A Declarative Formal Approach to Dynamic Reconfiguration

Marianne Simonot
Laboratoire CEDRIC
292 rue St. Martin
F-75141 Paris Cédex 03
marianne.simonot@cnam.fr

Virginia Aponte
Laboratoire CEDRIC
292 rue St. Martin
F-75141 Paris Cédex 03
maria-virginia.aponte_garcia@cnam.fr

ABSTRACT

Self-adapting software adapts its behavior in an autonomic way, by dynamically adding, suppressing and recomposing components, and by the use of computational reflection. One way to enforce software robustness while adding adaptative behavior is disposing of a formal support allowing these programs to be modeled, and their properties specified and verified. We propose FracL, a formal framework for specifying and reasoning about dynamic reconfiguration programs written in a Fractal-like programming style. FracL is founded on first-order logic, and allows the specification and proof of properties concerning both functional concerns and control concerns. Its encoding using the Focal proof framework, enable us to prove FracL soundness and to obtain a mechanized framework for reasoning on concrete architectures.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods*; F.3.1 [**Logics and meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Pre- and post-conditions*

General Terms

Verification, Design

Keywords

components, dynamic reconfiguration, self-management, formal methods

1. INTRODUCTION

Modern software systems often have to evolve in order to accommodate to changing environment and user requirements. Self-managing component systems [27] modify their architecture and behavior *during execution* to cope with these changing conditions: components have to be created, added, attached or detached at run-time. This behavior is commonly known as run-time evolution or dynamic reconfiguration [10]. The need for dynamic reconfiguration appears naturally and frequently: for instance, while performing automated deployment, when resource limitations (memory, battery) restrict the number or the quality of services of components that can be deployed simultaneously, or when adding behaviors as new components in response to new run-time conditions.

Component based design, *separation of concerns* and *computational reflection* are key technologies [27] enabling the realization of self-configuring systems. *Component based design* [36] supports modularity and reuse during software design; *separation of concerns* [30] simplifies development by clearly separating *functional behavior* from *control* concerns in applications (e.g., quality of service, fault tolerance, security). Finally, self-managing applications use *computational reflection* to observe their own structure and behavior (components and sub components, internal bindings, life cycle status, etc), as well as *reconfiguration* operations [10] in order to support changes. Fractal [11] is a popular language-independent component model that follows these key principles. It was designed for building dynamically reconfigurable distributed systems and has been broadly used [16, 25, 9] for building operating and middleware systems (application servers, grids, middleware communication frameworks, etc).

The Fractal component model [12] is defined as an extensible system of relations between component related concepts together with a corresponding API, that a Fractal component *may or may not implement*, depending on what the component intends to offer to other components. The Fractal API methods are organized in increasing *levels of control*, i.e., in increasing order of *reflective capabilities*. At the lowest level, a component does not provide any control capability and is therefore like an object. At the higher level, a component provides capabilities to introspect and modify its *content* (sub-components, bindings, state, etc). Thus, in Fractal applications, adaptative behavior can be programmed by calls to these API *control primitives*, relying on an informal specification [12], and on several implementations (Java, C, C++) [11, 16]. Nevertheless, no guarantee is given with respect to the safety of such adaptation programs: does the structure and non functional behavior of the application remain correct after adaptation?

Clearly, when adding adaptative behavior, formal techniques can be useful to ensure that the system will continue to execute in an acceptable or safe manner after adaptation. To this end, many formal approaches support dy-

IWOCE'09, August 24, 2009, Amsterdam, The Netherlands.

namic reconfiguration through the definition of reconfiguration operations and their formal semantics, as in [26, 32, 4]. However, the formal semantics underlying most of those approaches generally allow only for operational (as opposed to declarative) descriptions, and when equipped with verification tools, they focus mainly on model-checking techniques. The classical approach consists in mapping component and reconfiguration notations into a well-defined execution model, where these notions can be given a precise meaning. This can be interesting, for instance, for studying the operational basis of reconfigurable programming: whether its multiple concerns can be addressed with a small set of powerful constructions.

In this work we adopt a different approach. Our main goal is to build a formal framework allowing us (i) to describe what a valid reconfigurable component system is, (ii) to describe how control primitives behave by dynamically changing the structure and behavior of systems, (iii) and to add new adaptation programs to such systems, while ensuring their safety *prior to deployment time* and *by means of theorem-proving oriented and model-finder tools*. That is, we might want to use model finders such as ALLOY [19] to find errors and refine the user's initial component configuration; we might then pursue with more powerful tools, such as theorem provers, to prove the safety of user's adaptation programs.

As Fractal is popular and widely used by real applications, we take its API set of control primitives (at the highest level of control), and their informal description [12] as our starting point. As we want to focus on the safety of programmed adaptation *verified via theorem proving and model finders* we favour a declarative specification style, well adapted to these tools. Finally, as we wish to benefit from different kinds of verification tools, we remain dialect-tool independent and choose first-order logic as our specification language. Our proposed model, FracL, is a declarative formalism based on first-order logic, allowing the formal description of reconfigurable component systems, and the formal specification of adaptation scripts on them. FracL provides a first-order logic modeling of Fractal entities, with pre-conditions and post-conditions on Fractal operations, and with invariants to be respected by reconfigurable component systems. Our formalism provides direct support for reasoning, due to its logical nature, and is expressive enough to allow for the description of components in a property-oriented way. We used this core with a simple procedural script language to implement adaptation scripts that can be formally specified and verified. As a first experiment, we encoded FracL and proved its soundness [5, 33] using the formal proof framework FOCAL [15] (relying on the automated theorem prover Zenon [8], and Coq proof-assistant [6]), and we are currently encoding it in ALLOY [19].

The rest of this article is organized as follow. Section 2, gives a brief introduction to Fractal, while section 3 outlines the FracL approach. Section 4 introduces FracL syntactic entities, and section 5 provides an axiomatic semantic to FracL control primitives. Section 6 illustrates FracL by a toy example specifying the upgrade of an already installed plugin, and 7 presents an example of adaptation script. Related work is presented in section 8 and section 9 ends with concluding remarks.

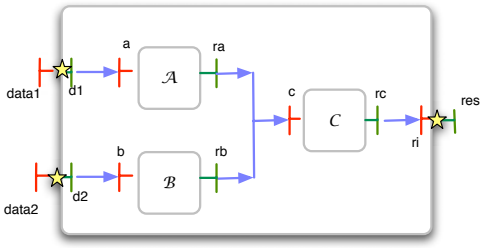

Figure 1: A Fractal composite component

2. BACKGROUND: FRACTAL

The main goals of the Fractal [11] component model are to implement, deploy and manage (i.e. monitor and dynamically reconfigure) complex software systems. These goals motivate the core features of the Fractal model: *base* and *composite* components (to have a uniform view of applications at various abstraction levels), shared components (to model resources), introspection capabilities (to monitor a running system), and configuration and reconfiguration capabilities (to deploy and dynamically reconfigure an application).

In Fractal, components are encapsulated, executable entities that interact with other components through well-defined interfaces. Interfaces, or ports, are access points to components, and bindings are established between ports to support component interactions. Ports can be *client* when they correspond to required interfaces, or *server*, when they correspond to provided interfaces.

Composite components are component assemblies, and components can appear in several composites. Ports are either *external* or *internal*. External ports are accessible from outside the component, while internal ports are accessible only from the component's sub components. Composites constitute a kind of *locality*: binding and naming are constrained by membership to a particular composite, e.g., *primitive* bindings cannot cross composite boundaries, thus ensuring encapsulation. Therefore, crossing of component boundaries must be controlled at the external/internal interface junctions. Figure 1 shows a typical Fractal composite component: arrows going from client ports to server ports model bindings and stars stand for junctions between internal and external ports.

Fractal supports reflection: components can expose, and provide control over their internal structure, their activity status, some configuration attributes, etc. Operations to introspect and modify Fractal entities and configurations are given by a set of methods known as *control primitives*, and given by an API informally described in [12].

Among the most interesting characteristics of Fractal are hierarchical-compositional capabilities, together with a clear separation between functional and non-functional aspects They both help the implementation and maintenance of complex software systems.

3. OUTLINE OF FRACL

Our formal approach involves the specification of the architectural structure of a system, including *invariants* stat-

ing what a valid configuration is, as well as the expression of dynamic reconfiguration actions as methods on these configurations.

To this end, our formal model FracL is given by: (1) a *syntactic domain*, expressed in terms of a first-order language, modeling the underlying FracL entities (*Comp*, *Port*, etc.), as well as their primitive non-functional capabilities (*owner*, *status*); (2) a *theory* on this domain, given by a set of constraints on this first-order language, as they follow from [12]. This theory constitutes the *invariant* of any valid FracL component configuration; (3) a notion of *configuration state* for a given FracL system, given by a particular model of this theory; (4) a *dynamic behavioral semantic* for FracL control operations in the sense of *behavioral contracts* [7]. They are described as functions mapping one configuration state into a new configuration state and are given by pre and post-conditions on the current system configuration.

Since pre and post-conditions on component configurations can be seen as characterizing sets of configuration states [17], these behavioral specifications can be viewed as summarizing the input/output behavior of each control primitive in FracL. As we proved (mechanically [5]) the soundness of these specifications with respect to the FracL invariant, we obtain a core logical framework where component configurations can be described and their correctness with respect to the invariant can be proved. As an example, we enrich this core FracL framework with a small procedural script language where adaptation programs can be encoded and proved later.

4. FRACL ENTITIES

FracL formally accounts for non-functional aspects of Fractal, following [12], at its highest level of control capabilities. In this section we describe entities and the notions needed for modeling. In its current version, our model does not include multiple bindings for a single port (*collections*), attributes, mandatory and optional ports nor composite bindings. Due to lack of space we do not include component types, but only port types.

A component system is made from *components* which can be either *primitives* or *composites*, from *ports* which can be *external* or *internal*, and from *signatures*. Signatures correspond to port types and are therefore language-dependent. They are modeled by any set equipped with a reflexive and transitive relation[1] $\sqsubseteq \in Sig \longleftrightarrow Sig$. Ports can be *client* (required interfaces), or *server*, (provided interfaces). Figures 2 and 3 show FracL entities and their structure. Each entity is defined by a particular set together with primitive relations and functions, and is constrained by some properties. The constraints on these entities are what we call the *invariant* of FracL.

4.1 Ports

Ports in FracL correspond to named component interfaces, i.e. access points to component services. They have a unique *name* (1), a unique *signature* (2), and a unique component *owner* (3). Their *role* (4) states whether they are *server* (provided) or *client* (required), and their *visibility* (5) is *external* within primitive components and can be either *external* or *internal* within composite components. External ports are accessible from outside a component while internal

[1]This relation can correspond to syntactic subtyping as in [31], refinement [1], or behavioral subtyping [17].

Sig, P_{name}	signatures, port names
C_{name}	component names
$Role = \{client, server\}$	port categories
$Comp, Primitive, Composite$	components
$Visibility = \{int, ext\}$	port visibility
$Status = \{stop, start\}$	component status
$PortC, PortS$	client, server ports
$PortI, PortE$	in/out ports

Figure 2: FracL entities

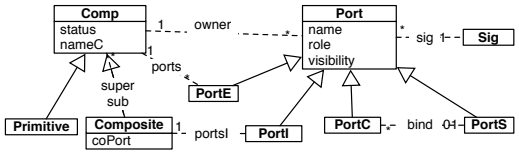

Figure 3: Structure of FracL entities

ports are only defined within composite components. Internal ports are used to connect the sub-components with the outside of the composite enclosing them. In particular, to each internal port in a composite c corresponds a port external to c. Each client port records its current bound port (if any): *bind* (6) models port introspection on bindings. We name $PortC$, $PortS$, $PortI$ and $PortE$ respectively the sets of client ports, server ports, internal ports, and external ports.

$$name \in Port \rightarrow P_{name} \quad (1)$$

$$sig \in Port \rightarrow Sig \quad (2)$$

$$owner \in Port \rightarrow Comp \quad (3)$$

$$role \in Port \rightarrow Role \quad (4)$$

$$visibility \in Port \rightarrow Visibility \quad (5)$$

$$bind \in PortC \nrightarrow PortS \quad (6)$$

In Fractal, *primitive bindings* can be established between one client and one server port, only *within the same address space*. Roughly speaking, this space is given by the set of composite components embracing the sub components possessing these ports (as external ports). Thus, a primitive binding preserves the encapsulation given by a composite: the binding cannot cross the composite boundaries except by explicit import or export connections made at external/internal port junctions (see section 2).

Definition 1 (Address space). *The function $AdrSpa \in Port \rightarrow \mathcal{P}(Comp)$ states the address space of ports. This space is given by its component owner if the port is internal and by the set of composite components embracing the components to which the port belongs if the port is external. Thus, the address space of an external port of a component outside any composite is empty. Two ports p_1, p_2 refer to a common composite component, noted $inCompo(p_1, p_2)$, if the intersection of their address spaces is non empty. We note $commonAdr(p_1, p_2)$ when two ports p_1, p_2 belong to a common address space, i.e., if both are external ports outside any composite or if they belong to a common composite.*

$$AdrSpa(p) \equiv \begin{cases} \{owner(p)\} & \text{if } p \in PortI, \\ super(owner(p)) & \text{if } p \in PortE \end{cases}$$

$$outCompo(p) \equiv p \in PortE \wedge AdrSpa(p) = \emptyset$$

$$inCompo(p_1, p_2) \equiv (AdrSpa(p_1) \cap AdrSpa(p_2) \neq \emptyset)$$

$$commonAdr(p_1, p_2) \equiv inCompo(p_1, p_2) \vee (outCompo(p_1) \wedge outCompo(p_2))$$

The following constraints belong to the FracL invariant. Signatures of connected ports must be compatible (7). Two bound ports must belong to a common address space (8). Two internal ports cannot be bound to each other (9). Within a component, two ports must have different names (10). To keep the specification short, this last constraint is stronger than the one in Fractal, where only internal (resp. external) ports from a single component cannot have the same name.

$$\forall p_c, p_s \in bind(sig(p_c) \sqsubseteq sig(p_s)) \quad (7)$$
$$\forall p_1, p_2 \in bind(commonAdr(p_1, p_2)) \quad (8)$$
$$\forall p_1, p_2 \in bind(p_1 \in PortE \vee p_2 \in PortE) \quad (9)$$
$$\forall p_1, p_2 \in Port(p_1 \neq p_2 \wedge owner(p_1) = owner(p_2) \Rightarrow name(p_1) \neq name(p_2)) \quad (10)$$

4.2 Components

A component c belongs either to *Primitive* or to *Composite* (11) and is characterized by a unique name $nameC(c)$, (12), its activity *status*, (13), and by the set of its external ports $ports(c)$, (14). If the component is composite, it has an associated set of sub-components $sub(c)$, (16), and a set of internal ports $portI(c)$, (17). A component can appear in several composite components. If c belongs to a composite, this composite is one of its *super* components (15). In a composite, each external port has an internal counterpart port via *coPort* (18).

$$Primitive, Composite \subseteq Comp \quad (11)$$
$$nameC \in Comp \rightarrow C_{name} \quad (12)$$
$$status \in Comp \rightarrow Status \quad (13)$$
$$ports \in Comp \rightarrow \mathcal{P}(PortE) \quad (14)$$
$$super \in Comp \rightarrow \mathcal{P}(Composite) \quad (15)$$
$$sub \in Comp \rightarrow \mathcal{P}(Comp) \quad (16)$$
$$portI \in Comp \rightarrow \mathcal{P}(PortI) \quad (17)$$
$$coPort \in Comp \rightarrow (PortI \rightarrow PortE) \quad (18)$$

Definition 2 (Client, server). *We define the functions mapping a component into respectively the set of its client ports, the set of its server ports, as well as the function giving the port corresponding to a name in a component.*

$$clientPorts(c) \equiv ports(c) \cap PortC$$
$$serverPorts(c) \equiv ports(c) \cap PortS$$
$$portOfN \equiv \{((n, c), p) \mid c \in Comp \wedge n \in P_{name} \wedge p \in ports(c) \cup portI(c) \wedge n = name(p)\}$$

A composite component corresponds to a *name space*, meaning that names of components belonging to it must be different. Thus, all components inside a common composite must have different names. This constraint holds also for components that do not belong to any composite: they must have different names as well. The following definition states whether two components reside in a common name space, namely, in a space where their names must be different.

Definition 3 (Name space). *We note $sameBox(c, d)$, two components c, d that belong to a common composite. We note $NSpace(c, d)$, when c, d resides in a common name space, i.e. either they reside outside any composite, or they reside in a common composite and $noBox(c)$, when c does not belong to any composite.*

$$noBox(c) \equiv super(c) = \emptyset$$
$$sameBox(c, d) \equiv (super(c) \cap super(d) \neq \emptyset) \vee (c = d)$$
$$NSpace(c, d) \equiv (noBox(c) \wedge noBox(d)) \vee sameBox(c, d)$$

Components are subject to the following constraints. The ports of a component have this component as their owner (19). Sub-components of a component have this component as their super-component (20). In a started component, bindings for its client ports[2] must be effective: emission and reception of method calls must be normally executed (21). Two components belonging to a common name space cannot have the same names (22). *super* is acyclic (23)[3].

$$\forall c \in Comp. \forall p (p \in ports(c) \Leftrightarrow owner(p) = c) \quad (19)$$
$$\forall c, d \in Comp. (d \in super(c) \Leftrightarrow c \in sub(d)) \quad (20)$$
$$\forall c \in Comp. (status(c) = start \Rightarrow (ports(c) \cup portI(c)) \cap PortC \subseteq dom(bind)) \quad (21)$$
$$\forall c_1 c_2 \in Comp. (NSpace(c_1, c_2) \wedge c_1 \neq c_2) \Rightarrow nameC(c_1) \neq nameC(c_2) \quad (22)$$
$$\forall c \in Comp. (c, c) \notin \sim super \quad (23)$$

Primitive and composite components are subject to the following constraints. *Primitive* and *Composite* partition *Comp* (24). A primitive component does not have sub-components nor internal ports (25). $coPort(c)$ is a bijection from internal ports of the composite c to its external ports (26). Internal and their corresponding external ports have inverted roles (27) and the same signature (28).

$$Composite \cap Primitive = \emptyset \wedge Composite \cup Primitive = Comp \quad (24)$$
$$\forall c \in Primitive. sub(c) = \emptyset \wedge portI(c) = \emptyset \wedge coPort(c) = \emptyset \quad (25)$$
$$\forall c \in Composite. coPort(c) \in portI(c) \longmapsto ports(c) \quad (26)$$
$$\forall c, p_i, p_e \in coPort. (role(p_i) = client \Rightarrow role(p_e) = server) \wedge (role(p_i) = server \Rightarrow role(p_e) = client) \quad (27)$$
$$\forall c, p_i, p_e \in coPort. (sig(p_i) = sig(p_e)) \quad (28)$$

Definition 4 (FracL invariant). *Constraints [1] to [28] constitute the FracL invariant.*

5. FRACL CONTROL PRIMITIVES

The set of control primitives specified here is strongly inspired from Fractal's API. We describe them as methods mapping a *FracL configuration state* into a new configuration state, by giving pre and post-conditions on this state. Formally, a *FracL configuration state* is given by any model of the FracL theory. Therefore, control primitives implicitly range on structures given by tuples:

$$< Comp, Composite, Primitive, Port, Sig, P_{name}, C_{name}, Status, Role, Visibility, name, nameC, role, sig, owner, sub, super, bind, ports, portI, coPort, status, visibility >$$

such that they verify the FracL invariant.

The FracL control primitives specified here provide an axiomatic semantic for a core language intended to program introspection and dynamic reconfiguration of hierarchical component systems. We name primitives as they appear in Fractal's API. We distinguish between two kinds of control primitives: *introspection operations* given by *accessors*

[2]This constraint is weaker than in Fractal: it only affects non optional ports. In this article, we do not distinguish between optional and mandatory ports.

[3]We note transitive closure by $\sim$, and bijection by $\longmapsto$.

```
P(PortI) getFcInternalInterfaces(Comp c)
  post: return portI(c)

P(Port) getFcInterfaces(Comp c)
  post: return ports(c)

P(P_name) listFc(Comp c)
  post: return name[(ports(c) ∪ portI(c)) ∩ PortC]

Status getFcState(Comp c);
  post: return status(c)

P(Comp) getFcSuperComponents(Comp c)
  post: return super(c)

P(Comp) getFcSubComponents(Comp b)
  post: return sub(b)
```

Figure 4: Accessors on components

```
P_name getFcItfName(Port p)
  post: return name(p)

Sig getFcItfType(Port p)
  post: return sig(p)

Role getRole(Port p)
  post: return role(p)

Comp getFcItfOwner(Port p)
  post: return owner(p)

boolean isFcInternalItf(Port p)
  post: return  p ∈ PortI

boolean isBound(Port p)
  post: return  p ∈ dom(bind)
```

Figure 5: Accessors on ports

```
PortE getFcInterface(P_name n, Comp c)
  pre:  ∃p. portOfN(n,c) = p ∧ p ∈ PortE
  post: return portOfN(n,c)

PortI getFcInternalInterface(P_name n, Comp c)
  pre:  ∃p. portOfN(n,c) = p ∧ p ∈ PortI
  post: return portOfN(n,c)

PortS lookupFc(Comp c, P_name n)
  pre:  ∃p. portOfN(n,c) = p ∧ p ∈ PortC
        ∧ p ∈ dom(bind)
  post: return bind(portOfN(n,c))
```

Figure 6: Access by name

to the current configuration state and *intercession, or reconfiguration operations* given by *modifiers* of this state. To keep the specification short, we concentrate on reconfiguration primitives and do not give in this paper accessors to signatures, to component types nor naming control and instantiation capabilities.

5.1 Introspection operations

Most of these accessors consist in total functions defined by the application of a single primitive function attached to a given FracL entity, with no pre-condition. In the following, when there is no particular pre-condition, we omit the assertion pre: `true` within the body of specifications. Figures 4, 5 and 6 respectively specify accessors on components, on ports and accessors by a particular entity name.

In figure 4, for a given component c, the first two primitives respectively retrieve the external and internal ports of c; `listFc` its set of client ports, and `getFcState` its activity state. The last two primitives, respectively retrieve the sets of super- and sub-components of a composite c. Remember that a component can appear in several composites: it can have several super-components.

In figure 5, for a given port p, the first three primitives respectively give access to its name, type and role. The primitive `getFcItfOwner` retrieves its owner component while the last two boolean primitives respectively test if p is internal, and if p is a client-bound port.

In figure 6, for a port name n and a component c, primitive `getFcInterface`(n, c) retrieves the external port named n in c, failing when it does not exist. `getFcInternalInterface` is similar for internal ports. `lookupFc`(c, n) retrieves the external and server port bound (if any) to a client port named n in c. It fails when there is no client port n in c, or when it is not bound or bound to an internal port.

5.2 Reconfiguration primitives

These primitives modify the current configuration state. They correspond to partial methods that either access ports by their name in a component, or that change bindings on ports, or the state of a component, or the content of a composite component.

Figure 7 presents the primitives modifying bindings on ports. `bindFc`(c, n, p_s) bounds the client port named n from component c to the server port p_s. The operation fails either if c does not have a client port of this name, if this port is already bound, if ports to be bound do not belong to a common address space, if they have incompatible signatures, or if c is not stopped. `unbindFc`(c, n) unbinds the client port named n in component c.

Figure 8 presents activity state modifications. `startFc` starts a component only if all its client ports are bound. Figure 9 details primitives modifying the content of a composite b by adding/suppressing a sub-component c to/from b. Addition and suppression of a component requires it to be stopped. The property $bound(c)$ below states whether a component c has at least one bound port.

$$bound(c) \equiv (clientPorts(c) \cap dom(bind)) \cup (serverPorts(c) \cap codom(bind)) \neq \emptyset$$

5.3 FracL soundness

FracL soundness is established by showing that the above specifications preserve the FracL invariant. Proof of this was realised with the Focal framework (relying on Zenon and Coq proof assistants) and can be found in [5]. The Focal modeling is described in [2].

6. A PLUGIN UPGRADING EXAMPLE

In this section we use FracL to specify a toy example inspired from a concrete dynamic component system. We specify a *Mozilla*-like application, able to dynamically upgrade already installed plugins. In this application, compo-

```
void bindFc (Comp c, P_name n, PortS p_s)
   pre: ∃p_c. portOfN(n,c) = p_c
          ∧ p_c ∈ PortC
          ∧ p_c ∉ dom(bind)
          ∧ sig(p_c) ⊑ sig(p_s)
          ∧ status(c) = stop
          ∧ commonAdr(p_c, p_s)
   post: bind := bind ∪ {(portOfN(n,c), p_s)}

void unbindFc (Comp c, P_name n)
   pre: ∃p_c, p_s. portOfN(n,c) = p_c
          ∧ bind(p_c) = p_s
          ∧ status(c) = stop
   post: bind:= bind − {(portOfN(n,c), bind(portOfN(n,c))}
```

Figure 7: Binding modifiers

```
void startFc(Comp c)
     pre:  clientPorts(c) ⊆ dom(bind)
     post: status(c):=start

void stopFc(Comp c)
     post: status(c):=stop
```

Figure 8: State modifiers

nents come equipped with *install manifests*, supplying information about its compatibility with other components, its version number, etc. Information relevant for updating is (a) the component id, (b) the component version number and (c) the set of *target applications* of the component. One *target application specification* for a given component c describes each component a being one of its possible targets and being compatible with c. This description is given by the target id (a's id), as well as its minimal and maximal acceptable version numbers. Install manifests in components are implemented as *functional methods* giving access to this information. Thus, adaptation here requires calling to both control methods *and* functional methods. This illustrates a key point of our work: the possibility to reason about programs combining both kinds of concerns.

We consider a toy system composed of four components: `M`, the main application (e.g., *Firefox*); `E`, an already installed plugin; `VM`, a version manager component and `Main` the composite encapsulating the previous three components. The application's architecture is shown in figure 10.

Install manifests are modeled by the signature `InstallMf` (figure 11). Components `M` and `E` each have a port h of signature `H`, respectively client and server. They also each have a server port im of signature `InstallMf`. `M` has an

```
void addFcSubComponents(Comp c, Comp b)
     pre:  c ∉ sub(b) ∧ ¬ bound(c) ∧ status(c) = stop
           ∧ b ∈ Composite
     post: sub(b) := sub(b) ∪ {c}

void removeFcSubComponents(Comp c, Comp b)
     pre:  c ∈ sub(b) ∧ ¬ bound(c) ∧ status(c) = stop
           ∧ b ∈ Composite(b)
     post: sub(b) := sub(b) − {c}
```

Figure 9: Composite content modifiers

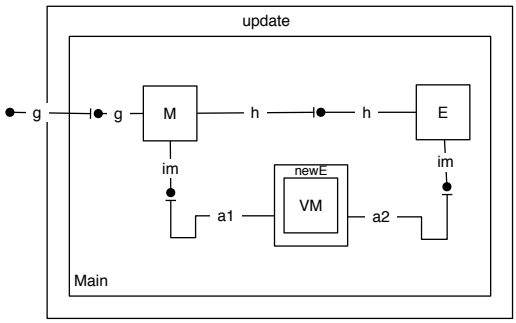

Figure 10: A plugin upgrade application

```
interface InstallMf{
   String getId();
   String getVersion();
   [] TargetsApp getTargetsApp();
}

interface TargetApp{
   String getId();
   String getMinVersion();
   String getMaxVersion();
}
```

Figure 11: Interfaces for updating

additional server port g of signature `G`, where g and h model business interfaces. `VM` is equipped with two client ports a_1 and a_2 of signature `InstallMf`. This component supplies `newE`, a control method modeling access to the update server. This method returns a component having the same type as the one bound to its port a_2, and being compatible (in the sense given by their corresponding install manifests) with the component bound to its other port a_1.

The `Main` composite exports business methods from `M` and supplies `update`, a control method implementing the upgrade. This method looks for a component with same id as `E`, having a more recent version and being compatible with `M`. In case of success, it replaces `E` with the component.

6.1 Modeling entities for upgrading

We first extend FracL with subsets of *Port*, *Comp*, *Sig*, etc, as illustrated in figure 12. The necessary signatures are: $G, H, InstallMf, TargetApp \in Sig$. Each port type is defined by a particular subset of *Port*. We proceed in a similar way for specifying components in the application. These specifications are given below.

Port specifications:

$$\begin{aligned}
\text{PcH} &= \{p \in PortC \mid sig(p) = H\} \\
\text{PsH} &= \{p \in PortS \mid sig(p) = H\} \\
\text{PsG} &= \{p \in PortS \mid sig(p) = G\} \\
\text{PcIm} &= \{p \in PortC \mid sig(p) = InstallMf\} \\
\text{PsIm} &= \{p \in PortS \mid sig(p) = InstallMf\}
\end{aligned}$$

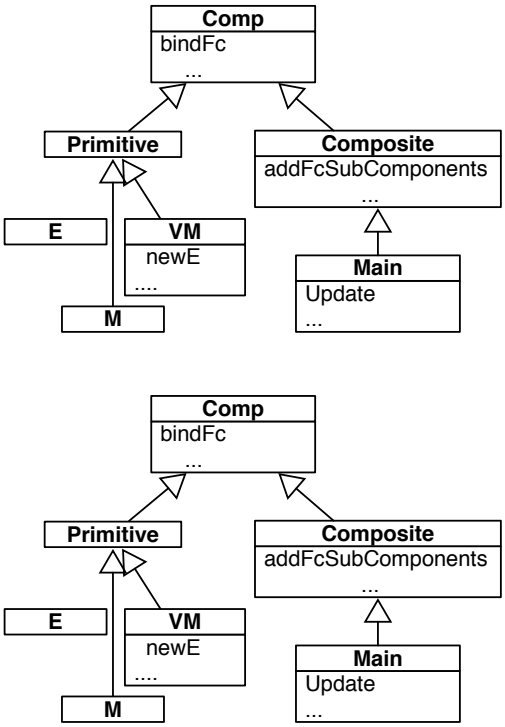

Figure 12: FracL components and port extension

Component specifications:

$$
\begin{aligned}
& h, a_1, a_2, g, i_m \in P_{name} \\
E =& \{c \in \mathit{Comp} \mid \exists p_1, p_2 (\mathit{ports}(c) = \{p_1, p_2\} \wedge p_1 \in \mathrm{PsH} \wedge \\
& p_2 \in \mathrm{PsIm} \wedge \mathit{name}(p_1) = h \wedge \mathit{name}(p_2) = i_m)\} \\
M =& \{c \in \mathit{Comp} \mid \exists p_1, p_2, p_3 (\mathit{ports}(c) = \{p_1, p_2, p_3\} \wedge \\
& p1 \in \mathrm{PsIm} \wedge p_2 \in \mathrm{PsG} \wedge p_3 \in \mathrm{PcH} \wedge \\
& \mathit{name}(p_1) = i_m \wedge \mathit{name}(p_3) = h \wedge \mathit{name}(p_2) = g)\} \\
\mathit{VM} =& \{c \in \mathit{Comp} \mid \exists p_1, p_2 (\mathit{ports}(c) = \{p_1, p_2\} \wedge \\
& p_1 \in \mathrm{PcIm} \wedge p_2 \in \mathrm{PcIm} \wedge \\
& \mathit{name}(p_2) = a_2 \wedge \mathit{name}(p_1) = a_1)\} \\
\mathrm{Main} =& \{b \in \mathit{Comp} \mid \exists m, e, v_m . (\mathit{sub}(b) = \{m, e, v_m\} \wedge \\
& m \in M \wedge e \in E \wedge v_m \in \mathit{VM} \wedge \\
& \exists p_1 \in \mathit{portI}(b) (\mathit{name}(p_1) = g \wedge p_1 \in \mathrm{PsH}))\}
\end{aligned}
$$

6.2 Methods specification

The $\mathit{newer}(e_1, e)$ property below states whether two ports e_1, e of signature `InstallMF` and common `Id`, are such that $e1$'s version is newer than or equal to e's version. The property $\mathit{compatible}(e, a)$ establishes that a is a correct target application for e.

$$
\begin{aligned}
\mathit{newer}(e_1, e) \equiv\ & \mathrm{getId}(e_1) = \mathrm{getId}(e) \wedge \\
& \mathrm{getVersion}(e_1) \geq \mathrm{getVersion}(e) \\
\mathit{compatible}(e, a) \equiv\ & \exists t_a \in \mathrm{getTargetsApp}(e). \\
& (\mathrm{getId}(t_a) = \mathrm{getId}(a) \wedge \\
& \mathrm{getMinVersion}(t_a) \leq \mathrm{getVersion}(a) \wedge \\
& \mathrm{getVersion}(a) \leq \mathrm{getMaxVersion}(t_a))
\end{aligned}
$$

```
E newE(VM v)
  pre:
      portOfN(a2, v) ∈ dom(bind) ∧
      portOfN(a1, v) ∈ dom(bind) ∧
      compatible(bind(portOfN(a2, v)),
                 bind(portOfN(a1, v)))
  post: returns e1 ∈ E such that
          noBox(e1) ∧ ¬bound(e1)
          ∃ pe . pe = bind(portOfN(a2, v)) ∧
          ∃ pm . pm = bind(portOfN(a1, v)) ∧
          newer(portOfN(im, e1), pe) ∧
          compatible(portOfN(im, e1), pm))
```

Figure 13: The `newE` method

```
void update (Main main)
  pre:  ∃m, e, vm .
          m ∈ M ∧ m ∈ sub(main) ∧
          e ∈ E ∧ e ∈ sub(main) ∧
          vm ∈ VM ∧ vm ∈ sub(main) ∧
          bind(portOfN(h, m)) = portOfN(h, e) ∧
          bind(portOfN(a1, vm)) = portOfN(im, m) ∧
          bind(portOfN(a2, vm)) = portOfN(im, e) ∧
          bind(portOfN(g, main)) = portOfN(g, m)

  post: ∃e1 ∈ E, e ∈ E, m ∈ M, vm ∈ VM.
          sub(main):=sub(main) − {e} ∪ {e1} ∧
          sub(main)={e1, m, vm} ∧
          bind(portOfN(h, m)) = portOfN(h, e1) ∧
          bind(portOfN(a1, vm)) = portOfN(im, m) ∧
          bind(portOfN(a2, vm)) = portOfN(im, e1) ∧
          bind(portOfN(g, main)) = portOfN(g, m) ∧
          newer(e1, e) ∧ compatible(e1, em)
```

Figure 14: The `update` method

In figures 13 and 14 we present specifications for the control methods `newE` from component `E` and `update` from component `Main`. Let v be a component of type `VM` having all ports bound, m be the component bound to v's client port a_1 and e be the component bound to the client port a_2 of v. The call `newE`(v) returns a component having the same type as e, being newer than e and compatible with m. In figure 14 the pre-condition and post-condition of the `update` method states the expected internal configuration of components from `Main` before and after upgrading, as well as compatibility conditions to satisfy.

Using FracL, we have specified an application able to dynamically upgrade one of its extensions. To this end, we stated the application's expected behavior concerning its dynamic reconfiguration capabilities. This specification is purely declarative: it does not indicate how the application realizes the intended functionality, but what the functionality is in terms of FracL core entities and primitives. In other words, properties that the `update` operation must fulfill constitute its specification.

To encode this operation, all we have to do is write methods making the appropriate calls to FracL control primitives. But first, we need (i) to enrich FracL with a script language allowing real programs to be built; (ii) to program and prove some general utilities for reconfiguration such as programs to safely substitute components, new kinds of introspection etc. Examples of such utility programs and script language are given in the following section.

```
P(Comp) siblings(Comp c)
    pre:  true
    post: return {c' | super(c') ∩ super(c) ≠ ∅}

P(PortC) bindingsTo(Comp c);
    pre:  ¬(noBox(c))
    post: return bind^-1[{serverPorts(c)}] ∩ PortE
```

Figure 15: `siblings` and `bindingsTo`(c) specifications

7. BUILDING NEW CONTROL METHODS

To specify and reason about new introspection and reconfiguration methods, we use a simple procedural script language consisting of assignment, sequence, choice, loops and having *certified method calls* as base instructions. A certified method is either one of the core FracL control primitives (defined in section 5), or a method for which we provide (i) a pre/post specification given in FracL (ii) a soundness proof stating that this specification preserves the FracL invariant, and (iii) a correctness proof stating that the code implementing this method realizes its specification. The proof can be developed using Hoare's logic. In this way, we obtain new control programs whose behavior is formally specified and proved and that can be used in turn to build more sophisticated programs.

7.1 Some useful introspection methods

We give here two examples of simple introspection methods. Fractal introspection allows direct access to server ports bound to client ports from a given component. We show how to specify and program the dual operation, `bindingsTo`(c), retrieving the set of client ports[4] bound to external server ports of component c. To program this operation, we need to access all composites containing c. As FracL invariant states that bindings are performed only on ports residing in a common address space, all we have to do is collect all client ports sharing an address space with every server port of c, and which are bound to it. To this end, we define a method `siblings`(c) returning the set of components belonging to a common composite with c, and the method `bindingsTo`(c) returning the set of client ports bound to external server ports from c. The corresponding specifications and programs are given in figures 15 and 16.

We do not detail here the correctness proofs for these programs. They are performed using Hoare's logic, and need to reason on loops and to exhibit a loop invariant. It is worth noticing that the pre-condition to `bindingsTo` is necessary to these proofs: FracL primitives do not allow the access to server ports bound to a client port for components outside any composite.

An incremental approach such as the one outlined here, enables the building of libraries composed of useful and certified control methods. Classical examples are methods allowing component replacement according to particular policies and able to manage sequences of control operations such as component stopping, binding and starting. This kind of certified library would ease the work of writing and proving concrete adaptation configurations.

[4]Which can appear in several composites.

```
P(Comp) siblings(Comp c)
    comps = ∅;
    bbs = getFcSuperComponents(c);
    for (b :bbs){
       comps = comps ∪ getFcSubComponents(b);
    }
    return comps;
}

P(PortC) bindingsTo(Comp c)
    pps = ∅;
    ccs =siblings(c);
    for(d : ccs){
       for(p : getFcInterfaces(d)){
          if (getRole(p) == client and isBound(p)) {
            s=lookupFc(d,getFcItfName(p));
            if (getFcItfOwner(s) == c){
                  pps = pps ∪ {p};
            }
          }
       }
    } return pps;
}
```

Figure 16: `siblings` and `bindingsTo` programs

8. RELATED WORK

Formal techniques have been extensively used to analyze component architectures and adaptation. Many works rely on operational descriptions of component behavior and reconfiguration operations and focus mainly on model checking verification tools.

More rare are works relying on theorem proving techniques and logic based analysis. Moreover, a majority of them only address static component configurations [29, 3, 22, 23]. These works concentrate on proving either business code correctness (with respect to their specification), or component composition correctness using refinement. None of them consider dynamic and self-adapting aspects of component systems.

The specification style we adopt, and the idea of using either model-finders or theorem-provers as verification tools, are already present in [35], and [20]. Both works specify the COM model, the first in Z, the second in Alloy. However, their purpose is to analyze general properties of the COM model, not to reason on concrete architectures nor adaptation.

The mapping of architectural styles expressed formally in the Acme ADL into Alloy models presented in [21] is closely related to our approach. Our core concepts and the related invariant are close except that [21] has a primitive notion of connector while, in concordance with Fractal, we don't. Their notion of concrete style and our notion of concrete architecture are very similar as they are all defined as subsets of the core component model. The main difference is that we add dynamicity to our model by formally specifying Fractal's API in order to manage adaptation, while they do not, but rather concentrate on analyzing general properties of style such as consistency, equivalence or overlapping between styles.

Among works addressing adaptive aspects of components, those based on logical specifications written in Alloy [19], such as [18, 13], are close to ours. We share with them similar core concepts, but they omit important configura-

tion and control aspects such as signature compatibility and component states, and as other works cited before, they do not consider the analysis of programs for adaptation.

This paper extends former work [2, 34] with notions of composite components, common address space, and with corresponding binding extensions. A more recent work [28] is the most closely related to ours. This work develops a formal specification of the Fractal component model and its API in ALLOY, by modeling the complete hierarchy of different configurations and optional levels present in this model starting from a very general core. The main difference with our work stems from our divergent goals: while they aim at using hierarchies of control levels to analyze and classify the numerous Fractal extensions and to compare Fractal with other component models with similar capabilities, our goal is to analyze and reason on concrete component systems and on adaptation programs. Indeed, this particular goal motivated our modeling of higher Fractal control capabilities only. On the other hand, they completely model the Fractal API, while our modeling is partial (but enough to reason on concrete applications). A final difference comes from their modeling of component behavior via labelled transition systems. Although this particularity increases their framework expressivity, they do not actually exploit it to model Fractal features. The reason is that it has a serious drawback: it can lead to state explosion during Alloy verifications.

9. CONCLUSIONS AND FUTURE WORK

We have presented FracL, a first-order logic and theorem-proving oriented framework for the specification and reasoning about component-based systems and adaptation programs written in a Fractal-like style. We have described what a FracL valid reconfigurable component system is by giving its invariant, and given behavioral semantics to Fractal control primitives in terms of pre- and post-conditions on configuration states. In previous works [33, 5] we proved their soundness. We have used these proved correct control primitives as central pieces to specify, and to incrementally program correct adaptation. Because of the behavioral style we choose, these programs can easily combine functional and non-functional concerns and their correctness can be proved via theorem provers using Hoare's logic.

We are considering several directions for future work. The first concerns extension of the model itself, in particular to capture naming, instantiation and component typing. Second, we plan to add an architecture navigation language such as Fpath [14] to our script language to facilitate the programming of architecture introspection. Third, a library of certified tools can be written following the principles outlined in section 7. Besides this, we believe that FracL can be used as a theory within different kinds of verification tools. We are currently working on its encoding into the model finder ALLOY. We also believe that FracL is suitable as a reasoning framework to provide a declarative semantic to Fractal's ADL [24]. We plan to map specifications written with this ADL into FracL's logic to allow the of use this theory to reason about properties of those specifications, either via ALLOY or via a theorem proving tools.

10. ACKNOWLEDGMENTS

This work was partially financed by the project ANR SSIA 2005 REVE.

11. REFERENCES

[1] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[2] V. Aponte, V. Benayoun, and M. Simonot. Modélisation de la reconfiguration dynamique en focal. In M. Filali, P. Michel, and C. Seguin, editors, *Approches Formelles dans l'Assistance au Développements de Logiciels (AFADL'2009)*, pages 105–119, ENSEEIHT, Toulouse (France), 26-28 janvier 2009. IRIT Press.

[3] R.-J. Back. Incremental software construction with refinement diagrams. In *AMAST*, page 1, 2006.

[4] T. Barros, R. Boulifa, A. Cansado, L. Henrio, and E. Madelaine. Behavioural models for distributed Fractal components. *Annals of Telecommunications*, accepted for publication, 2008. also Research Report INRIA RR-6491.

[5] V. Benayoun. Composants fractal avec reconfiguration dynamique : formalisation avec l'atelier focal. Technical report, ANR REVE - CEDRIC, http://reve.futurs.inria.fr/, 2008.

[6] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. 2004.

[7] A. Beugnard, J.-M. Jézequel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7):38–45, 1999.

[8] R. Bonichon, D. Delahaye, and D. Doligez. Zenon : An extensible automated theorem prover producing checkable proofs. In N. Dershowitz and A. Voronkov, editors, *LPAR*, volume 4790 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2007.

[9] S. Bouchenak, N. De Palma, D. Hagimont, and C. Taton. Autonomic management of clustered applications. In *IEEE International Conference on Cluster Computing*, Barcelona, Spain, Sept. 2006.

[10] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. 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.

[11] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java. *Softw., Pract. Exper.*, 36(11-12):1257–1284, 2006.

[12] E. Bruneton, T. Coupaye, and J. Stefani. The Fractal component model. Technical report specification, version 2.03, The ObjectWeb Consortium, 2004.

[13] R. Chatley, S. Eisenbach, and J. Magee. Modelling a framework for plugins. In *Specification and verification of component-based systems, September 2003*, 2003.

[14] P. David and T. Ledoux. Safe dynamic reconfigurations of fractal architectures with fscript. In *Proceeding of Fractal CBSE Workshop, ECOOP'06*, Nantes, France, 2006.

[15] C. Dubois, T. Hardin, and V. Donzeau-Gouge. Building certified components within focal. In H.-W. Loidl, editor, *Trends in Functional Programming*, volume 5 of *Trends in Functional Programming*, pages 33–48. Intellect, 2004.

[16] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. THINK: A Software Framework for Component-based Operating System Kernels. In *USENIX Annual Technical Conference*, 2002.

[17] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Muller, and M. Parkinson. Behavioral interface specification languages. A survey paper CS-TR-09-01, University of Central Florida, School of EECS, Draft, 2009.

[18] J. Hillman and I. Warren. An open framework for dynamic reconfiguration. *ICSE*, 0:594–603, 2004.

[19] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.

[20] D. 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.

[21] J. S. Kim and D. 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.

[22] O. Kouchnarenko and A. Lanoix. How to refine and to exploit a refinement of component-based systems. In I. Virbitskaite and A. Voronkov, editors, *PSI 2006, Pesrpectives of System Informatics, 6th Int. Andrei Ershov Memorial Conf.*, volume 4378 of *LNCS*, pages 297–309, Novosibirsk, Akademgorodok, Russia, June 2006. Springer.

[23] A. Lanoix, D. Hatebur, M. Heisel, and J. Souquières. Enhancing dependability of component-based systems. In *Reliable Software Technologies – Ada Europe 2007*, pages 41–54. Springer-Verlag, 2007.

[24] M. Leclercq, A. E. Ozcan, V. Quema, and J.-B. Stefani. Supporting heterogeneous architecture descriptions in an extensible toolset. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 209–219, Washington, DC, USA, 2007. IEEE Computer Society.

[25] M. Leclercq, V. Quéma, and J.-B. Stefani. DREAM: a Component Framework for the Construction of Resource-Aware, Reconfigurable MOMs. In *Proceedings of the 3rd Workshop on Reflective and Adaptive Middleware (RM'2004)*, Toronto, Canada, October 2004.

[26] Z. Liu and H. Jifeng, editors. *Mathematical Frameworks for Component Software - Models for Analysis and Synthesis*. World Scientific, 2006.

[27] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing adaptive software. *Computer*, 37(7):56–64, 2004.

[28] P. Merle and J.-B. Stefani. A Formal Specification of the Fractal Component Model in Alloy. Technical Report RR-6721, INRIA, Nov. 2008. hal.inria.fr/inria-00338987.

[29] B. Meyer. The grand challenge of trusted components. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 660–667, Washington, DC, USA, 2003. IEEE Computer Society.

[30] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[31] B. C. Pierce, editor. *Types and Programming Languages*. MIT Press, 2002.

[32] A. Schmitt and J.-B. Stefani. The kell calculus: A family of higher-order distributed process calculi. In C. Priami and P. Quaglia, editors, *Global Computing*, volume 3267 of *Lecture Notes in Computer Science*, pages 146–178. Springer, 2004.

[33] M. Simonot and M. Aponte. Modélisation formelle du contrôle en fractal. Deliverable D23, also RR 1563 - http://reve.futurs.inria.fr/, CNAM-CEDRIC, 2007.

[34] M. Simonot and M. Aponte. Une approche formelle de la reconfiguration dynamique. *L'Objet*, 14/4(4):73–102, 2008.

[35] 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.

[36] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, December 1997.