the user: if we want to prove the correct autonomous behaviour of our system, i.e. without user intervention, then we should consider that the reset button is never pressed. We express that by adding the restriction of non-reset action ($\neg ?\mathsf{I_{ext}}.\mathsf{reset}()$), then the formula becomes:

$$\mathbf{AG}_{true}[\mathsf{foo}(\mathsf{A.I_c}, \mathsf{B.I_p})]\ \mathbf{AG}_{\neg \mathsf{log}(\mathsf{C.I_{log}}, \mathsf{Logger.I_{log}})}\mathbf{EF}_{\neg (\mathbf{O}_D \vee ?\mathsf{I_{ext}}.\mathsf{reset}())} < \mathsf{log}(\mathsf{C.I_{log}}, \mathsf{Logger.I_{log}}) > true \quad (6)$$

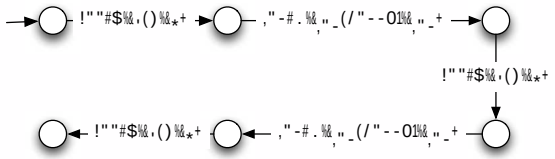

Figure 14: Property (6) diagnostic for **System**

Formula (6) is false in **System** and the tool give us the diagnostic shown in Fig. 14. The figure show us the presence of a deadlock because **Logger** is full. To empty **Logger**

it should be reset through the method call reset() in its I_1 interface and in conclusion, **System** can not autonomous behave correctly , it needs the user to take part.

We have shown in this section how we prove behavioural properties of a component or an application, where properties range from reachability of an error action to intricate temporal ordering of actions, including fairness properties. More research is needed to give to final users a more accessible language for expressing those properties, for example an extension of so-called "specification patterns" with specific constructs for component management.

3. **Non-structural Reconfiguration**:

   We would like to prove some preservation of properties when doing non-structural reconfigurations. For instance, we can prove that Property (5) is preserved for any non-structural reconfiguration that does not involve the interface $\mathsf{I}_{\mathsf{log}}$ in **System**. Let $\mathbf{C}_D$ be the deployment of **C** and $\mathbf{O}_{log}$ the binding operations involving $\mathsf{I}_{\mathsf{log}}$, i.e. $\mathbf{O}_{log} = \{?\mathsf{bind}(\mathsf{C.I}_{\mathsf{log}}, \mathsf{Logger.I}_{\mathsf{log}}), ?\mathsf{unbind}(\mathsf{C.I}_{\mathsf{log}}, \mathsf{Logger.I}_{\mathsf{log}})\}$. Then this property of preservation is successful verified by checking in the **System** controller the formula:

$$[\mathbf{C}_D]\mathbf{AG}_{\neg \mathbf{O}_C}[\mathsf{foo}(\mathsf{A.I_c}, \mathsf{B.I_p})]\ \mathbf{AG}_{\neg \mathsf{log}(\mathsf{C.I}_{\mathsf{log}}, \mathsf{Logger.I}_{\mathsf{log}})}\mathbf{EF}_{\neg \mathbf{O}_{log}} < \mathsf{log}(\mathsf{C.I}_{\mathsf{log}}, \mathsf{Logger.I}_{\mathsf{log}}) > true \quad (7)$$

4. **Structural 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 I_{p} interface using its $\mathsf{I}_{\mathsf{log}}$ interface. If we build this new system (using the method described in section 6.0.4), and try to prove again formula (3), it appears to be false, 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 $\mathsf{I}_{\mathsf{log}}$ of **B**. Since **B** did not use its interface $\mathsf{I}_{\mathsf{log}}$, the composition did not produced an undesired behaviour. However, the new **B2** uses its $\mathsf{I}_{\mathsf{log}}$ interface, and so it produces the error. So the update of **B** by **B2** should be followed by a binding of its $\mathsf{I}_{\mathsf{log}}$ interface. This example, likely to happen in real systems, shows the necessity of formal verification tools for checking reconfiguration requirements.

   If after the update and before starting the system, we bind the interface $\mathsf{I}_{\mathsf{log}}$ of **B2** to the internal interface $\mathsf{I}_{\mathsf{log}}$ of **C**, then the property is preserved.

7.1 Tools

Figure 15 shows the ADL description file for the upper level of hierarchy. In line 17 and 18 we suggest a way to introduce functional behavioural specification of primitive components.

We developed a tool prototype in Java which takes as inputs the system ADL and the functional behaviour of primitives to automatically generate the models described in this paper. We use the CADP [GLM02] tool-set to do the synchronisation product and the model-checking of formulas. From the ADL description, our tool prototype generates, once

System.fractal

```
1  <?xml version="1.0" encoding="ISO-8859-1" ?>
2  <!DOCTYPE .... >
3
4  <definition name="components.System">
5
6     <component name="C"
7           definition="components.C">
8        <interface name="log" role="client"
9              signature="components.LogInterface"/>
10       <interface name="l1" role="server"
11             signature="components.I1Interface"/>
12    </component>
13
14    <component name="Logger">
15       <interface name="log" role="server"
16             signature="components.LogInterface"/>
17       <content class="components.LoggerImpl">
18          <behaviour file="LoggerBehav"
19                format="Aldebaran"/>
20       </content>
21    </component>
22
23    <binding client="C.log"
24          server="Logger.log"/>
25 </definition>
```

Figure 15: System ADL

Component	Controller	Static Aut.
A	24/99	24/91
B	16/98	16/90
Logger	4/16	4/14
C	432/2168	12/39
System	36/151	6/19
B2	24/107	24/99
C {update(B,B2)}	1786/7082	20/58

Figure 16: Example of automata sizes (states/transitions) of the example

instanced, the synchronisation product in Exp-V2 format (and ASCII list of synchronisation vectors). The automaton describing the functional behaviour of primitives is taken directly from its file (line 18 in Fig. 15). In our example the primitive automata are in Aldebaran format (a simple ASCII representation of finite LTS). Our tool also generates a script to build the system (svl script from CADP). Finally the proofs are verify using evaluator, an on-the-fly model checking tool included in CADP. Table 16 shows some results for the generated automata in our example; the CADP tool-set allows us to handle systems with as much as 100 millions states at each level of the construction.

8 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.

8.1 Wright

Wight [AG97b] provides a formal basis for specifying the interactions among architectural components. They make the difference between implementation relationships and interaction relationships; they add that the components are logically independent of each other: the correctness of each component is independent of the correctness of other components with which it interacts.

The interactions between components are defined through connectors. A connector has a set of roles (or participants in the interaction) and a glue. Each role is provided with a behaviour in CSP and the glue is the way that these behaviour can be combined (in other words, the glue defines the acceptable traces of the interleaving of the roles).

Once defined the connectors, components are associated to them. Specifically, the ports (interfaces) of the components are associated to roles of the connectors. Those ports must satisfy the role specifications (they should be compatible), the idea is that a connector definition can be reused. This compatibility between ports and roles is the main issue addressed by Wright. Is defined as a variant (weaker) of the *refinement* relationship in CSP. Compatibility ensures absence of deadlock if the connector is deadlock free and conservative (conservative = the glue is a subset of the interleaving traces).

Summarising, the approach of Wright is to define the connectors as an SPEC and then through compatibility to check the conformance of components (implementations). They provides an automatic approach but they did not seem to have an implementation at the time of the paper. Wright does not tackle hierarchy or asynchronous communication and does not support dynamism.

8.2 Darwin (Tracta)

In the architecture description language Darwin [MDEK95]((TO VERIFY CITATION)), a distributed program is represented as a hierarchical composition of subsystems, with interacting processes at the leaves of the hierarchy. For analysis, the system developer additionally provides behavioural descriptions for the primitive components of the system, as well as a set of properties that the system needs to satisfy.

Behaviour of components is specified in terms of Labelled Transition Systems (LTS), which are finite state machines. Properties are separated into two classes: safety and liveness. Both classes of properties are described in terms of finite-state automata. Property automata are included in the compositional hierarchy of the system, to be composed with the component to which they relate.

Tracta [GKC99] is a compositional reachability analysis (CRA) technique used within Darwin. The Tracta CRA approach incrementally construct the overall behaviour of the system from that of its subsystems. At each intermediate step, a component behaviour is computed from the behaviour of its sub-components. Subsequently, actions that do not form part of the component interface are made internal, and the behaviour is minimised with respect to observational equivalence.

8.3 SOFA

Sofa [PV02] 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 besides the external communications, the communications among its direct sub-components.

The behaviour is given as a regular expression including the tokens concatenation (;), alternative (+), finite sequencing (*), $\wedge$-parallel (|), $\vee$-parallel (||) and restriction (\). Additionally its contains two composed operators, composition (Π_X) and adjustment ($_{|T|}$). The composition (Π_X) is equivalent to the parallel composition defined in CCS, but the internal communications becomes $\tau call$ instead of τ. The adjustment($_{|T|}$) is a synchronisation operator, the tokens in T are synchronised (by syntax match) and merged into a single label.

Asynchronous calls are supported by distinguishing the reception and the response of a call, and allowing actions between both, (syntax $[!?]m\{P\}$ where P is executed between the reception and response of m). The regular expression generates the set of all the possibles traces generated by a component.

Sofa uses a top-down approach, by considering a component primitive, then replacing by a composite and doing refinement. The frame protocol defines the accepted interplays of the actions for a component. The architecture protocol can be generated based in the frame protocols of its sub-components. Substitutability of components is based on trace language inclusion (through a compliance relation), though it is yet unclear how to compare with our bisimulation-based semantics

Sofa propose a way to detect errors resulting from incorrect composition. This errors can be of three types: bad activity, no activity and divergence. Bad activity and no activity involves deadlock situations whereas divergence involves a live-lock.

Sofa does not tackle deployment and structural reconfiguration.

8.4 Behavioural types

A quite different approach is advocated by Carrez, Fantechi and Najm in [CCN03]. 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.

8.5 Cadena

Cadena [HDD+03] is a development and verification environment for building real time systems using CORBA. They extend the Interface Definition Language to add light-weight specification of component behaviour and dependencies using a BIR-like language. Then they use Bogor, a specialised model-checking tool for Cadena, to verify properties expressed using logical patterns. Cadena does not work with hierarchical components, and the model-checking tool lacks compositionality. They assume a correct deployment and do not support dynamicity.

9 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[GLM02] 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 [ABM04], 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

[ABM04] 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.

[AG97a] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[AG97b] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

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

[BBM04] T. Barros, R. Boulifa, and E. Madelaine. Parameterized models for distributed java objects. In *Forte'04 conference*, Madrid, 2004. LNCS 3235, Spinger Verlag.

[BCM03] 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.

[BCS02] 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.

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

[CCN03] 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.

[CR94] Rance Cleaveland and James Riely. Testing-based abstractions for value-passing systems. In *International Conference on Concurrency Theory (CONCUR)*, volume 836 of *Lecture Notes in Computer Science*, pages 417–432. Springer, 1994.

[DNV90] 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.

[fra] FracTalk: Fractal components in SmallTalk. http://csl.ensm-douai.fr/FracTalk.

[GKC99] 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.

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

[HDD+03] John Hatcliff, Xinghua Deng, Matthew B. Dwyer, Georg Jung, and Venkatesh Prasad Ranganath. Cadena: an integrated development, analysis, and verification environment for component-based systems. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 160–173, Washington, DC, USA, 2003. IEEE Computer Society.

[jul] JULIA framework (fractal implementation). http://fractal.objectweb.org.

[Lak96] A. Lakas. *Les Transformations Lotomaton : une contribution à la préimplémentation des systèmes Lotos.* PhD thesis, Univ. Paris VI, june 1996.

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

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, London, UK, 1995. Springer-Verlag.

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

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

[PV02] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11), nov 2002.

Contents

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

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

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