A Framework for Reasoning on Component Composition

Ludovic Henrio[1], Florian Kammüller[2], and Muhammad Uzair Khan[1]

[1] INRIA – CNRS – I3S – Université de Nice Sophia-Antipolis
{mkhan,lhenrio}@sophia.inria.fr
[2] Institut für Softwaretechnik und Theoretische Informatik – TU-Berlin
flokam@cs.tu-berlin.de

Abstract. The main characteristics of component models is their strict structure enabling better code reuse. Correctness of component composition is well understood formally but existing works do not allow for mechanised reasoning on composition and component reconfigurations, whereas a mechanical support would improve the confidence in the existing results. This article presents the formalisation in Isabelle/HOL of a component model, focusing on the structure and on basic lemmas to handle component structure. Our objective in this paper is to present the basic constructs, and the corresponding lemmas allowing the proof of properties related to structure of component models and the handling of structure at runtime. We illustrate the expressiveness of our approach by presenting component semantics, and properties on reconfiguration primitives.
Key words: Components, mechanised proofs, futures, reconfiguration.

1 Introduction

Component models focus on program structure and improve re-usability of programs. In component models, application dependencies are clearly identified by defining interfaces (or ports) and connecting them together. The structure of components can also be used at runtime to discover services or modify component structure, which allows for dynamic adaptation; these dynamic aspects are even more important in a distributed setting. Since a complete system restart is often too costly, a reconfiguration at runtime is mandatory. Dynamic replacement of a component is a sensitive operation. Reconfiguration procedures often entail state transfer, and require conditions on the communication status. A suitable component model needs a detailed representation of component organization together with precise communication flows to enable reasoning about reconfiguration. That is why we present here a formal model of components comprising both concepts.

This paper provides support for proving properties on component models in a theorem prover. Our objective is to provide an expressive platform with a wide range of tools to help the design of component models, the creation of adaptation procedures, and the proof of generic properties on the component model.

Indeed most existing frameworks focus on the correctness or the adaptation of applications; we focus on generic properties.

In this context, introduction of mechanised proofs will increase confidence in the properties of the component model and its adaptation procedures. We start from a formalisation close to the component model specification and implementation; then we use a framework allowing us to express properties in a simple and natural way. This way, we can convince the framework programmer and the application programmer of the safety of communication patterns, optimisations, and reconfiguration procedures.

We write our mechanised formalisation in Isabelle/HOL but we are convinced that our approach can be adapted to other theorem provers. The generic metalogic of Isabelle/HOL constitutes a deductive frame for reasoning in an object logic. Isabelle/HOL also provides a set of generic constructors, like datatypes, records, and inductive definitions supporting natural definitions while automatically deriving proof support for these definitions. Isabelle has automated proof strategies: a simplifier and classical reasoner, implementing powerful proof techniques. Isabelle, with the proof support tool Proofgeneral, provides an easy-to-use theorem prover environment. For a precise description of Isabelle/HOL specific syntax or predefined constructors, please refer to the tutorial [20].

We present here a framework that mechanically formalizes a distributed hierarchical component model and its basic properties. We show that this framework is expressive enough to allow both the expression of component semantics and the manipulation of the component structure. Benefiting from our experiences with different possible formalisations, and from the proof of several component properties, we can now clearly justify the design choices we took and their impact[3]. The technical contributions of this paper are the following:

- formal description in Isabelle of component structure, mapping component concepts to Isabelle constructs,
- definition of a set of basic lemmas easing the proof of component-related properties,
- additional constructs and proofs to ensure well-formedness of component structures,
- proposal for a definition of component state, and runtime semantics for components communicating by asynchronous request-replies,
- application to the design and first proofs about component reconfiguration.

The remainder of the paper is organised as follows. Section 2 gives an overview of the context of this paper: it positions this paper relatively to related works and previous works on the formalisation of the GCM component model, which is also described in Section 2.2. Section 3 presents the formalisation of the component model in Isabelle/HOL highlighting design decisions and their impact on the basic proof infrastructure. We then summarize a semantics for distributed components with its properties, and present a few reconfiguration primitives in Section 4.1. Section 5 concludes and presents future directions.

[3] The GCM specification framework is available at `www.inria.fr/oasis/Ludovic.Henrio/misc`

2 Background

Component modelling is a vast domain of active research, comprising very applied semi-formal approaches to formal methods. In this section, we first give an overview of the domain, starting from well-known approaches, summarizing some community activities, and focusing on the most relevant related works. Then we present the GCM component model and existing formalisation of GCM. Finally we position this paper relatively to the other approaches presented here.

2.1 Related Work

Some well-known component models like CCA [11] are not hierarchical – their intent is the efficient building, connecting and running of components but they neglect structural aspects. We rather focus on hierarchical component models like Fractal[6], GCM[4], or SCA[5].

Recent years have shown several opportunities for the use of formal methods for the modelling and verification of component-based applications as shown in several successful conferences like FMCO, FOCLASA, or FACS.

For example, in [8, 9] the authors investigate the use of formal methods to specify interface adaptation and generation of interface adaptors, based on behavioural specification of interfaces to be connected. Also, in [10, 3] the authors focus on the verification of the behaviour of component-based application. They provide tools to specify the behaviour of a component application, and check that this application behaves correctly. Their model is applied to the GCM component model too but they prove properties of specific applications whereas we formalise the component model itself. In [18], the authors present a comprehensive formalisation of the Fractal component model using the Alloy specification language. Additionally, the consistency of resulting models can be verified through the automated Alloy Analyzer. These contributions are close to our domain but focus on the use of formal methods to facilitate the development and ensure safety of component applications, while our aim is to provide support for the design of component models and their runtime support.

SCA (Service Component Architecture) [5] is a component model adapted to Service Oriented Architectures. It enables modelling service composition and creation of service components. FraSCAti [21] is an implementation of the SCA model built upon Fractal making this implementation close to GCM. It provides dynamic reconfiguration of SCA component assemblies, a binding factory, a transaction service, and a deployment engine of autonomous SCA architecture. Due to the similarity between FraSCAti and GCM, our approach provides a good formalisation of FraSCAti implementation. There are various approaches on applying formal and semi-formal methods to Service Oriented Architectures (SOA) and in particular SCA. For example, in the EU project SENSORIA [1] dedicated to SOA, they propose Architectural Design Rewriting to formalize development and reconfiguration of software architectures using term-rewriting [7].

Creol [15, 16] is a programming and modelling language for distributed systems. Active objects in Creol have asynchronous communication by method calls

and futures. Creol also offers components; the paper [12] presents a framework for component description and test. A simple specification language over communication labels is used to enable the expression of the behaviour of a component as a set of traces at the interfaces. Creol's component model does not support hierarchical structure of components. In [2], the authors present a formalisation of the interface behaviour of Creol components. Creol's operational semantics uses the rewriting logic based system Maude [19] as a logical support tool. The operational semantics of Creol is expressed in Maude by reduction rules in a structural operational semantics style enabling testing of model specifications. However, this kind of logical embedding does not support structural reasoning.

2.2 Component Model Overview

Our intent is to build a mechanised model of the GCM component model [4], but giving it a runtime semantics so that we can reason on the execution of component application and their evolution. Thus we start by describing the concepts of the GCM which are useful for understanding this paper. We will try in this paper to distinguish clearly structural concepts that are proper to any hierarchical component model and a runtime semantics that relies on asynchronous requests and replies. Structurally, the model incorporates hierarchical components that communicate through well defined interfaces connected by bindings. Communication is based on a request-reply model, where requests are queued at the target component while the invoker receives a future. The basic component model has been presented in [13] and is summarized below.

Component Structure Our GCM-like component model allows hierarchical composition of components. This composition allows us to implement a coarse-grained component by composition of several fine-grained components. We use the term *composite component* to refer to a component containing one or more *subcomponents*. On the other hand, *primitive components* do not contain other components, and are leaf-level components implementing business functionality. A component, primitive or composite, can be viewed as a container comprising two parts. A central *content part* that provides the functional characteristics of the component and a *membrane* providing the non-functional operations. Similarly, interfaces can be functional or non-functional. In this work and in the following description, we focus only on the functional content and interfaces.

The only way to access a component is via its interfaces. *Client* interfaces allow the component to invoke operations on other components. On the other hand, *Server* interfaces receive invocations. A *binding* connects a client interface to the server interface that will receive the messages sent by the client.

For composite components, an interface exposed to a subcomponent is referred to as an *internal* interface. Similarly, an interface exposed to other components is an *external* interface. All the external interfaces of a component must have distinct names. For composites, each external functional interface has a corresponding internal one. The implicit semantics is that a call received on a

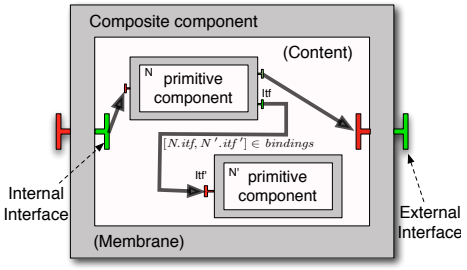

Fig. 1. component composition

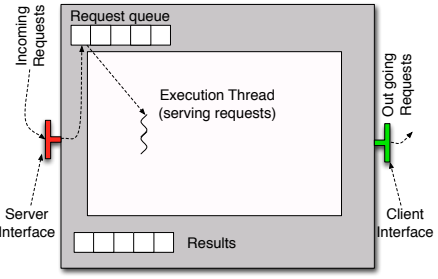

Fig. 2. component structure

server external (resp. internal) interface will be transmitted – unchanged – to the corresponding internal (resp. external) client interface.

The GCM model allows for a client interface to be bound to multiple-server interfaces. For the moment, in our model, we restrict the binding cardinality such that bindings connect a client to a single server. Note that several bindings can anyway reach the same server interface.

Figure 1, shows the structure of a composite component. The composite component contains two primitive subcomponents N and N'. The binding ($N.itf$, $N'.itf'$) connects the client interface itf of subcomponent N to the server interface itf' of subcomponent N'.

Communication Model Our GCM-like components use a simple communication model relying on asynchronous request and replies, as presented in [13]. Communication via requests is the only means of interaction between components. We avoid shared objects or component references, and use a pass-by-copy semantics for request parameters. A component receives the requests on its external server interface. The received requests are then enqueued in the *request queue*, which holds the messages until they can be treated.

Our communication model is asynchronous in the sense that the requests are not necessarily treated immediately upon arrival. Requests are only enqueued at the target component, then the component invoking the request can continue its execution without waiting for the result. Enqueuing a request is done synchronously but the receiver is always ready to receive a request. To ensure transparent handling of asynchronous requests with results, we utilise *futures*. Futures are created automatically upon request invocation and represent the request result, while the treatment of the request is not finished. Once the result of the computation is available, the future is replaced by the result value. Futures are first class objects: they can be transferred as part of requests or results.

Figure 2 gives the internal structure of a component. Incoming requests are enqueued in the *request queue*. The requests are dequeued by the execution threads, when computed; the results are placed in the *results list*.

Component Behaviour In our model, the primitive components represent the business logic and can have any internal behaviour. Primitive components treat all the requests they receive, choosing a processing order and the way to treat them. On the other hand, the behaviour of a composite component is more restricted: it is strictly defined by its constituent subcomponents and the way they are composed. A composite component serves its requests in a FIFO manner, delegating them to other components bound to it. A delegated request is delivered unchanged to the target component. Once the service of a request is finished, the produced result is stored in the computed results for future use. It can then be transmitted to other components, as determined by the reply strategy [17, 14].

2.3 Positioning

This paper provides formalisation of hierarchical components and their structure. At our level of abstraction, this structure is shared by several component levels like Fractal, GCM, and SCA. However most implementations of SCA (except FraSCAti) do not instantiate the component structure at runtime. By contrast, to allow component introspection and reconfiguration at runtime, we consider a specification where structural information is still available at runtime. This enables adaptive and autonomic component behaviours. Indeed, component adaptation in those models can be expressed by reconfiguration of the component structure. For example, reconfiguration allows replacement of an existing component by a new one, which is impossible or very difficult to handle in a model where component structure disappears at runtime.

Most existing works on formal methods for components focus on the support for application development whereas we focus on the support for the design and implementation of component models themselves. To our knowledge, this work is the only one to support the design of component models in a theorem prover. It allows proving very generic and varying properties ranging from structural aspects to component semantics and component adaptation.

A formalisation of our communication model along with the component semantics appear in [13]. An extended version of the formal semantics is presented in [14], providing formalisation of one particular reply strategy. Other possible strategies are discussed in [17]. Compared to our previous works, this paper relies on the experience gained in specification and proof and demonstrated in [13, 14] to design a framework for supporting mechanised proofs for distributed components. In particular this paper focuses on the handling of component structure, on a basic set of lemmas providing valuable tooling for further proof, and the illustration of the presented framework to prove a few properties dealing with component semantics and reconfiguration.

3 Formalisation of Component Model in Isabelle/HOL

Our component model is a subset of the GCM component model, but with a precisely defined structure and semantics. It incorporates hierarchical compo-

nents that communicate via asynchronous requests and replies. We start with formalising the structure of our components. Based on the structure defined, we present some of the various *infrastructure* operations that allow us to manipulate the components for proving properties. Then we formalise additional constructs to define component's state and request handling, and correctness of a component assembly. Finally we provide a set of very useful lemmas dealing with component structure and component correctness.

3.1 Component Structure

As we have seen in Section 2.2, a component in our model can either be a composite or primitive. A composite component comprises one or more subcomponents. On the other hand, a primitive component is a leaf-level component encapsulating the business logic.

```
datatype Component = Primitive Name Interfaces PrimState
    | Composite Name Interfaces (Component list) (Binding set) CompState
```

The above Isabelle/HOL datatype definition for `Component`s has two constructors `Primitive` and `Composite`. We present below the various elements that make up the structure of our components.

NAME: Each component has a unique name. We use this name as the component identifier/reference.

INTERFACES: Each component has a number of public interfaces. All communication between components is via public interfaces. An interface can be either client or server and by construction a component cannot have two interfaces with the same name.

SUBCOMPONENTS: Composite components have a list of subcomponents, given by the `Component list` parameter. Primitive components do not have subcomponents.

BINDINGS: In composite components, a binding allows an interface of one component to be plugged to an interface of a second component. `(N1.i1, N2.i2)`$\in$`bindings` if the interface `i1` of component `N1` is plugged to the interface `i2` of `N2` where `N1` or `N2` can either be component names or *This* if the plugged interface belongs to the composite component that defines the binding.

STATE: All components, primitive or composite have an associated state. Component state is discussed in more detail in Section 3.3.

Design decisions. In the Isabelle/HOL formalisation we chose to include the name of the component into the component itself. Like for interfaces, a first intuitive approach could be rather to define subcomponents as mappings from names to components. There are, however, major advantages to our approach. When we reason about a component we always have its name, which makes the expression of several semantic rules and lemmas more natural. The main advantage of maps is the implicit elegant encoding of the uniqueness of *Name*(s). As mentioned before, *Name*(s)are used as component references. Unfortunately,

this advantage of maps is quite low in a multi-layered component model because a map can only serve one level. As we want component names to be unique globally, a condition on name uniqueness is necessary.

Subcomponents are defined as lists rather than finite sets because lists come with a convenient inductive reasoning easing proofs involving component structure. Of course it is easy to define an equivalence relation to identify components modulo reordering. On the contrary the bindings of a component are defined as a set because no inductive reasoning is necessary on bindings, and sets fit better to the representation of this construct.

Having a formalisation of component structure alone, although useful, is not sufficient. An adequate infrastructure needs to be developed to help in reasoning on the component model. The next section describes some of the infrastructure operations that allow us to manipulate components inside component hierarchies.

3.2 Efficient Specification of Component Manipulation

This section provides various operations that allow us to effectively manipulate components. These include operation for accessing component fields, mechanisms for traversing component hierarchies, and means for replacing and updating components inside the hierarchical structure. All these operations are primitive recursive functions enabling an encoding in Isabelle/HOL using the `primrec` feature. Using this feature has great advantages for the automation of the interactive reasoning process. Automated proof procedures of Isabelle/HOL, like the simplifier, are automatically adapted to the new equations such that simple cases can be solved automatically. Moreover, the definitions themselves can use pattern matching leading to readable definitions.

Field access We define a number of operations for accessing various fields. These include the function GETNAME that returns the `Name` of the component.

```
primrec getName:: Component ⇒ Name   where
        getName (Primitive N itf s) = N |
        getName (Composite N itf sub b s) = N
```

Similarly, we define `getItfs` , `getQueue`, and `getComputedResults` for getting interfaces, request queues and replies. Requests and replies are part of the component state described in Section 3.3.

Accessing component hierarchy In order to support hierarchical components, we need a number of mechanisms to access components inside hierarchies. These range from simply finding a suitable component inside a component list to updating the relevant component with another component. The most useful of these operations are detailed below.

CPLIST: returns a list of all subcomponents of a component recursively. It uses the predefined Isabelle/HOL list operators `#` for constructing lists and `@` for appending two lists. Note that the following primitive recursive function is mutually recursive and needs an auxiliary operation dealing with component lists.

```
primrec cpList:: Component ⇒ Component list and
        cpListlist:: Component list ⇒ Component list
where
  cpList (Primitive N itfs s) = [(Primitive N itfs s)] |
  cpList (Composite N itfs subCp bindings s) =
              (Composite N itfs subCp bindings s)#(cpListlist subCp) |
  cpListlist [] = [] |
  cpListlist (C#CL) = (cpList C)@ cpListlist CL
```

CpSet: gives a set representation of the `cpList` of a component. This allows us to write properties in a much more intuitive way, for example, quantifying over sub-components is easily written as $\forall$ `C' ∈ CpSet(C)`. Note however that a few proofs require to stick to the `CpList` notation; indeed when switching to cpSet construct, one cannot reason on the coexistence of two identical components.

```
constdefs :: Component ⇒  Component set
             cpSet C == set (cpList C)
```

getCp: allows for retrieving a given component from a component list based on the component Name. The constructors `Some` and `None` represent the so-called `option` datatype enabling specifications of partial functions. Here, a component with the given name might not be defined in the list – this is nicely and efficiently modelled by a case distinction over the option type. Note the definition of `^` as an infix operator synonymous for `getCp`. This so-called pretty printing syntax of Isabelle supports natural notation of the form `CL^N = Some C'`.

```
primrec getCp:: Component list ⇒ Name ⇒ Component option where
   getCp [] N' = None |
   getCp (C#CL) N' = if (getName C=N') then Some C else (CL^N')
```

changeCp CL C: written `CL<-C` replaces the component in the list `CL` that has the same name as `C` by `C`; it does nothing if there is no component with the given name.

```
primrec changeCp::Component list ⇒ Component ⇒ Component list where
  changeCp [] C = [] |
  changeCp (C#CL) C' = if getName C=getName C' then C'#CL else C#(CL<-C')
```

removeSubCp C N: removes the subcomponent of `C` with name `N` but does nothing if there is no subcomponent with this name. Note, here the use of a `case` switch supporting again pattern matching in Isabelle/HOL definitions.

```
primrec removeSubCp:: Component ⇒ Name ⇒ Component where
  removeSubCp (Primitive N itf s) N' = (Primitive N itf s) |
  removeSubCp (Composite N itf sub b s) N' = (case sub^N' of
            None => (Composite N itf sub b s) |
            Some C => Composite N itf (remove1 C sub) b s)
```

Similar operations are needed for dealing with requests and results. This includes operations for building lists of all referenced requests inside a component (and its subcomponents), finding a result for a given future inside a component hierarchy, etc. In all we provide almost 30 functions and predicates to help express structured component specifications efficiently.

Design decisions. It is crucial for the reasoning process whether one chooses lists or sets to represent various parts of the specified component structure. As we have seen above the basic infrastructure we have built up to handle our hierarchical components is mainly based on lists. Consequently, we can define operations over components and their constituents by primitive recursion and thereby decisively improve automated support. However, sets come with a more natural notation. Often set theoretic properties can be simply decided by boolean reasoning that poses no problems for logical decision procedures integrated in Isabelle/HOL, and Isabelle/HOL comes with numerous lemmas for reasoning on sets. On the other side, inductive reasoning on finite sets is less convenient than on lists. In places where we want to combine the merits of both worlds, the `CpSet` function provides a convenient translation.

3.3 Component State

Our component model shall not only allow structural reasoning on hierarchical components but also reasoning about dynamic component state. While the preceding sections provided a good formalisation valid for any hierarchical component model, we now define component state in order to support communication by request and replies. Those constructs are used to define our component semantics, as shown in Section 4.1. Let us first focus on the high level definition of states which provide the constructs relating the component structure with the dynamic semantics[4]. We show below the two types of component states (for composite and primitive components) used in the definition of `Component` presented in Section 3.1.

```
record CompState =                    record PrimState =
Cqueue:: Request list                 Pqueue:: Request list
CcomputedResults:: Result list        PcomputedResults:: Result list
                                      PintState:: intState
                                      behaviour:: Behaviours
```

Each state contains a queue of pending requests, and a list of results computed by this component. Additionally, primitive components have an internal state and a behaviour for encoding the business logic, see below. We use the Isabelle/HOL `record` type constructor here; it automatically defines field projection as functions, e.g. for a `Compstate s`, `(Cqueue s)` accesses its request queue. Note that uniqueness of fields identifier required us to add a '`C`' or '`P`' prefix to fields of component states to distinguish them.

The definition of the component state relies on the definitions of requests (characterized by a future identifier, a parameter, and a target interface), and results (characterized by the future identifier and its value).

[4] The real definition of component states contains additional fields; only the fields of interest for this paper are shown here.

```
record Request =                    record Result =
id::Fid                             fid::Fid
parameter:: Value                   fValue:: Value
invokedItf:: Name
```

An interesting construct is the representation of component behaviour. Each primitive component has an internal state. A behaviour specifies how a primitive component passes from an internal state to another. It is defined as a labeled transition system between internal states of a component:

```
typedef  Behaviours={ beh::(intState × Action × intState) set.
   (∀ s s'. ((s,Tau,s')∈ beh ⟶ (set (PRqRefs s')⊆set (PRqRefs s))
                        ∧ PcurrentReqs s' = PcurrentReqs s)) ∧
 . . . }
```

The type `Behaviours` is defined as a set of triples (internal state, action, internal state). In our case actions are: internal transition (`Tau`, shown here), request service, request emission, result reception, and end of service which associates a result to a request. More than the precise definition of our actions, it is interesting to focus on the way behaviour can be defined and further refined by constraints. Additional rules are specified to restrain the possible behaviours, preventing incorrect transitions to occur; for example, we forbid replying to a non-existing request. In the piece of code above we require conditions on the internal state before and after an internal transition: the set of referenced futures can only be smaller after an internal transition, and the set of currently served requests is unchanged. More complex conditions are imposed for other actions.

Design decisions. Isabelle/HOL extensible records are the natural choice for representing states, requests, and results. They are better suited than simple products because they support qualified names implicitly. We did, however, not use the additional extension property of records which is similar to inheritance known from object-orientation. It could have been used to factor out the shared parts of primitive and composite components but this is not worthwhile – properties specific to the shared parts are few. Hence, there is practically no overhead caused by duplicating basic lemmas. The use of lists for requests and results is important for the efficient specification and proof of structural properties (see the design decisions in the previous section). The definition of behaviours in the internal state of primitive components uses an Isabelle/HOL type definition. This way, we can encapsulate the predicate defining the set of all well-formed behaviours into a new HOL type. These constraints are thereby implicitly carried over and can be re-invoked by using the internal isomorphism with the set `Behaviours`.

3.4 Correct Component

We presented the structure of our components in Section 2.2, while the various constructs designed to manipulate hierarchical components appear in Section 3.2.

However, we only reason on a subset of all possible components that can be constructed according to the described component structure. We refer to this subset of components as *correct components*. Correct components are not only well-formed, but they adhere to some additional constraints. The various well-formedness rules along with the correctness constraints are presented in the following.

We start with specifying the structure of a well-formed component. A composite component is considered as correctly structured if it passes the criteria specified by the function `CorrectComponentStructure` given below.

```
primrec CorrectComponentStructure :: Component ⇒ bool where
 CorrectComponentStructure (Composite N itfs sub b s) =
   ((∀ b ∈ bindings. (GetQualified(src b) (Composite N itfs sub b s =
                          Some (| kind=Client,cardinality=Single|))
    ∧ (GetQualified(dest b)(Composite N itfs sub b s) =
                          Some(| kind=Server,cardinality=Single|)))
    ∧ NoDuplicateSrc b
    ∧ distinct (map getName sub)
    ∧ (∀ Q∈ set (Cqueue s). (invokedItf Q)∈  dom itfs
                      ∧ kind (the (itfs (invokedItf Q))) = Server)
```

A composite component has a correct structure if: each binding only connects an existing client interface to another existing server interface; each client interface is connected only once; all subcomponents have distinct names; and all requests in the request queue of the composite refer to existing server interfaces. A primitive component has a correct structure if it follows the last requirement plus a couple of constraints relating its behaviour with its interfaces.

```
constdefs CorrectComponent :: Component ⇒ bool
CorrectComponent c == CorrectComponentStructure c ∧ distinct(RqIdList c)
        ∧ (ReferencedRqs c) ⊆ (set(RqIdList c))
        ∧ distinct (map getName (cpList c))
        ∧ (∀ f∈ set (RqIdList c). snd f ∈ set(map getName(cpList c)))
```

A correct component is a correctly structured component that also has uniquely defined request identifiers (`RqIdList c` gives all requests computed by `c` and its subcomponents), and all future referenced by the components should correspond to an existing request. Finally, names of all components in the composition should be unique. This differs from the well-formedness requirement which only requires the names of all direct subcomponents to be unique. The requirement of checking correct future referencing throughout the composition hierarchy is stronger than what is needed for most proofs, and can at times be relaxed resulting in a weaker correctness requirement `CorrectComponentWeak`. `CorrectComponentWeakList` gives similar constraints but for a list of components. Using `CorrectComponentWeak` eases proofs involving component hierarchy because if a component verifies `CorrectComponentWeak` then all its subcomponents also verify it.

```
constdefs CorrectComponentWeak:: Component ⇒ bool
CorrectComponentWeak c == CorrectComponentStructure c
    ∧ distinct (RqIdList c) ∧ distinct (map getName(cpList c))

constdefs CorrectComponentWeakList:: Component list ⇒ bool
CorrectComponentWeakList CL == (CorrectComponentStructureList CL)
  ∧ distinct (RqIdListList CL)∧ distinct (map getName (cpListlist CL))
```

3.5 Basic Properties on Component Structure and Manipulation

In this section, we present a few properties that we proved. They deal with the constructs presented in Section 3.2, and are unrelated to our definition of states presented in the last section. Those lemmas are the basic building blocks on which most of our proofs rely. On the set of more than 80 lemmas dealing with `cpSets` and `cpLists`, we focus on the most useful and significant ones. In particular, we choose to show rather lemmas dealing with the `cpSet` construct because it is a higher-level one and thus reasoning on sets of components is often preferable, when possible. Note however that most of the proofs dealing with distinctness of component names will rather use `cpLists`.

We start by an easy lemma quite heavily used and very easy to prove. It states that `C` is always in `cpSet(C)` (it is proved by cases on `C`).

```
lemma cpSetFirst: C ∈ cpSet C
```

The set of components inside a composite one can be decomposed as follows. It can be separated into the composite itself plus all the components in the `cpSet` of each sub-component.

```
lemma cpSetcomposite:
cpSet (Composite N itfs sub b s)={Composite N itfs sub b s}
                                ∪ {C.∃ C'∈set sub. C∈ cpSet C'}
```

This lemma is proved by an induction on lists of subcomponents. Conversely, we can prove that, if a component is in the `cpSet` of a subcomponent of a composite, it is in the `cpSet` of the composite. We also present a more general variant of this lemma stating that if `C''` is inside `C'` and `C'` is inside `C` then `C''` is inside `C`.

```
lemma cpSetcomposite_rev:
⟦ C∈ set sub; C'∈ cpSet C ⟧⟹ C'∈ cpSet (Composite N itfs sub b s)

lemma cpSetcpSet: ⟦C''∈ cpSet C';C'∈cpSet C⟧⟹ C''∈ cpSet C
```

Although those two lemmas are very easy to prove (by induction on the component structure), they are massively used in the other proofs.

Another theorem almost automatically proved by Isabelle, but exceedingly useful is the following one. It gives another formulation of the `getCp` construct.

```
lemma getCp_inlist: CL^N=Some C ⟹ C∈ set CL ∧ getName C=N
```

It is used to relate hypotheses in which a component name occurs and the component name, or the component structure. The reverse direction holds only if the component names inside CL are distinct as shown by the next lemma.

```
lemma getCpIdistinct:
⟦ distinct (map getName CL); getName C=N; C∈ set CL⟧ ⟹ CL^N=Some C
```

As the tools provided for the `distinct` construct in the Isabelle/HOL framework are a little weaker than for manipulating sets and lists, this proof is slightly longer and less automatic but still quite simple. Finally, the next lemma relates the `changeCp` primitive with the `getCp` one for the case that the name of the accessed component and the name of the changed one are different.

```
lemma upd_getCpunchanged: N≠getName C'⟹(CL <- C')^N = CL^N
```

Impact of design choices As a consequence of the mapping between component structure and Isabelle's structural support, it has been relatively easy to prove properties of component structure by automatic steps plus induction on the component structure. Consequently, the basic proofs on component sets and lists were relatively easy to handle: approximately 700 lines of code for the 80 lemmas dealing with component sets, component lists, and request identifiers, including the `getCp`, `getRecSubCp`, and `changeSubCp` primitives. By contrast, the proofs dealing with the semantics or correctness are generally much longer (several hundreds of lines per proof). However, the structural lemmas presented above are heavily used in the other proofs and strongly facilitate them.

3.6 Properties on Component Correctness

Based on the infrastructure for structural reasoning on the composition structure of components, we can now prove properties on the correctness of component structure presented in Section 3.4. The properties logically relate the degree of correctness of the structure. We present some of these lemmas here.

The lemma `CorrectCompWeak` establishes the relationship between `CorrectComponent` and `CorrectComponentWeak`.

```
lemma CorrectCompWeak: CorrectComponent C ⟹ CorrectComponentWeak C
```

`CorrectComponentListComp` establishes the correctness of the list of subcomponents given that the parent composite component is correct. Similarly, a member of a weakly correct component list is also weakly correct.

```
lemma CorrectComponentListComp:
     CorrectComponentWeak (Composite N  itfs subCp bindings s)
                                ⟹ CorrectComponentWeakList subCp
lemma CorrectComponentListComp_rev:
     ⟦CorrectComponentWeakList CL; C∈set CL⟧ ⟹ CorrectComponentWeak C
```

As a consequence, and as mentioned in Section 3.4, weak correctness entails weak correctness of subcomponents. Those lemmas imply that, when proving properties by induction, relying on weak correctness is very convenient as weak correctness can be used as the hypothesis of the recurrence hypothesis.

```
lemma SubComponent_CorrectComponentWeak:
    ⟦C'∈cpSet C; CorrectComponentWeak C⟧ ⟹ CorrectComponentWeak C'
```

The following property expresses a condition entailed in `CorrectComponentWeak`. `C^^N` returns the first subcomponent of `C` having the name `N`. If `C` is a weakly correct component, then there is a single component with that name, and thus the following hold:

```
lemma getRecSubCp_getName:
⟦CorrectComponentWeak C; C'∈ cpSet C⟧ ⟹ C^^(getName C') = Some C'
```

The proof of this property depends on properties on distinct names, and on the lemmas shown in this section and the preceding one.

Impact of design choices. The proofs in Isabelle/HOL are, for the most part of the correctness lemmas, almost automatic: unfolding the definitions, the proofs are mostly solved by applying the automatic tactic `auto`. Yet, these lemmas are important because they precisely relate different correctness conditions and consequently clarify subsequent proofs. They also entail properties of *compositionality*, i.e. what are the properties of a composite with respect to its constituents.

Other properties, like `getRecSubCp_getname` are harder to prove. Their proofs rely strongly on the provided infrastructure for structured components presented earlier in this section. Feasibility and readability of the proofs at the correctness level depends decisively on this clearly structured support with lemmas. Often the amount of automated proof work can be increased by adding our basic lemmas to the simplification sets of Isabelle/HOL.

4 Components at Runtime

4.1 Semantics

The formal semantics of our component model is given by a number of reduction relations defined by a set of inductive rules. These reduction relations along with the formal semantics of our component model appear in [13]; they were informally summarized in Section 2.2. This section illustrates the usefulness of the presented framework to specify and prove properties on the semantics by focusing on one reduction rule and one property. A smoothly working infrastructure of well-designed structural definitions and accompanying lemmas are prerequisite for mechanically proving properties over a structured component semantics.

We define a reduction relation $S \vdash C \rightarrow_R C', RL$ stating that in the component system S, a given component C reduces to a component C'. The list RL is used for specification of reply strategy that is not detailed here. We show below one specific communication rule CommChild, illustrated in Figure 3, and encoding the delegation of requests to a contained subcomponent.

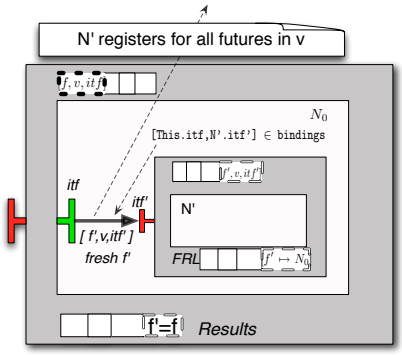

Fig. 3. CommChild rule

```
CommChild:
⟦ Cqueue s= R#Q; (|src=This(invkItf R),dest=N'.i2|)∈bindings;
                 f'∉set (RqIdList S) ; subCp^N' = Some C'⟧ ⟹
S⊢Composite N itf sub b s→R Composite N itf
    (sub<-(C'←(|id=f', parameter=(parameter R),invokedItf=i2|)) b
    (s(|Cqueue:=Q,CcomputedResults:=CcomputedResults s @
                                    [(|fid=id R,fValue=(0,[f'])|)]|),
                          (f,N)#(map (λ id.(id,N')) (snd(parameter R)))
```

The rule expresses request delegation between a composite component N and one of its subcomponents N'. The request R (shown as its constituents $[f, v, itf\,]$ in Figure 3) that has been sent to the parent N is dequeued from its request queue. A new future f' is created and added to the result list (`CcomputedResults`) of the parent as the result for this request R. A new request (shown as its constituents $[f', v, itf'\,]$) is enqueued in the subcomponent N'. In the Isabelle code snippet, we use the shortcut notation $\leftarrow$ for the enqueue operation. The target subcomponent is determined using the bindings: if *This.itf* is bound to $N'.itf'$ then the request is sent to the interface itf' of the subcomponent N', where *itf* is the external interface of N by which the request had arrived before. Note the use of the getCp primitive: `subCp^N'=Some C'` ensures that subcomponent of name `N'` exists and is `C'`. Also the `changeCp` primitive (`<-`) is quite useful here to update the subcomponent by enqueueing a new request to it.

Let us conclude this section by showing a property we proved in our framework that deals with component semantics. The following lemma shows that the set of names of components inside a component is unchanged by reduction.

```
lemma red_names_eq: ⟦S⊢c1→R c2, RL; CorrectComponentWeak c1⟧
        ⟹ getName ' (cpSet c2) = getName '(cpSet c1)
```

The proof is approximately 60 lines long, it is done by analysis on the reduction rule. It relies on a few lemmas relating names with reduction rules, and on most of the lemmas presented in Section 3. A crucial auxiliary lemma is the following one that is purely structural and unrelated with our semantics.

```
lemma upd_names_eq:
⟦CL^(getName c2)= Some c1; getName‘(cpSet c2)=getName‘(cpSet c1)⟧
          ⟹ getName‘(cpListset CL) = getName‘(cpListset (CL<-c2))
```

4.2 Reconfiguration

Reconfiguration represents all the transformations of the component structure or content that can be handled at runtime. We consider here mainly structural reconfiguration, which includes changes of the bindings, and of the content of a component. For example replacement of a primitive component by a new one is a form of reconfiguration that allows evolution of the business code.

In Fractal or GCM, configuration primitives are bind/unbind to manipulate bindings, add/remove to change the set of subcomponent of a composite component; also it is possible to start/stop a component.

Our framework enables reasoning on reconfiguration primitives and behaviour of a reconfigured component system. We illustrate below a few encodings of reconfiguration primitives and some theorems that can be proved in Isabelle/HOL thanks to our framework.

We illustrate reconfiguration capacities of our approach by defining two reconfiguration primitives and proving two related lemmas. But beforehand, we define the notion of *complete component*.

Completeness Similarly to [6], we say that a composite component is complete if all interfaces of its sub-components and all its internal interfaces are bound. This can be easily defined in Isabelle by the following primitive recursive predicate.

```
primrec Complete::Component⇒ bool where
 Complete (Primitive N itf s) = True |
 Complete (Composite N itf sub bindings s) =
   (∀ C∈set sub. allExternalItfsBound C bindings) ∧
   (allInternalItfsBound (Composite N itf sub bindings s) bindings) ∧
   (CompleteList sub)
```

Here, `allInternalItfsBound C b` checks that all external interfaces of `C` are bound by bindings `b`, and `allExternalItfsBound C b` that all internal interfaces of `C` are bound by bindings `b`. Finally, similar to `cpListlist` in Section 3.2, `CompleteList` recursively checks that all subcomponents are complete.

As there is no notion of optional interface in our model, this definition is really straightforward. For a complete component, any request emitted by a component will arrive at a destination component.

Unbind primitive The unbind primitive removes one of the bindings defined by a composite component.

```
primrec unbind:: Component⇒Binding⇒Component where
  unbind (Primitive N itf s) b = (Primitive N itf s) |
  unbind (Composite N itf sub bindings s) b =
          (Composite N itf sub (bindings-{b}) s)
```

Of course, un-binding does not maintain completeness, and this can be proved in our framework.

```
lemma unbinding_incomplete:
⟦b∈bindings; CorrectComponentStructure (Composite N itf sub bindings s)⟧
        ⟹ ¬ Complete (unbind (Composite N itf sub bindings s) b)
```

This lemma is proved in only 35 lines of simple Isabelle/HOL code, thanks to the properties presented in Section 3.5. The proof can be sketched as follows. `CorrectComponentStructure` imposes that in bindings `src b` is connected only once, thus, in `bindings-{b}`, `src b` is not connected anymore. Now, `src b` can be either `This N` if `b` connects an internal client interface to a sub-component, or of the form `CN.N` if it connects a sub-component to another interface. In the first case, the new component does not ensure `allInternalItfsBound` anymore, and in the second case, it is `allExternalItfsBound` that is not true for the component with name `CN`; note that `CorrectComponentStructure` ensures the existence of such a component.

Component replacement Let us now introduce a reconfiguration primitive that would automatically maintain completeness.

```
primrec Replace:: Component⇒Name⇒Component⇒Component where
Replace (Primitive N itf s) N1 C = (Primitive N itf s) |
Replace (Composite N itf sub binds s) N1 C = addSubCp (removeSubCp
(Composite N itf sub ((λb.RenameBinding b N1 (getName C))‘binds) s) N1) C
```

This primitive maintains completeness of a correct component as expressed in the following lemma:

```
lemma replace_complete:
 ⟦sub^(getName C')=None; sub^N'=Some oldC; getItfs oldC=getItfs C';
  Complete C'; Complete (Composite N itf sub bindings s);
  CorrectComponentStructure C';
  CorrectComponentStructure (Composite N itf sub bindings s)⟧
     ⟹ Complete (Replace (Composite N itf sub bindings s) N' C')
```

This lemma requires that all involved original components are correct and complete, that the replaced component is in the composition, but not the replacement one, and that those two components have the same interfaces. A similar lemma proving `CorrectComponentStructure` for the result of the replacement operation is also proved.

Of course, the replace primitive can be expressed by lower level reconfiguration operations, i.e. an unbind, remove, add, bind sequence. A lemma equivalent to the preceding one could also be proved. Such a lemma would be more general but a little more complex to express because it would need to relate the set of unbound bindings, the set of re-bound ones, and the component involved in the add-remove operations.

5 Conclusion

This paper presented the logical machinery of a mechanized framework for reasoning about structured component systems; especially targeting distributed components. We have first illustrated and motivated the specification of components and the provided proof infrastructure. Furthermore, we have shown this machinery in action by showing how reconfiguration of components can be formally specified, and how properties over component structure and reconfiguration can be handled. This paper also illustrated our approach by showing the specification of a semantics for components, and associated proofs. Overall, the developed framework consists of more than 4000 lines, including almost 300 lemmas and theorems, approximately 500 lines for defining the component model and its semantics, and 1800 lines focusing on properties specific to future registration which were not presented here. As usual with mechanised proofs, the main difficulty is the choice of the right structures providing the suitable level of abstraction. Some proofs are lengthy and technical but no major difficulty was encountered.

In contrast to existing works, our approach focuses on increasing confidence in global properties of component models. For this, we provide a framework and apply it to prove generally valid results. The established infrastructure of structured components with asynchronous communication provides an elegant abstraction from implementation detail while fully preserving the communication structure and defining a precise semantics. One limiting factor of our framework is that a precise semantics for components had to be chosen to allow mechanised proofs. Overall we have developed a reliable basis for the mechanical proofs of properties of hierarchical component models, and we have shown its adequacy to deal with first proofs entailing reconfiguration, or component semantics. We additionally provide subsequent support for distributed components communicating by asynchronous requests with futures.

A promising follow up project would be to analyse information flows based on this model, or properties entailing component synchronisation at reconfiguration time. More generally we expect to prove properties on reconfiguration that will entail reasoning simultaneously on component execution and on evolution of component structure. This would show the correctness of complex adaptation procedures that can be applied in autonomous component systems.

References

[1] Sensoria – software engineering for service-oriented overlay computers, 2005.

[2] Erika Ábrahám, Immo Grabe, Andreas Grüner, and Martin Steffen. Behavioral interface description of an object-oriented language with futures and promises. *Journal of Logic and Algebraic Programming*, 78(1-2):491–518, 2008.

[3] Toms Barros, Rabéa Ameur-Boulifa, Antonio Cansado, Ludovic Henrio, and Eric Madelaine. Behavioural models for distributed fractal components. *Annales des Télécommunications*, 64(1-2):25–43, 2009.

[4] Françoise Baude, Denis Caromel, Cédric Dalmasso, Marco Danelutto, Vladimir Getov, Ludovic Henrio, and Christian Pérez. GCM: A Grid Extension to Fractal for Autonomous Distributed Components. *Annals of Telecommunications*, accepted for publication, 2008.

[5] Michael Beisiegel, Henning Blohm, Dave Booz, Mike Edwards, and Oisin Hurley. SCA service component architecture, assembly model specification. Technical report, March 2007. `www.osoa.org/display/Main/Service+Component+Architecture+Specifications`.

[6] Eric Bruneton, Thierry Coupaye, and Jean Bernard Stefani. The Fractal Component Model. Technical report, ObjectWeb Consortium, February 2004. `http://fractal.objectweb.org/specification/index.html`.

[7] R. Bruni, A. Lluch Lafunete, U. Montanari, and E. Tuosto. Service oriented architectural design. In G. Barthe and C. Fournet, editors, *TGG 2007*, volume 4912 of *LNCS*, pages 186–203. Springer, 2008.

[8] Javier Cámara, Gwen Salaün, Carlos Canal, and Meriem Ouederni. Interactive Specification and Verification of Behavioural Adaptation Contracts. *2009 Ninth International Conference on Quality Software*, pages 65–75, August 2009.

[9] Carlos Canal, Pascal Poizat, and Gwen Salaün. Synchronizing behavioural mismatch in software composition. In *Proceedings of the 8th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 63–77, 2006.

[10] Antonio Cansado and Eric Madelaine. Specification and verification for grid Component-Based applications: From models to tools. In *Formal Methods for Components and Objects*, pages 180–203. 2009.

[11] CCA-Forum. The Common Component Architecture (CCA) Forum home page, 2005. http://www.cca-forum.org/.

[12] Immo Grabe, Martin Steffen, and Arild B. Torjusen. Executable interface specifications for testing asynchronous creol components. Technical Report Research Report No. 375, University Of Oslo, July 2008.

[13] Ludovic Henrio, Florian Kammüller, and Marcela Rivera. An asynchronous distributed component model and its semantics. In *FMCO 2008*. Springer, 2009.

[14] Ludovic Henrio and Muhammad Uzair Khan. Asynchronous components with futures: Semantics and proofs in isabelle/hol. In *Proceedings of the Seventh International Workshop, FESCA 2010*. ENTCS, 2010. To appear.

[15] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. In *SEFM '04: Proceedings of the Software Engineering and Formal Methods*, pages 188–197, Washington, DC, USA, 2004. IEEE Computer Society.

[16] Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. Creol: a type-safe object-oriented model for distributed concurrent systems. *Theor. Comput. Sci.*, 365(1):23–66, 2006.

[17] Muhammad Uzair Khan and Ludovic Henrio. First class futures: a study of update strategies. Research Report RR-7113, INRIA, 2009.

[18] Philippe Merle and Jean-Bernard Stefani. A formal specification of the Fractal component model in Alloy. Research Report RR-6721, INRIA, 2008.

[19] J. Meseguer. Conditional reqriting logic as a unified model of concurrency. *Journal of Theoretical Computer Science*, 96:73 – 155, 1992.

[20] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[21] OW2.Consortium. FraSCAti, Open SCA middleware platform. `https://wiki.objectweb.org/frascati/Wiki.jsp?page=FraSCAti`, 2009.