

LABORATOIRE



INFORMATIQUE, SIGNAUX ET SYSTÈMES
DE SOPHIA ANTIPOLIS
UMR 6070

CONTRACTING HIERARCHICAL COMPONENTS

Philippe Collet, Roger Rousseau

Projet OCL

Rapport de recherche
ISRN I3S/RR-2004-09-FR

Mars 2004

RÉSUMÉ :

Disposer de composants hiérarchiques pour construire des composants à partir d'un assemblage de composants est une nécessité du génie logiciel orienté composants. Assurer certaines propriétés d'un assemblage à partir des propriétés des sous-composants est alors un des principaux besoins. Actuellement, les approches contractuelles ne sont pas adaptées aux composants hiérarchiques et les notions de contrat et de spécification sont confondues, à tort. Le contrat est alors plutôt implicite, ce qui gêne son utilisation et sa gestion. Nous proposons donc Confract, un système de contrats pour composants hiérarchiques qui fournit à la fois des contrats d'interface sur les connections et des contrats de composition sur les assemblages. Le modèle distingue clairement les spécifications des contrats. Il a été validé par l'intégration d'un langage d'assertions dédié.

MOTS CLÉS :

Composants hiérarchiques, spécifications comportementales, Contrat, contrat

ABSTRACT:

Component-Based Software Engineering is now calling for hierarchical components to build components from components assembly. Ensuring properties of a resulting assembled component from properties of its subcomponents is then a major need. Currently contract-based approaches does not handle hierarchical components and mix up the notion of contract with the specification of a given property. The contract is then somewhat implicit, hampering its use and management. We propose confract, a contracting system for hierarchical components that supports both interface contracts on interface binding and composition contracts on assembly. The model also clearly distinguishes specifications from contracts and has been validated by the integration of a dedicated assertion language.

KEY WORDS :

Hierarchical Components, Behavioral specifications, Contract, Interface

Contracting Hierarchical Components*

Philippe Collet
OCL project / I3S laboratory
UMR 6070 CNRS/UNSA – BP 121
F-06390 Sophia Antipolis, France
Philippe.Collet@unice.fr

Roger Rousseau
OCL project / I3S laboratory
UMR 6070 CNRS/UNSA – BP 121
F-06390 Sophia Antipolis, France
Roger.Rousseau@unice.fr

ABSTRACT

Component-Based Software Engineering is now calling for hierarchical components to build components from components assembly. Ensuring properties of a resulting assembled component from properties of its subcomponents is then a major need.

Currently contract-based approaches does not handle hierarchical components and mix up the notion of contract with the specification of a given property. The contract is then somewhat implicit, hampering its use and management.

We propose *Confract*, a contracting system for hierarchical components that supports both interface contracts on interface binding and composition contracts on assembly.

The model also clearly distinguishes specifications from contracts and has been validated by the integration of a dedicated assertion language.

Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification—*Programming by contract, Assertion checkers*; D.2.1 [Software Engineering]: Requirements/Specifications; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs—*Assertions, Invariants, Pre- and post-conditions*; D.1.5 [Programming techniques]: Object-oriented Programming

General Terms

Languages, Verification, Reliability, Measurement

Keywords

Hierarchical Components, Behavioral specifications, Contract, Interface contract, Composition contract, Fractal, Java, Confract

*This research was supported by France Télécom under contract 422721832.

I3S Research Report #2004-09

Submitted to OOPSLA'04 Vancouver, British Columbia, Canada

1. INTRODUCTION

Component-Based Software Engineering (CBSE) intends to deliver the beneficial effects that the object-oriented approach failed to completely provide: reuse of out-sourced pieces of software and thus increased productivity. Components tend to be more reusable for several reasons: (i) they can be more easily (re-)used by non-expert programmers, (ii) they usually take into account distribution and network issues from the start, (iii) they are deployable entities, facilitating extensibility and evolvability of systems, (iv) they offer several interfaces, usually separating required and provided ones and thus providing more general binding policies and a better separation of concerns.

Currently, industrial component platforms [14, 36, 33] are quite efficient in their specific domain, but their applicability is limited, as composition is not recursively organized — a component cannot be built from a composition of other components — and its resulting behavior is weakly defined and verified. Checking some semantic properties on hierarchical components boil down to *compositional reasoning*, defined by Bachman et. al [2] as the ability “to ensure that the properties of a system of components can be predicted from the properties of the components themselves”. This problem is inherently complex, but adding to it distribution and dynamic reconfiguration mean an incomplete and changing knowledge of components, hence the difficulty to provide complete proofs [34]. Several recent work are willing to establish forms of compositional reasoning [8, 34]. While these properties are relevant to CBSE, none of these work, to our knowledge, handles the general specification and verification of behavioral properties for components while taking into account the entire context described above.

Getting back to the definition of a software component as “an unit of composition with contractually specified interfaces” [45], we believe the notion of contract appears as a natural solution to express and organize such behavioral descriptions. Nevertheless contracts in object-oriented systems are usually likened to behavioral specifications using executable assertions with pre, postconditions and invariants on classes [30, 26, 40, 3] and more rarely with invariants and rules on interactions between objects[23]. In the middleware domain, the notion of contract is also used to express quality of services [20, 47], inherited from network and multimedia backgrounds, and more generally to constrain non functional properties [4]. Several work introduce some contract support for components, focusing on interfaces or connec-

tors, specifying architectural constraints [39] or quality of services [47]. But for most of these work, the contract itself is always implicit, being an informal interpretation of an associated specification. Moreover, hierarchical components are never taken into account in those approaches.

To solve these problems, we propose a general contracting model, *Contract*, which considers contracts as in the real life: the contract is the result of an agreement on some services between several parties, each one coming with its own specification. Moreover, for each provision of the contract, one can identify what is required and/or provided by each party. This model enables to define contract, which can be based mainly on two kinds of specification:

- *specifications on interfaces*, for which the contract is the result of the matching of the required and provided interfaces when two components connect to each other.
- *specifications on composition*, which can be related to the interfaces of the composite component and the interfaces of the components it contains. The result is a *composition contract*.

Specifications can be expressed with different formalisms, as long as they fit in the *Contract* model, that is the formula can be translated to one of the two forms above, the responsibility of each formula can be automatically deduced, and a verification method is implemented for the formula in its context (interface or composite component).

The contributions of this paper are two-fold. First, we define *Contract*, a general contracting model for hierarchical components that can be used to organize the definition and verification of specifications on both interfaces and composite components. Second, we show its applicability through an example using an assertion-based language, CCL-J, adapted to hierarchical components and objects. Both the contracting model and the language have been implemented on top of *Fractal* [6], a general component platform supporting recursive composition. They have also been used to specify, contract and control a small reference application and some other validation applications are almost finalized. *Contract* is also intended to be used in several large *Fractal*-based applications.

The remainder of this paper is organized as follows. Section 2 introduces *Fractal*, the component framework on which our work is developed. Section 3 presents the requirements for a general contracting model for hierarchical components. Using an example, we describe our proposed model, *Contract*, in section 4, and the CCL-J assertion language in section 5. Examples of the reified contracts are given in section 6. Section 7 discusses the implementation, related work and open issues. Section 8 concludes this paper.

2. HIERARCHICAL COMPONENTWARE: A FRACTAL VIEWPOINT

Current industrial component platforms are quite efficient in addressing specific domain targets: J2EE [14] and .NET [33] are specialized frameworks for business applications, CCM [36] defines a more general component model but aims at

providing a superset of the CORBA 2 model, focusing on distribution and business component connections rather than on composition. The applicability of these models is thus limited, as composition is “flat”, that is a component cannot be built, maybe dynamically, from an assembly of other components. This property is indeed essential to obtain a uniform viewpoint on software components at different abstraction levels and thus build a real market of software components [44].

Our work is based on the *Fractal* framework [6], which supports recursive composition. The following paragraphs briefly present this framework, its underlying model and its current implementation. Despite the fact that our proposal is dedicated to *Fractal*, results are to a large extent applicable and useful to component-based models and platforms.

2.1 The Fractal Framework

The *Fractal* framework [6], developed as a project of the ObjectWeb consortium [35], provides a general software composition framework. It supports component-based programming, providing programmatic component management, e.g. definition, configuration, composition, and administration. The main features of the framework aim at achieving these goals: composite components, shareable components, introspection capabilities, configuration and dynamic reconfiguration capabilities.

The recent release of the second version of the *Fractal* specification brings some changes to the framework. It introduces different levels of compliance, which are associated with component control, from no control at all, which makes a component very close of an object, to some extensive control, close to the full-fledged version 1. Our proposal is based on the first version, which makes it compliant at the highest level with the new specification.

2.2 The Fractal Component Model

A *Fractal* component is considered as a run-time entity composed of two parts: a content and a controller. The content of a component is composed of other components, which are managed by the controller of the enclosing component. The component model is recursive and allows components to be made of other components at an arbitrary level. Primitive components, which contain no component, are distinguished from composite components that contain other components (see figure 1). A root component is thus defined as the unique top-level one. A component may be shared by different components, in order to easily design shared resources. A component interacts with its environment through its access points called interfaces. A component may have multiple provided interfaces, which define the functionality that the component offers to other components, and multiple required interfaces, which define the functionality the component requires from the surrounding environment. On a component, an interface is thus defined by its name ($i, j \dots$ on figure 1) and its signature, that is a set of methods, coming from the underlying object model by its name ($I, J \dots$ on figure 1). On a Java-based implementation of *Fractal*, I and J would be the names of some Java interfaces. Component interfaces may also be declared optional, i.e. their binding is not necessary for the component to be started.

As for components and interfaces, contracts must be typed, with compatibility rules based on substitution principles. Intuitively, different kinds of contract have to be distinguished, according to the place and nature of the comparison between the parties. Generally, there is a possibility of contract each time there is a possible confrontation between contradictory interests from the parties, which is particularly the case between provided and required services. In an *Fractal* assembly of components, these confrontation places are mainly the following:

- *functional interfaces*, where mandatory client interfaces need a binding with a server interface that is compatible both in type and in any constraint expressed in specifications as described above;
- *assemblies*, which are made by the content controller of a composite component. The composition of interactions between internal and external services has to satisfy the provided side of the component and depends on assembled components and their interfaces.

Hence, the contract model must not only support *interface contracts*, but also provide *composition contracts*, distributed in a hierarchical way along the composition of components.

Contracts must be made of specifications and responsibilities and both of them are only known at components assembly time. Consequently, contracts has to be built during assembly and some static compatibility checking must be done: if a required property is not satisfied by an element of the composition (such as an interface or a component), a negotiation could be initiated to decide whether to find a better component, to lower the exigence level or drop a unsatisfied service. Consequently, the distinction between the specification, expressed *a priori* by interface or component designers, and the contract, built during component assembly, would create an appropriate environment to enable negotiations. The issues related to computer-assisted negotiations are briefly discussed in section 7.3.

4. THE CONFRACT MODEL

4.1 Foundations

The *Confract* model describes concepts that are useful to specify and verify, in a contractual way, properties of applications built by assembling components on the *Fractal* platform. In this section, we give an informal definition of the model illustrated by an example.

4.1.1 Contract

A *contract* consists of a list of participants (the signatories) and a list of specifications (the provisions). Every specification must be under the responsibility of an unique guarantor and accessible to one or more beneficiaries. The guarantor of a specification is one of the participants and may be the beneficiary of another specification of the same contract.

The *Confract* model distinguishes three kinds of contracts¹:

¹We do not support operation contract like in the Eiffel

- *Library (unit) contracts* define the semantics of the underlying OO language units (interfaces or classes). These units are used in applications, but without being components interfaces. They often come from the language libraries, like data structures, windowing elements, etc. or they can be application specific. These units were called components before middleware components emerged (for example in [23, 32]) and the contracts are those of the OO languages [30, 26].
- *Interface contracts* are located at the junction point of required and provided component interfaces, such as interface *t* in figure 1. These interface contracts only refer to entities accessible in the interface and its methods.
- *Composition contracts* refer to entities from several external interfaces of the component or of its internal sub-components. For example, a property that concerns several external interfaces of a component, or one or more internal components, or external interfaces bound to internal components, has to be controlled by a composition contract. On figure 1, a contract that refers to specifications of *i* and *j*, or of *i* and *s*, or of *s* and *t*, is a composition contract. This kind of contract only refers to entities accessible from the contract controller of the component, which manages specifications.

The life cycle of contracts is organized in two stages : *development* and *operation*. During the development stage, a contract is open. The specifications of an open contract may vary, in particular during a negotiation process. The participants of an *open* contract may not be known. A contract is *closed* when all its specifications are stabilized and associated with an identified guarantor. The closure of a contract in the *Confract* model corresponds to the signing of a contract in real life. In a closed contract, none of its specifications may be modified nor a guarantor be changed. To modify a contract, it must be first canceled and a new one must be created. Interface and composition contracts can be closed, but library contracts are not necessarily closed when they do not explicitly define the beneficiaries of the provided services.

4.1.2 Specification

A *specification* describes one or more logic properties (or constraints) that must be satisfied during given periods or at given times at application runtime. Every specification is written in a given formalism, but the specifications of a same contract may use different formalisms, according to the nature of the properties and the adapted verification techniques. Thus, very stable functional aspects could be described in an algebraic specification language, QoS specifications in QML [20], some temporal aspects with TLA [27] for example. In the current implementation of *Confract*, only CCL-J specifications are supported (*cf.* section 5). When a specification is complex, it can be broken into several *clauses*, which can be identified by a *number* or a *label* to document contract violation.

language, as they are included in library contracts and are not separately negotiable. Other kinds of contracts, like on synchronization, could subsequently enhance the model.

Every specification must be verifiable, automatically or not, using proofs, static analysis tools or runtime checking. A specification has a life cycle and can go through different states during its construction and verification: unstable, stable, verified and possibly certified. At runtime, a stable specification can only be in one of three states: undefined, true or false. For example, a method precondition will be true or false at the method entry — its satisfaction period — and will be undefined during other periods.

The *context* of a specification is the scope of entities that can be referred to. It is determined from the position of the specification in the application, using the scope rules of the *Fractal* model. Therefore a specification may be located on an interface of the underlying language or on a component controller.

Two specifications written in the same formalism, with the same context and the same satisfaction period can be compared to determine their *compatibility*. A specification of an entity A (method, interface, library unit) is compatible with a comparable entity B if A can be substituted for B and can satisfy B for all execution.

Contracts are different from specifications on several points:

- they can use several different guarantors,
- they can use several formalisms on different specifications,
- Specifications are provided by developers in advance, when contracts are dynamically built (reified) by a contract controller, following a possible negotiation to compare required and provided properties.
- Specifications can be verified in advance, particularly syntactically, before their integration in the contracting system. They could also be thoroughly verified in case of certification. Contracts cannot be certified in advance, as all their guarantors are not generally known.

However specifications and contracts have common features:

- These are texts with a specific syntax, which can be internally reified or printed in a readable way.
- They are typed and can be compared against their compatibility.

4.1.3 Contract participants

The participants of a specification are:

- its *guarantor*, which is the responsible entity (in reality the developer) of the described properties. The term “responsible” is not bound to be interpreted here as a guilty party. If the properties is not satisfied, the guarantor must act at best to make the application run in a satisfactory way from both functional and non functional viewpoints. On a component platform,

the separation between qualities of correction and robustness — traditional in software engineering — is no more so relevant and faults that are detected for both aspects are uniformly handled by exception raising. The hierarchical nesting of component controllers provides a natural frame to intercept, handle or propagate those exceptions. The request of reconfiguration and the renegotiation on expected requirement levels are among the possible actions of a guarantor.

- one or more *beneficiaries*, which are entities that can “rely” on the properties described in the specification. It plays the role of working hypothesis for the beneficiaries. This separation of the responsibilities between guarantor and beneficiaries brings numerous advantages [30] : one knows which entity to notify in case of violation, no risk of verification oversight or repetition, decisions making is clarified.
- possibly, one or more *contributors*, which are entities indirectly responsible of some parts of the specification. They can act to make the specification true. Therefore, in case of invalidation of the specification, its guarantor can also *negotiate* with the contributors to make the specification true again.

The contracting system especially needs to know the guarantors of the different specifications. The set of all guarantors of all specifications of a closed contract is the set of its participants. A beneficiary of a contract's specification is a participant of this contract only if it is the guarantor of another specification of the contract. When a contract is open, some of its specifications have potential guarantors, which are not yet identified. The guarantors of library unit contracts are identifiable when the usage of the unit in the implementation code of components is known. For other contracts, the identification of participants is made at the time of interfaces binding (interface contract) or component assembly (composition contract). In particular, the following rules apply:

- An interface contract has two participants, the two connected interfaces;
- If a contract has more than two participants, it is necessarily a composition contract. However a composition contract can have only two participants, for example if the interfaces of its participants are not connected.

4.2 An example : a copier

We illustrate the *Confract* model with a simple example that simulates a black & white copier. The description of this example is very partial here, but a complete version will be available soon in an extensive *Confract* tutorial. The external view of the copier, considered as a black box, is given in figure 2.

The copier environment is a pseudo-component that corresponds to the real world and can be materialized as “plug” interfaces during testing. The copier provides two interfaces, a command panel and an output tray for copies.

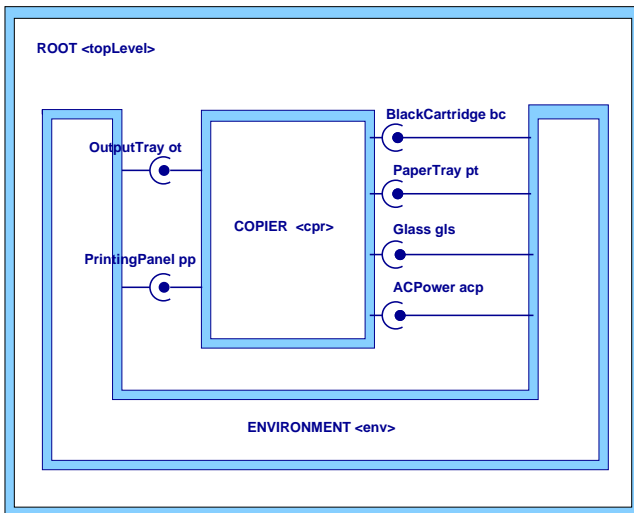


Figure 2: External architecture of COPIER.

Required interfaces manage everything the environment must provide to realize a copy: black ink, a blank paper tray, a glass for the original document and a power supply. All these interfaces are described in the support language of *Contract* and *Fractal*, currently Java². Specifications of the different constraints are given in the following section using CCL-J.

4.2.1 Library interfaces signatures

As seen before, library units describe the general properties which are independent of any assembly. These properties are used in the constraints on components and therefore must be known, but usually, they already exist.

As for the copier, the following interfaces must be known: Sheet (paper sheet), BWImage (black & white image), BWDot, Stack, ...

```

interface Sheet { // Sheet of Paper
  float width (); // inches
  float height (); // inches
  BWImage image (); // image in black and white on the sheet
  boolean isBlank (); // is the sheet blank ?
  boolean isStable (); // is the image stable ?
  float heightResolution (); // dots per inch (dpi) on height
  float widthResolution (); // dots per inch (dpi) on width
  boolean isSame (Sheet s); // is this sheet the same as s ?
}
interface BWImage {
  // Digital image, considered as a raster of dots in black and white.
  int width (); // number of dots on width
  int height (); // number of dots on height
  BWDot getDot(int x, y); // get the BWDot with x, y
  void setDot(int x, y, BWDot p); // set the BWDot with x, y
  boolean isEqual (Image other); // is this object equal to other ?
}
interface ColorImage extends BWImage {
  // Digital image, considered as a raster of color dots
  ColorDot getDot(int x, y); // get the Colordot with x, y
  void setDot(int x, y, ColorDot p); // set the Colordot with x, y
}

```

²To reduce notations, the **public** keyword is implicit.

```

}
interface BWDot { // Black and white dot
  boolean isWhite (); // is this dot white ?
}
interface ColorDot { // Color dot
  boolean isWhite (); // is this dot white ?
  int Red, Green, Blue; // the color of this dot in RGB system
}
interface PrintCartridge { // replaceable toner
  float level (); // current ink level in ml
  void setLevel (float newLevel);
  String color ();
}

```

Description 1: Library interfaces signatures.

4.2.2 Signatures of the copier external interfaces

The signatures of the external interfaces of the copier must then be described. Required ones are BlackCartridge, PaperTray, Glass and ACPower. Provided interfaces are PrintingPanel and OutputTray.

```

interface BlackCartridge { // Black cartridge support
  PrintCartridge cartridge ();
  void replaceCartridge(PrintCartridge c);
}
interface ACPower { // AC power for all components in the copier
  boolean isConnected (); // is AC power connected ?
  void connect (); // connect AC power
  void unConnect (); // unconnect AC power
}
interface PaperTray { // Input tray with blank sheets
  float trayWidth (); // inches
  float trayHeight (); // inches
  Stack<Sheet> tray (); // the full paper tray
  void loadTray (Stack<Sheet> new); // pushes new sheets onto the tray
  int capacity (); // max. nb. of sheets in paper tray
  Sheet topSheet (); // the sheet of the top of this tray
  boolean isPaperJam ();
}
interface Glass { // Glass for original documents
  Sheet original (); // document to copy
  float glassWidth (); // inches
  float glassLength (); // inches
  void loadOriginal (Sheet o);
  void removeOriginal ();
}
interface PrintingPanel { // Panel to command copies
  boolean isPowerOn (); // is AC power on ?
  boolean isBusy (); // is copier currently copying ?
  float copyDuration (); // effective duration of current copy
  void copy (int n); // asks n BW copies of the original
}
interface OutputTray { // Output tray with the asked copies
  Stack<Sheet> tray ();
  void removeSheets (); // remove all sheets of this tray
  int capacity (); // max. nb. of sheets in this tray
  Sheet topSheet (); // the sheet of the top of this tray
  boolean isPaperJam ();
}

```

Description 2: External interfaces signatures.

4.2.3 Internal architecture

We now describe the internal architecture of the copier. To simplify the example, we only use three main components that should be reusable in other devices, such as scanners, fax machines, or all-in-one printers. The component **SCANNING** produces a digital image from the original document. The component **PRINTING** prints the digital image onto a blank paper sheet, but the result is not yet stabilized. The component **FINALIZATION** stabilizes the fresh print, by drying for example. A component **DRIVER** is added to realize the working of these three components. This component is connected to the **PrintingPanel** and controls the three other components using three interfaces, **FinalizationDriver**, **PrintingDriver** and **ScanningDriver**.

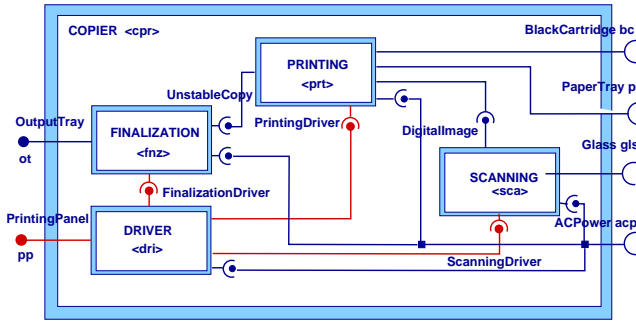


Figure 3: Internal architecture of COPIER.

4.2.4 Interfaces signatures of internal components

Some external interfaces of internal components are directly connected to external interfaces of the composite component COPIER³. This is the case for **BlackCartridge** *bc*, **PaperTray** *pt*, **Glass** *gls*, **ACPower** *acp*, **OutputTray** *ot* and **PrintingPanel** *pp*.

Some external interfaces of internal components are directly bound to each other, that is their provided and required interfaces are compatible. This is the case for interfaces controlling program flow, such as **FinalizationDriverClient**, **ScanningDriverClient** and **PrintingDriverClient**. These interfaces are bound to the corresponding server interfaces of the component **DRIVER**. The interfaces **UnstableCopy** and **DigitalImage** provides services that can be different whether they are required or provided interfaces. For example, the scanner used in a B&W copier could easily scan color images, a B&W projection of which would be used by a printing component that used only black ink. The comparison of required and provided interfaces is first done by the *Fractal* system which checks the static compatibility of signatures types. The *Confract* system then establishes an interface contract accepted by common consensus of the two components, following a more or less automated negotiation (cf. section 6).

```
interface ClientDigitalImage { // PRINTING side
    BWImage image (); // Digital image to proceed
    void imageProcessing (BWImage im);
    // put processed im into image
}
```

³These are interfaces that cross the boundary of the component COPIER.

```
interface ServerDigitalImage { // SCANNING side
    ColorImage image (); // Digital image to proceed
    void imageProcessing (ColorImage im);
    // put processed im into image
}

interface ServerUnstableCopy { // PRINTING side
    Sheet copy ();
    void setCopy (Sheet s);
}

interface ClientUnstableCopy { // FINALIZATION side
    Sheet copy ();
    void setCopy (Sheet s);
}

interface FinalizationDriver { // FINALIZATION side
    void finalize ();
    float finalizationDuration ();
    // duration to finalize one sheet (in ms)
}

interface PrintingDriver { // PRINTING side
    void print ();
    float printingDuration (); // duration to print one sheet (in ms)
}

interface ScanningDriver { // SCANNING side
    void scan ();
    float scanningDuration (); // duration to scan one sheet (in ms)
}
```

Description 3: Signatures of external interfaces of internal components.

5. THE CCL-J ASSERTION LANGUAGE

5.1 Rationale

The *Confract* model aims at supporting contracts with specifications that are expressed in different formalisms. We define CCL-J (Component Constraint Language for Java), a formalism of executable assertions with quantifications, which is dedicated to *Fractal* component interfaces and component composition. Of course, constraints on library units can also be defined with CCL-J, that is on Java interfaces. Besides CCL-J follows our previous work on assertions and their efficient evaluation [11, 12] and on the OCL language [46].

CCL-J provides five conventional kinds of specification [30, 15], which correspond to well identified points of the space-time and guarantors:

- a precondition **pre**: is valid at the entry of an interface method and the method caller is its guarantor ;
- a postcondition **post**: is valid at the exit of an interface method and the implementation of the method is its guarantor ;
- a specification **rely**: is valid during the interface method call and its guarantor is the method caller ;
- a specification **guarantee**: is also valid during the interface method call but its guarantor is the implementation of the method ;

- an invariant **inv:** is valid during an interface *observable periods*, that is from the instantiation to the destruction of the objects that implements the interface and only during inactivity periods of these objects (between method calls); These inactivity periods are managed by the life cycle controller of each *Fractal* component.

A CCL-J specification is a logic formula, with a syntax adapted to the Java language and the possibility of quantifications on collections, as in the OCL language. For documentation and simplification purposes, specifications can be broken into several labelled clauses, which are then implicitly conjuncted. On program 4, the precondition is broken into clauses which are identified by clauses such as `InkLevelOk`, `TrayOk`, etc. During runtime checking of these clauses, the *Confract* system can then name the labels of invalid clauses.

```
pre:
  InkLevelOk:
    bc.cartridge().level() >= 0.05; // ml
  TrayOk:
    pt.tray.size() > n && ! pt.isPaperJam();
  dimPaperOk:
    pt.width() >= pt.topSheet().width() &&
    pt.height() >= pt.topSheet().height() ;
  ...
```

Description 4: A specification broken into clauses.

CCL-J also enables to define variables local to specification (**let**) and provides an keyword **implies** with the conventional semantics.

In a postcondition, one can refer to the function result with a specific variable `result` and the state of an expression at method entry is accessible using the postfix operator `@pre`, just like in OCL.

```
post: result == image.height() / height() ;
post: getstack().size() == (getstack().size())@pre-1;
```

Description 5: Example of the result and @pre constructs.

In this example, the first postcondition defines the return value of the function and the second one states that the size of a stack has increased by one between method entry and exit.

5.2 Contexts and spatial quantifications

As CCL-J constraints are described separately from any signature or architecture descriptions, the context of these constraints must be specified. As in OCL, CCL-J uses a **context** construct but adapted to components needs:

- on Java interface signatures (library unit): the specification is valid whatever use is made in *Fractal*, at the component (*Fractal* interface) or implementation (Java class implementing a *Fractal* interface) level;
- on a method of a Java interface: the validity is the same as the previous case, but limited to the specified method;

- on an instance, a type or a template of a *Fractal* component: the specification is then valid for the specific instance or all instances generated by the component type or template. All interfaces accessible from the component controller can then be referred to, that is, external interfaces and, if the component is composite, all external interfaces of its subcomponents.

To designate *Fractal* interfaces, a construct **on** specifies on which component they are located. The names of component, component type or template are specified inside `<...>`, like with `<cpr>` ou `<COPIER>`. Java and *Fractal* interfaces are accessed with path expressions, which can be simplified by Java-like **import** clauses:

```
import contract.copier.*;
import contract.copier.ui.controller
import java.util.*;
on <COPIER> ...
```

Description 6: Import statements.

The following example shows the contexts of a Java interface and of one of its method, without specifying any component (no **on** clause). In the first case, only invariants can be specified. In the second case, one or more specifications of kind **pre:**, **post:**, **rely:** or **guarantee:** can be described.

```
context contract.copier.Sheet
  inv: height () <= 11 ; // inches
      width () <= 8.5 ; // inches

context boolean contract.copier.Sheet.isBlank()
  post: result == ( image /= void ) &&
        forEach( int w in 1 .. width() :
          forEach( int h in 1 .. height() :
            image().getDot(w,h).isWhite() ));
```

Description 7: Context statements of library units.

The **on** construct enables to specify properties on components (instances, types or templates) and also to factor different contexts of this component.

```
on <COPIER> ... // context on the component type COPIER

on <cpr> ... // context on an instance of COPIER
```

Description 8: Context statements of Fractal components.

The choice between an instance or a template/type of a component must always be done in favor of generality: the identity of a component is specified only if the specification depends on this identity.

```
on <COPIER>
  context pt
    inv: acp.isConnected();

  context Stack<Sheet> ot.outTray()
    post:
      result /= void implies
        forEach (int i in 0 .. result.size()
```

```

    result.elementAt(i).width() == pt.trayWidth() &&
    result.elementAt(i).height() == pt.trayHeight() );
end on

```

Description 9: Example of context statements of a component.

The above example first shows an invariant on the `pt` interface which depends on another interface `acp` of the same component. Then the postcondition states that the dimensions of all sheets of the output tray are the same as the ones of the input tray.

A specification can often be factored to several methods or even several interfaces. In order to facilitate the expression of constraints according to a specific viewpoint (a functional or non functional constraints for all operations or interfaces of some kind), CCL-J enables to factor a constraint to several contexts with a `*` generator. Regular expressions can then be defined on names to realize a spatial quantification.

```

on <COPIER>
  context *.*
  rely: acp.isConnected();
  context void PrintingPanel.*Copy (int n)
  pre: 0 < n < COPIESMAX;
end on

```

Description 10: Use of the `*` generator.

The above example first shows a constraint on the availability of the electricity supply for all operations of the component type `<COPIER>`. Then a precondition on the number of fired copies is defined identically on all copy commands (one can imagine both a `ColorCopy` and a `BlackCopy` method for a colour copier).

5.3 Quantifications on collections

To increase assertions expressiveness, CCL-J supports some simple collective operators on collections and some specific shortcuts. In particular, it is often useful to quantify over a numeric range, a type extent or some collections.

We use here a notation inspired from OCL, anchoring some specific operations on the Java 2 collection hierarchy and extending the Java syntax accordingly. The collective operations use the following syntax:

```

collection.operation ( Type v : boolean expression
                      with v )

```

The following methods are then provided:

- **select**, which filters a subset of a collection according to a given criterion (and **reject** as the complementary operation);
- **collect**, to get a collection resulting from the projection of the given expression onto the elements of the collection;
- **exists**, which corresponds to an existential quantification;
- **forAll**, which corresponds to an universal quantification.

A specific shortcut of **forAll** enables to easily range over integers:

```

forEach ( int v in low .. sup : expression with v )

```

```

import contract.example.copier
on <COPIER> context void pp.copy(int n)
  post:
  copy: // all n last copies have an image equal to that of the original.
  forEach (int i in size()-n+1 .. size():
    ot.tray.elementAt(i).image.isEqual(
      gl.original.image) )
  paperPath: // output is in reverse order of input
  let (int oTopPt=pt.tray.size()@pre)
  forAll (i in 1 .. n) :
    pt.tray.elementAt(oTopPt-i-1)@pre.
    isSame(ot.tray.elementAt(ot.tray.size()-i+1))
end// on <COPIER>

```

Description 11: Examples of quantifications

The above example shows a postcondition of the `copy` method. The first clause specifies that the `n` copies on top of the the output tray have an image identical to the original document. The second clause states that the resulting `n` sheets in the output tray are the ones from the paper tray in inverse order.

6. CONTRACTS

Contracts are built during the component assembly stage, using the various descriptions of the application: the architecture expressed in the *Fractal* ADL, Java interface signatures and provided specifications. In the remainder of this section, specifications will be considered as CCL-J constraints. The construction stage leads to the reification of contracts. It is then possible to easily produce a textual form, which can be used as documentation. We use this textual representation to present some resulting contracts on our running example, showing the main parts of a contract : participants, provisions and guarantors. For documentation purpose, interface signatures are also provided.

As seen in section 4, the *Contract* system supports three kinds of contract, on library units, interfaces and composition. The two last ones are the main contributions of the system.

6.1 Library contracts

Library unit contracts are not original and correspond to contracts in object-oriented languages [30] and their extensions [26, 46]. The participants are only identified when the library unit is used in the implementation code and the contracts mainly aim at detecting programming errors.

A contract is then associated with each library unit. The benefit of using quantifications in CCL-J is significant, as shown by the definition of the `nochange`: properties when modifying a dot, or the definition of the result of `isEqual` in the following example:

```

library contract BWImage #3 {
  provisions // Digital image, considered as a raster of dots in black and white.
  int width (); // number of dots on width
  int height (); // number of dots on height
  BWDot getDot(int x, y); // get the BWdot with x, y
}

```

```

void setDot(int x, y, BWDot p); // set the BWDot with x, y
pre:
  1 <= x <= width() ;
  1 <= y <= height() ;
  p /= void ;
post:
  change: p.isEqual(getDot(x,y)) ;
  nochange: forEach ( int w in 1 .. width() :
    forEach ( int h in 1 .. height() :
      w /= x && h /= y implies
        getDots(w,h).isEqual(getDots(w,h)@pre)))
boolean isEqual (Image other); // is this object equal to other ?
pre:
  other /= void ;
post:
  result = forEach ( int w in 1 .. width() :
    forEach ( int h in 1 .. height() :
      getDots(w,h).isEqual(other.getDots(w,h))))
}

```

Description 12: Library contract on BWImage.

6.2 Interface contracts

The *Contract* system builds an interface contract for each interfaces binding by comparing the required and provided sides of these interfaces. A first check is done by the *Fractal* platform to ensure that the type of the server interface (provided) is compatible with the client interface (required). If it is the case, the *Contract* system pursues the comparison using the available specifications.

6.2.1 Identical required and provided interfaces

The first simple case is when required and provided interfaces are the same, as the client and the server use the same interface type. The comparison is then trivial and the contract is built using all specifications associated with the interface. On the copier example, this is the case for some provided interfaces such as `PrintingPanel` or `OutputTray`, and required ones such as `BlackCartridge`, `PaperTray`, `ACPower` or `Glass`. All these interfaces have the environment `<env>` as a participant, and the copier `<cpr>` with inverted responsibilities according to the required or provided nature of the interface.

For example, the contract on the `PrintingPanel` interface is the following:

```

interface contract on <ROOT> #11 {
binding server = PrintingPanel <cpr>.pp,
  client = Main <env>.main
participants cpr, env
provisions
  boolean isPowerOn (); // is AC power on ?
  boolean isBusy (); // is copier currently copying ?
  float copyDuration (); // effective duration of current copy in ms
  void copy (int n); // ask n BW copies of the original
  pre: guarantor is env
    0 < n <= 500 ;
  post: guarantor is cpr
    copyDuration == time() - time()@pre
    copyDuration <= 60000*n/17; // required speed is 17 ppm
  rely: guarantor is env
    isPowerOn ();
  guarantee: guarantor is cpr
    isBusy ();
  inv: guarantor is cpr
    !isBusy ();
}

```

Description 13: Interface contract on `PrintingPanel pp`.

An interface contract is first made up of an identification part:

- contract type: interface;
- scope: component `<ROOT>`;
- an ID code: a number for example;
- components binding for its interfaces, on the server (`<cpr>.pp`) and client (`<env>.main`) side;
- the participants list, with only two participants in the case of an interface contract;
- and the provisions list of the contract, in the form of method signatures and associated specifications. In the case of CCL-J, specifications are **pre**:, **post**:, **rely**:, **guarantee**: and **inv**: clauses.

In the case of `PrintingPanel`, the semantics of the resulting interface contract is somewhat poor, as it only refers to properties that concern the interface itself. But the working of a printing depends on the different external interfaces of the copier: to make a copy, one need some electrical supply, enough paper, an original document correctly installed on the glass, some (black) ink. And then, the result can be observed in the output tray. Therefore, it is in the composition contract seen below that will be stated the key semantics of the panel.

On the contrary, in the case of `PaperTray`, the resulting semantics is richer, as its working mainly depends on the interactions between the copier, which consumes paper, and its environment, which has to reload the tray with blank paper from time to time.

```

interface contract on <ROOT> #12 {
binding server = PaperTray <env>.pt,
  client = PaperTray <cpr>.pt
participants env, cpr
provisions
  int capacity (); // max. nb. of sheets in paper tray
  boolean isPaperJam ();
  float trayWidth (); // inches
  float trayHeight (); // inches
  Stack<Sheet> tray (); // the full paper tray
  void loadTray (Stack<Sheet> ns);
  // pushes a new stack ns of sheets onto the tray
  pre: guarantor is env
    ns != void ;
    ns.size() + tray.size() <= capacity() ;
    ns.forAll (Sheet s : s.width() == trayWidth() &&
      s.height() == trayHeight());
    ns.forAll(Sheet s : s.isBlank());
  post: guarantor is cpr
    tray.size() == tray.size()@pre + ns.size() ;
    forEach (i in 1 .. tray.size()@pre:
      tray.elementAt(i).isSame(tray.elementAt(i)@pre));
    let ( int olast=tray.size()@pre: ;
      int nlast=tray.size() )
      forAll (i in olast+1 .. nlast :
        tray.elementAt(i).isSame(ns.elementAt(i-olast)@pre));
  Sheet topSheet (); // the sheet of the top of this tray
  post: guarantor is cpr
    tray.size() >0 implies tray.peek().isSame(result) ;
}

```

```

inv: guarantor is cpr
    tray /= void ;
    5.5 <= trayHeight() <= 11 ;
    4.25 <= trayWidth() <= 8.5 ;
    tray.size() <= capacity() ;
    tray.forAll(Sheet s : s.isBlank()) ;
    tray.forAll(Sheet s : s.width() == trayWidth() &&
        s.height() == trayHeight() ;
}

```

Description 14: Interface contract on PaperTray pt.

As a result, this contract defines how paper is loaded, ensuring that the environment satisfies the precondition: the provided paper stack does not overflow from the tray, the paper dimensions are appropriate... As postconditions, the copier guarantees that it has taken well the provided paper on top of the tray, allowing to maintain the tray invariant: paper dimensions inside the acceptable range and identical to the tray, and paper always blank. The condition `tray /= void` in the invariant ensures that the implementation of the `PaperTray` object instantiates a stack for the `tray` reference. This simplifies all specifications regarding `tray`.

6.2.2 Distinct but compatible required and provided interfaces

We now focus on distinct interfaces, such as `DigitalImage` in our example. The `PRINTING` component requires a B&W image and we suppose it needs an `imageProcessing` method (a smoothing operation for example).

```

// reminder of the specification given on description 3
import contract.example.copier
interface ClientDigitalImage { // PRINTING side
    ImageBW image () ; // processed image
    void imageProcessing (ImageBW im) ; // put im into image
}
on <PRINTING>
context ClientDigitalImage
inv:
    image.height() >= 14000 ; // dots
    image.width() >= 11000 ; // dots
context void imageProcessing (ImageBW im) ;
pre:
    im.height() >= 18000 ; // dots
    im.width() >= 15000 ; // dots
post:
    image.isEqual(im) ;
    image.height() >= 14000 ; // dots
    image.width() >= 11000 ; // dots
end on

```

Description 15: Signature and specification of ClientDigitalImage.

This required interface supplies to the `PRINTING` component a B&W image with a resolution of 14000×11000 dots grabbed from an original with a 18000×15000 dots resolution. The `SCANNING` component is also supposed to be more efficient than `<PRINTING>`: it should provide a color image with a better resolution according to less requirements on the original document.

```

// reminder of the specification given on description 3
interface ServerDigitalImage { // SCANNING side
    ImageColor image () ; // processed image
    void imageProcessing (ImageColor im) ; // put im into image
}

```

```

on <SCANNING>
context ServerDigitalImage
inv:
    image.height() >= 15000 ; // dots
    image.width() >= 12000 ; // dots
context void imageProcessing (ImageColor im) ;
pre:
    im.height() >= 16000 ; // dots
    im.width() >= 13000 ; // dots
post:
    image.isEqual(im) ;
    image.height() >= 15000 ; // dots
    image.width() >= 12000 ; // dots
end on

```

Description 16: Signature and specification of ServerDigitalImage.

The type comparison between the `ServerDigitalImage` and `ClientDigitalImage` interfaces is successful, as a color image is made of color dots that specialize B&W ones. The comparison of the associated specifications is then made to build a consensual contract on the binding. When types are compatible, this contract can be obtained using a simple algorithm, but which can be improved on. This algorithm applies the subtyping rules to assertions, as in the Eiffel language [30], but more formally defined in [17] as behavioral subtyping.

As the server interface type is a subtype of the client interface, the server specifications **pre:** and **rely:** must not be stronger and the specifications **post:** and **guarantee:** must be at least as strong as the client ones.

As for **pre:** and **rely:**, the contract connects the required and provided conditions with an operator **or else**, as in Eiffel. The following rule then applies for required conditions of the client (Req_{cli}) and the server (Req_{serv}):

$$(Req_{cli} \vee Req_{serv}) \wedge (Req_{cli} \Rightarrow Req_{serv})$$

The specifications on **post:** and **guarantee:** are connected in the contract by an operator **and then**, applying the following rule:

$$(Prov_{cli} \wedge Prov_{serv}) \wedge (Prov_{serv} \Rightarrow Prov_{cli})$$

in which $Prov_{serv}$, resp. $Prov_{cli}$, are provided conditions of the server, resp. the client.

Finally, the same rule as for **post:** and **guarantee:** can be adapted to check client and server invariants. It is expressed by the same operator in the presentation of the contract.

As a result, the consensual contract for the `DigitalImage` interfaces is the following:

```

interface contract on <COPIER> #17 {
binding server = ServerDigitalImage <sca>.sdi,
         client = ClientDigitalImage <prt>.cdi
participants sca, prt
provisions
  ImageWB image (); // scanned image
  void imageProcessing (ImageBW im);
  pre: guarantor is prt
    im.height() >= 18000 ;
    im.width() >= 15000 ;
  or else
    im.height() >= 16000 ;
    im.width() >= 13000 ;
  post: guarantor is sca
    image.isEqual(im) ;
    image.height() >= 14000 ;
    image.width() >= 11000 ;
  and then
    image.isEqual(im) ;
    image.height() >= 15000 ;
    image.width() >= 12000 ;
  inv: guarantor is sda
    image.height() >= 14000 ;
    image.width() >= 11000 ;
  and then
    image.height() >= 15000 ;
    image.width() >= 12000 ;
}

```

Description 17: Consensual interface contract for DigitalImage, in a raw form.

The possible incompatibilities between required and provided specifications will be detected during runtime checking. For example, if the client `prt` processes the image with an original color image and a 18000×15000 resolution, the precondition will be directly satisfied. The processing algorithm will be the one of the server `sca` and will provided a color image with a resolution higher than 15000×12000 , satisfying the postcondition of the `imageProcessing` method and the contract invariant. At the time of the assignment, the color image will be subsumed in a B&W image.

In some favorable cases, it is possible to simplify the raw contract, such as in the following example:

```

interface contract on <COPIER> #17 {
binding server = ServerDigitalImage <sca>.sdi,
         client = ClientDigitalImage <prt>.cdi
participants sca, prt
provisions
  ImageWB image (); // scanned image
  void imageProcessing (ImageBW im);
  pre: guarantor is prt
    im.height() >= 16000 ;
    im.width() >= 13000 ;
  post: guarantor is sca
    image.isEqual(im) ;
    image.height() >= 15000 ;
    image.width() >= 12000 ;
  inv: guarantor is sda
    image.height() >= 15000 ;
    image.width() >= 12000 ;
}

```

Description 18: Consensual interface contract on DigitalImage, in a reduced form.

But in the general case, the simplification is undecidable, as it is not possible to deduce whether a condition is weaker or stronger than another one without runtime checking.

6.2.3 Distinct and incompatible required and provided interfaces

When the required and provided interface types are incompatible or when types are compatible but specifications are invalidated by some dynamic admission control [13], a negotiation is necessary to build the consensual interface contract or declare the assembly impossible. This negotiation, which may use a human arbitration or some combinatorial optimization algorithms, is under study. We also expect to reuse or adapt work on e-negotiation [42, 1, 43].

6.3 Composition contracts

The *Contract* system builds a composition contract for each component that has specifications referring to unbound interfaces. At a more abstract level, two forms of composition contract can be distinguished:

- external ones, which only refer to external interfaces of a component, whether it is primitive or composite;
- internal ones, which refer to external interfaces of a composite component as well as external interfaces of its subcomponents. As a result, this form of composition contract is only applicable to component instances or templates, as *Fractal* component types only define the external interfaces.

As for checking techniques, there is no difference between these two forms which are under the control of the component contract controller. But if one wants to show the collaboration between the external interfaces of a component, the appropriate form can be extracted from the composition contract.

There can be two or more participants in a composition contract. As there is no direct binding between interfaces, there is no comparison between types or specifications, but only a conjunction of all properties that are expected to be true and must be checked.

In order to build the composition contract of a component, two kinds of specifications must be separated out: the ones related to interface contracts and the ones related to composition contracts. In the case of CCL-J, specifications with an context `on` are the composition-related specifications.

The example below shows the composition contract on the `<COPIER>` component. The key method is `copy` on the `PrintingPanel` interface, but one can imagine other control methods, on other server interfaces or, more rarely, on client interfaces. The composition contract thus groups together provisions interface by interface, in the example `PrintingPanel`, `OutputTray`, etc.

```

composition contract on <COPIER> #22
participants cpr, env, fnz, prt, sca
provisions on interface PrintingPanel pp {

```

```

boolean isPowerOn (); // is AC power on ?
  rely: guarantor is cpr
    acp.isConnected() ;
boolean isBusy (); // is copier currently copying ?
float copyDuration (); // duration of last copy command in ms
void copy (int n );
  // ask n copies in BW of the original with a speed of 17 ppm
pre: guarantor is env
  InkLevelOK: // minimim level is 0.05 ml
    bc.blackCartridge.level() >= 0.05 ;
  InputTrayOk: pt.tray.size()>n &&! pt.paperJam();
  OutputTrayOk: n <= ot.capacity() -ot.tray.size();
  originalOk: gl.original.image /= void;
post: guarantor is cpr
copy: forEach (int i in 1 .. n :
  ot.tray.elementAt(ot.tray.size()-i+1).image.
  isEqual(gl.original.image));
inkConsumption: // Ink consumption is proportional to n
  bc.blackCartridge.level@pre-0.02*n
  <= bc.blackCartridge.level
  <= bc.blackCartridge.level@pre;
paperPath: // output is in reverse order of input
  let (int oTopPt=pt.tray.size()@pre)
  forall (i in 1 .. n ) :
    pt.tray.elementAt(oTopPt-i-1)@pre.
    isSame(ot.tray.elementAt(ot.tray.size()-i+1))
copyDuration:
  copyDuration == <sca>.driver.scanningDuration() +
  n * (<prt>.driver.printingDuration() +
  <fnz>.driver.finalizationDuration() )
speed: // required speed is 17 ppm
  copyDuration <= 60000 * n / 17;
inv: guarantor is cpr
  pp.isPowerOn() implies acp.isConnected();
}

```

Description 19: Composition contract on the copier component

From the server interface `PrintingPanel`, the contract captures the interactions between the external interfaces provided (`OutputTray`) and required (`BlackCartridge`, `PaperTray`, `Glass ACPower`), and consequently between the participants `cpr` and `env`. The preconditions on `copy` state the hypotheses on other interfaces for the copy to work correctly: enough ink is needed, the paper tray is correctly loaded enough with blank paper, etc. In the same way, the postconditions of `copy` state the expected result on the output tray (the `n` top sheets of the paper tray have an image identical to the original document) and the effects on ink and paper consumption. These last specifications resemble more non-functional constraints on resource consumption than behavioral specifications.

Other non functional specifications, which are related to the copy duration, break the total duration into the time spent in the different internal components and also verify the expected printing speed. Some “data flow” constraints may also be expressed, on the path of the paper or on the successive conversion of the original image through the internal components `<SCANNING>`, `<PRINTING>` and `<FINALIZATION>`.

7. DISCUSSIONS

7.1 Implementation

An implementation of the *Confract* model has been developed on top of the *Fractal* 1.1 platform [19] and its imple-

mentation in Java, *Julia*. The current version of the system consists of 330 Java classes, representing around 24000 lines of code (with a selective counting of non empty and uncommented lines in methods body). It uses ANTLR to analyze the CCL-J language and the Beanshell, a full Java scripting language and interpreter, to easily prototype assertions checking at runtime. Both interface and composition contracts are completely supported and the integration of the CCL-J formalism is currently covering almost the entire language⁴.

The current prototype is organized in two layers. An underlying technical layer consists of a controlling and debugging API for *Fractal*. This API is based on event notification and combines debugging and control features with an architecture similar to the Java Platform Debugging Architecture. Those features are directly available through a specific *Fractal* controller, called a `ServiceController`. The upper layer uses this `ServiceController` to add a `ContractController` to each component. This controller manages interface contracts of possible subcomponents and the composition contracts of the component. As for the CCL-J integration, the contract controller drives interceptions on *Fractal* interfaces and on other controllers (`LifeCycle`, `Binding` and `Content`) in order to be notified of necessary events and check executable assertions according to their specification kind.

A complete version of the copier example has been developed with *Confract*, showing approximatively the same internal structure as in this paper, but with more interfaces on components, for example to provide both B&W and color ink cartridges from the copier environment.

Other validation examples are under construction and should be released soon. A portable multimedia player is complete and can be deployed on both classic J2SE and J2ME using the Connection Device Configuration on PDA. Using contracts, we are currently experimenting dynamic reconfiguration of this player according to available resources. A video streaming system is also under construction using the RMI version of *Fractal* and the RTP protocol for streaming. On this system, we expect to conduct the same kind of experiments taking into account network-oriented quality of services.

7.2 Related work

7.2.1 From assertion languages to software contracts
Our work builds on the foundations developed on assertion languages and software contracts. Pursuing Floyd’s work [18], Hoare first used pre and postconditions in a systematic way for programs proof [22]. Assertions have then been added to programming languages as annotations, like in ANNA [29] for Ada. Meyer made the next significant step by completely integrating executable assertions in the Eiffel language [31], taking into account class inheritance and proposing to organize class construction using contracts [30]. In this approach, contracts are implicit, both between developers using and implementing a class and between developers of an heir class and of its descendants. The contract is then a way to assign blame to developers and the specifica-

⁴**rely:** and **guarantee:** are the only specification clauses not supported yet.

tion with executable assertions is interpreted by a runtime mechanism to do so. Recent work by Findler and Felleisen [16] clarified this interpretation in class and interface hierarchies. Besides, the runtime checking of CCL-J specifications in *Contract* follows these rules.

Numerous annotation-based systems have also been proposed, such as Larch/C++ [9], iContract [26] and JML [28] for Java. As for Java, the reader may refer to [40] for an overview of available prototypes, compared by their integration means in a Java platform.

Besides, two different approaches can be distinguished among these work. Eiffel-like systems, such as iContract, enhance the underlying language to provide assertions whereas some others, such as Larch/C++, AsmL [3] or AAL [25], use a separate semantics to annotate the programming language. The Alloy Annotation Language (AAL) enables to describe behavior with a first-order logic of sets and relations and to statically analyze these descriptions. AsmL uses abstract programs that are interpreted in parallel with the code, whereas Larch/C++ annotates the code with algebraic specification. These systems all uses a principle of model-based specification, which can be more abstract and powerful than language-based specification but also more difficult to comprehend by the average developer. The Java Modeling Language (JML) [28] can be seen as a compromise between the two worlds. It mixes executable assertions with some abstract program features, enabling to build an executable model with an abstraction function on the specified class. Moreover, JML provides a lot of specific constructs to facilitate complete specifications, such as specification only declarations, making it an extensive experimental platform for object-oriented specification. Assertions have also been integrated in the UML notation with the Object Constraint Language (OCL) [46], with some support for runtime evaluation [12, 24].

Nevertheless, the common characteristics of all these work are to implicitly associate assertions with software contracts, while this is not a necessity, and contrary to *Contract*, to work on objects, classes or at best on component interfaces. The CCL-J language adapts the OCL syntax to the Java programming language and extends it to take into account the specification of hierarchical components. CCL-J is simpler than JML in terms of available constructs, but the current version of the language should be seen as a minimal but practical set for the validation of executable assertions language integrated in the *Contract* model.

7.2.2 Contracts on components

Work on programming by contracts have been adapted to components. Contracts on .NET assemblies have been proposed in [3], using AsmL as specification language (see above), but they only concern component interfaces.

The composition contract produced by *Contract* can be compared to collaboration contracts on objects proposed by Helm and Holland [21, 23]. The notion of views in the collaboration are similar to the role played by contract participants in *Contract*. Nevertheless, *Contract* makes a component carry the contract to take into account the possible hierarchy when the collaboration contract is a separate entity at the same

level as classes.

In the UML world, several work have proposed some forms of contract for UML components. In [39], contracts between service providers and service users are formulated based on abstractions of action and operation behaviour using the pre and postcondition technique. A refinement relation is provided among contracts but they only concerns peer to peer composition in this approach. In the same way, a graphical notation for contracting UML components is proposed in [47], focusing on expressing both functional (with OCL) and non functional (with QML [20]) contracts on component ports. Here again, only the connection of components is considered and checking means are not discussed. The forthcoming version 2 of the UML notation [38] supports a form of hierarchical components but contrary to *Fractal*, it focuses more on component connectors than on a composite structures. Moreover, version 2 of OCL [37] does not provide any extension to express compositional constraints. Consequently, it is likely to become quite cumbersome to express CCL-J like composition constraints with OCL.

Contracts have also been used in middleware systems, mainly to constrain non functional properties and quality of services. For example, the MAQS system [4] used a contract hierarchy to classify non functional contracts expressed with QML in order to manage priorities during negotiation. Contrary to CCL-J, non functional constraints in MAQS are very simple and cannot use functional properties or method calls.

7.3 Open issues

The *Contract* implementation has been designed to clearly separate the contracting system and the used specification formalisms. In the current version, this has only been validated by the CCL-J language. As only the integration of other formalisms will demonstrate the generality of the model, *Contract* will be handling TLA specifications in the near future, taking up existing work on *Fractal* [41].

The presented version of *Contract* model is only characterized by its operational semantics, but we focused on first delivering an experimental platform with some validating elements, such as the integration of CCL-J. We believe it is a necessary step to build a user community, get feedback with the integration of different formalisms and empirically study the practical value of our model.

As CCL-J is an executable assertions language, contracts are verified through runtime checking. Consequently, the compositional property linking external interfaces of a composite component and some interfaces of its sub-components cannot be deduced but only verified at runtime. This boils down to a weak form of compositional reasoning, but strong forms are only available, to our knowledge, for complete formal models [8]. Some forms of contracts have also been developed with formal models of peer-to-peer composition, to infer component types with provided and required sides [34], to ensure liveness and safety properties on components behavior [7].

The *Contract* model enables to construct contracts through a negotiation process, which would be necessary if the pro-

vided properties are weaker than the required ones. This negotiation is not developed in the current version, as it would have to take into account the full available context when adding a component into a composite structure or when connecting two components: functional (behavioral specification) and non functional (quality of service) aspects. In order to tackle this complex problem, we plan to build on existing work on e-negotiations [42, 5] and web services coupling [1].

8. CONCLUSION

Our work occurs within the context of specifying and checking hierarchical software components. In this context, deducing properties of an assembly from the properties of its parts is a key issue in Component-Based Software Engineering [2]. Our work aims at providing a form of compositional reasoning based on contracts. In this paper, we have presented *Confract*, a general contracting model for hierarchical components. This model first focuses on separating the contract from the specifications that are used in it. *Confract* enables to take into account conventional contracts on objects. The model also supports both interface contracts on interface binding and composition contracts on assembly. These composition contracts enable to define and verify constraints linking external interfaces of a composite component to interfaces of its subcomponents. Consequently, specifications become static entities provided by developers at configuration times, whereas contracts are runtime entities built according to compatibility rules. The *Confract* model then enables to handle elements of specification from different formalisms. Each kind of specification clearly defines a guarantor (a component) and some beneficiaries. For each kind of specification, a specific checker must be included in the *Confract* system by defining the times or periods the specification is expected to be satisfied. According to the kind of contract, *Confract* provides the appropriate checking context in order to verify each specification.

The *Confract* model is currently developed on the *Fractal* composition framework [6], which encompasses and enhanced conventional component platforms. However our work is likely to be applicable to other component models. The current version of the *Confract* model has been validated by the integration of an assertion language, named CCL-J, which is adapted to both objects and hierarchical components. This language adapts OCL to Java and adds specific constructs in order to support compositional specifications. The copier example presented in this paper has been completely implemented and is used as the main tutorial for the model. Other reference applications, such as a video player, are under finalization.

A formal semantics of the *Confract* model is not defined yet, as our main objective is to provide an experimental platform supporting compositional contracting for hierarchical components, in order to really integrate different formalisms implying different verification techniques (complete proof, static analysis, runtime checking). To our knowledge, our work is indeed unique in providing a contract framework for hierarchical components, in which contracts are runtime entities and can constrain both interfaces and composite structures.

Future work include the integration of other formalisms in the *Confract* system. We intend to first focus on a formalism to express synchronization constraints, in order to deduce in a compositional way liveness properties. Some extensions of the CCL-J language are also already planned to express architectural constraints on components. As stated in the paper, the *Confract* model is design to allow negotiation during binding and assembly but no such process is currently supported. Work on e-negotiation [42, 5, 1] are likely to provide a basis to develop a dedicated negotiation protocol. Finally, the generality of the model has to be validated by taking into account non functional properties on components [10].

9. ACKNOWLEDGMENTS

Thanks to Alain Ozanne for implementing the first *Confract* prototype and for profitable discussions on the integration of *Confract* in the *Fractal* platform. We would like to also thank Eric Bruneton, Thierry Coupaye and Nicolas Rivierre for their feedback on the *Confract* model.

10. REFERENCES

- [1] J.-M. Andreoli and S. Castellani. Negotiation as a generic component coordination primitive. In *4th intern. IFIP Working Conference on Distributed Applications and Interoperable Systems (DAIS'2003)*, Paris, November 17-21 2003. Also in techn. report num. 2003/064, Xerox Research Centre Europe, to appear in LNCS.
- [2] F. Bachman, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau. Technical concepts of component-based software engineering. Technical Report CMU/SEI-2000-TR-008, Carnegie Mellon Software Engineering Institute, May 2000. Volume 2.
- [3] M. Barnett and W. Schulte. Runtime verification of .net contracts. *Journal of Systems and Software*, 65(3):199–208, 2003.
- [4] C. Becker, K. Geihs, and J. Gramberg. Representation of quality of service preferences by contract hierarchies. Technical report, University of Stuttgart, Informatik, 1999.
- [5] M. Benyoucef and R. K. Keller. An evaluation of formalisms for negotiations in e-commerce. In *Distributed Communities on the Web, Third International Workshop, DCW 2000*, volume 1830 of *Lecture Notes in Computer Science*, pages 45–54. Springer, June 2000.
- [6] E. Bruneton, T. Coupaye, and J.-B. Stefani. The Fractal component model. Specification Draft 2.0-2, The ObjectWeb Consortium, September 2003.
- [7] C. Carrez, A. Fantechi, and E. Najm. Behavioural contracts for a sound composition of components. In H. König, M. Heiner, and A. Wolisz, editors, *23rd IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2003, IFIP TC 6/WG 6.1)*, volume 2767 of *Lecture Notes in Computer Science*, pages 111–126. Springer-Verlag, Berlin, Germany, Sept. 2003.

- [8] M. Charpentier and K. M. Chandy. Towards a compositional approach to the design and verification of distributed systems. In *FM'99 - Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 570–589. Springer Verlag, Sept. 1999.
- [9] Y. Cheon and G. T. Leavens. A quick overview of Larch/C++. *Journal of Object Oriented Programming*, 7(8):39–49, Oct. 1994.
- [10] P. Collet. Functional and non-functional contracts support for component-oriented programming. In *First OOPSLA Workshop on Language Mechanisms for Programming Software Components (OOPSLA'2001)*, page 3, October 2001.
- [11] P. Collet and R. Rousseau. Efficient implementation techniques for advanced assertion languages. *L'objet*, 5(3-4):417–442, Déc. 1999.
- [12] P. Collet and R. Rousseau. Towards efficient support for executing the object constraint language. In *International Conference on Technology of Object-Oriented Languages and Systems (Tools 30, USA'99)*. IEEE Computer Society Press (New York), August 1999.
- [13] P. Collet and R. Rousseau. Contrôle d'admission de composants avec des contrats comportementaux. In *LMO'2003 (Langages et Modèles à Objets)*, pages 31–44. Hermes Science Publications, *RSTI L'objet*, volume 9, numéro 1-2/2003, Jan. 2003. Aussi rapport I3S/RR-2002-49-FR.
- [14] L. G. DeMichiel, L. Ümit Yalçınalp, and S. Krishnan. Enterprise java beans specification - version 2.0. Technical report, Sun Microsystems Inc., August 2001.
- [15] D. F. D'Souza and A. C. Wills. *Object, Components and Frameworks with UML: The Catalysis Approach*. Addison-Wesley Publishing Co. (Reading, MA), 1998.
- [16] R. B. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *Proceedings of OOPSLA'2001*, 2001.
- [17] R. B. Findler, M. Latendresse, and M. Felleisen. Behavioral contracts and behavioral subtyping. In *Proceedings of Foundations of Software Engineering (FSE'2001)*, 2001.
- [18] R. W. Floyd. Assigning meanings to programs. In *Proceedings American Mathematical Society Symposium in Applied Mathematics*, volume 19, pages 19–31, 1967.
- [19] France-Telecom-R&D. Fractal web site, 2002. <http://fractal.objectweb.org>.
- [20] S. Frølund and J. Koistinen. Quality of service in distributed object systems design. In *4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS), Santa Fe (New Mexico)*. USENIX, April 27-30 1998.
- [21] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In N. Meyrowitz, editor, *OOPSLA/ECOOP'90*, pages 169–180, Ottawa, Canada, October 1990.
- [22] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, Oct. 1969.
- [23] I. M. Holland. Specifying reusable components using contracts. In *ECOOP'92*, volume 615 of *Lecture Notes in Computer Science*, pages 287–308. Springer Verlag, July 1992.
- [24] H. Hussmann, B. Demuth, and F. Finger. Modular architecture for a toolset supporting OCL. In *UML'2000 International Conference*, Oct. 2000.
- [25] S. Khurshid, D. Marinov, and D. Jackson. An analyzable annotation language. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 231–245. ACM Press, 2002.
- [26] R. Kramer. iContract - the Java design by contract tool. In *International Conference on Technology of Object-Oriented Languages and Systems (Tools 26, USA'98)*. IEEE Computer Society Press (New York), Aug. 1998.
- [27] L. Lamport. TLA+. Technical report, Digital Research, July 1995. <http://www.research.digital.com/SRC/personal/lamport/tla/tla.html>.
- [28] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999.
- [29] D. Luckham et al., editors. *Anna: A Language for Annotating Ada Programs*. Number 260 in *Lecture Notes in Computer Science*. Springer Verlag, 1987.
- [30] B. Meyer. Applying “design by contract”. *IEEE Computer*, 25(10):40–51, Oct. 1992.
- [31] B. Meyer. *Object-Oriented Software Construction*. The Object-Oriented Series. Prentice Hall Inc., 2nd edition, 1997.
- [32] B. Meyer. *Reusable Software: The Base Object-Oriented Libraries*. Prentice Hall Inc. (Englewood Cliffs, NJ), Apr. 1994.
- [33] Microsoft Corporation. .NET documentation. Technical report, <http://www.microsoft.com/net/>, 2001.
- [34] O. Nierstrasz. Contractual types. Technical Report IAM-03-004, Institut für Informatik, University of Bern, Switzerland, August 2003.
- [35] Objectweb consortium web site, 2000. <http://www.objectweb.org>.
- [36] OMG. CORBA component model. Technical Report ptc/2001-11-03, Object Management Group, November 2001.

- [37] OMG. UML 2 OCL final adopted specification. Technical Report ptc/03-10-14, Object Management Group, October 2003.
- [38] OMG. UML 2 superstructure final adopted specification. Technical Report ptc/03-08-02, Object Management Group, August 2003.
- [39] C. Pahl. Components, contracts, and connectors for the unified modelling language UML. In *FME 2001 - Formal Methods Europe*, volume 2021 of *Lecture Notes in Computer Science*, pages 259–277. Springer Verlag, 2001.
- [40] R. Plösch. Evaluation of assertion support for the java programming language. In *Journal of Object Technology, Special issue: TOOLS USA 2002 proceedings*, volume 1,3, pages 5–17, 2002. http://www.jot.fm/issues/issue_2002_08/article1.
- [41] N. Rivierre and T. Coupaye. Observing component behaviors with temporal logic. In *ECOOP 2003 Workshop on Correctness of Model-based Software Composition*, July 2003.
- [42] R. G. Smith. The contract net protocol : highlevel communication and control in a distributed problem solver. *IEEE Transactions on computers*, 29(12):1104–1113, December 1980.
- [43] M. Ströbel. *Engineering Electronic Negotiations Negotiation Technologies for the Design and Implementation Next-Generation Electronic Markets - Future Silkroads of eCommerce*. Kluwer, jan 2003.
- [44] C. Szyperski. *Component Software — Beyond Object-Oriented Programming*. Addison-Wesley Publishing Co. (Reading, MA), 1998.
- [45] University of Linz (Austria). *International Workshop on Component-Oriented Programming (WCOP'96)*, July 1996.
- [46] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison Wesley, 1999.
- [47] T. Weis, C. Becker, K. Geihs, and N. Plouzeau. A UML meta-model for contract aware components. In *UML 2001 - The Unified Modeling Language*, volume 2185 of *Lecture Notes in Computer Science*, pages 442–456. Springer Verlag, Oct. 2001.