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

ABSTRACT

Being a composite part of a larger system, a crucial feature of a component is its *interface*, as it describes the component's interaction with the rest of the system in an abstract manner. It is now commonly accepted that simple syntactic interfaces are not expressive enough for components, and the trend is towards *behavioral* interfaces.

We propose to go a step further and enhance components with *deontic contracts*, i.e., agreements between two or more components on what they are *obliged, permitted*, and *forbidden* to do when interacting. This way, contracts are modeled 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.

We take the object-oriented, concurrent programming language *Creol* as starting point and extend it with a notion of components. We then discuss a framework where components are accompanied by contracts and we sketch some ideas on how analysis of compatibility and compositionality could be done in such a setting.

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

General Terms

Verification

Keywords

Components, compositionality, contracts, interfaces, object-orientation, Creol, deontic logic

1. 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* to do when interacting. This way, contracts are modeled 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, allowing static and dynamic reasoning on component consistency and conformance, using a *contract logic*. In the following section we discuss some differences between objects and components. In Section 3 we clarify the notion of "contract" used on this paper. In Section 4 we sketch the three ingredients mentioned above, whereas in Section 5 we describe our proposed framework. We conclude in the last section.

2. COMPONENTS VS. OBJECTS

Even if there is no clear-cut definition of what exactly is a component, and what distinguishes the notion from a software module or just an object, we highlight here some essential differences between objects and components.

- Components are supposed to be self-contained units and independently deployable. This is not the case in general for objects, as they usually are not executable by themselves.

- If developed using the object-oriented paradigm, a component may contain many objects which are encapsulated and thus are not accessible from other components. If an object creates another object inside a component, this new object is not visible from the outside unless explicitly allowed by the interface. Objects in most languages do not have this feature.

- Components are static entities representing the main elements of the run-time structure, in contrast to objects, which are dynamic instantiations of classes. A purely class-oriented program does not identify the main elements of a system.[1]

In some sense the above may justify the definition of components as being *just* a collection of "circles" (objects) encapsulated inside a "box", which in turn could also be a kind of object typed by an interface. It is now accepted that such interfaces should not only take into account functional aspects but should take into account the history of interactions, or in other words be *behavioral*.

3. ON THE NOTION OF CONTRACTS

The term "contract" is understood in various ways by different research communities. We briefly recall some of its more common definitions or informal meanings.

1. *Conventional contracts* are legally binding documents, establishing the rights and obligations of different signatories, as in traditional judicial and commercial activities.

2. *Electronic contracts* are machine-oriented and may be written directly in a formal specification language, or translated from a conventional contract. The main feature is the inclusion of certain normative notions such as *obligations*, *permissions*, and *prohibitions*, be it directly or by representing them indirectly. In this context, the signatories of a contract may be objects, agents, web services, etc.

---

[1] However, early OO languages, including Simula and Beta, had a notion of block prefixing giving rise to static units which resemble components.

3. Some researchers informally understand contracts as *behavioral interfaces*, which specify the history of interactions between different agents (participants, objects, principals, entities, etc). The rights and obligations are thus determined by legal (sets of) traces.

4. The term "contract" is sometimes used for specifying the interaction between communicating entities (agents, objects, etc). It is common to talk then about a *contractual protocol*.

5. *Programming by contract* or *design by contract* is an influential methodology popularized first in the context of the object-oriented language Eiffel [6]. Contract here means a relation between pre- and post-conditions of routines, method calls, etc.

6. In the context of web services, "contracts" may be understood as a *service-level agreement* usually written in an XML-like language like IBM's Web Service Level Agreement (WSLA [10]).

We are mostly concerned with the first two meanings, though, to be able to reason and operate on contracts, it is natural to have the contracts written in a formal language, and thus the second meaning is more adequate. Obviously, the mentioned interpretations are not absolutely disjoint. The point we like to stress here is the importance of the mentioned normative aspects, which is very typical for (electronic) contracts capturing the spirit in which legal contracts are usually written. Besides those deontic aspects, electronic contacts in our sense also include behavioral aspects (making statements about the order of interactions at the interface), and may also relate the pre- and post-conditions of methods, as in point 5. But what is missing in usual interface and behavioral specifications are linguistic means to make the consequences explicit; e.g. what happens (or should happen) when the normative requirements are violated.

4. COMPONENTS, OBJECTS AND CONTRACTS

Creol

Creol is an object-oriented, concurrent programming and modeling language developed at the University of Oslo. For a deeper coverage of the language, its design and semantics, we refer to the Creol web pages [3] and to [4, 5]. The choice of Creol 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. This makes it an attractive basis for component-based systems.

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

Interfaces: Creol's notion of *co-interface* allows specification of required and provided interfaces. The language supports behavioral interfaces, based on assume-guarantee specifications expressed in terms of the communication history.

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.

Tool support: 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. The Maude tool may be used for simulation, model checking, and analysis.

Contract language

Formally, we let component interface descriptions be based on the contract language $\mathcal{CL}$ developed in [9]. $\mathcal{CL}$ 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
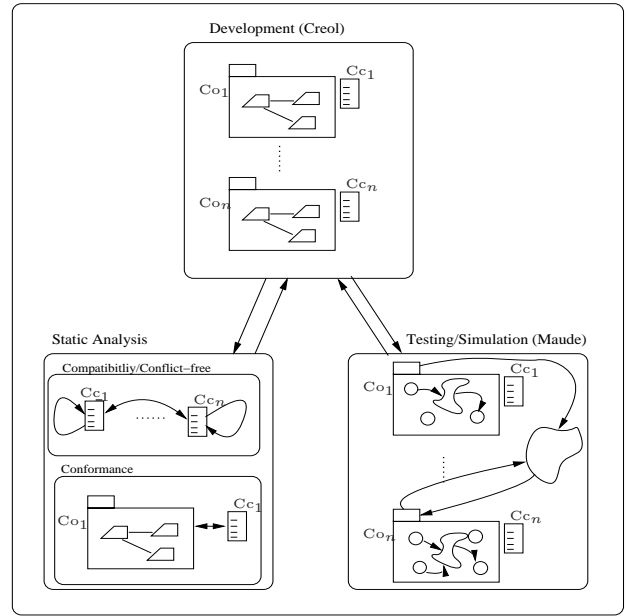

Figure 1: Development phase.

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.

5. PROPOSED FRAMEWORK

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 [8]. The combination of components, objects and contracts may be done as sketched in our proposed framework, involving both 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 (see Fig. 1):

Development: Each component 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.

Static Analysis: Before deployment, the contract is formally analyzed to guarantee that it is contradiction
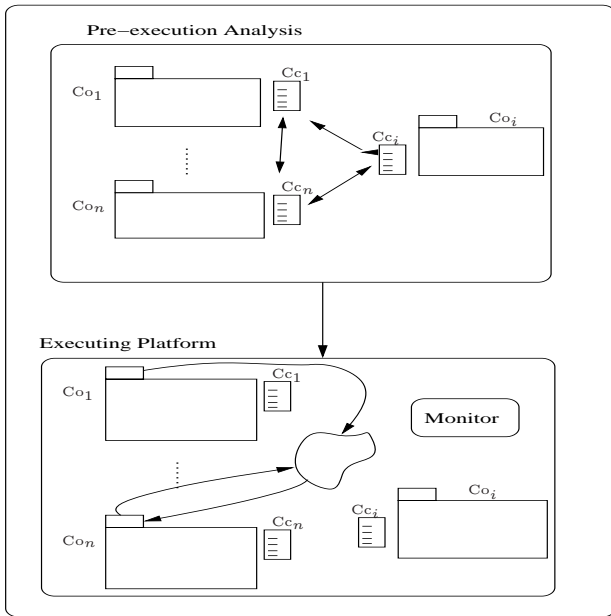
Figure 2: Deployment phase.

free. This might be done by using a proof system or by model checking. Static conformance between the component and its contract is also proved.

Testing/Simulation: Static analysis techniques cannot validate every aspect of a system. Testing and simulation are thus needed to complement the above. Since Creol has a 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. See Fig. 2.

Pre-execution Analysis: Before adding a new component to an existing context of other components, the corresponding contracts are checked to guarantee compatibility. If there are disagreements, a phase of negotiation may start, or the component is simply rejected. This phase may be considered as a kind of static analysis on the side of the execution platform.

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 must take the corresponding action as stipulated in the contract for such situation, or cancel the contract and disable the component.

6. FINAL DISCUSSION

In this paper we sketched how to enhance components with contracts as complementary to the latest ideas of using behavioral interfaces. In our opinion this approach would benefit from the fact that such contracts could be analyzed logically and model checked in order to find (local) inconsistencies, they could be negotiated and monitored. We believe component-based development and engineering will in some sense be reduced to the same kind of problems one finds in web services and other application domains where contracts are being studied.

The extension of Creol with primitives to define components is not difficult to do as most of the basic constructs are already defined in the language. For instance, contracts might be included as data-types in the language.

The successful use of contracts as we have proposed depends very much on the existence of a suitable formal contract language. We intend to further explore $\mathcal{CL}$ and its semantics to be used in this context. We expect to benefit from its formal semantics in the μ-calculus to further develop proof systems and to explore the possibility of use existing model checking tools.

Though we believe the first phase of the deployment phase could be achieved relatively easy, we are aware that obtaining a contract monitor, when executing a component could represent a big challenge if we intend to do so in real-time. We do not have a solution yet. A very interesting research direction would be to study how to combine meta-programming (e.g. in a reflective language) techniques with a formal (logical) framework for extracting a monitor from one or more contracts.

Related work and further details may be found in the accompanying technical report [7], representing the full version of the paper.

7. REFERENCES

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

[2] Credo. `www.cwi.nl/projects/credo/`, 2006.

[3] Creol. `www.ifi.uio.no/~creol`, 2007.

[4] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. In *Proc. 2nd Intl. Conf. on Software Engineering and Formal Methods (SEFM'04)*, pages 188–197. IEEE Computer Society Press, Sept. 2004.

[5] 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.

[6] B. Meyer. *Eiffel: The Language.* Prentice Hall, 1992.

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

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

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

[10] WSLA. `www.research.ibm.com/wsla/`.