Formal Methods for Component Software: The Refinement Calculus Perspective

Martin Büchi

Turku Center for Computer Science,
Lemminkäisenkatu 14A, 20520 Turku, Finland, mbuechi@abo.fi

Emil Sekerinski

Åbo Akademi University, Department of Computer Science,
Lemminkäisenkatu 14A, 20520 Turku, Finland, esekerin@abo.fi

Turku Centre for Computer Science
TUCS Technical Report No ?
August 15, 1997
ISBN ?
ISSN 1239-1891

Abstract

We exhibit the benefits of using formal methods for constructing and documenting component software. Formal specifications provide concise and complete descriptions of black-box components and, herewith, pave the way for full encapsulation. Specifications using abstract statements scale up better than pre-postconditions and allow for 'relative' specifications because they may refer to other components. Nondeterminism in specifications permits enhancements and alternate implementations. A formally verifiable refinement relationship between specification and implementation of a component ensures compliance with the published specification. Unambiguous and complete contracts are the foundation of any component market.

1 Introduction

The separation of specifications/interfaces and implementations of components is a prerequisite for the establishment of component software. It alleviates the necessity to distribute source code, thereby protects the implementation know-how and avoids overspecification. Overspecification basically prohibits future enhancements and alternate implementations. Furthermore, separate specifications enable the component integrator to understand the functionality without having to examine the source code.

The lack of easily and quickly understandable, concise, and complete specifications is the chief reason, why the advantages of the separation between specifications and implementations are not commonly exploited. Most current interface-description languages (IDLs) are limited to expressing syntactical aspects such as number, names, and types of parameters only. For example, the C library routines *strcpy* and *strcat* have the same interface (signature) and are, therefore, up to their name — a vague hint at the functionality — indistinguishable. The IDLs completely ignore the behavior of components which is usually given as incomplete, ambiguous, partly overspecific, and often outdated textual description[1]. Incompleteness forces the component integrator to derive additional properties by trial and error which might be invalidated in future versions of the component. Ambiguity often remains undetected in an informal setting and causes mysterious errors; if noticed, forces the integrator to make unfounded assumptions. Overspecification unnecessarily restricts future enhancements. Incompleteness, ambiguity, and overspecification hinder alternate implementations — the ground stone of any component market. Without the possibility of automated consistency verification, informal descriptions are rarely kept up to date. Due to this deficiencies, programmers request components as source code. Overwhelmed with implementation details — in addition to the aforementioned problems of this approach —, they then often choose to reinvent the wheel rather than reuse the badly specified existing components.

Formal specifications can solve these problems. The creator of a component can test, whether based solely on the specification the component may be appropriately used. Ambiguities can be detected by consistency proofs. Overspecification can more easily be detected in a concise formal language. Formal verification, here in the form of refinement proofs, guarantees that the implementation actually behaves as specified. Furthermore, a specification which is created before the component is implemented, can facilitate a structured development and, thereby, create more general, robust and efficient components and often also helps to save costs. "When quality is pursued, productivity follows," K. Fujino, as quoted in [18]. For a recent overview of formal methods success stories see [8, 16].

The adaptation of formal specifications has been slow because of difficult nota-

[1] We denote by interface the syntactical aspects only and by contract both the interface and the behavioral specification.

tions which differ too much from implementation languages and lack of tool support, but also due to ignorance and prejudice. We aim for a lightweight approach to formal methods, based on a rigorous and general formalism, but without an overwhelmingly rich expressiveness of the language, modeling, and analysis in favor of lower costs in terms of time, know-how, and money. We deviate from the classic dogmatic view on formal methods, without abandoning the foundations.

Section 2 makes a plea for formal specifications as contracts, Sect. 3 shows why nondeterminism is also relevant for practitioners. Refinement between specifications and implementations to ensure compliance and refinement between different versions of a specifications are the topics of Sect. 4. Section 5 points to related work and Sect. 6 draws the conclusions.

2 A Plea for Formal Contracts

The buyer of a microprocessor or another chip usually requests a detailed description in form of a data sheet and, quite commonly, also an executable specification in form of a VHDL program. These descriptions form a contract, if a sale takes place. They describe all relevant information for deployment, such as form factor, voltage, and signal delay in a standard way, that does not require interpretation or understanding of the physics required to build the chip. Contrast this with the typical description of an ActiveX software component: Incomplete plain textual descriptions augmented with a formal part that merely describes the number and types of parameters. The customer has no possibility to verify in advance whether the desired part meets the requirements. He often spends hours of trial and error to find out how the component must be used. He relies on testing of a few cases as the only way to gain confidence. Nobody and nothing guarantee that he uses only functionality which will continue to exist in future versions. Hence, there is an urgent need for better contracts! A good contract is clear, complete, and concise. A bad contract is ambiguous, misses important points, lays down irrelevant details, and is unnecessarily long. That current contracts, respectively in our terminology interfaces with textual addition, are too weak has also been acknowledged at WCOP'96 [26]. The lack of standardized contracts for software components is due to the high degree of freedom compared to hardware, the immaturity of the field, the difficulties in automated verification, and the — partly unnecessary — complexity and ill-definedness of common programming languages, which further complicates verification.

Jézéquel and Meyer [11] recently argued that the crash of the Ariane 5 [2] was due to a reuse specification error. A poorly documented limitation in a component originally designed for the Ariane 4 with different physical requirements caused the error. Jézéquel and Meyer conclude, that reuse without a contract is sheer folly. Yet, contracts are the most important non-practice in component software. Clearly, white-box components do not solve these problems for large systems as they overwhelm the designer with details, rather than providing suitable abstractions.

	Obligations	Benefits
Client	Ensure precond.	Assume postcond.
Server	Ensure postcond.	Assume precond.

Figure 1: Obligations and benefits from pre- and postconditions

A simple and popular form of contracts is that of assumptions, called preconditions, and promises, called postconditions. If a component is used in a correct manner, it has to satisfy its contract, i.e. establish the promised postcondition. If, however, its preconditions are not met, it has no obligations whatsoever and is free to behave arbitrarily (Fig. 1). If a system consisting of several components fails do perform the requested job, the failure can be attributed either to a component not fulfilling its contract or to the integration, i.e., even if all components behave as specified the whole system does not perform correctly. Without formal contracts, locating such errors is much more involved.

Pre- and postconditions that are only checked at runtime help to locate errors, but do not prevent them as static analysis does. A program can still fail at a customer's site with input values which have not been tested. Programmers annotate their programs with pre-and postconditions in order to make them more reliable. During test runs, they are checked at run-time, but for the production version, these checks are often disabled for efficiency reasons. This is like having lifeboats on a ship for a test cruise but getting rid of them for a transatlantic cruise with passengers because of the additional load. Static analysis, on the other hand, allows only the removal of lifeboats which will provably never be used. Because of the deficiencies of run-time only checking, programmers are not inclined to use specifications at all.

Pre- and postconditions being predicates, they cannot contain calls to other methods, except pure functions. This means that using pre- and postconditions one has to reinvent the wheel afresh for each method, rather than being able to build upon other specifications. Specifications in form of abstract statements are not affected by this scalability problem. Consider the partial specification of component Buffer using abstract statements:

```
component Buffer ...
  b : set of Item
  print(d: Device) = for all x in b do d.print(x) end
end Buffer
```

If we were to specify the same component using pre- and postconditions, we would have to expand the definition of the base type Device's print method incurring a number of disadvantages. The specification of how print ultimately sets the pixels on a device would be rather lengthy and not of our interest here. We loose the information that a method of d is invoked. Reasoning about the program, we cannot use the knowledge that d is of (behavioral) subtype of Device with a more deterministic specification. Pre-postcondition specifications contradict encapsulation and specialization.

Specifications by abstract statements come close to contracts as proposed by Helm et al. [10]. Contracts of Helm et al. specify "behavioral dependencies" between objects in terms of method calls and other constructs. Contracts are expressed in a special purpose language and then have to be linked to the underlying programming language. By contrast, we like to see abstract statements as a moderate extension of the underlying programming language for expressing contracts.

Changes to the specification of the print method, e.g. improved version decreasing nondeterminism, are not automatically reflected in the specification of Buffer. Pre-postconditions do not support 'relative' specifications in the sense of relying on previous specifications. The loss of self-containedness of abstract statement specifications can easily be compensated by a specification browser supporting in place expansion or hypertext-like facilities. Abstract statements also lend themselves to grey-box specifications, which reveal parts of the internals, such as call-sequences [6].

The process of writing a formal specification often leads to more generally useful, easier to integrate, and longer-lived components. Rough edges, special cases, and anomalies resulting from implementation difficulties and lack of overview during implementation can often be detected and eliminated by a specification.

For example, the above specification of Device states that for all elements in b, the method print is called in an arbitrary order. No element is printed twice, since a set contains an element at most once. If this is desired, we should have used a bag (multiset) rather than a set. If we like that the elements are printed always in the same order, we should have used a sequence rather than a set and an iteration in print. The specification also states that printing an empty buffer is a no–op rather than an error.

By writing the specification, or at least parts of it, before starting the implementation, one can benefit from the commonly advocated advantages of a structured development process [22]. Cliff Jones [13] argues that formalism employed to justify early data structure and design decisions which helps avoid the most costly errors is the most important application of formal methods. Specifications written after the implementation tend to lay down irrelevant details of the specific implementation and lack the desired degree of abstraction. However, as with frameworks, an iterative approach is usually preferred over the waterfall model in order to get very general components. The number of iterations can be reduced by formal specifications which help to eliminate implementation quirks.

For some reasons forced to use a component that comes without a formal contract, it might even be worthwhile to write a specification of it as it is perceived and used. Such a 'contract assumption' can greatly simplify testing and reduce the time to evaluate the suitability of new and alternate versions of the component.

A component should not only formally specify its own contract, but also the (minimal) contracts of its required components. A calendar component might require a database component which satisfies a certain contract [3, 27]. The component integrator can choose such a component, or — in a more dynamic scenario — the calendar component can 'shop' for the desired component at runtime. Formal specifications of required and existing components simplify also the creation of wrappers/adaptors.

3 Nondeterminism: Avoiding Overspecification

Nondeterminism is an approach to deliberately leave a decision open, to abandon the exact predictability of future states. As such, nondeterminism appears to be neither commonly desirable nor is it used in implementation languages. On the other hand, nondeterminism is a fundamental tool for specifications to avoid laying down unnecessary details[2].

A nondeterministic specification leaves more choice for the implementation, which can be used for optimizations. Even if this degree of freedom is not used in the envisaged first implementation, it greatly increases the likelihood that future enhancements and alternate implementations can be made compliant with the specification. The earlier specification of the component Buffer is an example.

Nondeterminism often enhances the comprehensibility of specifications because the reader does not have to wonder why something has to be exactly in a certain way, when other choices would be as good. Many things are actually nondeterministic and should be acknowledged and specified as such.

Nondeterminism from an outside perspective often stems form information hiding, where the actual implementation is deterministic. A SQL database query without any sorting options returns an arbitrarily sorted list of records; a square root function returns an arbitrary value satisfying the specified precision. Both implementations are deterministic, but the outcomes are determined by hidden state components and implementation details.

Nondeterminism can also be present in the implementation. This occurs typically when the implementation is concurrent, for example by distributed objects. Given only the specification, it does not matter for a user of the component if and when the nondeterminism is resolved. This way, a large variety of implementations become valid refinement of the specification.

We can also interpret nondeterminism as 'free will' of a component which can in no way be influenced from the outside. Writing a combined specification consisting of existing components and a custom 'glue component' which we implement ourselves, we have to distinguish between two forms of nondeterminism. Nondeterminism within existing components which is beyond our control, called demonic nondeterminism, and nondeterminism in our custom component which we can control in our favor, called angelic nondeterminism. In this sense, we can

[2]Nondeterministic constructs would leave more choices for compiler optimizations also. This could be beneficial especially in the case of portable executable formats and just-in-time compilers, where static optimizations for a specific architecture are not applicable.

consider program execution as a game, the rules of which are given by the specification [5]. Demonic choices are moves made by an opponent (the existing component), and angelic choices are our moves. The combined specification is correct, if we can make moves such that we can achieve the desired goal, no matter what the opponent does. Hence, such a combined specification can help to decide whether a given component is suitable to solve a certain task.

4 Refinement: Ensuring Compliance with Specification

Employing formal specifications, we want to make sure that the implementation actually complies with its specification or, more precisely, is a refinement thereof [5]. A statement T refines a statement S, if considering the observable input-output behaviour, the output of T for a given input would be possible with S as well. Taking the possibility of nondeterminism into account, we formally define that $S \sqsubseteq T$ (S is refined by T) as

$$S \sqsubseteq T \stackrel{\text{def}}{=} \forall q.wp(S,q) \subseteq wp(T,q)$$

where q is a set of states and $wp(X,q)$ denotes the weakest precondition of q with respect to statement X, i.e., the set of states from which X is guaranteed to terminate in a state of q. Refinement is reflexive, transitive, and antisymmetric [5].

A component D is a refinement of component C, if the observations we can can make when using D would be possible with C as well. Let $S[C]$ be a statement using component C. Formally, we define $C \sqsubseteq D$ (C is refined by D) by:

$$C \sqsubseteq D \stackrel{\text{def}}{=} \forall S.S[C] \sqsubseteq S[D]$$

If an implementation of a component constitutes a refinement of its specification, any client designed according to the specification will work with the implementation. This is exactly what is implied by the above definition. On the other hand, two implementations of a component which are both refinements of the same specification might not be in a refinement relationship to each other (Fig. 2 a). Hence, it is important that clients only rely on properties guaranteed by the specification. Testing cannot uncover reliance on unspecified implementation features, only formal analysis can.

Nierstrasz and Tsichritzis [23] remark that encapsulation is violated if clients of a software component must be aware of implementation details not specified in the contract in order to make correct use of a component. In particular, if changes in the implementation that respect the original contract may affect clients adversely, then encapsulation is violated. Clients which rely only on properties specified in the contract never fall into this trap. However, a contract may be too weak so that the corresponding components can not be used intelligently.

Unfortunately, fully automatic proving of refinements similar to type checking performed by a compiler does not seem to be feasible in the foreseeable future. Because of this, contracts are approximated by interfaces, guaranteeing only syntactic

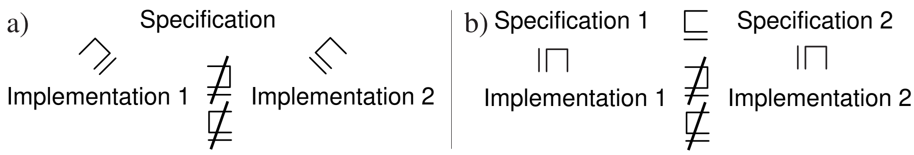

Figure 2: Refinement relationships

compatibility. The B-Toolkit [4] and Atelier-B [25], two environments for the B method [1], show, however, that semiautomated refinement proofs are feasible for industrial applications. Automatic proofs of refinement are an absolute necessity for components which at runtime shop for other components satisfying a required contract.

Instead of a full refinement proof, automatic test data generation from the specification, approximate reasoning with upper and lower bounds, finite state analysis with binary decision diagrams based on equivalence classes, and relative debugging can be employed to gain confidence in the correctness and to isolate parts meriting formal proofs [7, 12, 9]. In relative debugging, an alternative to a full refinement proof stemming from the field of scientific computing [21], the specification and the implementation are executed in lockstep and their states are compared after each statement according to a mapping relation.

If we want to better a component, we have to incorporate the improvements into the specification. Otherwise, they cannot be used by any clients which are only allowed to rely on features guaranteed by the specification. An advanced component should also be usable by an old client, which knows nothing about the improvements. If the new specification is a refinement of the old specification and the new implementation refines the new specification, then by transitivity of refinement it also refines the old specification. Hence, refinement is also an important relationship between different versions of a component's contract. However, an implementation of the new specification does not necessarily refine an implementation of the old specification (Fig. 2 b). In a world where progress is acceptable only if it is compatible with the current state, refinement is a crucial relationship.

We have not addressed performance, but the separation of specifications and implementations, which are refinements thereof, permits the replacement of inefficient components by efficient components which adhere to the same specification without any modification to the clients. It also allows for several components providing the same functionality which are tuned for different usages, e.g., set implementations as arrays and as lists. We could even specify an advanced set component as a refinement of the basic set component which provides a meta level method for controlling performance-relevant mapping decisions [14]. An implementation of the advanced set component might package the two set implementations into a single component and select the actual data structure according to the preferences set through the meta method.

5 Related Work

The Interface Specification Language (ISL), a pure extension of the CORBA IDL, developed in the Component-Based Software Engineering project at CSTaR Software Engineering Lab supports the descriptions of pre-/postconditions of operations, invariants and protocols of interfaces [15]. ISL opts for featurism, including multiple inheritance across components, rather than simplicity as we promote it. ISL is a sublanguage of the Architecture Specification Language, which also includes the Glue Specification Language and the Configuration Specification Language.

Participants in the Composable Software Systems project at Carnegie Mellon try to specify not only functional behavior of components, but also reliability, performance, and security aspects [24]. They call concepts similar to refinement behavioral subtyping for classes [17] and matching for components [27].

Bertrand Meyer propagates design by contract for component software, albeit of a less formal nature [19, 20]. The Object Systems Group at the University of Geneva under Nierstrasz and Tsichritzis have researched a number of applications of semi-formal methods to component software [23].

6 Conclusions

We have argued that only formal contracts paired with refinement can guarantee full encapsulation of software components, which is the base for improved and alternate implementations. Formal contracts lead to a more structured development, more orthogonal and, hence, longer-lived and more generally useful components, often at a lower cost.

Nondeterminism is a necessity for providing freedom of implementation. Refinement guarantees that implementations adhere to their specifications and that new versions are plug-compatible. Abstract statements do not have the scalability problems of pre-postcondition specifications because they allow for external calls.

Formal methods are needed to compensate the loss of the closed-world assumption and the impossibility to test a component in all possible environments. They are, however, no universal panacea nor is their application very simple, but we regard them as a necessity for the establishment of component software.

The paper presented at the workshop (http://www.abo.fi/~mbuechi/) also contains sections on the specification of invariants and temporal properties and on the design of specification languages, which are left out of this version.

Acknowledgments We would like to thank the referees for helpful comments, Wolfgang Weck and Ralph Back for a number of fruitful discussions and inspiring ideas, and Lars Nielsen and Marcel Mettler for their help with this paper.

References

[1] J. R. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] European Space Agency. Ariane 5: Flight 501 failure, July 1996. http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html.

[3] Ásgeir Ólafsson and Doug Bryan. On the need for "required interfaces" of components. In M. Mühlhaeuser, editor, *Special Issues in Object-Oriented Programming*, pages 159–165. dpunkt Verlag Heidelberg, 1997. ISBN 3-920993-67-5.

[4] B-Core. *B-Toolkit*. England, 1995.

[5] R. J. R. Back and Joackim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer Verlag, to appear 1997.

[6] Martin Büchi and Wolfgang Weck. A plea for grey-box components. In *Foundations of Component-Based Systems '97*, 1997. http://www.abo.fi/˜mbuechi/.

[7] Marsha Chechik and John Gannon. Automatic analysis of consistency between requirements and designs. In *Tech. Rep. CS-TR-3394.1, University of Maryland, College Park*, 1995. http://www.cs.utoronto.ca/˜chechik/.

[8] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(10), December 1996.

[9] Craig A. Damon, Daniel Jackson, and Somesh Jha. Faster checking of software specifications by eliminating isomorphs. In *Proceedings ACM Conference on Principles of Programming Languages*, 1996. http://www.cs.cmu.edu/afs/cs.cmu.edu/project/nitpick/www/home.html.

[10] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *Proceedings of OOPSLA/ECOOP '90 Conference on Object-Oriented Programming Systems, Languages and Application*, pages 169–180, October 1990.

[11] Jean-Marc Jézéquel and Bertrand Meyer. Put it in the contract: The lessons of ariane. *IEEE Computer*, pages 129–130, January 1997.

[12] Craig A. Damon, Somesh Jha, and Daniel Jackson. Checking relational specifications with binary decision diagrams. In *Proc. Foundations of Software Engineering '96*, 1996. http://www.cs.cmu.edu/afs/cs.cmu.edu/project/nitpick/www/home.html.

[13] Cliff B. Jones. A rigorous approach to formal methods. *IEEE Computer*, pages 20–21, April 1996.

[14] Gregor Kiczales. Why are black boxes so hard to reuse. In *Proceeding of OOPSLA'94*, 1994. http://www.parc.xerox.com/spl/projects/oi/towards-talk/transcript.html.

[15] W. Kozaczynski and J. O. Ning. Concern-driven design for a specification language. In *Proceedings of the 8th International Workshop on Software Specification and Design*, Berlin, Germany, March 1996.

[16] WWW Virtual Library. Formal methods. http://www.comlab.ox.ac.uk/ archive/formal-methods.html.

[17] Barbara H. Liskov and Jeanette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6), November 1994.

[18] Carlo Ghezzi, Dino Mandrioli and Mehdi Jazayeri. *Software Engineering*. Prentice Hall, 1991.

[19] Bertrand Meyer. Applying 'design by contract'. *IEEE Computer*, 25(10):40–51, October 1992. See also http://www.eiffel.com/doc/manuals/ technology/contract/index.html.

[20] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[21] David Abramson, Ian Foster, John Michalakes, and Rok Sosic. Relative debugging: A new methodology for debugging scientific applications. *Communications of the ACM*, 39(11):69–77, November 1996.

[22] Caroll C. Morgan. *Programming from Specifications*. Prentice Hall, 1990.

[23] Oscar Nierstrasz and Dennis Tsichritzis, editors. *Object-Oriented Software Composition*. Prentice Hall, 1995.

[24] David Garlan, Daniel Jackson, Mary Shaw, and Jeannette Wing. Composable software systems, 1996. http://www.cs.cmu.edu/˜Compose/ and ftp://ftp.cs.cmu.edu/project/compose/brief.ps.

[25] Stéria Méditerranée. *Atelier-B*. France, 1996.

[26] Clemens A. Szyperski and Cuno Pfister. Component-oriented programming: WCOP'96 workshop report. In M. Mühlhaeuser, editor, *Special Issues in Object-Oriented Programming*, pages 127–130. dpunkt Verlag Heidelberg, 1997. ISBN 3-920993-67-5.

[27] Amy M. Zaremsky and Jeannette M. Wing. Specification matching of software components. In *SIGSOFT Foundations of Software Engineering*, October 1995. Also CMU-CS-95-127.

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.abo.fi

University of Turku
- **Department of Mathematical Sciences**

Åbo Akademi University
- **Department of Computer Science**
- **Institute for Advanced Management Systems Research**

Turku School of Economics and Business Administration
- **Institute of Information Systems Science**