A formal specification of the Fractal component model in Alloy

Philippe Merle, Jean-Bernard Stefani

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A formal specification of the Fractal component model in Alloy

Philippe Merle, Jean-Bernard Stefani

N° 6721

November 2008

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6721--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

A formal specification of the Fractal component model in Alloy

Philippe Merle, Jean-Bernard Stefani

Thème COM — Systèmes communicants
Équipes-Projets ADAM, SARDES

Rapport de recherche n° 6721 — November 2008 — 44 pages

Abstract: This report contains a formal specification of the Fractal component model using the Alloy specification language. The report covers all the elements of the (informal) reference specification of the Fractal model. It provides a truly language-independent specification of the Fractal model, and lifts the ambiguities of the reference specification.

Key-words: Software components, component model, formal specification, software architecture, Fractal component model, Alloy specification language.

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Une spécification formelle du modèle de composants Fractal en Alloy

Résumé : Ce rapport contient une spécification formelle du modèle Fractal dans le langage de spécification Alloy. Le rapport couvre tous les éléments de la spécification (informelle) de référence du modèle Fractal. Il fournit une spécification du modèle Fractal réellement indépendante des langages de programmation et il résoud les ambiguités de la spécification de référence.

Mots-clés : Composants logiciels, modèle de composants, spécification formelle, architecture logicielle, modèle de composants Fractal, langage de spécification Alloy.

Contents

1 Introduction

Motivation The Fractal component model [9] is a programming-language-independent component model, which has been introduced for the construction of highy configurable software systems. The Fractal model combines ideas from three main sources: software architecture, distributed configurable systems, and reflective systems. From software architecture [24], Fractal inherits basic concepts for the modular construction of software systems, encapsulated components, and explicit connections between them. From reflective systems, Fractal inherits the idea that components can exhibit meta-level activities and reify through controller interfaces part of their internal structure. From configurable distributed systems, Fractal inherits explicit component connections across multiple address spaces, and the ability to define meta-level activites for runtime reconfiguration. The Fractal model has been used as a basis for the development of several kinds of configurable middleware, and has been used successfully for building automated, architecture-based, distributed systems management capabilities, including deployment and (re)configuration management capabilities [2, 11, 13, 14], self-repair capabilities [7, 25], overload management capabilities [8], and self-protection capabilities [12].

The Fractal model is currently defined by an informal specification [10]. The specification only briefly mentions the general foundations that constitute the Fractal model per se, and focuses mostly on default meta-level capabilities (or *controllers*, in Fractal parlance). The specification has been successfully implemented in different languages and environments, notably in Java and C, without giving rise to serious issues, which is a testimony to its consistency. However, there are aspects of the specification that remain decidedly insufficiently detailed or ambiguous. The present report attempts to correct these deficiencies by developing a formal specification of the Fractal component model which makes explicit the underlying general component model constituting the foundation of Fractal; which clarifies a number of ambiguities in the informal Fractal specification; and which identifies places where the informal Fractal specification may be overconstraining.

Beyond ensuring the consistency of the Fractal model, a formal specification for the Fractal model can serve several purposes:

- to provide a more abstract, programming-language-independent specification of the Fractal model;
- to allow a formal verification of Fractal designs;
- to provide the basis of a formal architecture description language for Fractal;
- to allow a formal specification and verification of Fractal tools;
- to allow a rigorous comparison with other component models.

The latter is important because the Fractal specification aims to define a very general component model (*e.g.*, meta-level capabilities in Fractal are not fixed, nor is the semantics of composition realized by composite components), from which more specialized component models can be derived and combined. A formal specification can

thus help in assessing whether a component model constitutes a proper refinement or specialization of the Fractal model.

Alloy The specification in this report is written in Alloy 4 [1, 15, 16], a lightweight specification language based on first-order relational logic. Alloy is interesting because of its simplicity and because of the straightforward usage of its analyzer, which acts essentially as a model checker and counter-example generator, and which enables rapid iterations between modelling and analysis when writing a specification (very much akin to debugging a specification). For a detailed introduction and motivation of Alloy, we refer the interested reader to the book [16]. An online tutorial for Alloy is also available on the Alloy Analyzer Web site [1].

Presentation The report is written in a litterate programming style: the specification is presented in its entirety, the (informal) commentary on the formal specification being interspersed with excerpts of the Alloy code. All assertions (Alloy *facts*) and theorems (Alloy *assertions*) have been checked with the Alloy analyzer, checking for the existence of finite models in the first case, and for the absence of counter-examples in models below a certain size in the second case. We do not introduce Alloy nor the Fractal model. Hopefully, the commentary running along the Alloy code excerpts will suffice.

Organization The report is organized as follows. Section 2 discusses related work. Section 3 details the Alloy specification of the core Fractal concepts. Sections 4 to 8 detail the Alloy specification of the different elements of the Fractal model, including the naming and (distributed) binding framework which is an integral part of Fractal (Section 4), the optional component controllers (meta-level functions) defined in the informal Fractal specification (Sections 5 and 6), the notion of component factory and bootstrap conditions for a Fractal system (Section 7), and the optional type system (Section 8). Section 9 shows that the overall specification is consistent, *i.e.*, that there exists a model for a component that combines all the features specified in Sections 3 to 8. Section 10 contrasts the formal specification presented in this report with the informal one, highlighting areas (and motives) of divergence between the two. Section 11 concludes the report with a mention of future work.

2 Related work

There have been several approaches to the formalization of component-based software and component models. Representative samples are provided by the two books [19, 20]. The two bodies of work closest to ours are: the co-algebraic approach developed by Barbosa, Meng et al. [3, 4, 21, 22], and the formal specification in Alloy of Microsoft COM component model developed by Jackson and Sullivan [17], following work by Sullivan et al. on the formal specification of the COM model in Z [27]. Although the presentation we give is relational, the notion of component or *kell* we

develop in this report is essentially coalgebraic in nature since a *kell* can be understood primarily as a set of transitions. Whereas Barbosa et al. develop a categorical framework, we prefer to adopt a simpler set-based approach: while we lose the benefit of dealing in the same way with multiple forms of behavior (*e.g.*, probabilistic, time-based, etc.) as in [3], the intuition is in our view better aided by a set-based presentation, and it is easier to understand for it directly generalizes the well-known notion of transition system. The COM specification presented in [17] focuses on the structural aspects of the COM model, and notably on the definition of its query interface and aggregation mechanism. While the Component controller in the Fractal specification provides much the same functionality as the query interface in COM, the Fractal model does not exhibit the COM-specific difficulty arising in the interplay between query interface and aggregation highlighted in [27], and possesses several forms of meta-level behavior (so-called *controllers* and *controller interfaces*). Thus, our work deals with a richer component model than the COM one, and deals in particular with the specification of meta-level behavior that subsumes and extends that of the COM model.

A recent work [26] presents a first-order relational logic formalization of the Fractal component model. The consistency of a part of this specification is proven manually while the consistency of our specification is checked by the Alloy Analyser. Moreover, this work deals only with certain structural aspects of the Fractal specification, while the present report covers the whole informal Fractal specification. Especially, AttributeController, ContentController, SuperController, NameController, GenericFactory, Factory, and TypeFactory interfaces are not formally specified in [26].

3 Foundations

This first part of the specification captures the underlying core of the Fractal model: a very general notion of component, called *kell* (a remote reference to the biological *cell*). At this level of abstraction, the notion of kell first emphasizes two facts:

- A kell has entry points, called *gates*. The notion of gate is an abstract form of the notion of interface in the Fractal specification. A gate constitutes a named point of interaction between a kell and its environment. The set of gates of a kell constitutes its sole means of interaction with its environment – *i.e.*, a kell is a unit of encapsulation.

- A kell may have subcomponents, called *subkells*. All transitions in a kell may act on the set of subcomponents, and modify it in arbitrary ways. This flexibility is key to allow different semantics for composition, and to support different kinds of meta-level operations – *i.e.*, operations operating on the internal structure and behavior of components.

The first primitive sets in the Alloy specification of the core Fractal model are given below [1].

[1] In Alloy, primitive sets are just sets of *atoms* – *i.e.*, elements which have no internal structure (and are not sets – atoms are sometimes called *urelements* in the logic litterature, *e.g.*, as in [5]). Primitive sets are

```
module fractal/foundations

sig Id {}
sig Val {}
sig Op extends Id {}
```

The above declarations introduce three primitive sets: Id, Val, and Op. They correspond to the set of *identifiers*, *base values*, and *operation names* respectively. Identifiers are just primitive forms of names or references. Base values represent values of some (unspecified) data types, such as integers, booleans, strings, etc. At this level of abstraction, the exact forms base values can take is of no import, hence their specification as just atoms [2].

The general notions of interface and component in the core Fractal model are given below. They are called, respectively, *gate* and *kell*.

```
sig Gate {
      gid: Id
}

sig Kell {
      gates: set Gate,
      sc: set Kell,
      kid: Id
}

fact GatesInKellHaveUniqueIds {
    all c:Kell | all i,j:c.gates | i.gid = j.gid implies i = j
}
```

A gate – *i.e.*, an element of the set Gate – is an entry point to communicate with a kell. The declaration above stipulates that a gate has an identifier [3]. A kell – *i.e.*, an element of the set Kell – is defined as having an identifier, given by the feature kid, a set of gates, given by the feature gates, and a set of sub kells, given by the feature sc. The fact that a kell has an identifier is necessary (*e.g.*, for management purposes) to manifest a notion of identity that persists throughout state changes. The Alloy fact named GatesInKellHaveUniqueIds expresses an invariant on kells[4], namely that gates belonging to a kell have distinct identifiers.

called *signatures* in Alloy, hence the keyword **sig** for introducing them. Note also the module declaration: in Alloy, specifications can be broken down into modules, which can then be imported for use in other modules using a declaration of the form: **open** moduleX **as** X, where X is some local name used, in the current module, as an abbreviation for the imported module.

[2] Keyword **extends** in Alloy indicates that a primitive set is declared as a subset of another one (and that it will form, with other subsets similarly declared, a partition of the set it *extends*).

[3] In Alloy, a declaration of the form gid:Id can be read as declaring a feature, or instance variable, of the class Gate; formally, it declares a binary relation gid : Gate $\rightarrow$ Id between the set of gates, Gate, and the set of identifiers, Id.

[4] This invariant takes the form of a simple first-order logical formula, where the keyword **all** denotes the universal quantifier $\forall$, where a declaration such as c:Kell denotes an arbitrary element c of the set Kell (likewise, i:c.gates denotes an arbitrary element i of the set of gates of the kell c – the dot notation c.gates is the standard notation for accessing a feature, or attribute, of an instance of a class). In a more classical logical notation, the GatesInKellHaveUniqueIds formula would read:

$$\forall \mathtt{c} \in \mathtt{Kell},\ \forall \mathtt{i}, \mathtt{j} \in \mathtt{gates}(\mathtt{c}),\ \mathtt{gid}(\mathtt{i}) = \mathtt{gid}(\mathtt{j}) \implies \mathtt{i} = \mathtt{j}$$

These elements provide the basic structure of a kell but do not explain how it behaves. This is captured by the definition of the set TKell below, a subset of Kell[5], which endows kells with *transitions*. Transitions are defined below as 4-tuples that comprise a set of *initial kells* (feature tsc), a set of *input signals* (feature sin), a set of *output signals* (feature sout), and a set of *residual kells* (feature res). Intuitively, the initial set of kells of a transition corresponds to subkells of the kell to which the transition belongs (the set of subkells on which the transition acts). The set of residual kells are the kells produced by the transition. The kell to which the transition belongs may or may not belong to the residual of the transition. This allows us to model *component factories*, as in the Fractal specification, *i.e.*, component that can create other components, or operations that delete or transform the target component. Effectively, this means that a kell can be seen as some sort of generalized Mealy machine (a labelled transition system, whose labels denote input and output signals handled during a transition).

```
sig TKell in Kell {
    transitions: set Transition
}

sig Transition {
    tsc: set Kell,
    sin: set Signal,
    sout: set Signal,
    res: set Kell
}

fact TransMayNotHaveDifferentSubComps { all c:TKell | all t:c.transitions | t.tsc = c.sc }
```

The invariant TransMayNotHaveDifferentSubComps ensures that the initial kells associated with each transition of a given kell c are indeed the subkells of c.

Signals are defined below as records of arguments (feature args), with a target gate (feature target), *i.e.*, the gate at which a signal is received (if it is an input signal) or emitted (if it is an output signal), and an operation name (feature op). In object-oriented terms, a signal looks very much like a reified method invocation.

```
sig Signal {
    target: Gate,
    operation: Op,
    args: Id -> set Arg
}

sig Arg in Id + Val + Gate + Kell {}

fact SignalsTargetGates { all c: TKell | c.transitions.(sin + sout).target in c.gates }
```

Signal arguments belong to the set Arg defined above as the union of four sets: identifiers, values, gates and kells. This means in particular that signal may carry gates (much as in the π-calculus messages may carry channel names [6]), and kells. The latter capability is not explicitly reflected in the Fractal specification, but is required to model

[5] In Alloy, **in** denotes the subset relation, or the set membership relation, + denotes set union, & denotes set intersection, − denotes set difference, # denotes set cardinality.

[6] Note that in the π-calculus, channels, *i.e.*, communication capabilities, are just names. Strictly speaking we could have avoided to include gates as possible arguments to signal, by just relying on identifiers. However, we have been careful in the specification to ensure that gates remain immutable, in contrast to kells

mobile agents and strong mobility, as well as deployment, checkpointing and reconfiguration capabilities[7]. The fact SignalsTargetGates expresses the invariant that all target gates in signals appearing in transitions of a kell c are gates of c.

Discussion

This completes the specification of the core Fractal model. As can be seen the core is very small, and it merely asserts that components are higher-order Mealy machines, that can be hierarchically organized. This core model allows a number of seemingly unusual, or unexpected, features:

- We allow kells to have no gates, and thus only internal behavior. As a result, the following assertion [8] is not valid:

  ```
  assert AllKellsHaveGates { all c:TKell | some c.gates }
  ```

  In contrast, the following assertion is valid:

  ```
  assert NoGateImpliesInternalActions {
    all c:TKell | (no c.gates) implies (no c.transitions.(sin + sout) )
  }
  ```

  It asserts that if a kell has no gate, then its transitions are merely internal as they involve no exchange of signals with the environment.

- We allow component structures with sharing – *i.e.*, a kell may be a subkell of two different kells. Thus, the following assertion to the contrary is invalid:

  ```
  assert SharingIsImpossible {
    all c1,c2:Kell | no cs:Kell {
      cs in c1.sc & c2.sc and (not c1 = c2) and (not c1 in c2.sc) and
        (not c2 in c1.sc)
    }
  }
  ```

  Component sharing is an original feature of the Fractal model, which as been found useful to model situations with resource sharing – *i.e.*, where components at different places in a component hierarchy need access to the same resource, such as a software library, or an operating system service.

(*i.e.*, no operation will transform a gate into another with the same identifier). Having a gate identifier or a gate itself as an argument are thus strictly equivalent, but allowing gates as signal arguments simplifies the specification.

[7] Alternatively, one could model this through marshalling and unmarshalling operations, allowing to transform a kell into a value, and vice versa. The above specification of arguments makes this higher-order character of signals and of the kell model explicit.

[8] An assertion in Alloy is written exactly like a fact, except for the use of the **assert** keyword to declare it. An invalid assertion is detected by the Alloy Analyzer when it generates a finite model that contradicts it (a counterexample). The Alloy keyword **some** denotes the existential qualifier. Thus, **some** c:Kell | P where P is some predicate, asserts the existence of some kell c verifying P. By extension, **some** s, where s is a set, asserts that s is not empty – *i.e.*, that there is some element in s.

- We allow component structures that are not well-founded – *i.e.*, where a kell may appear as a subkell of itself, or as a subkell of some of its subkells, etc. Thus, the following assertion to the contrary is invalid[9]:

```
assert ContainmentIsWellFounded {
    all c:Kell | (not c in c.*sc)
}
```

  This may seem counterintuitive, however this is not really different from allowing recursive procedure calls, and we therefore do not enforce it at the level of abstraction of the core model. The Fractal specification explicitly disallows this (ContainmentIsWellFounded would be written as an invariant – an Alloy fact), but this feature could be interesting to model recursive component structures. This is one occurrence where the present formal specification relaxes the constraints from the informal Fractal specification.

- We allow components to have a varying number of interfaces during their lifetime – *i.e.*, kells to have a varying number of gates in the course of their execution. Thus, the following assertion to the contrary is invalid:

```
assert NumberOfGatesInKellDoesNotVary {
    all c:TKell | all t:c.transitions | all cr:TKell {
        (cr in t.res and cr.kid = c.kid) implies (cr.gates = c.gates)
    }
}
```

- In contrast to most other component models (see *e.g.*, [18] for a discussion of recent ones), we do not need to introduce a notion of *connector* or *binding* to mediate the communication, or reify the communication paths, between components. Sharing, containment (*i.e.*, the kell-subkells relationship) and the fact that each component or kell institutes its own composition semantics suffice to make explicit communication channels in a component structure, and to define their semantics.

The specification of kells as Mealy machines has however one major drawback as an Alloy specification. Because transitions make explicit the state changes that a kell may go through, specifying state changes in the present specification amounts to require that certain facts hold, which would take the form of closure properties, such as "kells of this kind – and the kells that appear in the residues of their transitions – must have transitions of this sort". For instance, we would require all kells that support the Component gate to have certain transitions implementing the Component operations, and all the kells in the residues of their transitions to be kells of a similar kind. Closure properties of this kind are unfortunately instances of so-called *generator axioms* that may lead to a state explosion in models of the specification, which makes them impossible to analyze using the model checking approach of the Alloy Analyzer. This problem is an instance of the *unbounded universal quantifiers* problem discussed in Section 5.3 of [16], and needs to be avoided if we want to exploit the Alloy Analyzer

[9] In Alloy, *r of some relation r denotes the reflexive and transitive closure of r, while ^r denotes its transitive closure.

in assessing the consistency of the specification. Our approach in this specification is to not describe explicitly the set of transitions logically associated with a kell. Instead, we will define Alloy predicates that describe state changes on certain kells, but refrain from imposing that these state changes appear as explicit transitions in the supporting kells. In effect, for the purposes of this specification, we will deal only with elements of the Kell set, and will not consider elements of the TKell set. In the following sections, we adopt this approach: all properties and predicates considered will deal apply to elements of Kell.

Before moving to the specification of the (optional) default meta-level capabilities of the Fractal model, we gather here a number of declarations used in the rest of the specification. The distinction between Client and Server is here merely a primitive type distinction, which governs bindings between gates: to bind two gates together, one must be a dual of the other. The denominations Client and Server merely reflect that duality. The predicate isoKell can be interpreted as a strong identity predicate, whereby two kells are strongly identical if they have the same identifier, the same gates and the same subkells (they may differ in their internal state).

```
sig Client extends Gate {}
sig Server extends Gate {}
one sig NoSuchInterfaceException extends Val {}
one sig IllegalLifeCycleException extends Val {}
one sig Ok extends Val {}
pred isoKell(c:Kell, c1:Kell) {
    c.kid = c1.kid
    c.sc = c1.sc
    c.gates = c1.gates
}
```

4 Naming and binding

The naming and binding part of the specification captures the notions necessary for the construction of distributed configurations. We follow here the informal Fractal specification [10], Section 2.2.

The first concept is that of *name*. A name is merely an entity that is used to refer to another one. A name comes equipped with a reference to its naming context (feature context).

```
module fractal/naming

open util/relation as RR
open fractal/foundations as FF

sig Name {
      context: Id,
      pack: NamePickle
}

sig NamePickle {
    unpack: Name,
    context: Id
}
```

A name can also be *pickled* (or marshalled) to make it persistent or to communicate it between different machines. This is obtained through the combination of the pack feature and of operations encode and decode [10], defined as follows:

```
fact PackUnpackIdempotent { all n:Name | n.pack.unpack = n }

pred encode(n:Name, p:NamePickle, n1:name) {
    n1 = n
    p = n.pack
}

pred decode(nc:NamingContext, p:NamePickle, n:Name, nc1:NamingContext) {
   nc1 = nc
   p.context = nc.nid and p.unpack.context = nc.nid implies n = p.unpack
}

assert DecodingYieldsSameNameThanEncoded {
   all p:NamePickle, n:Name, nc:NamingContext {
         encode[n,p] and nc.nid = n.context implies decode[nc,p,n]
   }
}
```

Names exist only within *contexts*. Contexts are primarily associations between names and *referents* – *i.e.*, entities which are refered to by names. Making contexts explicit allows us to define different systems of names and to have them coexist and cooperate without the need to rely on a global naming authority to disambiguate independently created names. Contexts are defined as follows:

```
sig Referent in Gate + Name {}

sig NamingContext {
      nid: Id,
      exported: Name -> lone Referent
}
```

The feature exported in a naming context identifies the association between names in the context and their referents [11].

Two key invariants, given below, apply to names and contexts [12]. The first one merely asserts that names appearing in a context correctly refer to this context. The second one clarifies the fact that referents cannot be names that belong to the context (they can be names that belong to other contexts, though, thus allowing referral chains to be constructed across multiple contexts).

[10] Alloy allows the definition of first-order predicates, whose declarations start with keyword **pred** and are optionally followed by the list of the predicate arguments. Operations, that perform state changes in a system can be defined as predicates with some arguments corresponding to the initial state, *i.e.*, the state prior to the execution of the operation, and other arguments corresponding to the final state, *i.e.*, the state resulting from the execution of the operation. For a discussion on how to model state changes in Alloy, we refer the reader to Chapter 6 of [16].

[11] In Alloy, a declaration of the form exported: Name -> **lone** Referent denotes an injective binary relation between the set Name and the set Referent. The keyword **lone** is an example of a relation multiplicity. In our case, it signifies that a given name is to be associated with one, and only one, referent. Of course, two distinct names can have the same referent.

[12] Defined in the Alloy module util/relation, the function dom returns the domain of a binary relation and ran returns the range of a binary relation.

```
fact NameRefersToContext {
      all nc:NamingContext | all n:dom[nc.exported] | n.context = nc.nid
}

fact InContextNamesNotExported {
     all nc:NamingContext | all n:ran[nc.exported] | n.context != nc.nid
}
```

The main operation supported by a naming context is the export operation. Operation export returns a new name n for referent r in context nc. The name n is a valid name for referent r in the context nc. The naming context nc can for instance be a network context where remotely accessible interfaces are given names of a special form (*e.g.*, URLs for a Web service context). Note that a name can be exported as well. This is necessary to handle names across different naming contexts. A referent that is already a name in the target context cannot be exported. Operation export is specified below [13].

```
one sig NamingException extends Val {}

pred export(nc1, nc2: NamingContext, r: Referent, n:Name + NamingException) {
      let A = not (some s:Referent − r | n−>s in nc1.exported),
          B = (not r.context = nc1.nid) {
           (A and B) implies nc2.exported = nc1.exported + n −> r and nc1.nid = nc2.nid
           else n in NamingException and nc1 = nc2
      }
 }
```

One may verify a number of properties in relation to operation export. Here are a few self-explanatory ones:

```
assert ExportReturnsNewNameOrOldMap {
      all nc,ncc:NamingContext, r:Referent, n:Name |
          export[nc,ncc,r,n] implies
              let A = (not n in dom[nc.exported]),
                  B = (n.(nc.exported) = r),
                  C = (not r.context = nc.nid) {
                (A or B) and C
              }
}

assert ExportNameBelongsToContext {
      all nc,ncc:NamingContext, r:Referent, n:Name |
          export[nc,ncc,r,n] implies n.context = nc.nid
}

assert ExportExceptionLeavesContextUnchanged {
      all nc,ncc: NamingContext, r:Referent, n:NamingException |
          export[nc,ncc,r,n] implies nc = ncc
}
```

[13] Note the use of the Alloy construct **let** A = ... { S }, which just declares a variable A to stand as a denotation for some value, denotation which is then used inside the statement S. Note also the use of a nested implication of the form C1 **implies** F1 **else** F2, which is equivalent to (C1 **and** F1) **or** ((**not** C1) **and** F2). Note, finally, the keyword **one** which precedes the declaration of the NamingException value: it just signifies that the set NamingException is a singleton. In Alloy, set elements are essentially identified with singletons.

The following assertions highlight the fact that name resolution within a single naming context can be partial. To be complete, name resolution must typically take place across several naming contexts. However, we also allow partial name resolution across several naming contexts. This takes care of situations were name resolution cannot be carried out in full (*e.g.*, in disconnected situations) or need not be carried out in full (*e.g.*, when no access to a referenced interface or component is attempted).

```
assert ExportClosureIsJustExport {
    all nc:NamingContext | nc.exported = ^(nc.exported)
}

assert ExportClosureEndsInInterfaceOrNotInContextName {
     all nc:NamingContext, n:Name, r:Referent |
            r in n.(nc.exported) implies (r in Gate) or (r in Name and r.context != nc.nid)
}
```

A binder is a naming context that can resolve names and establish connections (bindings) towards entities referred to by resolved names. A binding is created typically by a bind operation. The creation of a binding results in the creation of a component that provides a (local) interface which corresponds to (*e.g.*, is a proxy to) the resolved name. A binder records the association (bindings) between resolved names and the (local) interfaces they refer to. Binders are specified below.

```
sig Binder extends NamingContext {
     bindings: Name -> lone Gate,
}

fact BindingNamesBelongToContext {
     all b:Binder | all n: dom[b.bindings] | n.context = b.nid
}

fact BindingsAndExportedDomainsDisjoint {
     all b:Binder | no (dom[b.bindings] & dom[b.exported])
}
```

The bind operation is specified below, together with some self-explanatory properties.

```
pred bind(b,b1:Binder, n:Name, i:Gate + NamingException) {
     b.nid = b1.nid
     n -> i in b.exported implies b = b1
     else i in Gate implies b1.bindings = b.bindings + n -> i
     else i in NamingException and b = b1
 }

assert BindExceptionLeavesBinderUnchanged {
      all b,b1:Binder, n:Name, i: NamingException | bind[b,b1,n,i] implies b = b1
}

assert BindReturnsNewInterfaceOrFromExported {
     all b,b1:Binder, n:Name, i:Gate | bind[b,b1,n,i] implies n -> i in b.exported + b1.bindings
}
```

Finally, one can prove a correct interplay between export and bind, namely that bind returns a local (in-context) gate referred to by a previously exported name.

```
assert BindReturnsPreviouslyExportedReferent {
    all b,b1:Binder, n:Name, i:Gate |
```

```
                export[b,b1,i,n] implies bind[b1,b1,n,i]
}
```

5 Basic introspection

The component controller in Fractal supports basic introspection capabilities: discovering all the interfaces associated with a component and their type. We follow here the informal Fractal specification [10], Section 3.

Our formal specification of the component controller begins with the declaration of the Type signature, with its subtype relation, noted sstypes. At this level of abstraction, the only property recorded of the subtype relation is that it constitutes a partial order [14].

```
module fractal/component

open util/relation as RR
open fractal/foundations as FF
open util/graph[Type] as GG

sig Type extends Val {
    sstypes: set Type
}

fact SubTypingIsPartialOrder { GG/dag[sstypes] }

sig ComponentType extends Type {}
sig InterfaceType extends Type {}
```

The next signatures declare the Component gates and the Interface gates. A Component gate is a server gate which also records the type of the component it belongs to. An Interface gate records its type, as well as the Component gate of the component it belongs to. As noted in the informal Fractal specification, this setting is similar to that adopted by the Microsoft COM model, with Component corresponding to the COM IUnknown interface.

```
sig Component extends Server {
    ctype: ComponentType
}

sig Interface in Gate {
    owner: Component,
    itype: InterfaceType,
}
```

A *ckell* is now defined as a kell with one gate which is an instance of Component (its other gates can be arbitrary gates). The fact CKellsHaveComponent constraints ckells to have only one Component gate.

```
sig CKell in Kell {
    comp: Component
}
```

[14] Note the use of the Alloy utility module graph and the predicate dag from this module.

```
fact CKellsHaveComponent { all c:CKell | c.comp = c.gates & Component }
```

Likewise, we define an *ikell* as a kell whose gates are all instances of Interface.

```
sig IKell in Kell {}

fact IKellsHaveInterfaces { all c: IKell | c.gates in Interface }
```

We now define *compkells* as ckells, which are also ikells, thus, as kells which have a Component gate, and whose gates are all instances of Interface.

```
sig CompKell in Kell {}

fact CompKellsAreCKellsAndIKells { CompKell = CKell & IKell }

fact InterfacesInCompKellsHaveOwner { all c: CompKell | all i:c.gates | i.owner = c.comp }
```

The basic properties of compkells are corroborated by the following simple, self-explanatory assertions.

```
assert OneComponentPerCompKell {
    all c:CompKell | one c.gates & Component
}

assert ComponentInCompKellsIsInterface {
    all c:CompKell | c.comp in Interface
}

assert CompKellsHaveOnlyInterfaces {
     no c:CompKell { some c.gates & (Gate − Interface) }
}

assert OwnersInCompKellsAreComponent {
    all c:CompKell | all i:c.gates | i.owner in Component & Interface
}
```

Before specifying the different operations that are attached to Component and Interface, we first define an equivalence predicate on compkells. Roughly, isoCKell indicates that two compkells have the same identifier, the same subcomponents, and the same gates – *i.e.*, their internal and external structures (but not necessarily their exact states) are the same. By virtue of the above CKellsHaveComponent invariant, two equivalent compkells have the same Component gate.

```
pred isoCKell(c:CompKell, c1:CompKell) {
    isoKell[c,c1]
}

assert IsoCompKellsHaveSameComponent {
    all c,c1:CompKell | isoCKell[c,c1] implies c.comp = c1.comp
}
```

We give below the different operations attached to a Component gate. Operation getInterfaces returns the set of gates is of a compkell c, given its Component interface o. In the process, compkell c becomes compkell c1, which is required to be equivalent to c – *i.e.*, have the same gates, the same identifier, and the same subkells. Operation

getInterface returns the gate i whose identifier iid is passed as argument to the operation. In the process, compkell c becomes compkell c1. Operation getCType returns the component type ct of compkell c.

```
// Operations from the Component interface

pred getInterfaces(c:CompKell, o:Component, is: set Interface, c1:CompKell) {
    o = c.comp
    is = c.gates
    isoCKell[c,c1]
}

pred getInterface(c:CompKell, o:Component, iid:Id, i:Interface + NoSuchInterfaceException,
                  c1:CompKell) {
    o = c.comp
    isoCKell[c,c1]
    i in c.gates implies iid = i.gid
    else i = NoSuchInterfaceException
}

pred getCType(c:CompKell, o:Component, ct: ComponentType, c1:CompKell) {
    o = c.comp
    ct = o.ctype
    isoCKell[c,c1]
}
```

The specification of the above operations provide examples of ambiguities that arise in the informal Fractal specification (in fact, in any informal specification), and which are difficult to weed out without a formal model. In fact, [10] leaves unspecified the exact postconditions of operations. Here we strike a middleground between a strong form which would require that the target compkell be left untouched, *i.e.*, that would specify c = c1 in place of our isoCKell[c,c1], and a very weak form which would only require c.kid = c1.kid. The strong form would forbid any kind of side-effect to such meta-level operations (such as setting a counter or updating a log of such operations), whereas the very weak form would make these introspection operations essentially useless (since the obtained information would be obsolete as soon as it is obtained).

We specify below operations associated with an Interface gate. Operation getOwner returns the Component gate associated with the compkell c that hosts the target Interface gate i. Operation getName returns the identifier of the target Interface gate i. Operation getType returns the interface type it of the target Interface gate i.

```
// Operations from the Interface interface

pred getOwner(c:CompKell, i:Interface, o:Component, c1:CompKell) {
    i in c.gates
    o = i.owner
    isoCKell[c,c1]
}

pred getName(c:CompKell, i:Interface, iid:Id, c1:CompKell) {
    i in c.gates
    iid = i.gid
    isoCKell[c,c1]
}
```

```
pred getType(c:CompKell, i:Interface, it: InterfaceType, c1:CompKell) {
    i in c.gates
    it = i.itype
    isoCKell[c,c1]
}
```

We give below two simple properties, which assert the consistency of the Component and Interface operations.

```
assert ComponentToInterfacesAndBack {
    all c,c1:CompKell | all i: Interface | all is: set Interface {
        getInterfaces[c,c.comp,is,c1] and i in is implies getOwner[c,i,c.comp,c1]
    }
}

assert InterfaceToComponentAndBack {
    all c,c1:CompKell | all i: Interface | all is: set Interface {
        getOwner[c,i,c.comp,c1] and getInterfaces[c,c.comp,is,c1] implies i in is
    }
}
```

The informal Fractal specification also defines the operation isInternal into the Interface interface. However for a better separation of concerns between the specification of controllers, this operation is defined later in Section 6.3. This is due to the fact that internal interfaces are managed by ContentController.

6 Configuration

6.1 Attribute control

The AttributeController interface in Fractal allows to configure properties of a component. We follow here the informal Fractal specification [10], Section 4.2.

We specify below kells with AttributeController gates – *i.e.*, elements of ACKell or *ackells*.

```
module fractal/attribute

open fractal/foundations as FF

sig AttributeController extends Server {}

sig ACKell in Kell {
      actrl: AttributeController
}

fact AttributeControllerInACKellIsExternalGate { all c:ACKell | c.actrl in c.gates }
```

We now define an equivalence predicate on ackells. Two such kells are equivalent if they have the same identifier, the same subkells, the same gates and the same AttributeController gate.

```
pred isoACKell(c:ACKell, c1:ACKell) {
    isoKell[c,c1]
    c1.actrl = c.actrl
}
```

The informal Fractal specification does not define operations for the AttributeController interface. However, the informal specification defines a convention for defining application-specific attribute controllers. We now give an example of how to formalize this convention. This example component provides both getter and setter operations for a foo property. We define two signatures: ExampleAttributeController as a subset of AttributeController gates, and ExampleKell as a subset of ackells containing the foo field and where the attribute controller is an ExampleAttributeController gate.

```
sig ExampleAttributeController extends AttributeController {}

sig ExampleACKell in ACKell {
    foo: Val
} {
    actrl in ExampleAttributeController
}
```

We specify below the getter and setter operations for the foo property.

```
pred getFoo(c:ExampleACKell, ac:ExampleAttributeController, r:Val, c1:ExampleACKell) {
   c.actrl = ac
   r = c.foo
   isoACKell[c,c1]
}

pred setFoo(c:ExampleACKell, ac:ExampleAttributeController, v:Val, c1:ExampleACKell) {
   c.actrl = ac
   c1.foo = v
   isoACKell[c,c1]
}
```

Finally we give a consistency property on these two operations.

```
assert GetFooAfterSetFoo {
  all c,c1: ExampleACKell, v:Val |
       setFoo[c, c.actrl, v, c1] implies getFoo[c1, c1.actrl, v, c1]
}
```

6.2 Binding control

The BindingController interface in Fractal supports the binding of client interfaces of a component to server interfaces. The effect of this binding is to allow the components that are connected via these bound interfaces to communicate. We follow here the informal Fractal specification [10], Section 4.3.

In our case, we do not specify the exact effect of binding a client and a server gate, since the semantics of this binding typically depends on the enclosing component where it takes place. However kells providing a BindingController gate record which client interfaces are bound (feature bindings in an instance of BCKell). Notice the different constraints that apply:

- a client gate is bound at most to a single server gate (**lone** multiplicity in bindings feature declaration);
- client gates must be gates of the hosting kell (fact ClientsInBindingCntrlAreBCKellGates);

- the bindings relation records the binding of client gates (fact BindingsBindClientGates).

```
module fractal/binding

open util/relation as RR
open fractal/foundations as FF

sig BindingController extends Server {}

sig BCKell in Kell {
    bctrl: BindingController,
    clients: set Client,
    bindings: Client -> lone Server
}

fact BindingsBindClientGates {
   all c:BCKell | dom[c.bindings] in c.clients
}

fact ClientsInBindingCntrlAreBCKellGates {
    all c:BCKell | c.clients in c.gates
}

fact BindingControllerInBCKellIsExternalGate {
    all c:BCKell | c.bctrl in c.gates
}
```

The following assertion is valid as it is implied by the conjunction of ClientsInBinding-CntrlAreBCKellGates and GatesInKellHaveUniqueIds invariants.

```
assert ClientsInBCHaveUniqueIds {
   all c:BCKell | all ci,cj:c.clients | ci.gid = cj.gid implies ci = cj
}
```

Before specifying the operations attached to BindingController gates, we define an equivalence predicate between kells with a BindingController gate. Two such kells are equivalent if they have the same identifier, the same gates, the same subkells, the same client gates and the same BindingController gate.

```
pred isoBCKell(c:BCKell, c1:BCKell) {
     isoKell[c,c1]
     c1.clients = c.clients
     c1.bctrl = c.bctrl
}
```

We specify below the different operations associated with BindingController gates. Operation list returns the set of client gate identifiers of the kell hosting the target BindingController gate; in the process, the hosting kell c evolves into kell c1. Operation lookup returns the server gate i that is bound to the client gate whose identifier iid is passed as argument. Operation bind binds the client interface whose identifier cid is passed as argument to the server gate si passed as argument. Finally, operation unbind unbinds the client gate whose identifier cid is passed as argument.

```
// Operations from the BindingController interface

one sig IllegalBindingException extends Val {}
```

```
sig BindingReturn in Ok + NoSuchInterfaceException + IllegalBindingException
                          + IllegalLifeCycleException {}

pred list(c:BCKell, bc:BindingController, r: set Id, c1:Kell) {
    c.bctrl = bc
    r = c.clients.gid
    isoBCKell[c,c1]
}

pred lookup(c:BCKell, bc:BindingController, iid: Id, i: Server + NoSuchInterfaceException,
            c1:Kell) {
    c.bctrl = bc
    isoBCKell[c,c1]
    iid in (c.clients).gid implies (some ci: Client { ci.gid = iid and ci in c.clients and
                                                      i in ci.(c.bindings) })
    else i = NoSuchInterfaceException
}

pred bind(c:BCKell, bc:BindingController, cid: Id, si:Server, r: BindingReturn, c1:Kell) {
     bc = c.bctrl
     some ci:Client {
        r = IllegalLifeCycleException implies isoBCKell[c,c1]
        else no cid & (c.clients).gid implies r = NoSuchInterfaceException and isoBCKell[c,c1]
        else some ci.(c.bindings) implies r = IllegalBindingException and isoBCKell[c,c1]
        else ( cid in (c.clients).gid and no ci.(c.bindings) and
                ci.gid = cid and ci in c.clients and
                r = Ok and c1.bindings = c.bindings + ci -> si and
                isoBCKell[c,c1] )
    }
}

pred unbind(c:BCKell, bc:BindingController, cid: Id, r: BindingReturn, c1:Kell) {
    some ci:Client, si: Server {
        c.bctrl = bc
        r = IllegalLifeCycleException implies isoBCKell[c,c1]
        else no cid & (c.clients).gid implies r = NoSuchInterfaceException and isoBCKell[c,c1]
        else no ci.(c.bindings) implies r = IllegalBindingException and isoBCKell[c,c1]
        else ( cid in (c.clients).gid and ci -> si in (c.bindings) and
                ci.gid = cid and r = Ok and
                c1.bindings = c.bindings - ci -> si and
                isoBCKell[c,c1] )
    }
}
```

We give below a number of properties that assess the mutual consistency of the different BindingController operations. Predicates getClient, getBoundClient, and getBoundServer are just abbreviations for some simple conditions. The last two properties UnbindAfterBind Possible and BindAfterUnbindPossible are commutation conditions on the bind and unbind operations.

```
assert LookupAfterBindYieldsCorrectServer {
   all c:BCKell, cid: Id, si:Server, r: Ok, c1:BCKell |
      bind[c,c.bctrl,cid,si,r,c1] implies lookup[c1,c1.bctrl,cid,si,c1]
}

pred getClient(c:BCKell, cid:Id, ci:Client) {
```

```
      cid = ci.gid
      ci in c.clients
}

pred getBoundClient(c:BCKell, cid:Id, ci:Client) {
    getClient[c,cid,ci]
    ci in dom[c.bindings]
}

pred getBoundServer(c:BCKell, cid:Id, si:Server) {
   some ci:Client | getBoundClient[c,cid,ci] and ci -> si in c.bindings
}

assert UnbindPossibleMeansBindingExists {
    all c:BCKell, cid:Id, c1:BCKell {
       unbind[c,c.bctrl,cid,Ok,c1] implies some s:Server { s in Client.(c.bindings) }
    }
}

assert UnbindAfterBindPossible {
   all c:BCKell, cid: Id, si:Server, c1:Kell |
      bind[c,c.bctrl,cid,si,Ok,c1] implies unbind[c1,c1.bctrl,cid,Ok,c]
}

assert BindAfterUnbindPossible {
   all c:BCKell, cid: Id, c1:BCKell, si:Server {
      unbind[c,c.bctrl,cid,Ok,c1] and getBoundServer[c,cid,si] implies
          bind[c1,c1.bctrl,cid,si,Ok,c]
   }
}
```

6.3 Content control

The ContentController interface in Fractal allows to introspect the internal structure of a component in the form of its so-called *internal interfaces* and of its subcomponents. We follow here the informal Fractal specification [10], Section 4.4.

We specify below kells with ContentController gates – *i.e.*, elements of CCKell. Internal gates appear only as a set of gates, which are not gates for interaction with the environment (*i.e.*, the exterior) of a kell. There are no further semantics associated with this notion of internal gate, since it typically varies with each kell (internal gates typically allow to explicitly connect subkells to some inner functionality of their parent kell). Making explicit internal gates allows to control, through the ContentController gate, the internal connections between a parent kell and its subkells. Instances of CCKell also provide access to (in general, a subset of) their subkells (feature subcomps in the CCKell signature). Notice that CCKell is defined as a subset of CKell – *i.e.*, each instance of CCKell has both a ContentController gate and a Component gate. All subkells of a kell in CCKell, or *cckell*, are ckells – *i.e.*, they all have a Component gate.

```
module fractal/content

open util/relation as RR
open fractal/foundations as FF
open fractal/component as FC
```

```
sig ContentController extends Server {}

sig CCKell in Kell {
      cctrl: ContentController,
      internals: set Gate,
      subcomps: set CKell
}

fact ContentControllerInCCKellIsExternalGate { all c:CCKell | c.cctrl in c.gates }

fact InternalsAreNotExternalsInCCKells { all c: CCKell | no (c.internals & c.gates) }

fact SubcompsAreSubComponentsInCCKells { all c: CCKell | c.subcomps in c.sc }

fact InternalsIdsAreDistinct { all c:CCKell | all g,g1:c.internals | g.gid = g1.gid implies g = g1 }

fact CCKellsHaveDistinctComponentsInSubComps {
      all c:CCKell | all c1,c2:c.subcomps {
          c1.comp = c2.comp implies c1 = c2
          c1.kid = c2.kid implies c1 = c2
      }
}

assert CCKellsHaveCompAsIdsInSubComps {
     all c:CCKell | all c1,c2:c.subcomps {
          c1.kid = c2.kid <=> c1.comp = c2.comp
     }
}
```

As mentioned previously, the Interface interface contains the operation isInternal, which returns true if an interface i is an internal interface or false if it is an external interface.

```
// Operation from the Interface interface

pred isInternal(c:CompKell, i:Interface, c1:CompKell) {
   c in CCKell and i in c.internals
   isoCKell[c,c1]
}
```

We now define an equivalence predicate on cckells. Two such kells are equivalent if they have the same identifier, the same subkells, the same gates, the same ContentController gate, the same internal interfaces, and the same subcomponents.

```
pred isoCCKell(c:CCKell, c1:CCKell) {
   isoCKell[c,c1]
   c.cctrl = c1.cctrl
   c.internals = c1.internals
   c.subcomps = c1.subcomps
}
```

We specify below the different operations attached to a ContentController gate. Operation getInternalInterfaces returns the set of internal interfaces of the cckell hosting the target ContentController gate. Operation getInternalInterface returns the internal gate whose identifier iid is passed as argument. Operation getSubComponents returns the set scc of subkells accessible via the ContentController gate. Operation addSubComponent adds a

ckell designated by its Component gate icc to the set of subcomps of the host cckell. Operation removeSubComponent does the reverse.

```
// Operations from the ContentController interface

one sig IllegalContentException extends Val {}

one sig ContentReturn in Ok + IllegalContentException + IllegalLifeCycleException {}

pred getInternalInterfaces(c:CCKell, cc:ContentController, sg: set Gate, c1:Kell) {
    cc = c.cctrl
    sg = c.internals
    isoCCKell[c,c1]
}

pred getInternalInterface(c:CCKell, cc:ContentController, iid: Id, ig: Gate, c1:Kell) {
    cc = c.cctrl
    ig.gid = iid
    ig in c.internals
    isoCCKell[c,c1]
}

pred getSubComponents(c:CCKell, cc:ContentController, scc: set Component, c1:Kell) {
    cc = c.cctrl
    scc = (c.subcomps).comp
    isoCCKell[c,c1]
}

pred addSubComponent(c:CCKell, cc:ContentController, icc: Component, r: ContentReturn,
c1:Kell) {
  some scc:CKell {
    cc = c.cctrl
    icc = scc.comp
    r = IllegalLifeCycleException implies isoCCKell[c,c1]
    else icc in c.subcomps.comp implies r = IllegalContentException and isoCCKell[c,c1]
    else r = IllegalContentException implies isoCCKell[c,c1]
    else r = Ok and c1.kid = c.kid and c1.comp = c.comp and c1.cctrl = c.cctrl and
         scc in c1.subcomps
  }
}

pred removeSubComponent(c:CCKell, cc:ContentController, icc:Component, r:ContentReturn,
c1:CCKell) {
   cc = c.cctrl
   r = IllegalLifeCycleException implies isoCCKell[c,c1]
   else no icc & c.subcomps.comp implies r = IllegalContentException and isoCCKell[c,c1]
   else r = IllegalContentException implies isoCCKell[c,c1]
   else some scc: c.subcomps {
            icc = scc.comp and
            r = Ok and
            c1.cctrl = c.cctrl and c1.comp = c.comp and c1.kid = c.kid and
            no scc & c1.subcomps
        }
}
```

Finally we give some consistency properties on operations.

```
assert RemoveAfterAddIsPossible {
```

```
    all c,c1: CCKell, icc:Component {
        (addSubComponent[c,c.cctrl,icc,Ok,c1] and
         c1.gates = c.gates and
         some scc:CKell {scc.comp = icc and c1.subcomps = c.subcomps + scc} ) implies
            removeSubComponent[c1,c1.cctrl,icc,Ok,c]
    }
}


assert AddAfterRemoveIsPossible {
    all c,c1: CCKell, icc:Component {
        (removeSubComponent[c,c.cctrl,icc,Ok,c1] and
         c1.gates = c.gates and
         some scc:CKell {scc.comp = icc and c1.subcomps = c.subcomps - scc} ) implies
            addSubComponent[c1,c1.cctrl,icc,Ok,c]
    }
}

assert GetSubCompSucceedsAfterAdd {
    all c,c1: CCKell, icc:Component {
        addSubComponent[c,c.cctrl,icc,Ok,c1] implies
            (getSubComponents[c1,c1.cctrl,c1.subcomps.comp, c1] and icc in c1.subcomps.comp)
    }
}
```

The following GetSubCompFailsAfterRemove property does not hold. Because of the weak conditions on removeSubComponent, it may well be that a component with the same Component gate icc exists as a subcomponent of a component from which a component with Component gate icc has just been removed.

```
assert GetSubCompFailsAfterRemove {
    all c,c1: CCKell, icc:Component {
        removeSubComponent[c,c.cctrl,icc,Ok,c1] implies no icc & c1.subcomps.comp
    }
}
```

The following assertions are not valid either – *i.e.*, adding or removing a component do not imply that all previous subkells, subcomponents, gates, and internal interfaces are preserved.

```
assert AddPreservePreviousSubKells {
    all c,c1:CCKell, icc:Component {
        addSubComponent[c,c.cctrl,icc,Ok,c1] implies
            some ck:CKell { ck.comp = icc and c1.sc = c.sc + ck }
    }
}

assert AddPreservePreviousSubComponents {
    all c,c1:CCKell, icc:Component {
        addSubComponent[c,c.cctrl,icc,Ok,c1] implies
            some ck:CKell { ck.comp = icc and c1.subcomps = c.subcomps + ck }
    }
}

assert AddPreservePreviousGates {
    all c,c1:CCKell, icc:Component {
      addSubComponent[c,c.cctrl,icc,Ok,c1] implies c1.gates = c.gates
```

```
    }
}

assert AddPreservePreviousInternals {
    all c,c1:CCKell, icc:Component {
      addSubComponent[c,c.cctrl,icc,Ok,c1] implies c1.internals = c.internals
    }
}

assert RemovePreservePreviousSubKells {
    all c,c1:CCKell, icc:Component {
      removeSubComponent[c,c.cctrl,icc,Ok,c1] implies
          some ck:CKell { ck.comp = icc and c1.sc = c.sc − ck }
    }
}

assert RemovePreservePreviousSubComponents {
    all c,c1:CCKell, icc:Component {
      removeSubComponent[c,c.cctrl,icc,Ok,c1] implies
          some ck:CKell { ck.comp = icc and c1.subcomps = c.subcomps − ck }
    }
}

assert RemovePreservePreviousGates {
    all c,c1:CCKell, icc:Component {
      removeSubComponent[c,c.cctrl,icc,Ok,c1] implies c1.gates = c.gates
    }
}

assert RemovePreservePreviousInternals {
    all c,c1:CCKell, icc:Component {
      removeSubComponent[c,c.cctrl,icc,Ok,c1] implies c1.internals = c.internals
    }
}
```

6.4 Super control

The SuperController interface in Fractal allows to introspect the super components of a component, *i.e.*, the components that contain a component. We follow here the informal Fractal specification [10], Section 4.4.

We specify below kells with SuperController gates – *i.e.*, elements of SCKell. Instances of SCKell provide access to (in general, a subset of) their super components (feature supercomps in the SCKell signature) [15]. All super components of a kell in SCKell, or *sckell*, are ckells – *i.e.*, they all have a Component gate.

```
module fractal/super

open fractal/foundations as FF
open fractal/component as FC

sig SuperController extends Server {}
```

[15] In Alloy, ~r of some binary relation r denotes the transpose of r, forming a new relation by reversing the order of atoms in each tuple.

```
sig SCKell in Kell {
      sctrl: SuperController,
      supercomps: set CKell
}

fact SuperControllerInSCKellIsExternalGate { all c:SCKell | c.sctrl in c.gates }

fact SupercompsAreSuperKells { all c: SCKell | c.supercomps in (~sc)[c] }

fact SCKellsHaveDistinctComponentsInSuperComps {
      all c:SCKell | all c1,c2:c.supercomps {
                  c1.comp = c2.comp implies c1 = c2
                  c1.kid = c2.kid implies c1 = c2
      }
}

assert SCKellsHaveCompAsIdsInSuperComps {
     all c:SCKell | all c1,c2:c.supercomps {
          c1.kid = c2.kid <=> c1.comp = c2.comp
     }
}
```

We now define an equivalence predicate on sckells. Two such kells are equivalent if they have the same identifier, the same subkells, the same gates, the same SuperController gate, and the same super components.

```
pred isoSCKell(c:SCKell, c1:SCKell) {
   isoKell[c,c1]
   c.sctrl = c1.sctrl
   c.supercomps = c1.supercomps
}
```

We specify below the operation attached to a SuperController gate. Operation getSuperComponents returns the set scc of super components accessible via the SuperController gate.

```
// Operation from the SuperController interface

pred getSuperComponents(c:SCKell, sc:SuperController, scc: set Component, c1:Kell) {
     sc = c.sctrl
     scc = (c.supercomps).comp
     isoSCKell[c,c1]
}
```

6.5 Name control

The NameController interface in Fractal allows to associate a name to a component. We follow here the informal Fractal specification [10], Section 4.4.

We specify below kells with NameController gates – *i.e.*, elements of NCKell. Instances of NCKell, or *nckells*, provide access to a name (feature name in the NCKell signature).

```
module fractal/name

open fractal/foundations as FF
```

```
sig NameController extends Server {}

sig NCKell in Kell {
      nctrl: NameController,
      name: Id
}

fact NameControllerInNCKellIsExternalGate { all c:NCKell | c.nctrl in c.gates }
```

We now define an equivalence predicate on nckells. Two such kells are equivalent if they have the same identifier, the same subkells, the same gates, and the same NameController gate.

```
pred isoNCKell(c:NCKell, c1:NCKell) {
   isoKell[c,c1]
   c.nctrl = c1.nctrl
}
```

We specify below operations attached to a NameController gate. Operation getName returns the name n of the nckell hosting the target NameController gate. Operation setName sets the name n of a component.

```
// Operations from the NameController interface

pred getName(c:NCKell, nc:NameController, n:Id, c1:NCKell) {
     c.nctrl = nc
     n = c.name
     isoNCKell[c,c1]
}

pred setName(c:NCKell, nc:NameController, n:Id, c1:NCKell) {
     c.nctrl = nc
     c1.name = n
     isoNCKell[c,c1]
}
```

Finally we give a consistency property on these two operations.

```
assert GetNameAfterSetName {
     all c,c1:NCKell, n:Id | setName[c,c.nctrl,n,c1] implies getName[c1,c1.nctrl,n,c1]
}
```

6.6 Lifecycle control

The LifecycleController interface in the Fractal model provides basic capabilities to control the execution of a component. The execution of a component from the point of view of this LifecycleController is abstracted as evolving between two macro-states, Started and Stopped. We follow here the informal Fractal specification [10], Section 4.5.

We specify first these two macro-states.

```
module fractal/lifecycle

open util/relation as RR
open fractal/foundations as FF
```

```
sig LFState extends Val {}
one sig Started extends LFState {}
one sig Stopped extends LFState {}
```

We then define the set LFKell that offers a LifecycleController gate. The feature ctrls identifies the set of "control" gates – *i.e.*, those gates whose operations are not inhibited when in the Stopped state.

```
sig LifeCycleController extends Server {}

sig LFKell in Kell {
    lfctrl: LifeCycleController,
    state: LFState,
    ctrls: set Gate
}

fact LFCtrlIsACntrlGate { all c:LFKell | c.lfctrl in c.ctrls }

fact CtrlGatesAreInGates { all c:LFKell | c.ctrls in c.gates }

fact LFStateIsStoppedOrStarted { all c:LFKell | c.state in Started + Stopped }
```

We define first an equivalence predicate between kells with LifecycleController gates. Two such kells are equivalent if they have the same identifier, the same subkells, the same gates, the same LifecycleController gate, the same macro-state, and the same control gates.

```
pred isoLFKell(c:LFKell, c1:LFKell) {
  isoKell[c,c1]
  c.lfctrl = c1.lfctrl
  c.state = c1.state
  c.ctrls = c1.ctrls
}
```

We specify below the operations attached to LifecycleController gates. Operation getState returns the macro-state s of the kell hosting the target LifecycleController gate lfc. Operation start places the kell c hosting the target LifecycleController gate lfc into the Started macro-state. Operation stop places the kell into the Stopped macro-state. This may imply all sorts of changes in c, hence the weak constraint on the resulting kell c1: it has the same identifier than c, and the same LifecycleController interface, and it is in the Started or Stopped macro-state respectively.

```
// Operations from the LifecycleController interface

sig LFReturn in Ok + IllegalLifeCycleException {}

pred getState(c:LFKell, lfc:LifeCycleController, s:LFState, c1:LFKell) {
    c.lfctrl = lfc
    s = c.state
    isoLFKell[c,c1]
}

pred start(c:LFKell, lfc:LifeCycleController, r:LFReturn, c1:LFKell) {
     c.lfctrl = lfc
     r = IllegalLifeCycleException implies isoLFKell[c,c1]
     else c.state = Started implies r = Ok and isoLFKell[c,c1]
```

```
        else c.state = Stopped and r = Ok and c1.state = Started and c1.kid = c.kid and
             c1.lfctrl = c.lfcrtrl
}

pred stop(c:LFTKell, lfc:LifeCycleController, r:LFReturn, c1:LFTKell) {
    c.lfctrl = lfc
    r = IllegalLifeCycleException implies isoLFKell[c,c1]
    else c.state = Stopped implies r = Ok and isoLFKell[c,c1]
    else c.state = Started and r = Ok and c1.state = Stopped and c1.kid = c.kid and
         c1.lfctrl = c.lfcrtrl
}
```

We give some consistency properties on these three operations.

```
assert GetStateReturnsStartedAfterStart {
  all c,c1:LFKell {
    start[c,c.lfctrl,Ok,c1] implies getState[c1,c1.lfctrl,Started,c1]
  }
}

assert GetStateReturnsStoppedAfterStop {
  all c,c1:LFKell {
    stop[c,c.lfctrl,Ok,c1] implies getState[c1,c1.lfctrl,Stopped,c1]
  }
}
```

Unfortunately, in this instance, the exact semantics of the Started and Stopped states, and hence of the start and stop operations, can only be given by reference to the behavior of the hosting kell. We specify below this semantics, exploiting the notion of transition. Essentially, the Stopped state is defined as one where no transition involving signals targetting non control gates is possible.

```
sig LFTKell in LFKell {}

fact LFTKellsAreTKells { all c: LFTKell | c in TKell }

fact LFTKellStoppedHasNoFunctionalTransitions {
   all c:LFTKell | all t:c.transitions | c.state = Stopped implies t.(sin+sout).target in c.ctrls
}
```

Finally, let's note that the following assertions are not valid – *i.e.*, starting or stopping a component do not imply that previous subkells and gates are preserved.

```
assert StartPreservePreviousSubKells {
  all c,c1:LFKell {
    start[c,c.lfctrl,Ok,c1] implies c1.sc = c.sc
  }
}

assert StartPreservePreviousGates {
  all c,c1:LFKell {
    start[c,c.lfctrl,Ok,c1] implies c1.gates = c.gates
  }
}

assert StopPreservePreviousSubKells {
  all c,c1:LFKell {
    stop[c,c.lfctrl,Ok,c1] implies c1.sc = c.sc
```

```
    }
}

assert StopPreservePreviousGates {
    all c,c1:LFKell {
        stop[c,c.lfctrl,Ok,c1] implies c1.gates = c.gates
    }
}
```

7 Instantiation

7.1 Factories

The Fractal model has two notions of *component factories*: a generic one, and a specific one. A *generic factory* is able to create arbitrary components from a type, a description of their content, and a description of their membrane (or set of controllers). A *factory* is only able to create components of a certain type. We specify these notions formally below. We follow here the informal Fractal specification [10], Section 5.

We first define the notion of factory. By definition, a factory is a component, hence a kell, that exhibits a Factory interface. A factory is specific to a component type, a controller description, and a content description. The notions of controller description and content description are just defined as simple signatures ControllerDesc and ContentDesc, with no further constraint[16].

```
module fractal/instantiation

open fractal/foundations as FF
open fractal/component as FC

sig ContentDesc extends Val {}
sig ControllerDesc extends Val {}

sig Factory extends Server {}

sig FKell in Kell {
    fg: Factory,
    insType: ComponentType,
    ctrlDesc: ControllerDesc,
    cntDesc: ContentDesc
}

fact FactoryInFKellIsExternalGate {
    all c:FKell | c.fg in c.gates
}
```

We define first an equivalence predicate between kells with Factory gates. Two such kells are equivalent if they have the same identifier, the same subkells, the same gates, the same Factory gate, the same instance type, the same controller and content descriptions.

[16] A future version of the Fractal model should improve on this, by defining in particular how these descriptions themselves can be manipulated, *e.g.*, for the purpose of specifying deployment processes, and what constraints apply to them.

```
pred isoFKell(c:FKell, c1:FKell) {
      isoKell[c,c1]
      c.fg = c1.fg
      c.insType = c1.insType
      c.ctrlDesc = c1.ctrlDesc
      c.cntDesc = c1.cntDesc
}
```

Operations associated with the Factory interface are specified below. The main operation is the newInstance operation, that creates a component, *i.e.*, a kell with a Component interface, of the type associated with the factory.

```
// Operations from the Factory interface

one sig InstantiationException extends Val {}

one sig FactoryReturn in Ok + InstantiationException {}

pred getInstanceType(c:FKell, f:Factory, t:ComponentType, c1:FKell) {
    f = c.fg
    t = c.insType
    isoFKell[c,c1]
}

pred getControllerDesc(c:FKell, f:Factory, d:ControllerDesc, c1:FKell) {
    f = c.fg
    d = c.ctrlDesc
    isoFKell[c,c1]
}

pred getContentDesc(c:FKell, f:Factory, d:ContentDesc, c1:FKell) {
    f = c.fg
    d = c.cntDesc
    isoFKell[c,c1]
}

pred newInstance(c:FKell, f:Factory, ci:CompKell, icc:Component, r:FactoryReturn,
                 c1:FKell) {
    f = c.fg
    r = InstantiationException implies isoFKell[c,c1]
    else r = Ok and icc = ci.comp and icc.ctype = c.insType and isoFKell[c,c1]
}
```

We give below a simple property, which asserts the consistency of the newInstance and getCType operations.

```
assert GetCTypeAfterNewInstance {
    all c:FKell, ci:CompKell, icc:Component {
        newInstance[c,c.fg,ci,icc,Ok,c] implies getCType[ci,icc,c.insType,ci]
    }
}
```

A generic factory is a component that provides the GenericFactory interface, with a single operation newInstance, that creates a new component, with a Component interface, given a component type, a content description, and a controller description.

```
sig GenericFactory extends Server {}
```

```
sig GFKell in Kell {
    gf: GenericFactory
}

fact GenericFactoryInGFKellIsExternalGate {
    all c:GFKell | c.gf in c.gates
}
```

We define an equivalence predicate between kells with GenericFactory gates. Two such kells are equivalent if they have the same identifier, the same subkells, the same gates, and the same GenericFactory gate.

```
pred isoGFKell(c:GFKell, c1:GFKell) {
     isoKell[c,c1]
     c.gf = c1.gf
}
```

```
// Operations from the GenericFactory interface
```

```
pred newInstanceGF(c:GFKell, f:GenericFactory, t:ComponentType, ctrld:ControllerDesc,
                   cntd:ContentDesc, ci:CompKell, icc:Component, r:FactoryReturn, c1:GFKell) {
    f = c.gf
    r = InstantiationException implies isoGFKell[c,c1]
    else r = Ok and icc = ci.comp and icc.ctype = t and isoGFKell[c,c1]
}
```

We give below a simple property, which asserts the consistency of the newInstanceGF and getCType operations.

```
assert GetCTypeAfterNewInstanceGF {
    all c:GFKell, ct:ComponentType, ctrld:ControllerDesc, cntd:ContentDesc,
            ci:CompKell, icc:Component {
        newInstanceGF[c,c.gf,ct,ctrld,cntd,ci,icc,Ok,c] implies getCType[ci,icc,ct,ci]
    }
}
```

7.2 Bootstrap

The informal Fractal specification specifies a bootstrap condition for any Fractal system: that there exists some generic factory accessible through some well-known name. We capture the first part of this bootstrap condition very simply: it merely mandates the existence of a generic factory. Operation getBootstrapComponent returns the GenericFactory interface associated with a generic factory component.

```
fact BootstrapCondition {
     some GFKell
}

pred getBootstrapComponent(f:GenericFactory) {
    some c:GFKell | f = c.gf
}
```

8 Typing

Fractal defines a simple type system for components and component interfaces, *i.e.*, ComponentType and InterfaceType respectively, and its associated type factory. We follow here the informal Fractal specification [10], Section 6.

8.1 Role, contingency and cardinality

We specify first the notion of *role*, *contingency*, and *cardinality* as abstract set signatures (*i.e.*, Role, Contingency, and Cardinality respectively) and define their possible values as singleton subsets – *i.e.*, server and client roles, mandatory and optional contingencies, singleton and collection cardinalities.

```
module fractal/type_system

open fractal/foundations as FF
open fractal/component as FC

abstract sig Role extends Val {}
one sig server, client extends Role {}
fact OnlyServerAndClientAreRoles { Role = server + client }

abstract sig Contingency extends Val {}
one sig mandatory, optional extends Contingency {}
fact OnlyMandatoryAndOptionalAreContingencies { Contingency = mandatory + optional }

abstract sig Cardinality extends Val {}
one sig singleton, collection extends Cardinality {}
fact OnlySingletonAndCollectionAreCardinalities { Cardinality = singleton + collection }
```

8.2 Component and interface types

We then define both ComponentType and InterfaceType sets. A component type is a set it of interface types. An interface type is made of a name, a signature, a role, a contingency, and a cardinality. Each interface type of a component type has a distinct name.

```
sig ComponentType extends FC/ComponentType {
  it: set InterfaceType
}

sig InterfaceType extends FC/InterfaceType {
  name: Id,
  signature: Type,
  role: Role,
  contingency: Contingency,
  cardinality: Cardinality
}

fact InterfaceTypesInComponentTypeHaveDistinctNames {
    all ct:ComponentType | all i,j:ct.it | i.name = j.name implies i = j
}
```

We specify below operations attached to the ComponentType interface. Operation getInterfaceTypes returns the set r of interface types of a component type. Operation

getInterfaceType returns the interface type r having the name n or the exception NoSuchInterfaceException.

```
// Operations from the ComponentType interface

pred getInterfaceTypes(ct:ComponentType, r: set InterfaceType, ct1: ComponentType) {
  r = ct.it
  ct1 = ct
}

pred getInterfaceType(ct:ComponentType, n:Id, r: InterfaceType + NoSuchInterfaceException,
                      ct1: ComponentType) {
   r in ct.it implies n = r.name else r = NoSuchInterfaceException
   ct1 = ct
}
```

We give below two simple properties, which assert the consistency of the ComponentType operations.

```
assert GetInterfaceTypesAndGetInterfaceType {
  all ct:ComponentType, its:set InterfaceType {
    getInterfaceTypes[ct,its,ct] implies all i:its | getInterfaceType[ct,i.name,i,ct]
  }
}

assert GetInterfaceTypeAndGetInterfaceTypes {
  all ct:ComponentType, i:InterfaceType, its: set InterfaceType |
    getInterfaceType[ct,i.name,i,ct] and getInterfaceTypes[ct,its,ct] implies i in its
}
```

We specify below operations attached to the InterfaceType interface. Operation getItfName returns the name of an interface type. Operation getItfSignature returns the signature of an interface type. Operation isClientItf returns true if the role of an interface type is equals to client and false if it is server. Operation isOptionalItf returns true if the contingency of an interface type is equals to optional and false if it is mandatory. Operation isCollectionItf returns true if the cardinality of an interface type is equals to collection and false if it is singleton.

```
// Operations from the InterfaceType interface

pred getItfName(it:InterfaceType, r:Id, it1:InterfaceType) {
  r = it.name
  it1 = it
}

pred getItfSignature(it:InterfaceType, r:Type, it1:InterfaceType) {
  r = it.signature
  it1 = it
}

pred isClientItf(it:InterfaceType, it1:InterfaceType) {
  it.role = client
  it1 = it
}

pred isOptionalItf(it:InterfaceType, it1:InterfaceType) {
  it.contingency = optional
```

```
  it1 = it
}

pred isCollectionItf(it:InterfaceType, it1:InterfaceType) {
  it.cardinality = collection
  it1 = it
}
```

8.3 Type factory

We specify below kells with TypeFactory gates – *i.e.*, elements of TFKell, or *tfkells*, are type factories.

```
sig TypeFactory extends Server {}

sig TFKell in Kell {
      tf: TypeFactory
}

fact TypeFactoryInTFKellIsExternalGate { all c:TFKell | c.tf in c.gates }
```

We now define an equivalence predicate between tfkells. Two such kells are equivalent if they have the same identifier, the same subkells, the same gates, and the same TypeFactory gate.

```
pred isoTFKell(c:TFKell, c1:TFKell) {
  isoKell[c,c1]
  c1.tf = c.tf
}
```

We specify below operations attached to the TypeFactory gates. Operation createItfType returns an interface type it with the name n, the signature s, the role r, the contingency co, and the cardinality ca. Operation createType returns a component type ct composed of the set its of interface types.

```
// Operations from the TypeFactory interface

pred createItfType(c:TFKell, tff:TypeFactory, n:Id, s:Type, r:Role, co:Contingency,
                   ca:Cardinality, it:InterfaceType, c1:TFKell) {
    c.tf = tff
    it.name = n
    it.signature = s
    it.role = r
    it.contingency = co
    it.cardinality = ca
    isoTFKell[c,c1]
}

pred createType(c:TFKell, tff:TypeFactory, its:set InterfaceType, ct:ComponentType, c1:TFKell) {
    c.tf = tff
    ct.it = its
    isoTFKell[c,c1]
}
```

8.4 Sub typing relations

We now specify the sub typing relations between types, interface types, and component types, based on substitutability. Predicate isSubTypeOf is true if a type t1 is a sub type of a type t2. This predicate is used to evaluate the sub typing relation between two signatures of interface types. Predicate isSubInterfaceTypeOf is true if an interface type it1 is a sub type of an interface type it2. Predicate isSubComponentTypeOf is true if a component type ct1 is a sub type of a component type ct2.

```
pred isSubTypeOf(t1, t2: Type) {
  t1 in t2.*sstypes
}

pred isSubInterfaceTypeOf(it1, it2: InterfaceType) {
    it1.name = it2.name
    it1.role = it2.role
    it2.role = server implies {
      isSubTypeOf[it1.signature, it2.signature]
      (it2.contingency = mandatory implies it1.contingency = mandatory)
    } else {
      isSubTypeOf[it2.signature, it1.signature]
      (it2.contingency = optional implies it1.contingency = optional)
    }
    (it2.cardinality = collection implies it1.cardinality = collection)
}

pred isSubComponentTypeOf(ct1, ct2: ComponentType) {
  all it1:ct1.it {
    it1.role = client implies one it2: ct2.it { isSubInterfaceTypeOf[it1, it2] }
  }
  all it2:ct2.it {
    it2.role = server implies one it1: ct1.it { isSubInterfaceTypeOf[it1, it2] }
  }
}
```

These previous three sub typing relations are reflexive and transitive.

```
assert IsSubTypeOfIsReflexive {
  all t:Type {
    isSubTypeOf[t, t]
  }
}

assert IsSubTypeOfIsTransitive {
  all t1, t2, t3:Type {
    isSubTypeOf[t1, t2] and isSubTypeOf[t2, t3] implies isSubTypeOf[t1, t3]
  }
}

assert IsSubInterfaceTypeOfIsReflexive {
  all it:InterfaceType {
    isSubInterfaceTypeOf[it, it]
  }
}

assert IsSubInterfaceTypeOfIsTransitive {
  all it1, it2, it3:InterfaceType {
```

```
    isSubInterfaceTypeOf[it1, it2] and isSubInterfaceTypeOf[it2, it3]
     implies isSubInterfaceTypeOf[it1, it3]
  }
}

assert IsSubComponentTypeOfIsReflexive {
  all ct:ComponentType {
    isSubComponentTypeOf[ct, ct]
  }
}

assert IsSubComponentTypeOfIsTransitive {
  all ct1, ct2, ct3:ComponentType {
    isSubComponentTypeOf[ct1, ct2] and isSubComponentTypeOf[ct2, ct3]
     implies isSubComponentTypeOf[ct1, ct3]
  }
}
```

9 Consistency

We have developed in the previous sections a formal specification of the Fratal component model. It mirrors the informal specification pretty truthfully (see Section 10 for a list of the differences between the two), and it has been written in a modular fashion, keeping the different controllers as independent from one another as possible, again following the philosophy of the informal specification. One may wonder however, whether the combination of all these different features in a single component may not give rise to inconsistencies. The following simple predicate, which asserts the existence of a component supporting all the different controllers defined in the present specification, can be shown to have an instance (and hence to be consistent). Specifically, the command listed below produces a kell which is an element of all the different sets listed below (ACKell, etc.). Note that a lesser scope (*e.g.*, a scope of 9 instead of 10 in the command) is not enough to obtain a model.

```
module fractal/consistency

open fractal/foundations as FF
open fractal/attribute as FAC
open fractal/binding as FBC
open fractal/component as FC
open fractal/content as FCC
open fractal/instantiation as FIC
open fractal/lifecycle as FLC
open fractal/name as FNC
open fractal/super as FSC
open fractal/type_system as FTS

pred ControllersAreConsistent {
  some ACKell & BCKell & CompKell & CCKell & FKell & GFKell
       & LFKell & NCKell & SCKell & TFKell
}

// Command
run ControllersAreConsistent for 10 but 30 Id, 20 Val
```

10 Discussion

We discuss in this section the main differences between the informal Fractal specification [10] and the present formal one. We also comment briefly on the use of Alloy and of the Alloy Analyzer.

10.1 Differences with the informal specification

Foundations The first difference with the informal specification is the elucidation of the core concepts found in the Foundations (Section 3). The informal specification only refers to them in passing, yet making them explicit is key to highlighting the generality of the Fractal model. In particular, they enable us to define precisely the general notion of behavior that can be expected from Fractal components. In formalizing these foundations, we have also clarified features of the Fractal model and on purpose relaxed certain constraints that appear in the informal specification. We list them below. Note that we flag explicitly each feature mentioned as an *extension* or as a *clarification*.

- [**extension**] We allow component graphs (corresponding to the subcomponent relation) to be arbitrary graphs. This means that we can have structures with sharing, but also non-well-founded structures, such as components which are in their own subcomponents. Non-well-founded structures are explicitly ruled out by the informal specification. We believe this to be an unnecessary restriction as non-well-founded structures can be useful – *e.g.*, in modelling reflective component structures.

- [**clarification**] We allow components to be passed by value in operations. This feature is not discussed in the informal specification, but the heavy Java bias of the IDL used in the informal specification does not explicitly cater for it.

- [**clarification**] We allow components to have a varying number of interfaces during their lifetime. This feature was not explicitly disallowed in the informal specification, but the Julia reference implementation does not support it.

General and specific differences We list below enhancements which the formal specification brings to the informal one, and a difference between the two specifications:

- [**clarification**] Pre and post-conditions of operations on the different interfaces defined in the specification are made explicit. For instance, the post-conditions on operations of the Component interface specify that the target component identity, gates (interfaces) and subcomponents must remain unchanged (through the use of the isoKell predicate). The informal specification is not explicit on these conditions.

- [**clarification**] The present specification clarifies the admissible effects of operations not only through operations pre- and post-conditions but also by highlighting *valid* and *invalid* assertions. Witness for instance the different commutation

properties and invalid assertions listed for side-effects of the ContentController and LifecycleController interfaces. The informal specification does not contain such a breadth of details on operations.

- [**clarification**] The subtyping relation used in basic introspection and typing is specified as being a partial order on types. The informal specification does not enforce this constraint. However, having a notion of subtyping that is not a partial order makes little sense.
- [**clarification**] The present specification enforces the unicity of the Component interface of a component, if it exists. We do not enforce the unicity of other kinds of controller interface. The unicity of the Component interface highlights the fact that it acts as an identity for a component. The informal specification is ambiguous on the unicity of controller interfaces.
- [**clarification**] The getInterfaces operation in basic introspection (Section 5) returns all the external interfaces of the target component. The informal specification is evasive on that subject. We believe this form is preferable for introspection purposes.
- [**difference**] The informal specification distinguishes between three kinds of bindings. We have left out this distinction in the formal specification, since it is essentially a classification with no operational import.

10.2 Modularity of the specification

The Fractal model and its informal specification promote a philosophy of separation of concerns, including between meta-level capabilities. The Alloy module system and Alloy's set-based inheritance features allowed us to follow this design philosophy, and to specify the Fractal model in a very modular way. Each of the module of our Alloy specification imports at least the foundations module (Section 3), since this module defines the core concepts common to all other modules, and at most the component module (Section 5), which provides basic introspection capabilities. The table below summarizes the dependencies between modules. This report can be seen as a vindication of the modularity claim of the Fractal specification.

However, we have highlighted that there exists one dependency that breaks this modularity between controllers: the Interface interface contains an isInternal operation related to internal interfaces. We therefore think that this operation should be removed from the next version of the Fractal specification to preserve orthogonality between the different component controllers.

Alloy module	module dependencies
attribute	foundations
binding	foundations
component	foundations
content	foundations + component
instantiation	foundations + component
lifecycle	foundations
name	foundations
naming	foundations
super	foundations + component
type_system	foundations + component

10.3 Using Alloy

Alloy fulfills its promise (discussed in [16]) of providing a fast *specification development / specification debugging* loop, thanks to the fully automatic character of the Alloy Analyzer. Coupled with a fast learning curve due to its first-order, set-based character, a well-thought out syntax, and extremely useful model visualization features in the Alloy Analyzer, developing specifications with Alloy is a very streamlined, and even enjoyable, process. The surprising amount of bugs (trivial or not), that were weeded out from the specification thanks to this rapid feedback loop, especially during the early stages of its development, is a testament to the efficacy and efficiency of the *lightweight specification* approach advocated by the Alloy designers.

Our use of Alloy is mostly classical. We have adopted an operational style for the specification of the different Fractal controllers, since this was both closer to the informal specification and the most natural style for the specification of mostly server-like behavior. We have made an abundant use of *invalid* assertions in order to signal possibly unexpected side effects or behavior of operations. Because of the purposedly weak semantics for operations adopted in the Fractal specification (which we have kept in the present work), we find these negative assertions to be invaluable to debunk misconceptions about the specification.

The one limitation of Alloy for our specification exercise is in the specification of mathematically oriented structures such as our notion of *transition* for the specification of component behavior. Unfortunately, we have had to eschew the use of transitions to characterize the behavior of components: this would have led us to systematically introduce *closure* conditions (see the discussion in Section 3); this would have, in turn, led to an explosion in the size of our models. For the most part, this does not constitute a hindrance, since we can capture the intended semantics of component operations using standard Alloy predicates. In one instance, however (the specification of the LifecycleController operations), the specification of the intended semantics could only be achieved by an explicit characterization of a component behavior in terms of transitions. For this kind of specification, dealing with conditions on mathematical structures, a higher-order theorem prover such as Coq [6] is probably much more suited.

11 Conclusion

We have presented in this report a comprehensive formal specification of the Fractal component model, that covers all the elements of the original informal reference specification of the Fractal model. The formal specification is written in Alloy, a simple and natural (for object-oriented modellers) formal specification language. We have used the Alloy Analyzer to check the consistency of the model, its key invariants, and several properties. Compared to the informal one, the formal specification given in this report has several advantages:

- It provides a formal description of the foundations of the Fractal component model, which are only alluded to in the informal specification.
- It is more liberal than the informal one in a number of aspects (*e.g.*, allowing components to be passed by value in operations).
- It provides a truly language-independent specification of the Fractal component model.
- It removes ambiguities from the informal specification, notably in the specification of the post-conditions of the controller operations.

We have concentrated in this report on providing a formal equivalent of the *full* informal Fractal specification, just leaving aside a formal specification of the notion of normal, export, import binding and of template (which will be covered by future work on a new Fractal ADL). We plan to extend this work in several directions, however:

- Using this specification as a basis for the development of a new ADL for Fractal, with a formal semantics.
- Extending this specification to cover concepts and constraints related to the handling of component packages and the specification of distributed deployment processes.
- Extending this specification to describe, formally and abstractly, meta-level and aspectual capabilities of current Fractal implementations, including the specification of component-based membranes [23].
- Using this specification for showing formally how several component models can be understood as Fractal personalities or Fractal specializations.

References

[1] Alloy Analyzer Web Site. Accessible at: http://alloy.mit.edu/.

[2] T. Abdellatif, J. Kornas, and J.B. Stefani. J2EE Packaging, Deployment and Reconfiguration Using a General Component Model. In *Component Deployment, 3rd International Working Conference, CD 2005*, volume 3798 of *LNCS*. Springer, 2005.

[3] L. Barbosa and J. Oliveira. State-based Components Made Generic. *Electronic Notes in Theoretical Computer Science, vol. 82, no.1*, 2003.

[4] L.S. Barbosa, S. Meng, B.K. Aichernig, and N. Rodrigues. *On the Semantic of Componentware: A Coalgebraic Perspective*, volume 2 of *Component-Based Software Development*, chapter 3. World Scientific, 2006.

[5] J. Barwise and L. Moss. *Vicious Circles*, volume 60 of *CSLI Lecture Notes*. CSLI Publications – Center for the Study of Language and Information, Stanford, California, 1996.

[6] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[7] S. Bouchenak, F. Boyer, S. Krakowiak, D. Hagimont, A. Mos, N. De Palma, V. Quéma, and J.B. Stefani. Architecture-Based Autonomous Repair Management: An Application to J2EE Clusters. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS 2005)*. IEEE Computer Society, 2005.

[8] S. Bouchenak, N. De Palma, D. Hagimont, and C. Taton. Autonomic Management of Clustered Applications. In *IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2006.

[9] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.B. Stefani. The Fractal Component Model and its Support in Java. *Software - Practice and Experience*, 36(11-12), 2006.

[10] E. Bruneton, T. Coupaye, and J.B. Stefani. *The Fractal Component Model*. ObjectWeb Consortium, 2.0.3 edition, 2004.

[11] D. Caromel, A. di Costanzo, and C. Delbé. Peer-to-Peer and Fault-Tolerance: Towards Deployment-Based Technical Services. *Future Generation Computer Systems*, 23(7), 2007.

[12] B. Claudel, N. De Palma, R. Lachaize, and D. Hagimont. Self-protection for Distributed Component-Based Applications. In *Stabilization, Safety, and Security of Distributed Systems, 8th International Symposium, SSS 2006*, number 4280 in LNCS. Springer, 2006.

[13] P.C. David, M. Léger, H. Grall, T. Ledoux, and T. Coupaye. A Multi-stage Approach for Reliable Dynamic Reconfigurations of Component-Based Systems. In *8th IFIP Int. Conf. Distributed Applications and Interoperable Systems, DAIS 2008*, volume 5053 of *LNCS*, 2008.

[14] A. Flissi and P. Merle. A Generic Deployment Framework for Grid Computing and Distributed Applications. In *OTM Confederated International Conferences, Grid computing, high performAnce and Distributed Applications (GADA 2006)*, volume 4276 of *LNCS*. Springer, 2006.

[15] D. Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2), 2002.

[16] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[17] Daniel Jackson and K. J. Sullivan. COM Revisited: Tool-Assisted Modelling of an Architectural Framework. In *ACM SIGSOFT Symp. on Foundations of Soft. Eng. (FSE)*. ACM, 2000.

[18] K.K. Lau and Z. Wang. Software Component Models. *IEEE Trans. Software Eng.*, 33(10), 2007.

[19] G. Leavens and M. Sitaraman, editors. *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

[20] Zhiming Liu and He Jifeng, editors. *Mathematical Frameworks for Component Software - Models for Analysis and Synthesis*. World Scientific, 2006.

[21] S. Meng, B. K. Aichernig, L.S. Barbosa, and Z. Naixiao. A Coalgebraic Semantic Framework for Component-based Development in UML. *Electr. Notes Theor. Comput. Sci.*, 122, 2005.

[22] S. Meng and L.S. Barbosa. Components as Coalgebras: The Refinement Dimension. *Theor. Comput. Sci.*, 351(2), 2006.

[23] L. Seinturier, N. Pessemier, L. Duchien, and T. Coupaye. A Component Model Engineered with Components and Aspects. In *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE'06)*, volume 4063 of *LNCS*. Springer, 2006.

[24] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[25] S. Sicard, F. Boyer, and N. De Palma. Using Components for Architecture-Based Management: The Self-Repair Case. In *30th International Conference on Software Engineering (ICSE 2008)*. ACM, 2008.

[26] M. Simonot and M.-V. Aponte. Modélisation formelle du contrôle en Fractal. ARA REVE Project Deliverable 2.3, CNAM-CEDRIC, December 2007.

[27] K. J. Sullivan, M. Marchukov, and J. Socha. Analysis of a Conflict between Aggregation and Interface Negotiation in Microsoft's Component Object Model. *IEEE Trans. Software Eng.*, 25(4), 1999.

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

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