Components, Objects, and Contracts

Olaf Owe
Department of Informatics
University of Oslo, Norway
olaf@ifi.uio.no

Gerardo Schneider
Department of Informatics
University of Oslo, Norway
gerardo@ifi.uio.no

Martin Steffen
Department of Informatics
University of Oslo, Norway
msteffen@ifi.uio.no

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software; D.1.3 [**Programming Techniques**]: Concurrent programming; D.1.5 [**Programming Techniques**]: Object-oriented programming; F.3.1 [**Logics and meanings of programs**]: Specifying and Verifying and Reasoning about Programs

Introduction

We propose to combine components with *deontic contracts,* i.e., agreements between two or more components on what they are *obliged, permitted* and *forbidden* when interacting. This way, contracts are modelled after legal contracts from conventional business or judicial arenas. Indeed, our work aims at a framework for *e-contracts*, i.e., "electronic" versions of legal documents describing the parties' respective duties. They go beyond standard behavioral interface descriptions, which typically describe sets of interaction traces. In particular, contracts, in the intended application domain, involve a *deontic* perspective, speaking about obligations, permissions and prohibitions, and also contain clauses on what is to happen in case the contract is not respected. This deontic aspect is typical for natural language legal contracts which we use as a starting point and which we aim to formalize.

The problem

We are concerned with finding a good programming and specification language, and appropriate abstractions for developing components in an integrated manner within the object-oriented paradigm. We are interested in enhancing components with more sophisticated structures than interfaces, targeted towards e-contracts. In that context, we address the following questions.

Design: How to develop components in a programming environment facilitating rapid prototyping and testing?

Composition and compatibility: How do we know that two or more components will not conflict with each other when put together?

Substitutability: How to guarantee that replacing a component will not introduce new unexpected behaviors?

Deontic specification: How to specify what a component is supposed to do, what it may do, and what it should not do?

Contract violation: How to react in case a component does what it is not supposed to do?

These issues are crucial in component-based software development and deployment. In fact, most of the questions, perhaps apart from the deontic aspect, are not new to the component-based software engineering community.

We propose a model combining the following ingredients: 1) As underlying object-oriented language, we use the concurrent language *Creol.* 2) As mentioned, we propose a notion of deontic contract, written in a *contract language.* 3) The contract is associated with the component model. 4) A *contract logic* allows static and dynamic reasoning on component consistency and conformance. 1In the rest of this paper, we sketch the four ingredients in turn.

Creol

Creol [2] as underlying language is motivated as follows:

Concurrency: It is a language for open, distributed systems, supporting concurrency and asynchronous method calls. The concurrency model is that of loosely coupled active objects with asynchronous communication.

Object-orientation: Creol is an object-oriented language, with late binding and multiple inheritance. It is strongly typed, supporting subtypes and sub-interfaces.

Interfaces: The language at the current state, already supports behavioral interfaces, based on assume-guarantee specifications. In particular, its notion of *co-interface* allows specification of required and provided interfaces.

Formal foundations: Creol has a formal operational semantics defined in rewriting logic. The core of the language has an operational semantics consisting of only 11 rewrite rules. This makes it easy to extend and modify the language and the semantics. We may reuse the operational semantics when formalizing the extension to components. Based on the formal semantics, the language comes with a simple reasoning system and composition rules. Creol has an executable interpreter defined in the Maude language and rewriting tool. This provides a useful test-bed for the implementation and testing of our component-based extension.

Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007), September 3-4, 2007, Cavtat near Dubrovnik, Croatia.
Copyright 2007 ACM ISBN 978-1-59593-721-6/07/0009 ...$5.00.

Contract language

Formally, we let component interface descriptions be based on the contract language $\mathcal{CL}$ developed in [5]. $\mathcal{CL}$ is a language tailored for electronic contracts (e-contracts) with formal semantics in an extension of the μ-calculus. The language follows an *out-to-do* approach, i.e. where obligations, permissions and prohibitions are applied to actions and not to state-of-affairs. The language avoids the main classical paradoxes of deontic logic and it is possible to express (conditional) obligations, permissions and prohibitions over concurrent actions keeping their intuitive meaning. Moreover, it is possible to represent (nested) CTDs (*contrary-to-duty*, i.e. what happens when an obligation is not fulfilled) and CTPs (*contrary-to-prohibitions*, i.e which action to be performed in case of violating a prohibition).

Components and Contracts

We list some of the main features of contracts in the context of component-based development and deployment. Contracts associated with components enhance behavioral interfaces and give the following added value:

1. If written in a formal language with formal semantics and proof system, a contract can be proved to be conflict-free, both by model checking and logical deduction techniques. The automatic checks can also reveal incompleteness in the specification, for instance it may indicate that no escalation is agreed upon in case one of the partners acts contrary to its contract.
2. The use of contracts may assist the developer during the development phase to check whether a component may enter into conflict with others, through a static analysis of contract compatibility. The appropriate notion of compatibility in the presence of obligations, permissions, and prohibitions needs to be developed.
3. A well-founded theory of contracts should provide the following kinds of analysis:
   - Determine whether a contract is *covered* by another one, i.e. a well-defined notion of sub-contract. This will help deciding whether a component may be replaced by another one in a safe manner.
   - Allow decisions on whether paying a penalty in case of one contract violation is beneficial or not when sub-contracting. Assume component A has a contract with component B where it is stipulated that A must "pay" x to B in case of contract violation. Suppose now that such violation depends on a service provided by C to A and that there is a contract between A and C stating that C must pay y to A in case of their own contract violation. Then a theory of contracts would allow A to determine whether it is good to compose with B. During the development phase this kind of information may help defining sub-contracting which are not against a component's own interest.
   - A negotiation phase could be added prior to the composition of two or more components. In this phase a contract could be negotiated before the final signature, as in the context of web services.
4. A run-time contract monitor will guarantee that the contract is respected, including the penalties and escalations in case of contract violation (CTDs and CTPs). We expect such a monitor could be extracted from the components contracts in a (semi-)automatic way.

Contract logic

The logical semantics of $\mathcal{CL}$ opens the way to use the logic proof system of μ-calculus, as well as existing model checkers. Initial work on model checking a contract has been presented in [4]. We propose to use the contract logic both during the component's development and deployment phase, using Creol as the development platform.

Development Phase During this phase our framework may be summarized as follows: a) *Development:* Each components has associated one or more contracts in the sense discussed above, i.e., specifying the obligations, permissions, and prohibitions in the component's interacting behavior. b) *Static Analysis:* Before deployment the contract is formally analyzed to guarantee it is contradiction free. This might be done by using a proof system or by model checking. Similarly, conformance between the component and its contract can be proved. c) *Testing/Simulation:* It is well known that static analysis techniques cannot validate every aspect of a system. Testing and simulation are thus needed to complement the above. Since Creol has formal semantics in rewriting logic we propose to use the Maude environment to simulate and test each component separately and its interaction with other components being developed.

Deployment Phase After the component is released there is still no complete guarantee of it being well suited for the yet unknown platform where it will be executed. We propose the following framework to increase confidence on the component's compatibility with its future environment. a) *Pre-execution Analysis:* Before adding a new component to an existing context where it will be composed with other components, the corresponding contracts are checked to guarantee their compatibility. If there are disagreements, a phase of negotiation may start, or the components is simply rejected. This phase may be considered as a kind of static analysis on the side of the execution platform. b) *Execution:* If the component is accepted after the analysis of the previous phase, then it is deployed. A contract monitor is launched to guarantee that the components behave according to the contracts. In case of contract violation, the monitor is responsible of taking the corresponding action as stipulated in the contract for such situation, or cancel the contract and disable the component.

Further details may be found in the accompanying technical report [3], representing the full version of the paper.

Acknowledgment. This work is partially supported by the Nordunet 3 project *Contract-Oriented Software Development for Internet Services* [1] and the EU-project *Credo*.

1. REFERENCES

[1] COSDIS. `www.ifi.uio.no/~gerardo/nordunet3`, 2007.

[2] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *TCS*, 365(1–2):23–66, Nov. 2006.

[3] O. Owe, G. Schneider, and M. Steffen. Components, objects, and contracts. Technical Report 363, Dept. of Informatics, Univ. of Oslo, Norway, August 2007.

[4] G. Pace, C. Prisacariu, and G. Schneider. Model checking contracts — a case study. In *ATVA'07*, volume 4762 of *LNCS*, pages 82–97, 2007.

[5] C. Prisacariu and G. Schneider. A formal language for electronic contracts. In *FMOODS'07*, volume 4468 of *LNCS*, pages 174–189, 2007.