

# Towards Practical Support for Component-Based Software Development Using Formal Specification

-- Position Statement --

Heinrich Hussmann

Dresden University of Technology  
Department of Computer Science  
Email: h.hussmann@computer.org

## Abstract

Starting from an analysis of the situation of a software developer using pre-fabricated components, it is investigated in which form techniques and formalisms from the area of formal specification can provide practical aid in development. Several different dimensions of precise component specification are identified. While formal specifications can be helpful for several of these dimensions, it is argued that the most relevant application area may be a flexible mechanism for creating different but consistent views on a complex system. Ideas for concrete tool support based on the Object Constraint Language (OCL) are sketched.

## 1 Introduction

Building software from pre-fabricated components has been discussed since a long time (e.g. [McIlroy68, Cox90]) as the ultimate breakthrough towards industrial production of software. It took until the end of the 90s that a number of technologies have appeared, mainly based on object-oriented principles, which make the component-based construction of complex software practically viable. A good overview of the current state of the art is given in [Szyperski97]. The term “component” is used in several variants. For this paper, we are concentrating on those approaches to component-based software development, where (relatively small) components are designed and produced with the explicit and sole purpose of later composition to applications. Examples of important recent technologies of this kind are Java Beans [JavaBeans97], Delphi Components, Enterprise Java Beans [Monson-Haefel99], or CORBA components [CC99].

The normal working situation of an application developer in component-based programming is to place a number of components on a graphical design surface, to configure them by setting property values and to interconnect them by event mechanisms. Designing an application becomes very similar to drawing a diagram with a special software tool. Of course, the range of possible applications that can be produced in this way is limited by the possibilities for component configuration, i.e. the properties (parameters) of the pre-fabricated components and the foreseen support for creating events and reacting to events in a component.

It is relatively easy to deal with software components in this style as long as these components mirror concrete elements of the user interface (so-called *visual components*). So for the realisation of Graphical User Interfaces (GUIs), component-based approaches are already accepted as a standard technology and are supported by

many commercial software development tools (e.g. JBuilder, Delphi, NetBeans). As soon as non-visual components are used, developers often feel less at ease with the component-based approach. Components acting as an information source for visual components (e.g. access to a database) are still relatively easy to integrate (and frequently used in practice). For most other kinds of components, in particular so-called business components representing facts and procedures of real-life businesses, significant effort has to be spent on learning the concepts that are behind the given components. However, the technology is mature enough to cover also server-side components, including issues of persistence, transaction and security (e.g. in Enterprise Java Beans).

As a starting point for the further discussion, let us analyse the situation of a software developer using the above-mentioned new technologies. The following observations can be made:

- When a specific use case is to be realised, the developer has to find the appropriate components to achieve the desired effect. In a large and sophisticated component library, the effort for finding an appropriate component and learning its use is quite high, often comparable to the effort to be spent for a direct implementation of the functionality. Search engines for component libraries can provide some help here. However, tools can never really save the effort that has to be spent on learning the actual application domain of the components and the conceptual ideas behind the components being on offer. Moreover, once a potentially suitable component has been found, it is extremely important to get precise information on the preconditions under which one can assume the component or a specific method of a component to work properly.
- The resulting network of interconnected components is not easy to understand. There is no obvious overall structure of the application “program”. The design is created interactively, and is often produced in a series of iterations. The graphical tools are not sufficient for showing the logical structure: Showing all interconnections graphically tends to give the optical impression of a bowl of spaghetti. Not showing the interconnections makes it difficult to trace the dependencies among components.
- One of the main motivations for using components is an economic one. Based on component technology, the evolution of a market is expected where third-party software developers offer components to application developers. One of the crucial issues in this model is the contractual situation. A component somehow provides a contract stating that it delivers some functionality when the environment meets certain preconditions. But how can a software developer trust the quality of the software components he is buying and using, when these contracts are not made explicit?

As can be seen from the list above, the main problems in dealing with components lie in the adequate specification of components, in particular regarding their interaction. In the remainder of this text, this topic is discussed further. In section 2, it is claimed that formal specification techniques are a relevant technology for improving component technology. Section 3 contains seven theses on the way how some of the well-developed techniques from formal specification may be applied to bring practical aid for component-based software development. Section 4 sketches potential tool support following the ideas from section 3.

## 2 Formal Specification and Components

Extensive research in formal specification languages (i.e. specification languages providing a formal syntax with a mathematical semantics) has been carried out now for several decades. Interestingly, the effect of this research on practical software development has been low up to now. [Meyer97] argues that the main reason is that the effort spent on a precise specification does not pay back appropriately in the current market situation where software is produced with very high time pressure, relatively low quality standards and a very limited extent of reuse. He claims that the situation will change completely when component-based development is applied on a large scale.

- For component developers and component users, an urgent need exists to rely on precise contracts about the functionality of the software. To protect the component developers, software components are usually delivered as compiled code, so the source code cannot be used for clarifying issues about the detailed functionality. It is a clear strength of formal specification techniques to describe software as a “black box”, but with absolute precision. So it seems logical to have a formal specification of a component (system) as part of the contract.
- For component developers, high investments into quality of the components pay back, since they can expect high-quality components to sell in very large quantities. So it may be economically feasible to spend extra effort on formal specification and even verification.

These ideas have led to the formation of an informal group of persons interested in high-quality components, the “Trusted Components Initiative” [TCI]. However, compared to the fast development of software technology (in particular in object-oriented and component-based approaches), it is astonishing that only relatively little progress has been made in this area during the last three years.

Further attempts to establish a precise component specification language have been made also by people from a less theoretical background, for instance the “BOCA” (Business Object Component Architecture Initiative) [Digre98]. BOCA aimed at defining a standardised component definition language where the semantics of components can be precisely specified by making reference to an underlying semantic model of the problem. This initiative somehow was torn apart by competing forces in the OMG standardisation process (the BOCA approach was rejected by OMG in summer 1998).

So altogether one can state that precise specifications for components have not yet taken up a role as a driving technology, despite of the potentially good perspective. One reason for the relatively slow progress may lie in the “fastly moving scenario” which is referred to in the motto of this workshop. The main reason for using components is to shorten development time and to ease application adaptation, but without introducing new (potentially time-consuming) technologies for quality assurance.

So it seems that simply specifying the behaviour of components with existing formal specification approaches is too naive an approach for being applied in practice. Below, we try to discuss more sophisticated and restricted ways how formal specification can be of practical help, keeping the “fastly moving scenario” in mind.

## 3 Seven Theses on the Practical Usefulness of Formal Specification for Components

In the following, a number of theses are put together regarding the potential of formal specification techniques for component specification.

### **3.1 Thesis 1: Components cannot be considered in isolation.**

At a first look, it may seem logical to consider a component as an entity that is completely described by its (formally specified) interface. However, components tend to be tied together in specific groups quite closely. As an example from GUI components (like Java Swing), MenuBars are closely tied to MenuItems, Separators and other components. In business objects, an order processing component has to know about products and customers, which are most probably dealt with in other components. So when talking about precise specification of components, one is always talking about joint specification of a group of components (a *component-based application framework*).

The idea that was used in the BOCA approach seems to be logical here: A specification of a component system consists of a specification of the semantic domain plus individual contracts for components using the terms of the semantic domain.

### **3.2 Thesis 2: Full formal specification of business domain semantics is too much.**

Based on the idea of a semantic domain specification, it seems to be necessary to fully specify the application domain of a group of specified components. However, this turns out as an extremely complex task, which is not needed in all circumstances. So for instance, specifying the exact behaviour of GUI components requires extensive descriptions of the geometry of windows. However, many of the operations are just straightforward in their semantics, like an action being called when a button is pushed. There is no doubt about the required functionality here, so formal specification of these aspects would not enhance the quality of component description. A similar situation arises in business objects, where many rules and algorithms, e.g. for accounting, are defined as standards of the business domain, so the only contract for the component needed is to refer to these (mostly informal) standards.

When considering the evolution of reuse in programs for a specific business domain, there an increase in abstraction levels can be observed (see e.g. [Pree 97]). In the beginning, solutions for individual problems of the application domain are implemented (and specified). In a later stage, libraries for frequently needed functions are designed, leading e.g. to object-oriented application frameworks. When such a framework is mature enough, it is possible to design a group of components for this application domain in such a way that its adaptation to individual problems does not require any programming language code anymore. This means that the components are introduced at a level where the application domain is very well understood and where the components themselves mirror very closely the concepts of the application domain. So a relatively small semantic gap remains between the business domain and the component framework. The need for a formal specification on this level disappears mostly, since the concepts are thoroughly understood by the contract partners. There is no need for instance in components targeted at accounting systems to formally specify algorithms for computation with interest rates, since the relevant algorithms are well known to everybody working in this area and can just be referenced by using the appropriate term used in the business domain.

As explained above, there are two strong forces that prohibit the *full* formal specification of a business domain: the sheer effort required to do so, and the limited usefulness in a well-known business domain. However, *some aspects* of a system, in particular co-ordination aspects, require formal specification, see below. For practical application, it seems in most cases sufficient to give a semi-formal specification of the

full business domain. This means to define e.g. UML class diagrams plus a number of informal explanations. On such a basis, it is also possible to specify selected important conditions formally, which have to be enforced as part of the contract. Examples are invariants regarding security conditions or mission-critical consistency conditions. So formal rigour can be applied just to a few selected aspects of the domain.

### **3.3 Thesis 3: Specification of execution semantics is too much.**

In the literature, some approaches can be found which map component architectures and concrete components into formal specifications. The focus is here on concurrent processes and event processing. This kind of work is of course very important in order to develop an overall understanding of the semantic concepts in various programming paradigms. Nevertheless, it is not directly helpful for the work of a component developer or a software developer using components. These developers usually have a quite clear understanding of the operational semantics of the component architecture they are working in. The problem is not so much to define the semantics of the component architecture but to understand the co-ordination and co-operation issues in a given component configuration.

### **3.4 Thesis 4: Critical component co-ordination issues require precise specifications.**

The last two theses left the impression that there is not much to be achieved by formal specification techniques, which is of practical relevance. However, the demand for precise specifications still exists even when we rule out the aspects mentioned in the preceding two theses. When configuring components to an application, there are many questions of the following kinds:

Questions to be answered about the static component configuration:

- In order to make use of component X, which other components have to exist and how are they to be connected to X?
- When configuring the properties of the component X, which rules have to be obeyed in order to keep the component working properly? Which dependencies exist to property values of other (connected) components?

Questions to be answered at design time, but related to effects appearing at run time:

- In order to invoke some operation of a component, which preconditions have to be ensured?
- When having executed an operation of a component, which changes to the preconditions of other operations can be inferred?

In general, questions about the right *configuration* of components are important for practical usage of components. So a contract for a component may be rather sloppy on the specification of the actual algorithms contained in the component (as long as these algorithms are well known and do not need further explanation). But a contract has to be very explicit about the constraints that have to be obeyed in the configuration of components and in interconnecting the components.

More generally speaking, there are different aspects of a specification for a business domain. One aspect is the detailed description of all details and functions, which was considered less important in thesis 2 above. Another aspect is the system of rules governing the co-ordination and configuration of the components. Component co-

ordination languages are also subject of recent related research work (see e.g.[MS00] from this workshop).

It is important to see that precise specification of configuration and co-ordination of course requires *some* degree of precise specification of the business domain and of the component runtime semantics (which looks like a contradiction to theses 2 and 3 above). However, it is sufficient to provide these specifications on a much simpler abstraction level where much of the detail information is left out. This abstraction saves time and effort, and contributes to economic viability.

### **3.5 Thesis 5: Non-functional requirements are important.**

The arguments above were essentially restricted to the functional requirements for a software system. In fact, there are several other dimensions of requirements that have to be taken into account in specification of software components.

The purely functional aspect of component specification (which effect is achieved when I invoke some operation?) can be separated into two aspects, as it was indicated above in thesis 2: “Functional essence” of the application domain, which is in many cases of limited relevance, and selected, particularly critical aspects.

Besides the functional aspects, several *non-functional* aspects are relevant for the practical usability of a component:

- What are the resources needed by the component (e.g. memory, hardware/software platform)?
- What is the (average, maximum) response time of the component to an input event?
- Which error rate is guaranteed for the component (based on a classification of error types)?
- Which security/confidentiality properties are guaranteed by the component (e.g. encryption and authentication in communication with other components)?

The above-mentioned aspects of a component belong to a practically useful component specification. The specification has the form of a rely/guarantee contract, i.e. depending on the non-functional properties of the container a component is allocated to. A close integration with the (critical) functional aspects is possible (e.g. performance or confidentiality dependent in individual parts of the functionality).

In the remainder of this text, we will not stress further the issues of non-functional requirements, in order to keep to a limited scope.

### **3.6 Thesis 6: Formal specifications may provide a flexible mechanism for browsing/viewing component configurations.**

When considering the situation of a developer who is configuring a concrete set of components to create an application, and assuming that the components are specified along the lines sketched above, the following opportunities appear:

- Some of the formally specified conditions on component configurations can be checked directly (e.g. presence of some required other components), and feedback can be given immediately to the developer.
- For many other conditions, general consistency rules (like runtime preconditions and invariants) can be instantiated according to the current component configuration. If a formal language with a precise logical semantics is used, the

instantiated rules can be simplified automatically. These more specific versions of the rules are easier to understand for a developer.

- Since components are interconnected in a complex way, the investigation e.g. of a precondition of an operation may span over several components. So the instantiation, composition and simplification of formal component specifications leads to a dynamic creation of textual explanations for the current component configuration, based on some developer-given query. The idea is here to provide a “semantics-directed browser” of the current component configuration.

### **3.7 Thesis 7: Formal specifications shall be interpreted by machines and still be readable for human beings.**

This last thesis is of a very general nature. As it can be seen from the relatively well-functioning system of laws and justice, natural language is in most cases sufficient for establishing contracts. The advantage of using a mathematically precise specification language for contracts is that the specification becomes machine-processable. So it is possible to monitor at runtime whether the contract is fulfilled, it is possible to use sophisticated browsers as mentioned above and it should be possible to automatically create appropriate tests whether a component fulfils its contract.

## **4 Ideas for Tool Support**

### **4.1 Tool Functionality**

Basically, advanced tool support based on formal specifications should be integrated into support tools that follow the state of the art of component-based development. This means that components are represented graphically and special browsers (inspectors) are used to deal with the formal specification of a component in the same way as normal properties are adjusted. So the formal specifications are stored as local parts of the component and packaged together with the component.

Typical examples for formal component constraints in the sense of the discussion above may be, in the context of an order processing system:

- An *OrderProcessing* component assumes that its local properties *customerManagement* and *productManagement* are set to defined components (of the correct type).
- The *CustomerManagement* component can only deal with *Customer* components which have a *customerNumber* and *customerStatus* attribute.
- A precondition for the local method *createOrder* of the *OrderProcessing* component is that the *customerManagement* has checked the respective customer status (whether he/she pays his /her bills, for instance).

These examples show that besides constraints rooted in the business domain, there are also many constraints that are of a more syntactical nature and therefore can be checked mechanically. A support tool can help the developer in many different ways here:

- By documenting and maintaining the constraints (which helps to understand the configuration rules);
- By instantiating the constraints according to the current configuration of component instances and applying static simplifications; (For instance, the constraint that the *customerManagement* property is defined can be removed as soon as it is fulfilled in the current configuration.);
- As a generalisation of the above-mentioned functionality, by providing a flexible browser for the interdependencies among the components;

- By providing an intelligent help function in resolving open issues regarding the component configuration; (e.g. providing a checklist of unresolved constraints);
- By compiling those constraints, which cannot be resolved statically, into dynamic runtime checks.

So the overall functionality envisaged here is not related to verification or any other advanced use of logical calculi. Instead, the focus is on such constraints, which can mechanically be evaluated during the application design and testing process. According to the theses from above, it is this kind of support which is most helpful for the developer.

## **4.2 Specification Language**

For the design of support tools, an important question is the concrete choice of the specification language. A language is needed with the following properties:

- easy to understand and learn for software developers;
- well integrated with the object-oriented paradigm underlying component technology;
- compatible with object-oriented business domain models, described e.g. in UML;
- compatible with the emerging Component IDL for CORBA;
- applicable in the level of business models as well as on the level of meta-models (for description of component configurations);
- fully machine-executable.

A suitable starting point for the choice of a language with these properties is the Object Constraint Language (OCL) [WK99], which is part of the UML. OCL is able to express various kinds of constraints on objects, including constraints on meta-level (using a reflection mechanism). OCL is explicitly designed for fully automatic mechanical checking of the constraints. First tools exist which evaluate OCL expressions at object configuration time and at runtime [HDF00, RG00].

## **5 Conclusion and Outlook**

In this position statement, it has been argued that there is a clear need for precise semantic component specification. However, a specific approach has been suggested which carefully distinguishes between separate aspects of component specification. The suggested approach does not aim at a general specification of the application domain for a set of components, but tries to give support for understanding the complex interactions of components. It has been stressed that non-functional properties are at least as relevant as functional properties for components.

Some of the ideas mentioned in this paper are currently explored further in research projects at Dresden University of Technology. So an initiative exists for investigating non-functional aspects of components in a co-operation of several computer science disciplines. Moreover, first building blocks for tool support using OCL exist already, in the form of an OCL parser, typechecker and OCL-to-Java compiler [DOCL00]

*Acknowledgement:* I would like to thank Klaus Bergner, 4Soft München, for interesting email discussions.

## References

- [CC99] OMG, CORBA Components, Joint revised submission, OMG TC document orbos/99-02-05, 1999
- [Cox90] B. Cox, Planning the software industrial revolution. *IEEE Software* (7)6, November 1990.
- [Digre98] T. Digre, Business component architecture, *IEEE Software* (15)5, September/October 1998, 60-69.
- [DOCL00] F. Finger, B. Demuth, H. Hussmann, Dresden OCL Compiler, see <http://www-st.inf.tu-dresden.de/ocl>
- [HDF00] H. Hussmann; B. Demuth, F. Finger: Modular Architecture for a Toolset Supporting OCL, to appear in Proceedings <<UML>>2000, Conference, October 3-6, 2000, York, UK
- [JavaBeans97] Sun Microsystems, JavaBeans specification, <http://java.sun.com/beans/doc/spec.htm>
- [McIlroy68] M. D. McIlroy, Mass produced software components, Proc. Nato Software Eng. Conf., Garmisch, Germany (1968) 138-155.
- [Meyer97] B. Meyer, The next software breakthrough, *IEEE Software* (30)7, July 1997, 113-114.
- [Monson-Haefel99] R. Monson-Haefel, Enterprise JavaBeans, O'Reilly 1999.
- [MS00] C. Montangero, L. Semini, Specification and composition of software components: formal methods meet standards, Proc. Monterey Workshop 2000, part of this volume.
- [Pree 97] W. Pree, Komponentenbasierte Softwareentwicklung mit Frameworks, dpunkt 1997.
- [RG00] M. Richters, M. Gogolla, Validating UML Models and OCL Constraints, to appear in Proceedings <<UML>>2000, Conference, October 3-6, 2000, York, UK.
- [Szyperski97] C. Szyperski, Component software, Addison-Wesley 1997.
- [TCI] The Trusted Components Initiative, see <http://www.trusted-components.org>
- [WK99] J. Warmer, A. Kleppe, The Object Constraint Language, Addison-Wesley 1999.