A Plea for Grey-Box Components

Martin Büchi and Wolfgang Weck

Turku Centre for Computer Science, Lemminkäisenkatu 14A, FIN-20520 Turku
Martin.Buechi@abo.fi, Wolfgang.Weck@abo.fi

Turku Centre for Computer Science
TUCS Technical Report No 122
August 1997

ISBN 952-12-0047-2
ISSN 1239-1891

Abstract

Interface Description Languages (IDLs) describe the syntactic part of a component's interface, but they do not help to specify semantics. Additional informal descriptions or pre- and postconditions of operations are often not precise enough. For instance, they cannot properly describe call-back scenarios, that is, the states, at which external calls are made, and the sequence of calls. Revealing the full implementation, that is, the source code, on the other hand, overspecifies the component and, thus, prohibits alternative implementations and future enhancements. What is needed is a language that allows to reveal as much of an implementation as is required to use a component, but not more.

Such a language can be found in the theory of program refinement, but this is rarely used with commercial software because of the human factor: the notation is very symbolic instead of resembling a known implementation language. The lack of tool support for simulating specifications and for automatic refinement proofs also limits the enticement of writing specifications.

Keywords: semantic specification of components, grey-box components, IDLs, component specification languages, abstract statements; presented at the Workshop on Foundations of Component-Based Systems, Zürich, September 26, 1997

TUCS Research Group
Programming Methodology Research Group

1 Introduction & Background

It is an old hope to be able to compose large software systems from prefabricated components, the same way as in other engineering disciplines products are assembled from third-party components. The expected advantages are well-known. Among them are a high degree of reuse to cut down production costs, high flexibility by choosing from different implementations, better adaptability by replacing individual components, reduction of the overall complexity of a system by breaking it down into smaller parts, and many more.

From the analogy to other engineering disciplines we can learn about a very important property components need to have: they need to provide abstractions, or specifications. Deployers of a component want to know what it can do for them and how to use it, but not how it is constructed internally. For instance, computer manufacturers use integrated circuits and compose them according to data sheets, but they do not want to be bothered by all the physics one has to understand to design the integrated circuit itself.

Similar abstraction is needed for software components. If component composition is supposed to be substantially easier than building systems from scratch, components have to be easier to deploy than they are to construct or to understand in full. Further, it has to be abstracted from certain implementation details, to make components replaceable. When one constructs an ActiveX control, one wants it to work in any container. Thus, it must be specified, how to interact with any container, independently of which implementation is used in the end.

On the other hand, abstraction must not go too far. As said in [Weg97], it is insufficient to state a component's input-output behavior. One needs to specify what longer, complex, interactions yield. Wegner speaks about marriage contracts opposed to business contracts.

We have ourselves experienced such problems with insufficiently specified components. A particularly precisely documented component environment, which we and our students use, is the Oberon/F framework [Obe95]. Though Oberon/F component interfaces are well documented with pre- and postconditions, important things are left unclear. As we will see, this is not a matter of poor specification, but a principal limit of the black-box component approach. The problems we experienced become even bigger with only informally specified environments, such as JavaBeans [Sun97] or Microsoft's OLE [Mic97]. When we created reusable or extensible components ourselves (for example in [Wec96]), we also recognized how difficult it is, to describe the essentials of our implementation in terms of relations between input and output only.

Our experience with formal methods, in particular with Ralph Back's refinement calculus [BvW97, BvW90], provided us with some insights, how such specifications could be provided in a form that is easier to write, better understandable, and more complete. The main idea is to provide an operational description of services, rather than a purely descriptional one. This allows us to reveal part of the implementation, one could say in a coarse form. Such specifications can be very detailed where needed, but they can be abstract enough where overspecification is to be avoided. This partial disclosure of implementation forms a middle ground between black-box components, of which only pre- and postconditions are accessible, and white-box components, showing all details. We therefore propose the term *grey-box components*.

Section 2 explains why we need a middle ground between black-box and white-box components. In Sect. 3 we argue that specification languages should be close to implementation languages in order to widen their acceptance. Given the requirements for component specification, we make a brief survey of different specification approaches in Sect. 4 and state why in our opinion abstract statements are the most promising foundation. Section 5 points to related work, Sect. 6 details our future plans, and Sect. 7 sums up and provides the conclusions.

2 We Need Grey-Box Components

2.1 Black-Box Specifications Are Insufficient

The term *black box* is a metaphor for a specific kind of abstraction. Black-box abstraction means, that none of the internal workings are visible, and that one can only observe output as reaction to some specific input (Fig. 1). Black-box testing, for instance, works this way. Test cases are selected without knowledge about the implementation. They are run, and the delivered results are checked for correctness.

Figure 1: Black-box component

For black-box specification of software components, pre- and postconditions are adequate. They describe the circumstances under which a component can be activated and the result of this activation.

Unfortunately, black-box specifications are insufficient for more interactive components. The result of an operation may depend not only on the input, but also on results of operations external to the specified component, which are called during execution (Fig. 2).

The external operation, activated by the component, needs to be specified, too. This specification is not needed to see how the operation is to be applied, but to see how it needs to be implemented. It is a specification of duties, sometimes referred to as required interfaces [AOB97]. Often, such an external operation will depend on the state of the component calling it. As an example of this, in [Szy97] the Observer Pattern from [JV95] is analyzed. To perform its task, the observer needs

Figure 2: Black-box component, which makes an external call

to request state information from the observed object. If the observed object is a text and the observer a text view component, the observer needs to know whether it sees the old or the new version of the text. In other words, one needs to know, whether the observer is called before or after the data is changed. Specifying the text manipulation operations, such as delete, as black boxes does not reveal when the observer is called and what the state of the caller is at that time (Fig. 3). In this simple example, the intermediate state of the black-box at the time of the call could be described verbally, but in more involved cases this approach usually fails.

One may think, that the above problems occur only, if intermediate state of a black-box operation can be observed, but this not true. If an operation involves more than one external call, the order of these may be important when implementing the called components. Consider the copying of a view with a separate model in the model-view-controller pattern: Has the model already been copied when the view is asked to copy its own state?

class TextModel;
  var text: **seq char**; observers: **set** Observer •
  ...
  procedure Delete (pos: **N**) =
    pre
      $0 \leq$ pos $<$ #text
    post
      text' = text[0...pos-1] + text[pos + 1...#text - 1]
    rem
      o.DeleteNotification(pos) is called for all o $\in$ observers
end

class Observer;
  var text: TextModel •
  ...
  procedure DeleteNotification (pos: **N**) =
    post
      text^' = text^
end

Figure 3: Black-box specification of observer pattern

Sometimes, programmers will try to retrieve information in addition to the specification by experimenting with the component [PS94]. Of course, such behavioral assumptions might not actually hold and depending on them ruins the possibility of later substitution. However, black-box descriptions often force this additional information gathering to be able to use a component.

2.2 Components Should not Be White Boxes

The opposite of a true black box would be a *white box*, which reveals all details (Fig. 4). With a white box, all information to select a component and connect it to a given environment is available. The deployer of a white-box component can gain arbitrarily detailed knowledge about the component's internals. This knowledge can be used to decide the components applicability in a specific situation, but it can also be used when creating a context for it. If the implementation is the only available interface description, the usage of that complete information cannot be restricted in any sensible way.

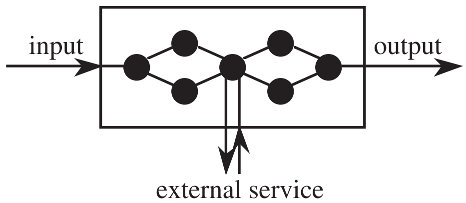

Figure 4: White-box component

Why should such a restriction be necessary? To answer this question, we need to reconsider why we are interested in components at all. One reason may be to use the same snippets of program text over and over again. Indeed, if this is the only goal white boxes do the job. For an industrial style of component usage, however, this is not satisfying. There, components should reduce the complexity of larger systems and they should help to reduce the product's cost. The former requires that a component can be deployed without understanding the full complexity of its implementation, whereas the latter requires, besides others, that components can be selected from a free market and that they can be replaced individually, for instance to adapt to new requirements or to upgrade to new technology.

For both problems, abstraction from the implementation is the solution. Hiding away the details allows the reader to understand the essentials faster. A specification saying in one line that some data is sorted is much faster to grasp than the full implementation of, say, Insertionsort. Reconsidering the observer pattern, the black-box specification given in Fig. 3 is much shorter and simpler to understand than its white-box counterpart, the implementation, which e.g., uses text pieces.

Also, details that are not known by the deployer can be changed later without requiring the deploying system to be changed. For example, it may become interesting to replace the aforementioned implementation using Insertionsort by one based on Heapsort. If the full implementation would be made public before, some client may depend on the fact that elements with equal keys are not reordered. If this observation may be used by clients, Heapsort would not be a valid implementation. Thus, excluding the detail how elements with equal keys are treated from the specification results in higher flexibility.

This flexibility allows system composers to select from a set of different but alike implementations or to replace individual components later, for instance, for incremental upgrades.

In a nutshell, the problem with white-box components is that their deployers cannot know which of the provided information they may use without endangering later substitutability. Of course, such a substitution can always be done by reconsidering the entire system, that is, the new component together with the entire particular context. This could become a complicated task, however. It may become entirely impossible in a context that is created only very late, for example, by the user of a network computer [Szy96]. Additionally, by distributing source code the developer cannot protect the implementation know-how.

2.3 Components Should Be Grey Boxes

Black-box components providing too little information and white boxes revealing too much, we need to go for a middle ground, which we may call *grey-box* components. A grey box reveals parts of its internal workings, not just relations between input and output. The information can become as detailed as necessary where needed, for instance, to state under what conditions external components are called. In other places it may remain very abstract and simply state a condition that is established (Fig. 5).

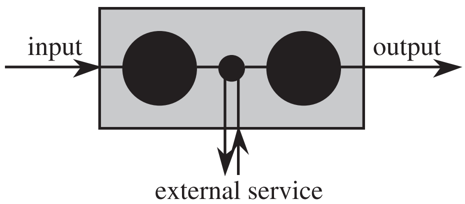

Figure 5: Grey-box component

Taking up the observer pattern from Fig. 3 we give the corresponding grey-box specification in Fig. 6, which clearly states that the delete notifications are sent after the modification.

Revealing some internal information might also help the client programmer improve the performance of the complete system by taking advantage of the knowledge which call backs are made in what state. Kiczales argues that the abstraction and the resulting loss of control of performance-relevant details makes black boxes

```
procedure Delete (pos: N) =
  assert 0 ≤ pos < #text;
  text := text[0...pos-1] + text[pos + 1...#text - 1];
  for o ∈ observers do o.DeleteNotification(pos) od
end
```

Figure 6: Grey-box specification of observer pattern

hard to reuse [Kic94]. He suggests a metaobject protocol for providing client control over the internal workings. Grey boxes might be a complimentary solution.

In summary, we can say that the main purpose of a specification is to make clear what a client can rely on, resp. what every implementation of that specification has to provide. For this, stating relations between input and output values is not always sufficient. Thus, we need a mechanism to express those parts of the implementation that are relevant to the client programmers in a way that does not fix any unnecessary details.

3 The Human Factor

In general, a specification should be clear, concise, and unambiguous. It must be decidable whether a given implementation fulfills a certain specification, and whether a specification reveals enough, for the corresponding components to be used intelligently. Overspecification, lack of orthogonality, and ambiguities should be easily detectable.

Formal methods provide ways to check these requirements. For this, a specification language should be based on a formal semantics. This does not, however, imply that it has to use a very symbolic mathematical notation.

Specification formalisms are often optimized for expressiveness and brevity, leading to symbolic notations which are not easily readable for the average programmer. Many specification techniques dictate a functional style of thinking, whereas most programmers are more accustomed to the imperative paradigm. Likeness to programming languages flattens the learning curve and improves acceptance. The difficulties and the number of errors increase with the extent of the transformation between the way the we think and the paradigm and notation in which we express our thoughts. This is not to say that we should not be exposed to different paradigms and notations to broaden our horizons. But in reality, changing the way we think is very difficult. In this paper we concentrate on components implemented in imperative programming languages. We do not address the specification of components written in functional or logic programming languages and the integration of components using different paradigms.

Specifications must not only be easy to write, but also to read. Component specifications are read by many people with different backgrounds.

The lack of tool support degrades specifications to mere documentation. The inclination to write a specification which can be executed/simulated and which can serve as a rapid prototype is higher than to write a mere documentation specification. Additionally, if tool support provides for refinement proofs between different versions of a specification and between specifications and implementations the usefulness of specifications and their usage increase. We have experienced ourselves that students, who could not be motivated to write a single specification using pencil and paper, spent days writing B specifications [Abr96, BC95], because the latter can be simulated and provide a base for machine-assisted correctness proofs of their implementations. The symbolic notation and the lack of tool support are the key factors why formal semantic contracts are the most important non-practice in component software.

4 Specification Approaches

In this section we describe common specification approaches and investigate their applicability for describing grey-box components and their applicability for imperative implementation languages.

4.1 Pre- and Postcondition Specifications

Pre- and postcondition specifications are probably the most common form of component contracts. The precondition expresses an assumption that must be satisfied when calling the component. The postcondition states the promises of what the component guarantees to establish.

Unfortunately, postconditions tend to get very big and unreadable. Because postconditions cannot contain any procedure calls, the semantics of the called procedures has to be expanded into the postcondition. Hence, the approach does not scale well. It is impossible, to specify the state at which external calls are made or the order of calls (Sect. 2.1, Fig. 3). Hence, pre- and postcondition specifications are ill-suited for grey-box component descriptions.

4.2 Algebraic Specifications

The classic example of algebraic specifications is the definition of a stack, which includes the axiom pop(put(s, x)) = s. Implicit definitions in forms of axioms stating the properties of the component free us from selecting a representation. Algebraic specifications have a number of appealing mathematical properties. Their practical use, however, is rather cumbersome. Operations are not specified individually. It is difficult to find a sufficient and minimal set of axioms for a given component. The forced style of thinking does not match well with the imperative paradigm, in which most programmers think and implement. External calls cannot be handled in a satisfactory manner. Thus, algebraic specifications do not meet the requirements for grey-box specifications.

4.3 Abstract Statements

Specifications can also be expressed in imperative languages, enriched by specification statements and data types like sets and sequences. Specification statements can express things like 'set x, y, z such that P holds' [BvW97, Mor90]. Other nondeterministic constructs help to avoid overspecification. The imperative style allows to break down an operation into a sequence of statements.

External calls can be included indicating to the client programmer which calls are made in which sequence and in which states. Hence, it fulfills our requirements for grey-box specifications. However, it is far from trivial to write a specification that includes all external calls and does not overspecify the component.

The notation of abstract statements may range from very mathematical to a syntax similar to implementation languages. In accordance with Sect. 3 we argue for the latter. Using abstract statements, we get a seamless refinement approach without a paradigm change from initial specification to implementation. An implementation only differs from a specification by the absence of nondeterministic constructs and special data types. This seamless approach also allows for intermediate steps, which as in a calculation can reduce errors.

Specifications by abstract statements come close to contracts as proposed by Helm et al. [HG90]. 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.

4.4 Invariants and Temporal Properties

In addition to specifications of single operations, it is often desirable to specify invariants and history properties that hold for the whole component. An invariant is a predicate restricting the set of possible states. Invariants must also be respected by additional operations added in future versions. Without explicit invariants in a component contract, client programmers tend to assume the strongest invariant derivable from the visible interface. This practice is dangerous, because added operations in a later version might violate this invariant and because supplemental operations might only be visible to other selected clients.

Temporal properties restrict the set of possible transitions. We might state that a counter is never decreased. As with invariants, additional operations must respect these properties. Again, it is not safe to assume that the set of traces will remain unchanged; hence, only explicitly stated temporal properties must be assumed.

5 Related Work

VDM-SL [Jon86], VDM++ [DvK92] and a number of other specification languages allow a mixture of explicit and implicit specification for a single procedure.

The rational behind this is to simplify the creation of specifications. Admitting this benefit, we stress the necessity to reveal selected information about a component, as argued in Sect. 2.

The human factor in specification language design has been taken into consideration by e.g., Larch/C++ [Lea96], which is a superset of a subset of C++. Other attempts to make the specification language similar to the envisaged implementation language, respectively make it part of it include 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 and Eiffel [Mey92].

6 Future Work

We propose the following agenda to investigate and validate the idea of a grey-box specification language, suitable for engineers. First, we intend to extend an existing programming language, that is, the component implementation language Component Pascal [Obe97], with a few constructs for nondeterministic specification. We expect to need few extra data types, in particular, sequences and sets of objects, a nondeterministic choice statement, and specification statements (Sect. 4.3).

Reusing an existing language has a couple of advantages. Most importantly, we expect to flatten the learning curve and rise the acceptance by programmers using the same language as for component implementation. Further, by making the specification language a true superset of the implementation language, it becomes easy to express the required details. We acknowledge the danger of overspecification, but have not yet a solution at hand to guide the specifiers away from that abyss. Finally, using a complete existing language, accelerates the experiment, because we can concentrate on the essential extensions. Picking a language, with which we have a lot of experience and which is particularly tailored for component programming, is a matter of convenience in the first place.

With the first specification language in place, we intend to run a case study by specifying completely an existing subsystem of Oberon/F [Obe95]. The idea behind that is not to fall into the 'toy example trap'.

After this case study, we will probably need to refine our language definition. This refined language should form a sound basis for further research, like tool support or transfer to other environments.

7 Conclusions

Components need semantic specifications. Black-box specifications, revealing only input-output behavior are not sufficient for interactive components making external calls. White-box specifications require detailed understanding and fix too many details, making alternate implementations and future enhancements impossible. Grey boxes, as a middle ground, reveal the states at which external calls are made and the sequence of calls, while abstracting from implementation details.

Specification languages should be based on the same paradigm and use a similar notation as the implementation language to improve acceptance and reduce transformation difficulties. Tool support for simulation and proving greatly enhances the appeal of specification languages.

Of the surveyed specification approaches, abstract statements best fit the requirements for grey-box specifications and for wide acceptance. The seamless approach between specification and implementation provided by the refinement calculus is an added bonus.

Acknowledgments We would like to thank Ralph Back, Emil Sekerinski, and Clemens Szyperski for a number of fruitful discussions.

References

[Abr96] J. R. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[AOB97] 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.

[BC95] B-Core. *B-Toolkit*. England, 1995.

[BvW90] R. J. R. Back and J. von Wright. Refinement calculus I: Sequential nondeterministic programs. In J. W. deBakker, W. P. deRoever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, pages 42–66. LNCS 430, Springer Verlag, 1990.

[BvW97] R. J. R. Back and Joackim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer Verlag, to appear 1997.

[DvK92] E.H. Dürr and J. van Katwijk. VDM++ — a formal specification language for object-oriented designs. In *Computer Systems and Software Engineering, Proceedings of CompEuro'92*, pages 214–219. IEEE Computer Society Press, 1992.

[HG90] 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.

[Jon86] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International, 1986.

[JV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Kic94] 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.

[Lea96] Gary T. Leavens. An overview of Larch/C++ behavioral specifications for C++. In Haim Kilov and William Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, pages 121–142. Kluwer Academic Publishers, 1996.

[Mey92] 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.

[Mic97] Microsoft. OLE Development, 1997. http://www.microsoft.com/oledev/.

[Mor90] Caroll C. Morgan. *Programming from Specifications*. Prentice Hall, 1990.

[Obe95] Oberon microsystems, Inc. *Oberon/F*. http://www.oberon.ch, 1995.

[Obe97] Oberon microsystems, Inc. Component Pascal Language Report, 1997. http://www.oberon.ch/docu/language_report.html.

[PS94] Dick Pountain and Clemens Szyperski. Extensible software systems. *Byte*, 19(5):57–62, May 1994. http://www.byte.com/art/9405/sec6/art1.htm.

[Sun97] Sun Microsystems, Inc. Java Beans, 1997. http://splash.javasoft.com/beans/.

[Szy96] Clemens A. Szyperski. Independently extensible systems — software engineering potential and challenges. In *Proceedings of the 19th Australasian Computer Science Conference, Melbourne*, 1996.

[Szy97] Clemens A. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, to appear 1997. ISBN 0-201-17888-5.

[Wec96] Wolfgang Weck. *On Document-Centered Mathematical Component Software*. PhD thesis, ETH Zürich, 1996. Diss ETH 11817, ftp://ftp.inf.ethz.ch/pub/publications/dissertations/th11817.ps.gz.

[Weg97] Peter Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, 40(5):80–91, May 1997.

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**