On Contracting Different Behavioral Properties in Component-Based Systems

Philippe Collet
University of Nice - Sophia Antipolis, I3S Laboratory, France
philippe.collet@unice.fr

Alain Ozanne
France Telecom R&D Division
MAPS/AMS Laboratory
Issy les Moulineaux, France
alain.ozanne@francetelecom.com

Nicolas Rivierre
France Telecom R&D Division
MAPS/AMS Laboratory
Issy les Moulineaux, France
nicolas.rivierre@francetelecom.com

ABSTRACT

Using different specification formalisms together is necessary to leverage better reliability on component-based systems. The ConFrac system provides a contracting system for hierarchical software components, but currently only executable assertions are supported. In this paper, we describe how to integrate other kinds of formalism in ConFract. We propose a domain specific language and integration tools that enable designers to describe the observations needed to appropriately verify their specifications.

Categories and Subject Descriptors
D.2.4 [Software Engineering]: Software/Program Verification—formal methods, programming by contract

Keywords
component, contract, verification, TLA, assertion

1. INTRODUCTION

Software engineering is now concerned with more complex, dynamic, evolving and long-living systems. Recently, the concept of component has been revisited to provide a more adapted framework to master software complexity. The notion of contract is then part of the definition of software components [9], in order to organize the guarantee of properties all along the software life cycle. Combining different specification formalisms is desirable to leverage reliability on component-based systems, but this task is rather complex, as the very term behavioral is even differently interpreted according to approaches. Works on executable assertions provide behavioral contracts as state-oriented expressions before and after method calls [6]. On the other hand, works on static approaches [1, 5] describe the component behavior as protocols or interaction sequences and provide formal verifications using theorem proving or model checking.

ConFrac. In this context, the ConFrac system [3] provides a contracting model for the Fractal hierarchical components platform [2] and aims at combining different specification techniques. It refines three types of contracts, building them from specifications on components and interfaces. There are interface contracts, constraining interfaces on a binding, internal composition contracts, constraining interfaces of a composition, and external composition contracts, between a component exposed interfaces.

Requirements. Currently, according to its first integrated formalism, ConFrac makes verification by checking assertions in the context of the execution flow, using directly the computation data. But, as the means of specification may differ from a formalism to another, the contract system should embed different kinds of verification technique. Moreover, the integration of a formalism is concerned with the definition of appropriate observations on a system. Considering the temporal logic TLA [4], an instance of sequence-based formalism, specifications can be verified on a model or serve to check execution traces acquired from specific system observations [8].

In this paper, we thus propose a Domain Specific Language (DSL) that is dedicated to the definition of observations with associated verification methods, thus acting as a pivot model in the ConFrac system. This language provides a generic way to apply specifications of arbitrary formalisms to a system, but requires integration tools. We show how, for each formalism, the means to bind together the descriptions of observations, the specifications, and the relevant contract, are provided, and finally associated with the appropriate verification techniques.

2. DSL DESCRIPTION

Our observation DSL enables designers to define a set of rules. A rule describes where and when observations occur, what values they capture, and the verifications to be made.

Rule definition. A rule is defined according to the following syntax pattern:

```
On <a component>
  Observe (val: <some value> at: <some times> +)
  Verify <some properties>
```

The On block defines what spatial domain of the system is visible to the rule, that is a component scope. The Observe
block describes the observations operated in the scope. The `Verify` block describes the checking part, which can of course use the observed values. The `Observe` part contains a list of observations that are defined by the statements `val... at:...`. The `at` block gives the times at which the values described in the `val` block can be observed. Whereas the `On` blocks always refer to a component, the `val` part contains a sequence of functional expressions that can be evaluated in the `On` scope at defined times. Results may be bound to names if needed. Finally, the `Verify` property is a predicate that takes the `val` values as parameters.

**Event definition.** In order to define when observations should occur, some atomic observable events are provided. Inside a rule, events are considered as sets. Basic regular expressions enable designers to denote more easily sets of events that encompass several method calls or several interfaces. For example, the following definition:

```plaintext
at: entry * sns.*(*) , exit * sns.*(*)
```

considers the set of all the events determined by an entry or exit of any method on the `sns` interface, as commonly done in aspect-oriented systems.

Classic set operations are provided (union, intersection, etc.) and intervals can be defined as sets of all occurring events between two events. Open intervals can also be defined by referring to all events before or after another one. Moreover, events that are specific to components life cycle are also manipulable so that design or configuration events can be taken into account. For example, adding or removing a component to/from a composite one, binding and unbinding between two interfaces, starting or stopping a component. As the `Fractal` platform provides these control features through extensible interfaces [2], it is quite straightforward to be notified of these events.

It should be noted that a rule can refer to design time verifications that are done, for example, on an architectural description. Indeed, main architectural properties (containment, binding) can be translated into successive configuration events, thus enabling the checking system to trigger appropriate verifications.

### 3. INTEGRATION INTO CONFRACT

ConFract has been first used with an assertion language. Using the proposed DSL, the observations and associated verifications are now made explicit. In the specific case of assertions, `ConFract` generates the observation rules from the specifications (see figure 1). Verification is then simply the evaluation of the assertions.

The integration of TLA can be done through the description of the multiple observation points necessary to each specification, and the specification of the associated verification method, model or runtime checking [8]. Designers have to provide first the observation rules associated to the specifications. This is illustrated in figure 1 by the three input files on the right. The `ConFract` system then interprets the rules to operate the observations and their associated verifications. As the system does not impose any naming convention on the specifications, a descriptor must also be provided to bind rules terms to TLA specifications. In this contract descriptor, the type of contract and the concerned components are described so that responsibilities can be deduced in the `ConFract` system [3]. On the contrary, the verification methods (runtime and model checking in the case of TLA) are provided once and for all when the formalism support is integrated into `ConFract`.

### 4. CONCLUSION

The work presented here targets both design and runtime verifications. Based on a DSL, our framework supports the integration of different formalisms for contracting component behaviors. As an illustration, we have integrated both an assertion language and a temporal logic formalism, combining different specifications concerns. To validate our model, we plan to integrate other formalisms, well-suited to specify communication among components, such as the SOFA behavior protocols [7]. This latter is based on regular-like expressions, support specifications composition and refinement in components assemblies and can be checked at configuration and run times. Languages dedicated to specify extra-functional properties will also be considered.

### 5. REFERENCES


