UNIVERSITY OF OSLO
Department of Informatics

Components, Objects, and Contracts

Research Report No. 363

Olaf Owe

Gerardo Schneider

Martin Steffen

ISBN 82-7368-321-4
ISSN 0806-3036

August 2007

Components, Objects, and Contracts*

Olaf Owe, Gerardo Schneider, Martin Steffen
Dept. of Informatics, Univ. of Oslo
P.O. Box 1080 Blindern, N-0316 Oslo, Norway
E-mail: {olaf,gerardo,msteffen}@ifi.uio.no

August 2007

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, functional 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.

1 Introduction

Even without general agreement about what concretely constitutes a component, one thing is for sure: components are intended for composition.

*Partially supported by the Nordunet3 project *Contract-Oriented Software Development for Internet Services*, and the EU-project IST-33826 *Credo: Modeling and analysis of evolutionary structures for distributed services.*

Whence the central role of *interfaces* as an abstraction mechanism for hiding internal details. The interface description is the basis for composition both from a theoretical point of view — to semantically understand composition and to formally reason about components — as well as a practical concern — only when agreeing on well-defined interfaces, there is hope to separately develop and deploy software that works together.

But then again, within this general picture, there is a large design space what exactly constitutes a good interface abstraction, and the choice depends on the underlying language, the communication model, and the properties of interest. At any rate, it is now commonly accepted that simple, functional interfaces, listing the method signatures, are not expressive enough for components, and the trend is to define components together with *behavioral* interfaces.

We propose to go a step further and enhance components with *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. Contracts, in the intended application domain, involve a *deontic* perspective, speaking about obligations, permissions and obligations, also containing clauses on what is to happen in case the contract is not respected.

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.

Outline of the paper

The paper is organized as follows. In the next section we further elaborate on the motivation of our work. In Section 3 we describe the object-oriented programming (and modeling) language Creol. In Section 4 we expand on the definition and the use of contracts in other domains not necessarily related to components. Section 5 is the main part of this paper and consists on two parts. We first show how Creol may be extended to be used as a programming language for components, whereas in the second part we describe a framework that combines components, objects and contracts, and how contracts may be used to facilitate interaction of components. We conclude in the last section.

2 Motivation

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. However, the practice of component frameworks clearly distinguishes them. The conceptual confusion is perhaps due to some similarities, and also since both object-orientation and component-based technologies, each at their time, where both hailed with similar promises as the next silver bullet to solve the software crisis. For instance, both objects and components are usually typed by interfaces, and the services they offer are only accessible through such interfaces. Hence, both concepts provide abstraction in supporting encapsulation and hiding.

We highlight here some essential differences between objects and components. (1) Components are supposed to be self-contained units, and independently deployable. This is not the case in general with objects, as they are instances of a class and usually are not executable by themselves. (2) A component may (and in general will, if developed using the object-oriented paradigm) 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. (3) Components represent 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*.

In this paper we will discuss the relation between objects and components, by sketching how components could be defined in the object-oriented programming and modeling language Creol [8]. We will also present some ideas on the use of *contracts* as complement to behavioral interfaces to help the development and deployment of components, to guarantee among other things their correctness and compositionality.

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

2.1 The problem

We are concerned with finding a good programming language and appropriate abstractions for developing components in an integrated manner within the object-oriented paradigm. We are also 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?

Relevance to component-based software development

The issues mentioned above are crucial in a component-based software development and deployment. In fact, most of the questions in the previous section, apart from perhaps the deontic aspect, are not new to the component-based software engineering community who is trying to answer them in one or another way.

Towards a solution

We propose here to use *contracts* as a means to specify and to (partially) guarantee the safe coexistence of components. A contract in our setting is modeled after real contracts, as one might find in law or judicial arenas. In this sense, it is more than a *behavioral interface*[2] as it contain clauses about the *obligations*, *permissions* (or *rights*), and *prohibitions* of the signatories.

The basic idea is that components are deployed not only with its usual interface specification, but together with a deontic contract. To assure non-conflicting interaction between two components, their respective contracts must agree in well-defined ways as explained in more detail in Section 5.

[2]See for instance the series of OOPSLA workshops on behavioral interfaces for further information.

3 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 [8] and to [**?**, 20]. The choice of Creol as underlying language is motivated as follows:

Concurrency: Creol is a language for open, distributed systems, supporting concurrency and asynchronous method calls. The concurrency model therefore is that of loosely coupled active objects with asynchronous communication. This makes it an attractive basis for component-based systems. It has been argued also elsewhere, that an asynchronous communication model of entities, loosely coupled by message passing, is well-suited in such settings.

Object-orientation: Creol is an object-oriented, class-based language, with late binding and multiple inheritance, as well as user defined data types and functions. 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 at the current state already supports behavioral interfaces, based on assume-guarantee specifications expressed in terms of the communication history.

Formal foundations: A formal operational semantics, defined in rewriting logics, allows us to formalize the extension to components by reuse of the operational semantics. The core of the language has a small kernel with an operational semantics consisting of only 11 rewrite rules. This makes it easy to extend and modify the language and the semantics. 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.

4 Contracts

The term "contract" is becoming a buzzword, and different research communities understand it in various ways. 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.

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 [22]. 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 [35]).

We are mostly concerned with the first two meanings, though, as we will see later, 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. Of course, beside those deontic aspects, electronic contacts in our sense also include behavioral aspect (making statements about the order of interactions at the interface) or may 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; i.e. what happens (or should happen) when the normative requirements are violated.

5 Components, Objects, and Contracts

In this section we first propose a way to extend the object-oriented programming language Creol to deal with components. We then present 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 framework.

5.1 Creol as a Component-Based Language

The core Creol language is centered around classes and dynamically generated objects. A simple notion of component representing singleton entities encapsulating a subsystem, is obtained by the construct

component C **implements** list-of-interfaces
  body
end

where the body contain ordinary Creol declarations and code, including a number of attributes setting up the initial internal object structure. All internal structure is hidden, except from the communication primitives stated in the interfaces in the **implements** clause. In particular, objects generated by the structure inside a component are considered to be part of this internal structure. A component may be thought of as an abstraction of a subsystem where the implements clause defines visible events in the interaction with the environment. Communication to the component is done using the name of the component. Routing of incoming calls may be done automatically, and in principle non-deterministically, or, if desired, by explicit programming of the handling of the re-routing. The input and output events of a Creol component are method invocation and reply events. In order to obtain platform independent units one would need to lift the interaction model to the level of ports.

5.2 Components and Contracts

Before describing our proposed framework, we list some of the main features of contracts in the context of component-based development and deployment. Formal contracts associated with components complement 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 what it is specified in the contract.

2. The use of contracts may assist the developer during the development phase to check whether a component may enter into conflict with other components, 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 in case of sub-contracting. Assume component A has a contract with component B where it is stipulated that A must "pay" x (according to a certain notion of quantified penalty) to B in case of contract violation. Suppose now that such violation depends on a service/product 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 or not 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 previous to the composition of two or more components. In this phase a contract could be negotiated before final "signature", as in web services context.

4. A run-time contract monitor will guarantee that the contract is respected, including the penalties and escalations in case of contract violation. I.e., what should happen if one of the signatories acts "contrary to duty" or "contrary to permission" (abbreviated as CTD and CTP). We expect such a monitor could be extracted from the components contracts in a (semi-)automatic way, at least partially.

The above list already gives an idea on how we intend to combine contracts and components. Contracts may be used both at the development and deployment phase.

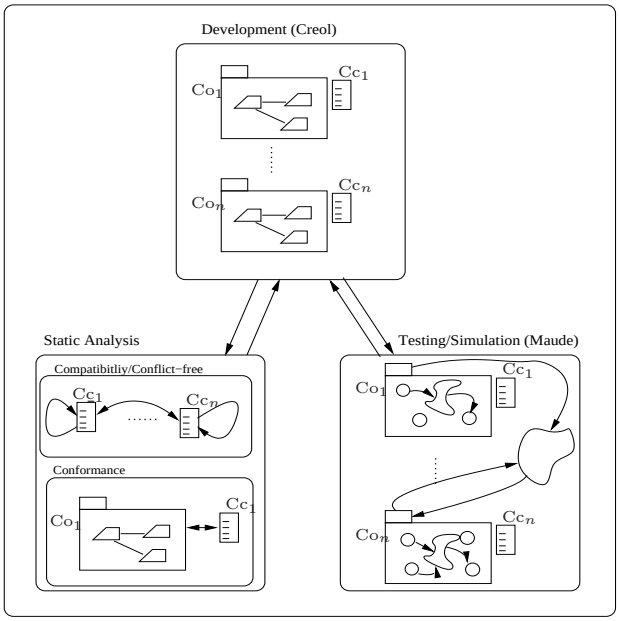

Figure 1: Development phase.

Development Phase During this phase our framework may be summarized as follows (see Fig. 1):

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. We propose Creol as a development platform.

Static Analysis: Before deployment, the contract is formally analyzed to guarantee that it is contradiction 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: 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 a formal semantics in rewriting logic and implemented in Maude, 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

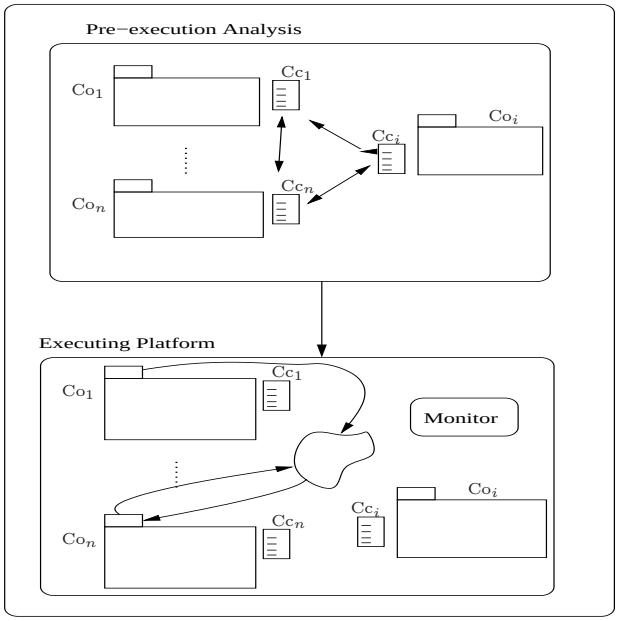

Figure 2: Deployment phase.

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 where it will be composed with 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 contract. 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.

6 Related work

Object-orientation Two main interaction models for distributed processes are remote method invocation (RMI) and message passing. RMI is the ap-

proach adopted by Java, and may lead to unnecessary waiting in a distributed setting. Moreover, Java's thread concept forces the programmer to choose between reduced parallelism (using the `synchronized` keyword) and shared-variable interference, and makes reasoning highly complex [2]. Mechanisms based on synchronous message passing also result in unnecessary delays [19]. Asynchronous message passing, as popularized by the actor model [4, 17], is very flexible but lacks the structure and discipline of object-oriented method calls. Moreover, actors have no direct notion of inheritance or hierarchy. In contrast, Creol objects are concurrent, each with its own virtual processor and internal process control, and communicate using synchronous or asynchronous, i.e., non-blocking method calls. This provides the efficiency of message passing systems, while keeping the structuring benefits of methods and object-oriented programming. A distinguishing feature of the language in this respect is also that in Creol, the client object, i.e., the caller, decides, whether to invoke the service asynchronously or whether to block. Furthermore, conditional processor release points provide a high-level synchronization mechanism that allow combination of active and reactive object behavior.

Components With the size of software systems ever increasing, there is no lack of proposals for component models, frameworks, and platforms, including various proposals within UML [7], Java component models, for instance EJB [32], and different component models put forward by Microsoft. As stressed above, composability and, as a consequence, the notion of interfaces are central. Furthermore finding the right level of abstraction is crucial, especially when developing a formal approach to components (cf. [21] for a collection of formal approaches to component-based software). A conceptually clear and elegant approach to get a grip on interface behavior is to consider the notion of being replaceable as a definitorial starting point: Two components, seen as syntactically *composable* units of a language, are considered equal, when they can replace each other with no observable difference. This corresponds to an observable, contextual perspective on equality as definitorial yardstick. As a black-box notion, it is appropriate for a component-based setting and has been employed for many languages and calculi, but obviously the definition leaves the *actual* interface description implicit. The task remains then to develop an explicit interface semantics, and ideally, to prove that it coincides with the implicit, contextually given one. That corresponds to the well-known problem of full abstraction.

In [31], for instance, such a semantics is developed in a class-based object-oriented setting, without an *explicit* notion of component. In other words, a

component is seen just as a set of classes, without linguistic support. The notion of observation is based on may- and must-testing [24]. Furthermore, the mode is based on a more tightly-coupled model of communication, namely that of multi-threaded Java. Currently [3], the communication model ported from multi-threading as in Java to a more loosely coupled model with asynchronous messages passing and active objects, corresponding to Creol. In particular, futures [12] and promises. With similar goals, [26] presents a behavioral interface semantics for a class-based object-oriented calculus, however without concurrency. The language, on the other hand, achieves a better modularization of the program. In particular, it curbs the unstructuredness of the heap by imposing ownership-structure. Another interaction semantics of components, in this case based on the actor model of concurrency, is presented in [33]. None of the mentioned approaches, however, is tailored towards deontic aspects, as aimed for in our setting.

Contracts Due to the great influence of the *design by contract* introduced by Bertrand Meyer and popularized first in the context of the object-oriented language Eiffel [22], we briefly discuss here some related works. Contract here means that every feature or method, created by the software developer (the *supplier*) starts with a precondition that must be satisfied by the software user (the *consumer*) of the routine. Moreover, each feature ends with post-conditions which the supplier guarantees to be true, if the preconditions were satisfied. The approach has been used for other languages, as well, for instance in the context of C$^{\#}$ language [13]. Relatively well-known here is the Spec$^{\#}$-language [5] and more recently Sing$^{\#}$[14] as extension of Spec$^{\#}$. Sing$^{\#}$, the core language of the Singularity operating system [18], is a type-safe, object-oriented language based on message-passing communication.

To use contracts in the context of component-based development and deployment as we have sketched in the previous sections we need to be able to write a contract in a formal language to be amenable to formal analysis, negotiation and monitoring.

There are currently several different approaches aiming at defining a formal language for contracts. Some works concentrate on the definition of contract taxonomies [1, 6, 34], while others look for formalizations based on logics (e.g. classical [11], modal [10], deontic [16, 25] and defeasible logic [15, 30]). Other formalizations are based on models of computation (e.g. FSMs [23] and Petri Nets [9]). None of the above has reached enough maturity as to be considered *the* solution to the problems of formal definition of contracts. Some provide a good framework for monitoring but lack a formal semantics and a reasoning system; others have nice proof systems and

model theory, but not mechanisms for monitoring or negotiation; many of the deontic-based approaches put too much emphasis on the logical properties and neglect the practical side, including monitoring. None of them captures all the intuitive properties of e-contracts we have described, while avoiding the most important paradoxes.

Since we intend to pursue our research by extending the contract language $\mathcal{CL}$ developed in [28], we describe the main features of this language in more detail. $\mathcal{CL}$ is a language tailored for electronic contracts (e-contracts) with formal semantics in μ^a-calculus, which is an extension of the μ-calculus with actions. A nice feature is that it opens the way to use the logic proof system, as well as existing model checkers. Since the μ^a-calculus is an action-based logic 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) obligation, permission and prohibition over concurrent actions keeping their intuitive meaning. Obligation of disjunctive and conjunctive actions is defined compositionally and it allows the representation of CTDs and CTPs. On the other hand, there is no mechanism for monitoring nor negotiation in the current state of development. No reasoning system is provided, though it seems quite straightforward to use the proof system as well as existing model checkers of the underlying μ-calculus. The approach is intended to be restricted to the context of e-contracts, so it is not practical for more general contracts, though we believe it can be used in the context of components. The underlying action algebra has been studied in [27] and initial works to show how to model check contracts has been presented in [29].

7 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. As mentioned in the related work section 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.

8 Acknowledgments

Marcel Kyas has contributed with valuable discussions about components.

References

[1] J. Aagedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, Dept. of Informatics, Faculty of Mathematics and Natural Sciences, University of Oslo, 2001.

[2] E. Ábrahám, F. S. de Boer, W.-P. de Roever, and M. Steffen. An assertion-based proof system for multithreaded Java. *Theoretical Comput. Sci.*, 331, 2005.

[3] E. Ábrahám, I. Grabe, A. Grüner, and M. Steffen. Abstract interface behavior of an object-oriented language with futures and promises. 2007. In preparation.

[4] G. A. Agha. *ACTORS: A Model of Concurrent Computation in Distibuted Systems*. MIT Press, 1986.

[5] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Proceedings of In CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

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

[7] J. Cheesman and J. Daniels. *UML Components*. Addison-Wesley, 2000.

[8] The Creol language. `http:www.ifi.uio.no/~creol`, 2007.

[9] A. Daskalopulu. Model Checking Contractual Protocols. In L. Breuker and Winkels, editors, *JURIX 2000*, Frontiers in Artificial Intelligence and Applications Series, pages 35–47. IOS Press, 2000.

[10] A. Daskalopulu and T. S. E. Maibaum. Towards Electronic Contract Performance. In *Legal Information Systems Applications, 12th International Conference and Workshop on Database and Expert Systems Applications*, pages 771–777. IEEE C.S. Press, 2001.

[11] H. Davulcu, M. Kifer, and I. V. Ramakrishnan. CTR-S: A Logic for Specifying Contracts in Semantic Web Services. In *Proceedings of WWW2004*, pages 144–153, May 2004.

[12] F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In R. de Nicola, editor, *Proceedings of Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Vienna, Austria.*, volume 4421 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[13] ECMA International Standardizing Information and Communication Systems. *C# Language Specification*, 2nd edition, Dec. 2002. Standard ECMA-334.

[14] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of EuroSys 2006, Leuven, Belgium*. ACM SIGOPS, 2006.

[15] G. Governatori. Representing business contracts in RuleML. *International Journal of Cooperative Information Systems*, 14:181–216, 2005.

[16] G. Governatori and A. Rotolo. Logic of violations: A Gentzen system for reasoning with contrary-to-duty obligations. *Australasian Journal of Logic*, 4:193–215, 2006.

[17] I. A. M. Gul A. Agha, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1), Jan. 1997.

[18] G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fähndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Traditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

[19] 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.

[20] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.

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

[22] G. T. Leavens and M. Sitaraman, editors. *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

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

[24] C. Molina-Jimenez, S. Shrivastava, E. Solaiman, and J. Warne. Runtime Monitoring and Enforcement of Electronic Contracts. *Electronic Commerce Research and Applications*, 3(2):108–125, 2004.

[25] R. D. Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Comput. Sci.*, 34:83–133, 1984.

[26] G. Pace, C. Prisacariu, and G. Schneider. Model checking contracts –a case study. In *5th International Symposium on Automated Technology for Verification and Analysis (ATVA'07)*, volume 4762 of *LNCS*, pages 82–97, Tokyo, Japan, October 2007. Springer-Verlag.

[27] A. Paschke, J. Dietrich, and K. Kuhla. A Logic Based SLA Management Framework. In *4th Semantic Web Conference (ISWC 2005)*, 2005.

[28] A. Poetzsch-Heffter and J. Schäfer. A representation-independent behavioral semantics for object-oriented components. In *9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, volume 4468 of *LNCS*, pages 157–173. Springer, 2007.

[29] C. Prisacariu and G. Schneider. An algebraic structure for the action-based contract language CL. 2007. Submitted.

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

[31] I. Song and G. Governatori. Nested rules in defeasible logic. In *RuleML*, volume 3791 of *LNCS*, pages 204–208, 2005.

[32] M. Steffen. *Object-Connectivity and Observability for Class-Based, Object-Oriented Languages*. Habilitation thesis, Technische Faktultät der Christian-Albrechts-Universität zu Kiel, 2006. submitted 4th. July, accepted 7. February 2007.

[33] Sun Microsystems Inc., USA. *JSR-220 Enterprise JavaBeans Specification*, version 3.0 edition, May 2006.

[34] C. L. Talcott. Interaction semantics for components of distributed systems. In *1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems, FMOODS'96*, 1996. `http://www-formal.stanford.edu/MT/96fmoods.ps.Z`.

[35] V. Tosic. On Comprehensive Contractual Descriptions of Web Services. In *IEEE International Conference on e-Technology, e-Commerce, and e-Service*, pages 444–449. IEEE, 2005.

[36] WSLA: Web Service Level Agreements. `www.research.ibm.com/wsla/`.