Specifying Self-configurable Component-based Systems with FracToy

Alban Tiberghien, Philippe Merle, Lionel Seinturier

Specifying Self-configurable Component-based Systems with FracToy

Alban Tiberghien, Philippe Merle, and Lionel Seinturier

INRIA Lille - Nord Europe
University of Lille 1 - LIFL CNRS UMR 8022
Villeneuve d'Ascq, France

firstname.lastname@inria.fr

Abstract. One of the key research challenges in autonomic computing is to define rigorous mathematical models for specifying, analyzing, and verifying high-level self-* policies. This paper presents the FracToy formal methodology to specify self-configurable component-based systems, and particularly both their component-based architectural description and their self-configuration policies. This rigorous methodology is based on the first-order relational logic, and is implemented with the Alloy formal specification language. The paper presents the different steps of the FracToy methodology and illustrates them on a self-configurable component-based example.

Key words: Alloy, Autonomic Computing, Component-based Systems, Formal Specification, Methodology Self-Configuration

1 Introduction

Autonomic computing gathers *systems that can manage themselves given high-level objectives from administrators* [12]. The idea is to design software which can provide efficient and continuous services to users without any human intervention. Self-configurability is a key property of any autonomous system, and means the capability of such a system to configure itself according to high-level policies automatically. For instance, software components [16] and connectors can be added or removed to/from a running software system according to evolutions of runtime conditions. These dynamic modifications of running software architectures can be described by high-level self-configuration policies. Here, one of the key research challenges is to define rigorous mathematical models for specifying, analyzing, and verifying such autonomous systems. Such a model must allow to detect errors and inconsistencies of high-level policies early at design time instead of during execution of targeted autonomous systems.

To tackle this problem, this paper presents the FracToy formal methodology to specify, analyse, and verify self-configurable component-based systems. This rigorous methodology is based on the first-order relational logic, and is

implemented with the Alloy formal specification language [10]. This methodology is iterative and divided into two main steps. The first step consists in specifying the component model used to build applications. This step is itself divided into three sub-steps. This first sub-step consists in defining the formal syntax of the component model, both its core concepts and the relations between these concepts. Secondly, this component model is constrained in order to define its static semantics, i.e., the set of constraints that any application must satisfy. Thirdly, it is necessary to specify the dynamic semantics of the component model, i.e., the set of operations allowing to update the architecture of running applications. Here, this dynamic semantics must be defined in a way allowing self-configurability of applications. Then, the second step of the FracToy methodology is to specify self-configurable component-based applications. Their components are defined by extending the core concepts of the component model and their self-configuration policies are defined as first-order logic constraints. Furthermore, the FracToy methodology allows to highlight and verify properties like the consistency of both the static semantics and self-configuration policies, and the commutativity of dynamic operations.

This paper is organized as follows. Section 2 presents the FracToy methodology and its different steps. Section 3 illustrates the methodology on a self-configurable component-based "Room" example. Section 4 discusses related works. Finally, Section 5 concludes and draws perspectives of FracToy.

2 The FracToy framework

FracToy is a framework that introduces a methodology, based on the Alloy [9], for the formal description of self-configurable component-based systems.

2.1 Alloy in a nutshell

The Alloy formal specification language fits with the fist-order relational logic [10]. The manipulated concepts are *sets* (Alloy signatures) that can be brought together using *relations* (Alloy signature fields). Alloy models are described with these two concepts and are constrained using facts or predicates. A fact is an expression that the whole model must always satisfy. A predicate is a parametrizable constraint which is applied only when invoked. As facts, predicates can be applied on the whole systems but also just on a specific signature. Furthermore, the language provides a model analyser. The Alloy Analyser can be used as a model finder (invoked with the Alloy `run` command) that instantiate all the models that satisfy the Alloy specification. It can also be used as a counter-example finder (invoked with the Alloy `check` command) in order to counter-example models that don't satisfy assertion (defined with the Alloy `asset` keyword). The combined use of the model finder and of counter-example finder allows fast iterative debugging, during the design process.

2.2 The FracToy methodology

The FracToy methodology proposes a use of Alloy for specifying, verifying and analysing self-configurable component-based systems. This rigorous and iterative methodology is divided into the two following steps and illustrated by the Figure 1:

1. **Specification of the component model** composed of three sub-steps.
   (a) **The formal syntax**: This step consists in defining each core concept of the component model and the relations between these concepts. Each concept is an Alloy abstract signature. Alloy signature fields define how and what concepts can be bound to a given concept. At this step, the model is not constrained but basic restrictions are nervertheless specified using the `one`, `lone` and `set` keywords in order to define the cardinality of the relations.
   (b) **The formal static semantics**: The static semantics of the component model is the set of constraints restricting the model. These constraints can be facts establishing what is possible to model with the component model. They can also be predicates in order to define finer-grain constraints that just concern certain concepts.
   Consistency checking: Once the static semantics is specified, it is possible to run a consistency test in order to verify that the constrained component model is instantiable. If tests don't pass, a correction/refinement loop can be performed on this step or on the previous step.
   (c) **The formal dynamic semantics**: Operations that dynamically update the running system must be specified in a way to preserve the self-configurable nature of the architecture. It is important to clearly identify the different states that the systems can reach. By fixing the pre-conditions and post-conditions, these operations define what is preserved during the changes of state of the system.
   Properties checking: These checks ensure that the dynamic of the system is well specified. For example, they ensure that add/remove operations are commutative. Indeed, all operations of the system have its inverse operations and the couple of operations must be commutative in order to have the certainty that it is possible to roll back in a stable state after applying an operation on the system.
2. **Specification of the self-configurable system**: Each component of the self-configurable architecture is a signature extending a concept of the component model. In the context of component-based architecture specification, the declaration part of the signature is dedicated to the declaration of services, references and/or sub-components. The Alloy `one`, `lone` and `set` keywords are used to specify the cardinality of these relations. The constraint part of the signature is dedicated to the definition of the assembly. In this part, constraints are used to map the previously declared fields to the concepts of the component model. Additional constraints can be added in order to limit the use of components in the case of the component model is not enough restrictive.

Self-configuration policies definition: Self-configuration policies are directly defined in the constraint part of the component signature. Indeed, in our approach, self-configuration is managed by components themselves.
Consistency/Properties checking: Here, it is possible to check the consistency on the full specified architecture and to verify that the self-configuration policies are efficiently applied and conform to the requirement.

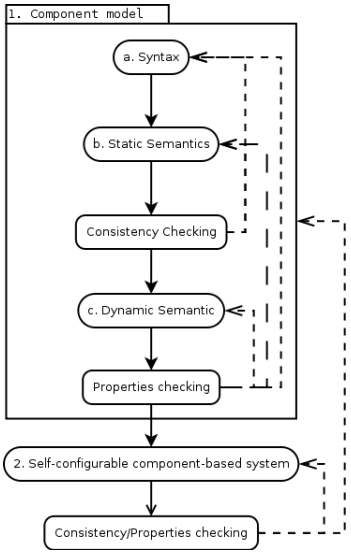

Fig. 1. The FracToy methodology

3 FracToy in action

Following the methodology presented in Section 2, this section provides the specification of a self-configurable component-based system, the "Room" use case, presented in Section 3.1. First, the component model is specified in Section 3.2 and, then, the "Room" self-configurable architecture is specified in Section 3.3. Verification and analysis are performed in Section 3.4.

3.1 The "Room" scenario

The scenario describes the case where a mobile user enters a room and wants to keep in touch with news and services provided by the room. The user's mobile device can receive news from the room and once s/he has obtained the expected information, s/he can visualise them on a screen or print them, according to the features available on her/his mobile device.

More precisely, there is a news provider that broadcasts news in the room. The room provides two kinds of output devices: screen and printer. The room is aware of the presence of all mobile devices. When a new mobile device (e.g. PDA, smartphones, etc.) enters the room, it is automatically connected to the news provider and to the screen and/or the printer devices according to the type of output devices it supports. For example, a PDA can print and display whereas a smartphone can only print because of power and energy restrictions. Finally, several mobile devices can be in the same room at the same time.

A component-based architecture description The *Room* scenario can be reified as a self-configurable component-based systems. The room and all devices are components. Each component has a variable number of input and output ports (communication points) respectively called services and references. The *NewsProvider* component has no service and its number of references (of type *News*) is not statically defined and can evolve according to the number of *MobileDevice* components contained in the *Room* component. Each *MobileDevice* component has one *News*-typed service and the number of references and their types (either *DisplayableNews* or *PrintableNews*) are specific to each type of *mobileDevice*. According to the informal definition, the *PDA* component has a reference of type *DisplayableNews* and a reference of type *PrintableNews* whereas the *Smartphone* component has only a reference of type *PrintableNews*. Self-configuration is performed when a *MobileDevice* component is added to the *Room* component. In this case, all bindings are automatically established between the *MobileDevice* components and other components.

3.2 Specification of the component model

Informal specification Our use case is not based on an existing component model, the presented component model remains consistent with the Szyperski component definition given in [16] in the way that "a component is a unit of composition with contractually specified interfaces and context dependencies only". The elementary entity of our model is *Component*. As this component model is hierarchical, a component can be either *Composite*, i.e. a component that can contain sub-components, or *Primitive*, i.e. a component implemented in a programming language. *Port* represents typed communication access points to a component. A port is either a *Service* (providing functionality) or a *Reference* (requiring functionality). Finally, it is possible to bind a reference to a service in order to explain communication channels between components. As our work takes place in a context of dynamic environments, this component model has to deal with this concern. That is why it is important to notice that when we use the term "component" or "port" it must be understood "a state of a component" or "a state of a port". Indeed an instance of a component models a certain state of the component. Each state is identified by an *Id*. Components have a *cid* and ports have a *pid*. If two different component instances have the same id, that means that we are semantically dealing with the same component but in different states.

Figure 2 represents the UML-like diagram of the key-concepts of the component model and their relations.

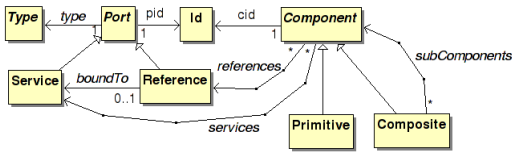

Fig. 2. UML-like diagram of the component model

The formal syntax We first declare the signature named *Component* (line 2). The fields *services* (line 4) and *references* (line 5) allow to respectively put in relation a component to its set of *Services* and *References*. Each component has a field *cid* (line 3) which represents the identity of the component (line 0). Two signatures specialize ("extends" in Alloy) the concept of component. The *Primitive* signature (line 7) just allows to directly manipulate this concept and to have a type for this kind of component. It is the same principle for the *Composite* signature (line 10) whereas it is possible, with this set, to associate (line 11) a component to other components (semantically its sub-components).

```
0  sig Id {}
1
2  abstract sig Component {
3    cid : one Id,
4    services : set Service,
5    references : set Reference
6  }
7  abstract sig Primitive extends Component{}
8
9
10 abstract sig Composite extends Component{
11   subComponents : set Component
12 }
```

In this component model, the *Port* signature (line 14) is a typed (line 16) communication access point of a component. In the same way as component, ports have an identity *pid* (line 15). *Service* (line 18) and *Reference* signatures (line 20) correspond to the functionality that a component provides and requires, respectively. The *boundTo* field (line 21) allows to bind a reference to a service. A reference can be bound to zero or one service and as a consequence a reference can exist even if it is not bound (specified with the Alloy `lone` keyword).

```
12 abstract sig Type {}
13
14 abstract sig Port {
15   pid : one Id,
16   type : one Type
17 }
18 sig Service extends Port {}
19
20 sig Reference extends Port {
21   boundTo : lone Service
22 }
```

The formal static semantics In addition of the formal syntax, the static semantics of the component model is defined as a set of constraints in order

to avoid certain use cases. The fact *AllPortsAffectedToOneComponent* (line 22) forces that all ports of a system are owned by one and only one component i.e. can be shared by two distinct component instances only if they have the same identity. The fact *AllBindingsInTheSameComposite* (line 29) ensures that all references of a sub-component are bound to a service of a sub-component of the same composite. The fact *NoBindingBetweenUncompatibleTypes* (line 36) just forbids that a binding is established if the types of the reference and the service are not the same. The fact *CompositeNotContainItself* (line 39 avoids that a composite contains itself in its sub-components. The *bind* predicate (line 42) declares a binding between a reference and a service. This statement chooses a reference in the set of references and binds it to the service (line 43).

```
22 fact AllPortsAffectedToOneComponent {
23   all p : Port {
24     all c, c' : Component {
25       (p in c.( services+references) and p in c'.( services+references)) implies c.cid = c'.cid
26     }
27   }
28 }
29 fact AllBindingsInTheSameComposite{
30   all c  : Composite {
31     all ref : c.subComponents.references {
32       ref.boundTo in c.subComponents.services
33     }
34   }
35 }
36 fact NoBindingBetweenUncompatibleTypes {
37   all r : Reference, s : Service | r.type != s.type implies r.boundTo != s
38 }
39 fact CompositeNotContainItself {
40   all c : Composite | c not in c.subComponents
41 }
42 pred Composite.bind[references : set Reference, service : one Service] {
43   one ref : references {
44     ref.boundTo = service
45   }
46 }
```

A test of consistency can be performed on the formal specification of this component model. This test consists in asking to the analyser to instanciate a model in an arbitrary (but coherent) scope. Here, *ComponentModelConsistency* test can be run, i.e, the analyser is able to instanciate a model that satisfy all the defined constraints. In other words, this core of concepts is consistent and can be a sure basis for more complicated architectures.

```
ComponentModelConsistency : run {} for 20
```

The formal dynamic semantics The last part of the specification of the component model is its dynamic semantics. In the context of our example, the dynamic semantics of the addition and the removal of a component in a composite has been formally specified. The two predicates *addComponent* (line 46) and *removeComponent* (line 53) are semantically commutative and are built following the same logic. In order to modelize the dynamicity of an addition (removal resp.), a predicate formalizes the change of state due to the operation execution. The two first parameters of these predicates, *c1* and *sc1*, symbolize the state of

the system before the operation execution, and the two last parameters *c2* and *sc2*, symbolize the state of the system after the operation execution. A semantics for these actions is to formalize that the resulting state of a component addition (removal resp.) is the start state plus (minus resp.) the component to add (remove resp.) and there is nothing more nothing less element in the architecture. This semantics is too strong in our case of self-configurable component-based system. Indeed, according to our *Room* example, when a *MobileDevice* component is added in the *Room* composite, the self-configuration policies are applied and as a consequence bindings are created between components and, thus, there is more that the new *MobileDevice* component in the *Room* composite. That is why it is important to notice that these operations don't ensure the strict equality of the system state (modulo the addition/removal of the component) but are based on the notion of state equivalence. Indeed both operations ensure the preservation of at least all that were present in the initial state of the system but it is not forbidden that the final state contains more elements.

Based on this logic, the *addComponent* predicate constrains the component *sc1* not to be in the sub-components of the component *c1* (line 47). The final composite *c2* is constrained to be equivalent to the initial composite *c1* (line 48) and the final component *sc2* to be equivalent to the initial added component *sc2* (line 49). Finally, the component *sc2* must be in the sub-components of the composite *c2* (line 50). It is exactly the opposite for the *removeComponent* predicate.

```
46  pred addComponent[c1 : Composite, sc1 : Component, c2 : Composite, sc2 : Component] {
47    sc1 not in c1.subComponents
48    compositeEquiv[c1, c2]
49    componentEquiv[sc1, sc2]
50    sc2 in c2.subComponents
51  }
52
53  pred removeComponent[c1 : Composite, sc1 : Component, c2 : Composite, sc2 : Component] {
54    sc1 in c1.subComponents
55    compositeEquiv[c2, c1]
56    componentEquiv[sc2, sc1]
57    sc2 not in c2.subComponents
58  }
```

The relation of equivalence used for the formalization of the addition and the removal of a component in a composite is specified through the three following predicates. Two components are equivalent (line 58) if they have the same identity (line 59) and if their services and references are equivalent (lines 60 and 61). Two port sets are equivalent (line 63) if all ports of the first set (line 64) have an equivalent port in the second set (line 65). Two ports are equivalent is they have the same identity (line 66) and the same type (line 67). Finally, two composites are equivalent (line 71) if they are equivalent components (line 72) and if all sub-components of the first composite have its equivalent in the second composite (lines 73-75). This formalization allows to support self-configuration policies as shown in Section 3.3.

```
58  pred componentEquiv(c1 : Component, c2 : Component) {
59    c1.cid = c2.cid
60    portEquiv[c1.references, c2.references]
61    portEquiv[c1.services, c2.services]
```

```
62  }
63  pred portEquiv(portSet1 : set Port, portSet2 : set Port) {
64    all p1 : portSet1 {
65      one p2 : portSet2 {
66        p1.pid = p2.pid
67        p1.type = p2.type
68      }
69    }
70  }
71  pred compositeEquiv(c1 : Composite , c2 : Composite){
72    componentEquiv[c1, c2]
73    all sc1 : c1.subComponents {
74      one sc2 : c2.subComponents {
75          componentEquiv[sc1,sc2]
76      }
77    }
78  }
```

An important property can be checked thanks to the Alloy analyser on the dynamic addition and removal of a component in a composite. Semantically the *addComponent* and *removeComponent* are two commutable operations. The *AddRemoveCommutable* assertion checks that adding a component in a composite then removing it keep the system in the same state. In other words, this assertion tests that the two predicates are commutable.

```
assert AddRemoveCommutable {
  all c1, c2  : Composite, sc1, sc2 : Component {
    addComponent[c1, sc1, c2, sc2] implies removeComponent[c2, sc2, c1, sc1]
  }
}
check AddRemoveCommutable for 10 expect 0
```

3.3 Specification of the self-configurable *Room* system

In a general way, the "Room" example is specified by extending the component model. Three singleton types,i.e. , *News*, *DisplayableNews*, and *PrintableNews*, are first defined. They respectively correspond to the type of each service and reference port (singletons are obtained thanks to the Alloy `one` keyword).

```
83  one sig News, DisplayableNews, PrintableNews extends Type {}
```

Specification of the primitive components *NewsProviders* is a primitive component (line 83). It provides no service (line 86) but requires a set of references named r (line 84). All these references are of type *News* (line 87) and it can not require other references than r (line 88).

```
83  sig NewsProvider extends Primitive {
84    r : set Reference
85  } {
86    no services
87    r.type = News
88    references  = r
89  }
```

Printer and *Screen* are two other primitive components (lines 89 and 97 resp.). Both require no reference (lines 92 and 100 resp.) but they provide a service named s (lines 90 and 98 resp.). This service is of type *PrintableNews* for

the *Printer* primitive (line 93) and of type *DisplayableNews* for the *Screen* one (line 93). They can not provide other services than *s* (lines 94 and 102 resp.).

MobileDevice is an abstract primitive component (line 96) for modeling any mobile device.

```
89  sig Printer extends Primitive {
90    s : one Service
91  } {
92    no references
93    s.type = PrintableNews
94    services = s
95  }
96  abstract sig MobileDevice extends Primitive {}
97  sig Screen extends Primitive {
98    s : one Service
99  } {
100   no references
101   s.type = DisplayableNews
102   services = s
103 }
```

Specification of the *Room* composite After having defined the different primitive components of the architecture, the *Room* composite can be specified (line 103). As this composite is autonomous, it doesn't declare neither services (line 109) nor references (line 110). It contains at least three primitives declare as a relation between the *Room* and the primitive sets (lines 104-106). Here the relation name represents the name of the sub-component. The Alloy `one` keyword means that there can be only one *NewsProvider*, one *Printer*, and one *Screen*. The *mobileDevices* field declares a pool of *MobileDevice*. Indeed, as the *Room* composite is open to different incoming/outcoming mobile devices, we have modelised this by the use of a set of *MobileDevice* (line 107). The constrain in line 111 specifies that these components are effectively declared as sub-component of the composite and that it can not have other kind of components in a *Room*.

In our methodology, the self-configuration policies are expressed as a contraint. These policies are declared in the signature of the composite that manages the self-configuration. Thus, the self-configuration policy of this use case specifies that, for all mobile devices contained in a room (line 114), all services of this mobile device (line 115) and of type *News* is bound from one reference of the *NewProvider* component (line 116). Regarding the mobile device references, there are two cases. If the reference is of type *DisplayableNews*, this reference is bound to the service provided by the *Screen* component (line 119). If the reference is of type *PrintableleNews*, this reference is bound to the service provided by the *Printer* component (line 120).

```
103 sig Room extends Composite {
104   newsProvider : one NewsProvider,
105   printer : one Printer,
106   screen : one Screen,
107   mobileDevices : set MobileDevice
108 } {
109   no services
110   no references
111   subComponents = newsProvider + printer + screen + mobileDevices
112
113   //SELF-CONFIGURATION POLICY
114   all md : mobileDevices {
115     all serv : md.@services {
116       serv.type = News implies bind[newsProvider.r, serv]
117     }
118     all ref : md.@references {
119       ref.type = DisplayableNews implies bind[ref, screen.s]
```

```
120         else ref.type = PrintableNews implies bind[ref, printer.s]
121       }
122     }
123   }
```

The specification of specific mobile devices *PDA* and *SmartPhone* are both *MobileDevice* components. Both provide only one service *s* of type *News* (lines 124, 128, 131 and lines 135, 138, 141). The difference is done by the reference that these mobile devices require. Both require one reference of type *PrintableNews* (lines 125, 129 and lines 136, 139) but, in addition, the PDA requires one reference of type *DisplayableNews* (lines 126, 130).

```
123 sig PDA extends MobileDevice {
124   s : one Service,
125   r1 : one Reference,
126   r2 : one Reference
127 } {
128   s.type = News
129   r1.type = PrintableNews
130   r2.type = DisplayableNews
131   services = s
132   references = r1 + r2
133 }
134 sig SmartPhone extends MobileDevice {
135   s : one Service,
136   r : one Reference
137 } {
138   s.type = News
139   r.type = PrintableNews
140   services = s
141   references = r
142 }
```

The whole self-configuration specification is completed and a more realistic test of consistency can be performed. The *SelfConfigurableArchitectureConsistency* tries to instantiate a model conform to the "Room" use case when a PDA is present in the room.

```
SelfConfigurableArchitectureConsistency: run {
  one myRoom : Room, pda : PDA | pda in myRoom.mobileDevices
}
for exactly 1 Composite, exactly 4 Primitive, exactly 6 Port, exactly 3 Type, exactly 11 Id
```

3.4 Analysis of the *Room* architecture

Static properties checking The *AllReferencesAreBound* assertion (line 1) specifies that a mobile device contained in a room (line 3) implies that all its references are bound to a service provided either by a printer or a screen (line 4). This assertion is verified on all the instantiable model in a large scope (line 7). The analyser doesn't find any counter-example and that is why it assures that when a mobile device is added to the room all the expected bindings are well established. This assertion shows that the self-configuration policy specification produces the expected result.

```
1 assert AllReferencesAreBound {
2   all room : Room, md : MobileDevice {
3     room.component[md]
4     implies all ref : md.references | ref.boundToin room.(printer+screen).services
5   }
6 }
7 check AllReferencesAreBound for 10 expect 0
```

Dynamic properties checking A more interesting use of the Alloy Analyser is to find non-explicit dynamic properties. The following assertion specifies that a *MobileDevice* primitive dynamically added in a *Room* composite implies that this primitive is also in the *mobileDevices* set of the *Room* composite. The analyser doesn't find any counter-example and it proves that an explicit constraint on the component model implies an implicit constraint on the self-configurable architecture. Indeed the *addComponent* predicate formalizes the adding of a component in a composite by preserving the state of the composite. The following satisfied assertion proves that if this predicate is applied on a *Room* composite and a *MobileDevice* primitive it implicitly implies that the *MobileDevice* primitive is also contained in the *mobileDevices* set of the *Room* composite. Even if this fact result from the conjunction of all constraints of the whole system, we want to highlight the fact that this constraint has never been expressed and that is a consequence of other constraints.

```
1  assert AddComponentImpliesMobileDeviceInRoom {
2    all room1, room2 : Room, md1, md2 : MobileDevice {
3      addComponent[room1, md1, room2, md2]
4      implies md2 in room2.mobileDevices
5    }
6  }
7  check AddComponentImpliesMobileDeviceInRoom for 10 expect 0
```

4 Related Work

In [3], Bradbury et al. highlight that formal methods are used to provide formal specification languages for designing dynamic software architectures. Works presented in [1], [5] and [6] are also based on logic-based formalisms but they aim at providing formal specification languages where our work provides rigorous and formal methodology to specify, verify and analyse self-configurable component-based systems on top of the use of a formal specification language.

In the domain of CBSE, Architectural Description Language (ADL) have been proposed in order to describe the configuration and the assembling of component-based systems [14]. Generally, the semantics of the underlying component model and of the description language are not clear and are hard-coded in their compiler/interpreter. Nevertheless, two works aim to describe dynamic architectures. The Plastik framework [8] provides a unique formalism (extending Acme/Armani ADL) to specify dynamic architecture (implemented with the OpenCOM component model [4]). Armani (now full part of Acme) allows to set invariants on architectures and some additional statements allows to imperatively describe the architectural reconfigurations Wright [2] is an ADL based on formal method, i.e., the Communicating Sequential Processes (CSP) process algebra and allows to formalize the dynamic behaviour of architectural connections. FracToy approach explicitly focuses on the description of component-based systems and allows to describe and reason on the architectural evolution of the system. The use of Alloy provides an unified, declarative, and constraint-based way of description.

Among Alloy community, Alloy has been already used in CBSE. In [7], Darwin ADL has been formalized with Alloy. This work presents a formalization of the Darwin component model and specifies an architecture built on to top of this model. In this work, constraints are only to express static invariants on the architecture. In [13], a way to formally express and verify properties of Acme architectural styles. Acme styles are mapped to Alloy in order to use the Alloy Analyser to check consistency and properties on these styles. In this work, the dynamic nature of software is not considered. Other works focus on the way to modelize existing component models using Alloy. It is the case for COM in [11] and Fractal in [15]. These works aim to formally specify component models that are originally specified in natural language. Thereby, they can highlight properties on the model that are ambiguous in the textual specification. The FracToy approach is not dedicated to a specific component model and allows, in addition, to specify, verify, and analyse both the component model and the self-configurable architecture built on top of these component model.

5 Conclusion and future work

In this paper, we have presented FracToy, a rigorous and formal methodology for specifying, verifying and analysing self-configurable component-based systems. This methodology is divided into two main steps: specify the component model and specify the self-configurable architecture.

The FracToy methodology was applied to design the *Room* self-configurable component-based system, both the underlying component model and the self-configurable component-based system. This example has shown how to efficiently use the Alloy analyser in order to exhibit static/dynamic and not necessary explicit properties on the architecture. The Alloy formal specification language proves that it fits to the specification of such systems. Indeed the underlying theory of Alloy, i.e., the set theory, is closed to the component-based programming and its analyser allows fast analysis, debugging, and visualizing. Moreover, this approach provides a unique paradigm for specifying, verifying and analysing systems. In addition, the first-order relational logic approach allows to design self-configurable systems in a declarative and constraint-based way without considering syntactic and technical concerns. Thus, specifications describe what the system should be, not how the system should do it. The system is described according to the different states that it can reach instead of describing the sequence of operations to execute to reach a certain state.

Nevertheless, the FracToy approach is limited by a built-in limitation of Alloy. Indeed, as other model finder, all Alloy model instantiations has to be performed in a defined scope. As a consequence, highlighted properties are fully true only in this scope. Moreover, by writing the *Room* use case in Alloy, we have identified some recurring syntactic patterns and that specification auto-generation can be expected. That is why, on the short term, we plan to add syntactic sugar on top of the FracToy description to fill this gap.

References

1. Nazareno Aguirre and Tom Maibaum. A Temporal Logic Approach to the Specification of Reconfigurable Component-Based Systems. In *ASE '02: Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, page 271, Washington, DC, USA, 2002. IEEE Computer Society.
2. Robert J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, May 1997.
3. Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. A Survey of Self-Management in Dynamic Software Architecture Specifications. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systems*, pages 28–33, New York, NY, USA, 2004. ACM.
4. Geoff Coulson, Gordon Blair, Paul Grace, Francois Taiani, Ackbar Joolia, Kevin Lee, Jo Ueyama, and Thirunavukkarasu Sivaharan. A Generic Component Model for Building Systems Software. *ACM Transactions on Computer Systems*, 26:1–42, 2008.
5. V. C. C. de Paula. *ZCL: A Formal Framework for Specifying Dynamic Software Architectures*. PhD thesis, Federal University of Pernambuco, 1999.
6. M. Endler and J. Wei. Programming generic dynamic reconfigurations for distributed applications. In *Proceedings of the International Workshop on Configurable Distributed Systems*, pages 68–79. IEE, 1992.
7. Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. Self-Organising Software Architectures for Distributed Systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 33–38, New York, NY, USA, 2002. ACM.
8. Antônio Tadeu A. Gomes, Thais V. Batista, Ackbar Joolia, and Geoff Coulson. Architecting Dynamic Reconfiguration in Dependable Systems. *Architecting Dependable Systems IV*, 4615/2007:237–261, 2007.
9. Daniel Jackson. Alloy: a Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.
10. Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, April 2006.
11. Daniel Jackson and Kevin Sullivan. COM Revisited: Tool-Assisted Modelling of an Architectural Framework. *SIGSOFT'00/FSE-8: Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 149–158, 2000.
12. Jeffrey O Kephart and David M Chess. The Vision of Autonomic Computing. *Computer*, 36:41–50, 2003.
13. Jung Soo Kim and David Garlan. Analyzing Architectural Styles with Alloy. In *ROSATEA '06: Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis*, pages 70–80, New York, NY, USA, 2006. ACM.
14. Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26:70–93, 1997.
15. Philippe Merle and Jean-Bernard Stefani. A formal specification of the Fractal component model in Alloy. Technical Report RR-6721, INRIA, November 2008.
16. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Number 0-201-74572-0. 2002.