The Greybox Approach: When Blackbox Specifications Hide Too Much

Martin Büchi

Turku Centre for Computer Science
Lemminkäisenkatu 14A, FIN-20520 Turku
Martin.Buechi@abo.fi, http://www.abo.fi/˜Martin.Buechi/

Wolfgang Weck

Oberon microsystems AG
Technoparkstrasse 1, CH-8005 Zürich
weck@oberon.ch, http://www.abo.fi/˜Wolfgang.Weck/

Turku Centre for Computer Science
TUCS Technical Report No 297
August 1999
ISBN 952-12-0508-3
ISSN 1239-1891

Abstract

Development of different parts of large software systems by separate teams, replacement of individual software parts during maintenance without changing other parts, and marketing of independently developed software components require interface descriptions. Interoperation is impossible without sufficient description; only abstraction leaves room for alternate implementations.

Specifications that only relate the state prior to service invocation (precondition) to that after service termination (postcondition) do not sufficiently capture external calls made during operation execution. If other methods called in the specification cannot be fully specified, it is not sufficient that the implementation only performs the specified state transformation. The implementation must also make the prescribed external calls in the respective states.

We show how to specify both state change and external call sequences using simple extensions of programming languages. Furthermore, we give a formal definition of the correctness of implementations with respect to such specifications and show how to prove correctness in practice with data refinement in context.

Keywords: behavioral interface specifications, component software, layered specifications, greybox specification, specification of state transformation and external call sequences, refinement, specification extensions of imperative programming languages, Java

TUCS Research Group
Programming Methodology Research Group

1 Introduction

Independently developed and marketed software components and individually replaceable software parts quickly gain importance. The interfaces between those have to be specified so that independent readers arrive at the same conclusions. The necessity for complete interface descriptions is especially big in the realm of independently developed software components. Hence, we expect current bad experiences with too fuzzy specifications to raise mainstream acceptance of the overhead it takes to write more systematic and formalized specifications of software component interfaces. Broad acceptance, however, can only be expected, if the required extra effort, both in time and intellectual, does not outweigh the experienced —or expected— gain.

The contributions of this paper are an interface specification approach that captures both state transformations and component interactions via method calls and accompanying refinement rules. The latter can be used both informally in the back of the head as well as for formal proofs. To enhance practical applicability and acceptance, our specification language is defined as a slight extension of an imperative programming language, in the case of this paper Java. Considering that even pre- and postconditions, for which some tool support already exists [31], are rarely adopted, we put special emphasis on bringing specifications closer to the mind setting of an imperative programming language user.

The refinement calculus foundation guarantees us semantic soundness.

1.1 Interface specifications are abstractions

It is an old observation that there is a need to present abstractions of software building blocks to make them reusable by third parties and to allow for alternate implementations. A long time ago, programming languages started to provide means for syntactic encapsulation, but even today only few can represent semantic abstractions. Almost all approaches to the latter rely on relating the system's states prior to an operation invocation to that after termination. Pre- and postconditions are the most prominent example here. In this paper, we discuss a situation in which these approaches are unsatisfactory and suggest to draw on the theory of program refinement and to use abstract programs as specification formalism.

Already modular programming introduced by Parnas in 1972 [41, 40] includes information hiding or encapsulation to separate concerns between implementing and using a module. This simplifies the analysis of complex software systems, because software using a specific module M can be described without explaining M's implementation details. As a further consequence, the implementation of M can even be changed later on, as long as it still meets the same abstraction. On the syntactic level, modules are supported by several programming languages, such as Modula-2 [51], Modula-3 [37], and Ada [50]. Programmers decide which identifiers (variables, procedures) are visible (exported) to clients of their module and which are hidden and can be accessed by code within the same module only.

Encapsulation is also one of several pillars on which object-oriented programming (OOP) rests. As with modules, object implementations are decoupled from their interfaces. In addition, however, the interfaces may be changed or extended by subclasses. The separation between specification and implementation is also crucial to achieve polymorphism, another main pillar of object orientation. Only because an object's client does not depend on one specific implementation, it can work with instances of subclasses as well.

Syntax level encapsulation is provided by most OOP languages, for example, C++ [47], Eiffel [30], and Java [18]. Modula-3 [37], Oberon [52], and Component Pascal [39] combine object-orientation with modules.

As a combination of modular and object-oriented programming, component software relies on encapsulation and abstraction as well [49]; and again, syntax level abstraction is supported by interface description languages (IDLs), as defined for Microsoft's COM [44] and OMG's CORBA [19] standards.

Syntax level encapsulation is extremely effective when it comes to ensuring that certain internal invariants are never invalidated by client modules or classes. Not granting access to data or functionality means that all access is under local control. If things go wrong, the reason must be in the own software building block.

1.2 From syntactic to semantic abstractions

Syntax level encapsulation and abstraction is not enough. For those identifiers that are visible to client programmers we often need to describe how, under which circumstances, or in which way they are to be used.

Syntactic abstraction must, therefore, be complemented with semantic abstraction. If variables can be accessed, assignments may need to be constrained with invariants. If operations can be invoked, it must be said under which circumstances they may be invoked and what they then can be expected to do. Prominent languages and formalisms for this are Eiffel's pre- and postconditions [30, 31], Larch C++ [26], and Parnas' tables [23].

All these approaches consider everything between an operation's invocation and termination as completely hidden. All available information deals with what is before and after an operation's execution. No information is given about what happens in-between (Fig. 1). We call such specifications *blackbox specifications*.

Often, this is sufficient, but there are cases in which blackboxes are too black and more information is needed. Often software libraries are provided with com-

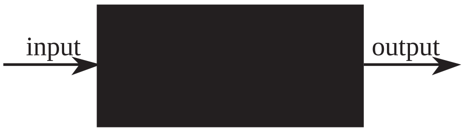

Figure 1: Blackbox Specification

plete implementation source code for last reference. Unfortunately, source code spoils most advantages of encapsulation. We refer to source code also as *whitebox specification*. In this paper we discuss situations in which blackboxes are too dark and propose a way to lighten up blackboxes to become greyboxes, which combine the advantages of black- and whiteboxes.

We develop formal refinement rules to prove the correctness of implementations with respect to their specifications. Even for projects where the (expected) cost savings in testing and maintenance don't outweigh the cost of proofs, safety is not critical, and the time is tight, we believe that it's worthwhile to at least have these rules in mind when coding. We look at them like loop invariants and termination functions, which are rarely written down —let alone formally proved— yet are in some programmers' mind when coding.

Overview. Section 2 illustrates a typical component interaction case where blackbox specifications hide too much. Furthermore, it introduces the observer-pattern, which is used as an example throughout the paper. In Sect. 3 we show a first specification approach that shows mandatory call-backs without giving up abstraction. In the following section we iron out the remaining problems and present the final greybox specification style. Section 5 talks about implementations. Section 6 presents the full refinement rules to assert correctness of implementations with respect to specifications. Aiming for practical applicability, we show how to prove correctness of frequent special cases in Sect. 7. Readers who are content with an informal, intuitive explanation of the correctness of implementations with respect to their specifications may skip Sects. 6.5 and 7. Section 8 summarizes the required additions to imperative programming languages for the purpose of greybox specification. Finally, Sect. 9 discusses related work and Sect. 10 draws the conclusions.

2 The Problem: When Black is Too Dark

Above, we defined *blackbox specifications* as descriptions that only relate the state before and after an operation (Fig. 1). It is impossible to draw conclusions about what happens between these two observation points unless there is some trace left in the observable state.

Such blackbox specifications are insufficient when it comes to call-backs. Callbacks activate functionality external to the specified component (Fig. 2). Typically such functionality is installed by the calling client or even third party software, as it is the case with the observer example detailed later in this section.

That blackboxes do not provide enough information to deal with call-backs has been stated and discussed in detail in [49]. In the following we shall briefly recapitulate that discussion.

2.1 Call-backs

A call-back mechanism allows clients of a library to register operations for activation under certain circumstances. Figurative, the client instructs the library *to call it back* upon occurrence of a certain event (Fig. 2).

Call-backs are used to make systems extensible. In layered system architectures they occur as calls from lower into higher layers in which case they are known as up-calls [12]. Up-calls allow the programmers of higher layers to modify the behavior of the lower layers of which they are clients.

In a similar way, methods implemented in a subclass but called in the respective superclass can be interpreted as call-backs from reusable into reusing software components.

A representative application of call-backs is the observer design pattern [17]. It allows software components that need to react to certain events, such as particular state changes, to register an *observer* object with the *observed* object. The observed object then calls a notify method of each registered observer object upon occurrence of the respective events.

A prominent application of the observer pattern is in the Model-View-Controller architecture (MVC), developed originally for Smalltalk [25, 17]. The MVC architecture provides a separation of concerns between internal representation and manipulation of data (model), data presentation to the user (view), and command interpretation (controller). Because of that separation, the way of presenting the data to users can be changed by replacing the view component while keeping —or reusing— the model component. Most implementations of the MVC architecture allow more than one view to present the same model at a time. These views may even display the data differently, for instance, as a spreadsheet and as a pie chart. In this paper, we use a simplified version of the MVC pattern without a separate controller.

If we want the model component to work with zero, one, or several simultaneous views and not to depend on what the presentation actually looks like, it must not be hard-wired to any specific view component. Simply defining the view as a subclass of the model does not work because the model could not be shared among different views. Whenever the data stored in the model is changed because some client activates an operation, all the views have to be updated. This is done

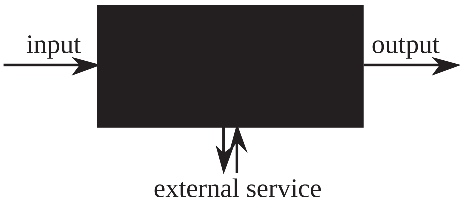

Figure 2: Blackbox Specification with External Call

by having each view register with the model as an observer to be notified on data changes.

2.2 Example: part of a text system

Szyperski [49] presents a simple text system as an example of the above. In this paper we shall only look at those functions needed to delete characters. As the form of presentation we use Java syntax together with pre- and postconditions (Fig. 3).

All our specifications are model based. That is, we add a model state, such as registeredObservers and text, to express the specifications. We use the modifier **private** to mark model fields and methods, i.e., members that are only present for specification purposes and are not accessible to clients and classes implementing the interface.

We give the specification additions directly rather than as comments with a special start symbol [27]. A pre-parser can easily remove the additions so that the stripped version can still be processed with a normal Java compiler.

We use the abstract data types **setof** and **seqof** for sets and sequences of objects. The empty set is denoted by '{}' and the empty sequence by '<>'. The length of a sequence is given by len, the ith element of sequence s can be accessed as s[i] with indices ranging from 0 to len(s)-1. In accordance with Java, we use '==' for equality and '=' for assignment. The keyword **all** denotes universal quan-

```
interface ITextModel {
  private setof ITextObserver registeredObservers={};
  private seqof char text=<>;

  int length();
    pre true
    post result==len(text)

  char charAt(int pos);
    pre 0<=pos && pos<len(text)
    post result==text[pos]

  void deleteCharAt(int pos);
    pre 0<=pos && pos<len(text)
    post (all i: 0<=i && i<pos: text[i]==text′[i]) &&
          (all i: pos<=i && i<len(text): text[i]==text′[i+1]) &&
          len(text)==len(text′)-1

  ...

  void register(ITextObserver obs); //specification omitted
  void unregister(ITextObserver obs); //specification omitted
}
```

Figure 3: Pre-/Postcondition Specification of Interface ITextModel

```
interface ITextObserver {
  void deleteNotification(int pos);
    // pre character that was at pos has just been deleted
    // (informal comment, not tool checkable)
}
```

Figure 4: Specification of Interface ITextObserver

tification. Primed identifiers in postconditions refer to the state before invocation (see below). The keyword **result** denotes the result value.

According to the MVC pattern, instances of classes implementing ITextModel are data stores. Changes, such as deleting a character, can be initiated by any client calling the respective method. Views need to call the register method once to subscribe to notifications about text changes.

The above specification of ITextModel.deleteCharAt does not say that and when observers are notified. Because it is not reflected by the state of the text data before or after calling the operation, pre- and postconditions of the deleteCharAt operation cannot capture this. The only solution would be a textual comment like 'call deleteNotification of all registered observers after deleting character'. Additionally, we can put the corresponding comment to ITextObserver.deleteNotification (Fig. 4). However, none of these plain English comments are machine checkable and enforceable.

As an aside: Whereas the prime notation for the value in the pre state works well with variables, it is unsharp for functions and actually dangerous for operations with side effects. In the latter case, using a function in a predicate must not be confused with an actual call to that function, because the corresponding side effects are not happening. With pure functions, problems occur if the result values depend on other state than the explicit parameters. The length function, if used instead of len(text), would be an example thereof; its result depends on the hidden state of the text it is associated with. The usual workaround is to introduce additional bound variable that denote the pre state.

2.3 Analysis of the example

In the ITextModel and ITextObserver example several aspects are unspecified or specified informally only. For instance, in what order are the observers notified? Is it the same order every time? Can the observers register additional observers upon being notified of a state change? Are these newly registered observers notified in the current round?

Often, such things will be left open intentionally and in this case the above specification is just fine, as long as readers don't make any assumptions about the unspecified facts, which in practice often happens because the intention of non-specification is not made explicit. It signals to the implementer of a ITextObserver that no assumptions must be made, while the the implementer of a ITextModel has full freedom. Of more interest are situations in which certain aspects shall be specified and other intentionally be left open.

The above problems illustrate the need for unambiguous interface specifications. They can, however, all be addressed with a pre- and postcondition-based approach with a complete and clear semantics. As hinted at before, there are, however, some fundamental issues that cannot be specified in this way. As an example, where blackbox specifications fails, we will in the remainder of this paper consider the deleteNotification method.

2.3.1 Blackbox specifications are about state changes only

The problem is rooted in the fact that we are specifying an interface of an open, extensible system. While we know and can prescribe many aspects of the model's state, we want to retain full flexibility for the observer. It is the very idea of this pattern that one has not to define what observers may do upon a notification. This openness presents a dimension of extension.

Without talking about the observer's state at all, we cannot specify the conditions of a notifier call, because the call does not affect the model's state. Theoretically, we could encode the calls made by an operation into history/trace variables. For example, each observer could be equipped with a counter to be implicitly incremented during each notification call. With this we could add a conjunct to the postcondition of ITextModel.deleteCharAt expressing that the counters of all registered observers have been incremented. However, with only these counter we could still not specify in which order, in which states, and with which parameter values the calls are to be made. Are the observers notified before or after the actual deletion? What are the parameter values of the calls?

Although theoretically possible, a trace variable encoding capturing all these aspects is very complicated and unusable for practical specifications. Furthermore, it does not give the desired results in combination with data refinement methods for abstract data types based on observational substitutability: Without enquiry operations for all trace variables, the latter can just be 'forgotten'. With enquiry operations, their values could be computed differently. Only a mandatory 1-to-1 data refinement relation on trace variables and syntactic rules to forbid explicit assignment to these variables could, theoretically, rest the case. In conclusion, blackbox specification of callbacks through encoding is not practical.

2.3.2 Whitebox specifications would work but aren't abstract enough

Whitebox specifications, that is source code, show exactly when and in which order the observers are notified (Fig. 5). However, they ruin abstraction by fixing too many details. For example, we might want to leave the notification order open so it can be changed in future versions. In Sect. 3 we will discuss how abstraction can be reintroduced into such whiteboxes.

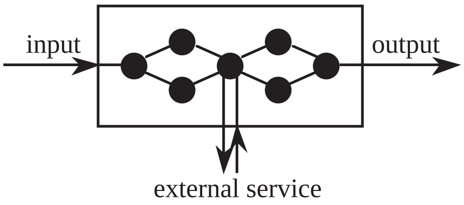

Figure 5: Whitebox Specification

2.3.3 Informal specifications are not good enough

Pure informal specifications might be clear enough in very simple cases, but they also have their share of disadvantages. They cannot be used as input to any tool, such as automatic test case generation [11], automatic pre- and postcondition checking [31, 15], or formal theorem provers [1].

Informal sentences are subject to interpretation, which in turn depends on the particular context of the reader. Often, these contexts vary resulting in mismatches of interpretations by independent vendors and eventually leading to incompatible software components.

This problem is aggravated by the fact that component interfaces need to abstract and for this often intentionally leave certain aspects undefined. However, with informal specifications it is often not clear what is intentionally left unspecified.

3 A Pragmatic Approach: Layering White on Black

In this section we discuss a simple and pragmatic way to specify the circumstances under which call-backs are to be made. The idea is to decompose an operation containing the call-backs into a set of private operations and a public operation calling the former. None of these auxiliary operations contain any call-backs. Hence, they can be specified as complete blackboxes. On top of that layer of blackbox specification we put a single whitebox specification of the original operation (Fig. 6). The latter contains only calls to the blackboxes of the lower layer, the call-backs, and

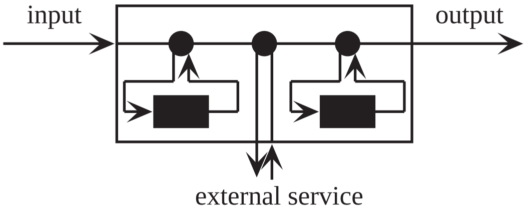

Figure 6: White-on-Black Layered Specification

eventual loops and conditionals in which call-backs occur. In our ITextObserver example, we decompose the deleteCharAt method into just one blackbox operation changing the text data and a whitebox operation. The latter calls the blackbox operation and notifies the observers (Fig. 7).

```
interface ITextModel {
  private ITextObserver[] registeredObservers;
  private int nofObservers=0;
  private seqof char text=<>;

  private void removeCharacter(int pos);
    //Blackbox specification:
    post (all i: 0<=i && i<pos: text[i]==text′[i] &&
          (all i: pos<=i && i<len(text): text[i]==text′[i+1]) &&
          len(text)==len(text′)-1

  void deleteCharAt(int pos) pre 0<=pos && pos<len(text) {
    // Whitebox specification:
    int i;
    removeCharacter(pos);
    for(i=0; i<nofObservers; i++){
      registeredObservers[i].deleteNotification(pos);
    }

  ...
}
```

Figure 7: Layered Specification of Interface ITextModel

The general idea used in the example can be summarized as follows. To make clear, under which circumstances the observer is notified, we need to give a whitebox specification of all services that contain such calls; in our case this is the deleteCharAt method. Specifying the entire service as a whitebox, however, would be too detailed. Hence, wherever we want to give an abstraction instead of an actual implementation, we include a call to another private service, which we specify as a blackbox. Without loss of generality we assume a single call-back only in the general specification pattern illustrating the above idea (Fig. 8).

The main advantages of this approach is that it can readily be deployed with formalisms such as Eiffel [30] or JML [27] that allow both black- and whitebox specifications. Although external calls can be specified, neither of these approaches comes with refinement rules that preserve the external call sequence.

This layered approach also has some disadvantages. We need to make every data structure, such as registeredObservers, used in the whitebox part concrete. Leaving certain aspects, such as the notification order, unspecified may be difficult. Also, the need to introduce a blackbox operation for every block makes specifications less readable.

```
interface LayeredSpecPattern {
  private void partOne(...);
    // Blackbox specification:
    // pre Pre1
    // post Post1

  private void partTwo(...);
    // Blackbox specification:
    // pre Pre2
    // post Post2

  void publicService(...) {
    // Whitebox specification:
    partOne(...);
    callBack(...);
    partTwo(...);
  }
}
```

Figure 8: Pattern for Layered Specifications

4 Abstract Programs: Shades of Grey

Specification statements [4, 7] can be used instead of auxiliary blackbox methods. A specification statement is of the form **any**(T y; P) {S;} where T is a type, y a (bound) variable, P a predicate (boolean expression), and S a statement. Upon execution, an arbitrary value for y is chosen such that P holds and then S is performed. For example, **any**(float y; y>=0 && 0.98*y*y<x && x<1.02*y*y) {s=y;} assigns the square root of x, computed with a precision of 2 % to y and subsequently to s. If the value of x is 16, then that of s will be between 3.96 and 4.04 after the statement has been executed.

In our example (Fig. 7), we can replace the call to removeCharacter in deleteCharAt by

```
any(seqof char txt; (all i: 1<=i && i<pos: txt[i]==text[i]) &&
    (all i: pos<i && i<=len(text): txt[i]==text[i]) && len(txt)==len(text)-1) {text=txt;}
```

Whereas the above square root example was truly nondeterministic, the text example isn't because there is exactly one possible outcome for every initial value of text and pos satisfying the precondition. Hence, the specification can in this case be written as:

```
text=text[0..pos-1]+text[pos+1..len(text)-1]
```

As another specification construct, we add a **do** loop over sets. The body of a **do** loop is executed exactly once with the iterator bound to each element of the initial value of the set.

```
interface ITextModel {
  private setof ITextObserver registeredObservers={};
  private seqof char text=<>;
  public int maxObservers=10;

  invariant // denoted by I in the text
    card(registeredObservers)<=maxObservers

  inquiry int length() {
    return len(text);
  }

  inquiry char charAt(int pos) pre 0<=pos && pos<len(text) {
    return text[pos];
  }

  void deleteCharAt(int pos) pre 0<=pos && pos<len(text) {
    text=text[0..pos-1]+text[pos+1..len(text)-1];
    do(o in registeredObservers) {
      o.deleteNotification(pos);
    }
  }

  ...

  void register(ITextObserver obs) pre obs!=null && not(obs in registeredObservers) &&
        card(registeredObservers)<maxObservers {
    registeredObservers=registeredObservers+{obs};
  }
}
```

Figure 9: Greybox Specification of Interface ITextModel

Our focus has been on the notification calls, which we want to specify so that they must be made in an implementation. However, a specification may also make optional calls. Say, we use a call to a square root function of a math component. An implementation should be free to either make the same call or compute the result itself. To distinguish between mandatory and optional calls, we use the following approach. *Inquiry methods* are declared with the modifier **inquiry** in the specification. All other methods are called *modification methods* —whether they actually modify the state or not. Inquiry methods may not modify the state or contain any calls to modification methods. In our example, length and charAt are inquiry methods. Calls to modification methods in the specification are referred to as *mandatory calls*; calls to inquiry methods as *optional calls*. Implementations must make the same mandatory calls as their specifications and may not make any additional calls to modification methods of the mentioned component instances. Additionally, they can make arbitrary calls to inquiry operations.

Figure 10 gives the general pattern for greybox specifications with the same

```
interface GreyboxSpecPattern {
  void publicService(...) {
    any($T_1$ $w_1$, ..., $T_m$ $w_m$; Post1′) {$u_1$=$w_1$; ...; $u_m$=$w_m$;};
    callBack(...);
    any($T_1$ $w_1$, ..., $T_n$ $w_n$; Post1′) {$u_1$=$w_1$; ...; $u_n$=$w_n$;};
  }
}
```

Figure 10: Pattern for Greybox Specifications

semantic meaning as Fig. 8. Post1′ and Post2′ correspond to Post1 and Post2 with u_i replaced by w_i. We have here generalized the **any** statement for multiple variables. That is, an arbitrary tuple of values will be chosen for $w_1, \ldots, w_m$.

Constructors are used in the creation of objects to explicitly assign instance variables. A class may have many (overloaded) constructors with different signatures. Unlike Java, we also allow constructor specifications in interfaces. These specifications only prescribe certain constructors; no instances of interfaces can be created.

Figure 11 illustrates the greybox specification approach.

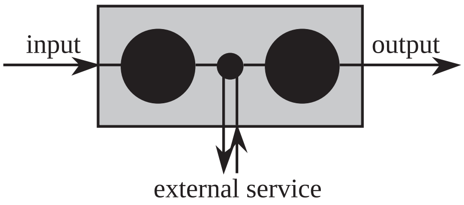

Figure 11: Greybox Specification

4.1 Consistency of specifications

Specifications that may abort or invalidate their own invariant when executed are of questionable use. Therefore, we require the following five consistency conditions to hold for greybox specifications: If an operation is started in a state and with actual parameters that satisfy both the invariant and the precondition, then

1. The method may not abort, i.e., try to access a **null** reference (We consider throwing an exception as abortion.).
2. The invariant is guaranteed to hold after termination.
3. All external calls are made with parameters that satisfy the respective preconditions.
4. The own invariant holds whenever an external call is made.

5. An operation either terminates or makes infinitely many external calls to modification operations.

The reason for the fourth condition is explained in Sect. 6.1. As we do not allow reentrant calls to modification methods (Sect. 6.1), external calls do not change the local state; there only influence is via return values. Constructors have to satisfy the above five consistency requirements when called with parameters satisfying the precondition.

If there are no possible values for the bound variables (y) in an **any** specification statement such that the predicate (P) holds, then the statement behaves like **magic**, that is it satisfies any postcondition. Of course, **magic** cannot be implemented. Still it has some practical applications in specifications [7].

5 Implementing Greybox Specifications

Implementations are coded as normal Java classes (Fig. 12). The only addition is the invariant. The latter consists of two —possibly intermingled— parts, the local and the gluing invariant. The local invariant restricts the 'legal' values of the local variables. Its aims are the documentation of desired properties and the simplification of proofs. The gluing invariant links the values of the concrete variables in the implementation to the abstract variables of the specification. For ex-

```
class CTextModel implements ITextModel {
  private ITextObserver[] regObs=new ITextObserver[maxObservers];
  private int nofObs=0;
  private StringBuffer t=new StringBuffer();
  invariant // denoted by I′ in the text
    0<=nofObs && nofObs<=maxObservers &&
    ITextModel.registeredObservers==regObs[0..nofObs-1] &&
    (String)ITextModel.text==(String)t

  public void deleteCharAt(int pos) {
    int i;
    ... // remove character from t
    for(i=0; i<nofObs; i++) {
      regObs[i].deleteNotification(pos);
    }
  }

  public void register(ITextObserver obs) {
    regObs[nofObs]=obs;
    nofObs++;
  }
  ...
}
```

Figure 12: Implementation CTextModel

ample, ITextModel contains a set of observers. This set is implemented with an array (Fig. 12). To prove (see below) that the implementation adheres to its specification, we need to say how the values of the set and the array are related. The invariant conjunct ITextModel.registeredObservers==regObs[0..nofObs-1], where regObs[0..nofObs-1] is the set {regObs[0]} ∪ {regObs[1]} ∪ … {regObs[nofObs-1]}, states this.

To allow different implementations, we require that all fields are private. Thus, all accesses go through methods.

6 Refinement of Greybox Specifications

An implementation of a component must adhere to its specification; otherwise the specification is useless for a client. Correctness of an implementation with respect to its specification is established by proving greybox refinement. The basic idea of greybox refinement is simple: The implementation of every method must make the same sequence of mandatory calls to other component instances in the same respective states as the specification, perform the same overall changes to the local state, and return the same value. If the specification is nondeterministic, then the call sequence, local state transformation, and return value of the implementation must correspond to one choice in the specification. The refinement calculus [4, 7, 36] provides a formal base, but not yet any rules, to prove this conformance.

A specification can be viewed as a contract between a client and a provider who implements it. To refine a specification means to improve it from the client's point of view. A refined version must be at least as good as the original. Often, refinement is defined by observational substitutability: The client, that thinks it uses the original version, does not notice that it actually uses a refined version.

In this section we first clear the field by discussing a number of critical issues surrounding greybox refinement and then formalize the latter.

6.1 Reentrance

Reentrance occurs if method m of component A calls method n of component B and n calls —directly or indirectly— method o of A (Fig. 13) [49, 32]. For example, an observer could in its implementation of deleteNotification call ITextModel.charAt to enquire the current value of text.

Both for the consistency of specifications (Sect. 4.1) and for the refinement rules (Sect. 6.5) we assume the invariant of an object to hold whenever one of its methods is called. Hence, we need to establish the invariant before making calls to other components, so that the own invariant holds upon reentrant calls.

We follow [13] in banning reentrant calls to modification methods. Although such calls can be handled in theory, they make components very difficult to understand. For example, assume that in Fig. 13 o would be a modification method. Then the effect of method m on the local state of an instance of A would not de-

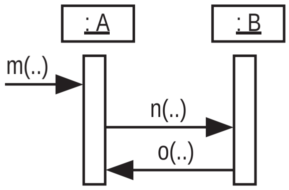

Figure 13: Reentrance Scenario

scribed by m alone. Instead all called external modification methods would have to be examined for possible reentrant calls to modification methods.

Mutual recursion between inquiry methods of components A and B could happen if A.m calls B.n and B.n call A.m. Because the calls to inquiry methods are not visible from specifications, such mutual recursions cannot be detected by modular reasoning. They are problematic if they are infinite. Documented solutions to this problem include imposing a partial order on inquiry methods and allowing calls only to ‘smaller’ methods [1]. In this paper we simply assume this problem to be addressed by some mechanism.

6.2 Not-mentioned component instances

Often implementations use instances of additional components not mentioned in the specification. For example, an observer may use a window instance, in which it paints the text and a model may use a set from a collection framework to represent the set of observers. This use of additional component instances comprises a variety of problems.

We call component instances referred to in the specification as *mentioned component instances* and all others as *not-mentioned component instances*. For example, from the perspective of ITextModel and of classes implementing only the former, any instance of a component CSet is classified as not-mentioned because neither CSet nor its specification is named. Furthermore, any instance of ITextObserver not registered as an observer is labeled not-mentioned.

As motivated above, allowing calls to modification methods of not-mentioned component instances is absolutely necessary. However, such calls may lead to reentrant calls of modification methods. Consider the following scenario: Method CTextModel.deleteCharAt calls a modification method event of a (not-mentioned) instance of a CLog component, the specification of which does not mention any other components. Since, event is a modification method, it may itself call other modification methods, including modification methods of our CTextModel instance and the latter’s registered observers. In both instances refinement of ITextModel by CTextModel may be violated. This problem is similar to the one of mutual recursion above. It cannot be solved by modular reasoning. Solutions that restrict the set of objects whose methods may be called trade safety for flexibility. In this paper we assume this problem to be solved by some strategy.

6.3 Self calls

Self calls, that is calls to methods of the same component instance, deserve special treatment. The idea of greybox specifications is to prescribe what calls have to be made between different component instances in addition to the changes of the instance state. If a specification contains self calls, it is simply for the purpose of factoring out parts that —ignoring recursion— could be textually substituted for the call. Hence, self calls are never mandatory. An implementation of a specification method containing self calls must simply make the same external calls and local state modification as the specification method including the own called methods. It is not required, but often beneficial, to establish the invariant before making self calls.

6.4 Additional methods and constructors

Classes may have more public methods and constructors than one of their implemented interfaces. For example, CTextModel could also have a method deleteCharsBetween(int from, int to), which deletes all characters between positions from and to. Every possible execution of an additional method must constitute a refinement of the external call sequence and state modifications performed by a finite sequence of interface methods with possible local computation of parameters in-between. In this case, other component instances cannot tell the difference between the single call to the new method and the sequence of calls (mumbling invariance) [9]. Additional enquiry methods can be added freely (stuttering invariance), as covered by the above definition with the empty sequence.

A class may implement multiple interfaces. In this case, the methods prescribed by other interfaces have to be considered as additional methods. For example, if CTextModel were also to implement an interface ILog, all implementations of methods prescribed by ILog would have to satisfy the above criteria with respect to ITextModel and vice versa.

Especially if a class implements several interfaces, the above requirement for additional methods sometimes turns out to be too strict. To overcome this problem, we allow interfaces to contain a special method others. This method is not to be implemented by classes, it simply shows what other modifications —with accompanying external calls— are allowed in additional methods.

Likewise, we can have additional constructors. Every possible execution of an additional constructors must constitute a refinement of the external call sequence and state modifications performed by an interface constructor followed by a finite sequence of interface methods with possible local computation of parameters in-between.

6.5 Formalizing greybox refinement

In this subsection, we give the general formalization of greybox refinement using a trace semantics, partly inspired by the trace semantics for action systems [6]. We formalize the conformance of call sequences and state changes. Predicate transformer semantics for basic programming language constructs can be found in the literature [7].

First we review some fundamentals of predicates, relations, and product and sum types.

Predicates and relations. Predicates (boolean expressions) on a type Γ are functions from elements of type Γ to Bool. For example, for variable x of type int, x>0 is a predicate on int. A predicate determines a subset, e.g. the positive integers. Thus we use $\subseteq$ as order between predicates, e.g. x>5 $\subseteq$ x>0. Predicates being functions, they can be applied to values, e.g. (x>5)(6) is true.[1]

Relations of type $\Sigma \leftrightarrow \Gamma$ are functions of type $\Sigma \to \Gamma \to Bool$. An invariant I′ of an implementation with state space Γ refining a specification with state space Σ can also be considered as a relation of type $\Sigma \leftrightarrow \Gamma$. We can apply such a relation to a state σ of the specification and a state γ of the implementation: I′(σ, γ). For relation R and predicate p the relational image im(R, p) is defined as {y | (**exists** x: R(x,y) && p(x))}. We use && for conjunction of predicates and relations.

Product and sum types. The product type int × char denotes the type of tuples of integers and characters. The tuple (5, ‘a’) is an element of it. We use the projection functions fst and snd for tuples, e.g. fst((5, ‘a’)) is 5 and snd((5, ‘a’)) is ‘a’.

The sum type int $\oplus$ char denotes a disjoint union of an integer and a character (corresponding to a variant record in Pascal or a union with a type tag in C). A variable of this type has either an integer or a character value. We leave the injection and projection functions for sum types implicit.

Definition of greybox refinement. With these notions we can define greybox refinement. Without loss of generality, we assume in our treatment that methods have no local variables, otherwise we turn them into instance variables. Furthermore, we only allow constants and variables as actual parameters.

When we animate (execute) method m of the sample interface ITest, it generates a behavior. A behavior is a sequence of states, where the successor state is computed by executing the next atomic statement.[2] Let Σ be the type of the state space of the specification, for example **setof** ITextObserver × **seqof** char × ITextObserver for ITextModel, where the last ITextObserver stems from the local variable o of deleteCharAt. When executing ITextModel.deleteCharAt(2) the following is a possible behavior: <({o1, o2}, [texst], null), ({o1, o2}, [text], null), ({o1, o2}, [text], o1), ({o1, o2}, [text], o2)>.

[1] We omit the explicit λ if the binding is clear due to the names or position as in the example, i.e., x gets bound to 6.

[2] External method calls are split into two atomic statements as explained below. Otherwise, the grain of atomicity is not important.

As defined so far, we do not record which modification methods of which objects are called with which parameters. Let Ω_i for i **in** 1..e denote the types of references to mentioned component instances in our specification ITest. Furthermore, let $\Delta_{i,j}$ for j **in** 1..f$_i$ denote the parameter types of the modification methods of component type i. Finally, let Ω_0 denote the type Unit with the single element unit and let f$_0$==0. By extending every state in a sequence by an element of type

$$\bigoplus_{i=0}^{e} \Omega_i \times \bigoplus_{j=1}^{f_i} \Delta_{i,j}$$

we can indicate for every state, whether in this state a mandatory call is made and if so to which method of which object and with which parameters. That is, an element of the above type is a tuple of a reference to a component and a parameter value for one of the modification methods of the referenced component.

Thus a *simple behavior* is of type:

$$\textbf{seqof}\left(\Sigma \times \left[\bigoplus_{i=0}^{e} \Omega_i \times \bigoplus_{j=1}^{f_i} \Delta_{i,j}\right]\right)$$

A mandatory method call w=e.m(c) generates two states σ_i and σ_{i+1}, such that σ_i==(fst(σ_{i-1}), (e,c)) and σ_{i+1}==((fst(σ_{i-1}))[w:=e.m(c)], (unit, unit)). (fst σ_{i-1})[w:= e.m(c)] stands for fst(σ_{i-1}) with the value of w replaced by the return value of e.m(c). A state is a *call state* if the second component is not (unit, unit).

For every method ITest.m we generate the set sbeh(ITest.m) of all simple behaviors where the first state and the value of the parameter satisfies the invariant and the precondition.

Let Π be the type of the input parameter of method in question, for example int for ITextModel.charAt, and Φ the result type, that is char for the aforementioned method. A *full behavior* additionally contains the values of the parameters and the result. A full behavior is of type:

$$\textbf{seqof}\left(\Sigma \times \left[\bigoplus_{i=0}^{e} \Omega_i \times \bigoplus_{j=1}^{f_i} \Delta_{i,j}\right]\right) \times (\Pi \times \Phi)$$

The set of full behaviors of method ITest.m is denoted by beh(ITest.m). For full behavior b, the corresponding simple behavior is fst(b).

We distinguish four kinds of behaviors: (normally) terminating, aborting, miraculous, and infinite behaviors. A terminating behavior is generated if the method returns control to its caller via a return statement or if its result type is void also by executing the last statement. An aborting behavior is generated if the method aborts, e.g., tries to access an array at an index outside its boundaries. Since aborting behaviors are undesirable, we actually require both specifications and implementations to be free of them. A miraculous behavior is generated if at some point **magic** is executed. An infinite behavior is generated if the operation neither

ITest	interface
CTest	class
Σ	type of state space of ITest
Γ	type of state space of CTest
I$'$	invariant of CTest
Ω_i	types of references to mentioned components
$\Delta_{i,j}$	parameter types of modification methods of Ω_i
$\oplus$	sum type
$\times$	product type
fst, snd	first and second projection of tuple
m	prescribed method
n	additional method
K	prescribed constructor
L	additional constructor
sbeh, beh	set of simple/full behaviors
str, tr	set of simple/full traces
b	behavior
s, s$'$	simple traces
t, t$'$	full traces
$\frown$	concatenation of simple traces
str(ITest)*	transitive closure of all simple traces of ITest
$\mathsf{str}(\widehat{\mathsf{ITest}})$	union of simple traces of constructors
im(I$'$, I)	relational image of I under I$'$

Figure 14: Summary of Notation and Conventions

terminates nor aborts. The terminating (beh$_+$(ITest.m)), aborting (beh$_\perp$(ITest.m)), magic (beh$_\top$(ITest.m)), and infinite behaviors (beh$_\infty$(ITest.m)) form a partitioning of all full behaviors. The same holds for their simple counterparts.

Traces are the observable parts of behaviors. They are generated by removing non-observable states (states where no external calls are made) from behaviors. We get the set of full traces tr(ITest.m) (resp. simple traces str(ITest.m)) by doing the following two operations on each behavior b in beh(ITest.m) (resp. sbeh(ITest.m)):

1. If b **in** beh$_\infty$(ITest.m) ends in an infinite sequence of non-call states, then we remove b from beh$_\infty$(ITest.m) and add it with the infinite sequence of non-call states removed to beh$_\perp$(ITest.m).

2. Remove all non-call states, except for the first state and if b is final the last state, from b.

Simple/full traces are of the same type as simple/full behaviors. The trace set is the union of the terminated, aborted, magic, and infinite traces.

Likewise we generate the set of traces of the implementation CTest.m under question. We assume that the state space of CTest is Γ. The initial states are given by im(I′, I && p). When computing traces for implementations, we only consider mentioned component instances and their types. Thus the types of the traces of ITest.m and CTest.m$_{\mathsf{ITest}}$ only differ in the first component of the sequences, which is Σ, respectively Γ. The subscript for CTest.m is necessary because CTest may implement multiple interfaces (Sect. 6.4).

We say that two simple traces s and s′ correspond at position i, if the state parts of the ith sequence elements are related by the invariant of the implementation I′, taken as a relation, and the external call parts are identical:

$$\mathsf{s} \propto^{i}_{I'} \mathsf{s'} \quad \overset{\text{def}}{=} \quad \mathsf{I'(fst(s[i]),\ fst(s'[i]))\ \&\&\ snd(s'[i])==snd(s[i])}$$

We define *trace approximation* under the refinement relation I′ as follows: The simple trace s approximates s′ under I' (written s $\preceq_{I'}$ s′) if one of the following conditions holds:

- s and s′ are terminating, len(s)==len(s′), and (**all** i: 0<=i && i<len(s): s $\propto^{i}_{I'}$ s′).
- s′ is magic and (**all** i: 0<=i && i<len(s′): s $\propto^{i}_{I'}$ s′).
- s and s′ are infinite and (**all** i: 0<=i: s $\propto^{i}_{I'}$ s′).

Greybox refinement for method m holds if for every trace of CTest.m$_{\mathsf{ITest}}$ there is a corresponding trace of ITest.m:

$$\begin{aligned} \mathsf{ITest.m} \leq_{I'} \mathsf{CTest.m}_{\mathsf{ITest}} \quad \overset{\text{def}}{=} \quad & \textbf{all}\ \mathsf{t'\!:\ t'}\ \textbf{in}\ \mathsf{tr(CTest.m_{ITest})}\mathsf{:} \\ & \textbf{exists}\ \mathsf{t\!:\ t}\ \textbf{in}\ \mathsf{tr(ITest.m)}\mathsf{:} \\ & \mathsf{(fst(t)} \preceq_{I'} \mathsf{fst(t'))\ \&\&\ snd(t)==snd(t')} \end{aligned}$$

To summarize: This condition requires that CTest.m preserves the invariant, refines its specification (call sequence, state transformation, return value), and establishes the invariant before each external call.

Additional methods. If CTest contains an additional public method n beyond those prescribed by ITest, there must for every simple trace of CTest.n be a finite sequence of concatenateable traces of methods of ITest that approximates the former. For the starting states we distinguish two cases: If n is not prescribed by any interface implemented by CTest and it has the precondition p in the implementation, then the starting states are im(I′, I) && p. If n is prescribed by another interface ITest2 with invariant I$_2$ and state space Σ_2, then I′ is actually a function of type $\Sigma \to \Sigma_2 \to \Gamma \to Bool$. The starting states are {γ | **exists** σ, σ_2: I′(σ, σ_2, γ) && I(σ) && I$_2$(σ_2) && p(σ_2)}.

We define concatenation ($\frown$) of simple traces as follows: Two simple traces u and v can be concatenated if u is terminating and if fst(u[len(u)-1])==fst(v[0]).

In this case we first add the simple traces and then remove the two intermediary non-call states. The concatenated trace belongs to the same kind (terminating, aborted, magic, infinite) as v. We define str(ITest) to be the union of all simple traces of its modification methods —including others (Sect. 6.4)—, and str(ITest)* the transitive closure thereof with respect to concatenation (including the empty trace).

With these definitions we can formally express the refinement condition for additional methods:

$$\begin{array}{lcl} \mathsf{ITest}^* \leq_{I'} \mathsf{CTest.n}_{\mathsf{ITest}} & \overset{\text{def}}{=} & (\textbf{all}\ \mathsf{s}'\text{: } \mathsf{s}'\ \textbf{in}\ \mathsf{str}(\mathsf{CTest.n}_{\mathsf{ITest}})\text{:} \\ & & \textbf{exists}\ \mathsf{s}\text{: } \mathsf{s}\ \textbf{in}\ \mathsf{str}(\mathsf{ITest})^*\text{: } \mathsf{s} \preceq_{I'} \mathsf{s}') \end{array}$$

Note that for additional methods there is no equivalence criteria for parameters or return values.

Constructors. The conditions for constructors are analogous to those for methods. The initial states are arbitrary. For simplicity, we consider the initialization in the field declaration as part of every constructor. The condition for prescribed constructor K[3] is as follows:

$$\begin{array}{lcl} \mathsf{ITest.K} \leq_{I'} \mathsf{CTest.K}_{\mathsf{ITest}} & \overset{\text{def}}{=} & \textbf{all}\ \mathsf{t}'\text{: } \mathsf{t}'\ \textbf{in}\ \mathsf{tr}(\mathsf{CTest.K}_{\mathsf{ITest}})\text{:} \\ & & \textbf{exists}\ \mathsf{t}\text{: } \mathsf{t}\ \textbf{in}\ \mathsf{tr}(\mathsf{ITest.K})\text{:} \\ & & \mathsf{fst(t)} \preceq_{I'} \mathsf{fst(t')}\ \&\&\ \mathsf{snd(t)}{=}{=}\mathsf{snd(t')} \end{array}$$

Additional constructors are like additional methods, except that the first simple trace of the concatenated sequence must be that of a constructor. We define $\mathsf{str}(\mathsf{I}\widehat{\mathsf{Test}})$ to be the union of all simple traces of the constructors of ITest. The rule for additional constructor L then becomes:

$$\begin{array}{lcl} \mathsf{ITest}^* \leq_{I'} \mathsf{CTest.L}_{\mathsf{ITest}} & \overset{\text{def}}{=} & (\textbf{all}\ \mathsf{s}'\text{: } \mathsf{s}'\ \textbf{in}\ \mathsf{str}(\mathsf{CTest.L}_{\mathsf{ITest}})\text{: } \textbf{exists}\ \mathsf{s}\text{:} \\ & & \mathsf{s}\ \textbf{in}\ (\mathsf{str}(\mathsf{I}\widehat{\mathsf{Test}})^\frown \mathsf{str}(\mathsf{ITest})^*)\text{: } \mathsf{s} \preceq_{I'} \mathsf{s}') \end{array}$$

7 Refinement Proofs in Practice

The above trace refinement rules are difficult to apply directly. In cases where the mandatory external calls in the specification and the implementation are embedded in similar structures (loops, conditionals), we can use simpler data refinement in context rules for corresponding blocks (Fig. 15). Of course, the blocks between the external calls (S, T, S′, and T′) may contain additional structure, but this can be ignored for our purposes.

[3] In most languages, constructors have the same name as the containing classes. To avoid overloading, we use different identifiers.

7.1 Data refinement

We review some fundamentals of the weakest preconditions and refinement following [7] and of data refinement following [5].

Weakest precondition. For statement S and predicate q, $wp(S,q)$ denotes Dijkstra's weakest precondition, that is the set of states from which S is guaranteed to terminate in q. For S and predicate p the strongest postcondition $sp(S, p)$ denotes the smallest set of states in which S may terminate if started from p. Formally the strongest postcondition is defined as:

$$sp(S, p) \overset{\text{def}}{=} (\cap\ q : p \subseteq wp(S, q) : q)$$

Assert and guard. The assert statement **assert** p skips if the boolean expression p holds and aborts otherwise. The guard statement is the dual of the assert. For predicate p, **guard** p skips if p holds and magically establishes any postcondition if p does not hold.

Algorithmic refinement. Statement S' refines statement S, written $S \sqsubseteq S'$, if it establishes any postcondition q from any state where S establishes it:

$$S \sqsubseteq S' \overset{\text{def}}{=} \textbf{all}\ q\text{: } wp(S, q) \subseteq wp(S', q)$$

Data refinement. Data refinement is a general technique to change the data representation in a refinement. For relation $R : \Sigma \leftrightarrow \Gamma$ let $[R]$ denote a nondeterministic relational update, that is a statement from Σ to Γ such that the states are related by R. It is the same as $\textbf{any}(\gamma'; R(\sigma, \gamma'))\ \{\gamma = \gamma';\}$ except that it also changes the state space. Statement S' data refines statement S under relation R, written $S \sqsubseteq_R S'$:

$$S \sqsubseteq_R S' \overset{\text{def}}{=} S; [R] \sqsubseteq [R]; S'$$

7.2 Piecewise data refinement in context

In case of structural similarity (Fig. 15), we can establish greybox refinement by proving data refinement in context of the corresponding blocks, the parameters of external calls, and of the result values. Let S, S', T, and T' be statements without any calls to modification operations of mentioned component instances. Let n denote a modification operation, and c, c', r, and r' either variables or constants. Furthermore, let $\_c$, $\_c'$, $\_r$, and $\_r'$ be fresh variables of the same types as their correspondents without the initial underscore.

We consider separately the first block and the following blocks. For each we develop a sequence of increasingly more general, but also more complex rules.

```
interface A {
  X x;
  W w;
  E e; // reference to other object
  invariant I
  int m(Z z) pre p {
    S;
    w=e.n(c);
    T;
    return r;
  }
}
```

```
class B implements A {
  X′ x′;
  W w′;
  E e′; // reference to other object
  invariant I′
  int m(Z z) {
    S′;
    w′=e′.m(c′);
    T′;
    return r′;
  }
}
```

Figure 15: Structure-Preserving Refinement

Thus, if one of the stronger rules holds, they are often easier to show, especially if done informally.

First block. The first block S including the value of the method parameter has to be data refined by S':

$$\mathsf{S};\ \_\mathsf{c}=\mathsf{c} \sqsubseteq_{\mathsf{I}'\ \&\&\ \_\mathsf{c}==\_\mathsf{c}'} \mathsf{S}';\ \_\mathsf{c}'=\mathsf{c}' \tag{1a}$$

The assignments to $\_\mathsf{c}$ and $\_\mathsf{c}'$ and the extension of the refinement relation guarantee that the values of the actual parameters of the method calls $\mathsf{e.n}$ and $\mathsf{e'.n}$ are the same.

This condition is sufficient, but too strong. We only require data refinement in a context where the invariant I and the precondition p hold. We express this context with an **assert** statement:

$$\textbf{assert}\ \mathsf{I}\ \&\&\ \mathsf{p};\ \mathsf{S};\ \_\mathsf{c}=\mathsf{c} \sqsubseteq_{\mathsf{I}'\ \&\&\ \_\mathsf{c}==\_\mathsf{c}'} \mathsf{S}';\ \_\mathsf{c}'=\mathsf{c}' \tag{1b}$$

That we only consider contexts where I' holds is already determined by the refinement relation.

Following blocks. In the rules for the second and the following blocks, it is a bit more complicated to express the minimal context in which data refinement must hold. The first rule does not limit the context:

$$\mathsf{w=e.n(c)};\ \mathsf{T};\ \_\mathsf{r}=\mathsf{r} \sqsubseteq_{\mathsf{I}'\ \&\&\ \_\mathsf{r}==\_\mathsf{r}'} \mathsf{w'=e.n(c')};\ \mathsf{T}';\ \_\mathsf{r}'=\mathsf{r}' \tag{2a}$$

Here we use $\_\mathsf{r}$ and $\_\mathsf{r}'$ to assure refinement of the return values. The preceding method calls $\mathsf{e.n}$, respectively $\mathsf{e'.n}$, are required because the refinement relation is only required to hold after termination of S and S', but not after the method calls.

Condition (2a) is sufficient, but too strong. We only require refinement to hold in contexts that are reachable by executing S in states where $\mathsf{I}\ \&\&\ \mathsf{p}$ holds. This gives us the sharper condition:

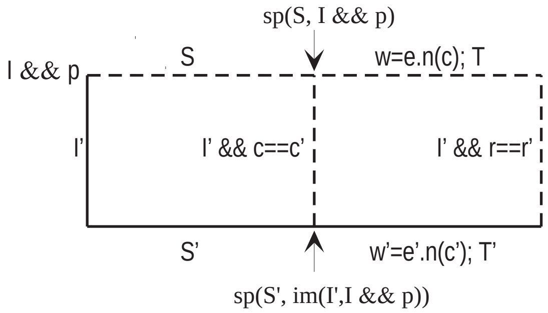

Figure 16: Piecewise Data Refinement

$$\textbf{assert}\ \mathsf{sp(S,\ p\ \&\&\ I);\ w{=}e.n(c);\ T;\ \_r{=}r} \sqsubseteq_{\mathsf{I'\ \&\&\ \_r{=}{=}\_r'}} \mathsf{w'{=}e.n(c');\ T';\ \_r'{=}r'} \quad \text{(2b)}$$

This is the sharpest context we can describe with asserts; using guards we can express an even weaker condition that is still sufficient. Often, $\mathsf{S'}$ is more deterministic and, therefore, gives us more context information than S. Consider the following correct refinement example:

I: true	p: true	S: **any**(int y; true) {x=y;}	T: x=1
I′: x==x′ && e==e′		S′: x′=1	T′: skip

This cannot be proved correct with the above rule, but with the following weaker rule:

$$\begin{array}{l} \mathsf{w{=}e.n(c);\ T;\ \_r{=}r} \sqsubseteq_{\mathsf{I'\ \&\&\ \_r{=}{=}\_r'}} \\ \qquad \textbf{guard}\ \mathsf{sp(S',\ im(I',\ I\ \&\&\ p));\ w'{=}e.n(c');\ T';\ \_r'{=}r'} \end{array} \quad \text{(2c)}$$

This rule is sufficiently complete for most practical applications. A complete rule, requiring additional concepts and notation, as well as as a proof of completeness are beyond the scope of this paper.

Let condition (1) be true if (1a) or (1b) holds and let condition (2) be true if (2a), (2b), or (2c) holds. Then we get the following theorem:

Theorem 1 (Soundness of piecewise data refinement) *If conditions (1) and (2) hold, then* $\mathsf{A.m} \leq_{I'} \mathsf{B.m}$.

Figure 16 shows this piecewise data refinement in context, where solid lines mean ‘for every choice’ and dashed lines ‘there exists a choice’. The strongest postcondition expressions are remarks to illustrate conditions (2b) and (2c).

Insufficient conditions. To sharpen the intuition, we also list the following two alternatives for condition (2) that are sometimes wrongfully believed to be sufficient. In the first case we require data refinement of the complete method:

$$\mathsf{S;\ w{=}e.n(c);\ T;\ \_r{=}r} \sqsubseteq_{\mathsf{I'\ \&\&\ \_r{==}\_r'}} \mathsf{S';\ w'{=}e.n(c');\ T';\ \_r'{=}r'} \tag{f1}$$

The following counterexample shows that the above condition (f1) together with (1) is insufficient. The conditions hold, but greybox refinement doesn't because <0, 1> is not a legal sequence of states for x.

I: true	p: true	S: **any**(int y; true) {x=y;}	T: skip
I′: x==x′ && e==e′		S′: x′=0	T′: x′=1

The second insufficient replacement for condition (2) is 'derived' from the layered specification pattern (Fig. 8). It says that partTwo, standing for T, must be data refined by partTwo′, that is T′. This condition, $\mathsf{T} \sqsubseteq_{\mathsf{I'}} \mathsf{T'}$, as well as condition (1) hold in the following example: p: true and

I: true	S: skip	w=e.n(c): w=1	T: w=0; x=1
I′: w==w′ && w′==0 && x==x′	S′: skip	w′=e′.n(c′): w′=1	T′: w=0

However, trace refinement does not hold. The problem of this condition is that it wrongfully assumes I′ to hold *after* the call to e.n. If we impose and prove the additional, unnecessarily restricting consistency requirement on specifications that the invariants also hold after the external method calls, then we can use this rule. For layered specifications (Sect. 3) this means that we can produce a correct implementation by implementing —or if we write the specification as an abstract class inheriting— the whitebox layer unchanged and proving data refinement in context (without any calls) of the blackboxes.

Sufficient conditions for additional methods and for constructors using data refinement in context follow the same pattern as conditions (1) and (2).

8 Towards a Greybox Specification Language

In this paper we have used invariants, preconditions, specification statements, abstract data types, and loops over sets as extensions to Java for formulating component contracts. The final definition of a greybox specification language is subject to future research. To get some feedback what constructs such a language needs to embrace, we have conducted a number of small to medium sized case studies within our group. The most notable example [48] is the specification of a part of the text subsystem of the commercial BlackBox Component Framework [38]. The language used for this was an extension of Component Pascal [39], the implementation language of the BlackBox Component Framework. The greybox specification was shown to be consistent with the original blackbox specification provided with the product.

There are two reasons why it is important to fix a greybox language and not just use ad-hoc notations. First, tool support can only be provided for a clearly defined language. Second, ad-hoc notations are subject to different interpretations. Below we summarize the specification extensions used in our case studies:

- Invariants and method preconditions with universal and existential quantifications (Sects. 2.2, 4).

- Fields, private members (methods, fields), and constructors in interfaces (Sects. 2.2, 4).

- Modifier **inquiry** for inquiry only methods (Sect. 4).

- Special method others to provide for more modifications in additional methods (Sect. 6.4).

- Abstract data types set and sequence of objects together with the common operations (Sect. 2.2).

- Loop (**do**) to iterate over sets and sequences (Sect. 4).

- Specification statement **any** (Sect. 4).

- Non-deterministic control structure. The statement **choose** $\{\mathsf{S}_1 \mid \mathsf{S}_2 \mid \ldots \mid \mathsf{S}_n\}$ nondeterministically chooses one of the S_i's and executes it.

9 Related Work

In this section, we discuss whether and how other specification methods can be utilized to (1) specify both state transformations and mandatory calls and (2) prove refinement of both aspects in implementations.

Pre/post specifications. As discussed, methods that are based on pre/post specifications (without an explicit encoding of the external call trace) cannot specify mandatory external calls that the component must make. This restriction also applies to Meyer's design by contract [29, 31] and the Java Modeling Language (JML) [27], although they are especially targeted at component-based development, respectively a language for this paradigm. As pointed out (Sect. 3), we can express layered specifications with notations, such as Eiffel, that express both white- and blackbox specifications. However, none of these approaches comes equipped with refinement rules that also preserve the external call sequence.

We are not aware of any pre/post specification-based methods that actually encode the call sequence with the respective states into trace variables (Sect. 2.3.1) to achieve the same expressiveness as greybox specifications.

B, VDM, and Z. The B method [1] uses a combination of preconditions and abstract statement sequences to specify operations. The operation bodies can contain calls to operations of imported modules. However, in refinement steps only the overall state transformation, but not the external call sequence needs to be preserved. Thus, B does not solve the problem at hand. Related methods such as Z

[46], VDM [24], VDM++ [14], and RAISE [42] do not give a better grip on the problem.

Class refinement. Mikhajlova and Sekerinski [34, 33, 35] also use a refinement calculus-based extension of Java with nondeterministic constructs and abstract data types for their treatment of class refinement. Although possible in their framework, they do not consider mandatory external calls, but only concentrate on local state transformations. Implementation correctness is based on data refinement of state transformers only. Because they equal non-termination with abortion, they do not have a practically useful treatment of methods that make infinitely many external calls either. Their condition for additional methods is weaker than ours: Additional methods must simply preserve the strongest invariant implied by existing methods; thus, preserve absolute rather than relative (from the current) state reachability as we demand. This would not be a sensible option for greybox refinement because no refinement of external call sequences could be demanded. Relative reachability is mentioned as a proof technique in [33]. No provisions, such as our special others method, are made for additional methods to perform supplementary modifications.

Class refinement has also been studied under the name behavioral subtyping in less formal settings guaranteeing only partial correctness by America [2] and by Liskov and Wing [28]. However, because pre/post specifications are used these approaches are not suitable for the problem at hand either.

Refinement calculi. As exemplified by our own proposal, the refinement calculi of Back [4, 7] and Morgan [36] with their abstract statement notation can be used to specify external calls. However, their refinement rules only take the state transformations, but not the calls, into account.

Contracts of Helm, Holland, and Gangopadhyay. Helm et al. define a mostly syntactic notion of interaction contracts [20, 21] for the object-oriented design of components. External calls can be explicitly specified. However, lacking the distinction between modification and enquiry operations, all specified calls are mandatory. As a consequence, not only the call to deleteNotification in ITextModel.deleteCharAt, but also the call corresponding to our getCharAt in ITextObserver.deleteNotification is explicitly mentioned to be mandatory in their treatment of the observer pattern.

Call sequences can be specified using sequential and parallel composition as well as conditionals. The local state change is indicated by a combination of postcondition and place where the modification should be executed:

```
SetValue(Value val) {Δvalue; Notify();} [value==val]
```

This means that first, indicated by the Δvalue, value should be set so as to satisfy the postcondition, that is to val. This notation does not work if several changes should be made. For example, the following specification, presented in our notation, can not be expressed in theirs:

```
SetTwoValues(Value val1, Value val2) {value=val1; Notify(); value=val2;}
```

Hence, their notation cannot satisfactorily be used to express the states in which external calls have to be made. Furthermore, no operation preconditions can be expressed.

Holland's thesis [21] gives part of an operational semantics for an object-oriented programming language and for interaction contracts. However, no semantic reasoning is done. There is a notion of contract refinement to design specialized contracts; however, no clear semantic conditions are listed and the examples are such that not even the state transformation aspect can be captured by any standard notion of semantic refinement. No conditions for the correct implementation of specified call sequences and state transformations is given. The notion of contract refinement is not applicable for proving implementation correctness, because there is a fundamental dichotomy between contracts (specifications) and class implementations in their work. The OOram method [43] partly expands on these ideas, but does not solve the problems discussed in this paper.

UML. Different kinds of diagrams from the Unified Modeling Language (UML) [45] can be used to specify both state changes and call sequences. Sequence and collaboration diagrams —collectively called interaction diagrams— show interactions of fixed sets of objects, including the messages sent among them. The more common instance form describes one actual sequence of message interchanges; thus, it is not appropriate for general specifications. On the other hand, the generic form describes all possible sequences using loops and branches. Using loops, we can also indicate messages sent to an a priori unknown set of objects, such as our observers. There is no notation to distinguish between mandatory and optional calls. The main focus of interaction diagrams are the possible message sequences. State changes can be indicated by placing a copy of an object icon showing those modifications. No invariants or operation preconditions can be expressed in collaboration diagrams. Thus no consistency check (Sect. 4.1) is possible. Figure 17 shows the collaboration diagram approximating ITextModel.deleteCharAt.

In our experience, interaction diagrams that make use of loops, branches, and object icon duplication to express greybox specifications quickly become crowded and unreadable. We are not aware of any use of interaction diagrams in the sense of greybox specifications. UML has no formal semantics. Furthermore, it lacks a notion of refinement and, therefore, cannot be used to assert the correctness of implementations based on UML diagrams. The object message sequence chart [10] and the message sequence chart notations [22], from which UML sequence diagrams are derived, have the same limitations.

Activity diagrams can also be used to model operations. They give flowchart-like representations as used in visual programming languages. However, even the

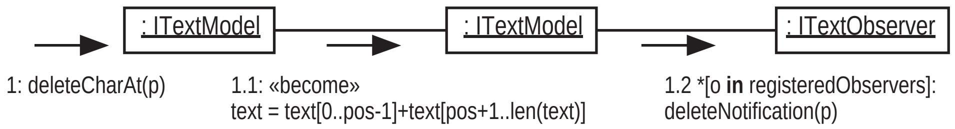

Figure 17: Collaboration Diagram for ITextModel.deleteCharAt

principal authors of UML admitted that this is usually more cumbersome than a textual representation [8]. Furthermore, there is no clear notation to indicate external calls and activity diagrams lack both a formal semantics and refinement rules.

Catalysis. Catalysis [16], a method specifically targeted at the development of components and frameworks, contains a several possibilities to indicate what external calls must be made during the execution of a method. For most cases, the preferred way is to use UML statechart diagrams. Statecharts show state machines that emphasize the flow of control from state to state. Although state changes, external calls, conditionals, and loops can all be encoded in the Catalysis version of statecharts, they are really meant for higher levels of abstraction and, therefore, quite cumbersome to use for greybox-like specifications. Catalysis does not provide a clear semantics and refinement rules that preserve all relevant aspects.

Another option, time indexes provide a way to refer to refer to values of variables at different times. For example, x@j==x@i +3 in a postcondition expresses that the value of x at time j must equal to the value of x+3 at time i. This, in our opinion cumbersome encoding, suffers from the same semantic problem in conjunction with methods that refer to global state as the primed notation in Sec. 2.2.

Sequence expressions can express sequencing constraints on external method calls using sequential composition, alternative, arbitrary iteration, and concurrency, but no conditionals. Time indexes can be used for parameters of calls in message sequences, but it is impossible to indicate in which states external calls have to be made.

Catalysis differentiates between optional and mandatory calls. Unlike in our approach where this distinction is based on the kind of the called method, Catalysis lets the call specifier optionally indicate mandatoryness. Catalysis lacks a formal semantics and has only vague informal refinement rules for asserting the correctness of an implementation.

No other surveyed method, such as Fusion and OOAD, gives a better grip on the problem.

Algebraic specifications. Algebraic specifications suffer from the same deficiencies as pre/post specifications. Consider the typical stack example. For stack s and element e, s==pop(put(s, e)). Here we cannot specify what external method calls methods pop and put must make.

10 Conclusions

Specification approaches that only relate the state prior to operation invocation to the state after operation termination are insufficient to cope with call-backs in extensible systems: The sequence of external calls and the respective states in which the latter must be made cannot be specified. An encoding with auxiliary trace variables could theoretically solve this problem, but in practice this would be very complex and almost unreadable.

Specifications that only relate pre- and post-operational states are called blackbox. As the other extreme, whitebox specifications contain all implementation details which often makes them too restrictive. On the middle ground there are two possibilities. In a discrete combination of the concepts one can layer a whitebox on blackboxes. On a continous scale, we recommend a new method which we call greybox.

To specify external calls, we proposed component interface specifications to draw on abstract programs rather than on pure blackbox views. Formally, this approach has a sound basis in the refinement calculus. Practically, abstract programs are very close to the programmers' intuition. To increase acceptability further, we recommend to define a greybox specification language as a natural extension of an implementation language. In this paper we used such an extension of Java.

Finally, we have given refinement rules for establishing the correctness of implementations with respect to specifications. These rules can be used for fully formal reasoning, but also give an intuition for informal justifications.

Greybox specifications also have a number of 'soft' advantages [3]: They are (usually) shorter than source code, tend to be more readable than large postconditions —even without trace encoding—, scale better, and allow to indicate enough detail for resource-efficient reuse. Here, we have on purpose not discussed these advantages in order not to distract from the fundamental problem solved by greybox specifications.

Acknowledgments. We would like to thank Ralph Back, Dominik Gruntz, Cuno Pfister, Clemens Szyperski, Anna Mikhajlova, and Leonid Mikhajlov for a number of fruitful discussions.

References

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

[2] Pierre America. Designing an object-oriented programming language with behavioral subtyping. In *Foundations of Object-Oriented Languages, REX School/Workshop*, Lecture Notes in Computer Science 489, pages 60–90, 1991.

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

[4] Ralph Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Center Tracts*. Mathematical Centre, Amsterdam, 1980.

[5] Ralph Back. Changing data representation in the refinement calculus. In *21st Hawaii International Conference on System Sciences*. IEEE Press, 1989.

[6] Ralph Back and Joackim von Wright. Trace refinement of action systems. In *CONCUR 94*, pages 367–384. LNCS 836, Springer Verlag, 1994.

[7] Ralph Back and Joackim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer Verlag, 1998.

[8] Grady Booch, Jim Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[9] Martin Büchi and Emil Sekerinski. Formal methods for component software: The refinement calculus perspective. In Wolfgang Weck, Jan Bosch, and Clemens Szyperski, editors, *Proceedings of the Second Workshop on Component-Oriented Programming (WCOP)*, volume 5 of *TUCS General Publication*, pages 23–32, Short version in ECOOP'97 workshop reader LNCS 1357, June 1997. http://www.abo.fi/˜mbuechi/publications/FMforCS.html.

[10] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture – A System of Patterns*. John Wiley & Sons, 1996.

[11] Marsha Chechik and John Gannon. Automatic analysis of consistency between requirements and designs. In *Proceedings of COMPASS'95*, pages 123–132, 1995. http://www.cs.utoronto.ca/˜chechik/.

[12] D.D. Clark. Structuring a system using up-calls. In *Proceedings of the 10th ACM Symposium on Operating System Principles (SOSP)*, ACM Operating System Review, 19(5), pages 171–180, 1985.

[13] Derek Coleman et al. *Object-Oriented Development: The Fusion Method*. Prentice Hall, 1994.

[14] 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.

[15] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical report, Compaq SRC Research Report 159, 1998.

[16] Desmond F. D'Souza and Alan Cameron Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison Wesley, 1998. http://www.catalysis.org.

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

[18] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.

[19] Object Management Group. The common object request broker: Architecture and specification, 1997. Revision 2.0, formal document 97-02-25, http://www.omg.org.

[20] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *Proceedings of OOPSLA/ECOOP '90*, pages 169–180, 1990.

[21] Ian M. Holland. *The Design and Representation of Object-Oriented Components*. PhD thesis, Northeastern University, 1993.

[22] International Telecommunication Union. Z.120: Message sequence chart (MSC), October 1992. http://www.itu.int.

[23] R. Janicki, D.L. Parnas, and J. Zucker. Tabular representations in relational documents. In C. Brink, W. Kahl, and G. Schmidt, editors, *Relational Methods in Computer Science (Advances in Computing Science)*, chapter 11, pages 184–196. Springer Verlag, 1997. Also as CRL Report 313, McMaster University.

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

[25] G.E. Krasner and S.T. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, 1988.

[26] 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.

[27] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06d, Iowa State University, Department of Computer Science, April 1999.

[28] Barbara H. Liskov and Jeanette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[29] 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.

[30] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, second edition, 1992.

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

[32] Leonid Mikhajlov, Emil Sekerinski, and Linas Laibinis. Developing components in the presence of re-entrance. In *Proceedings of Formal Methods 99*. LNCS, Springer Verlag, September 1999. http://www.tucs.abo.fi/publications/techreports/TR239.html.

[33] Anna Mikhajlova. *Ensuring Correctness of Object and Component Systems*. PhD thesis, Turku Centre for Computer Science, 1999. http://www.tucs.fi.

[34] Anna Mikhajlova and Emil Sekerinski. Class refinement and interface refinement in object-oriented programs. In *Proceedings of FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, pages 82–101. LNCS 1313, Springer Verlag, 1997.

[35] Anna Mikhajlova and Emil Sekerinski. Ensuring correctness of Java frameworks: A formal look at JCF. Technical Report 250, Turku Center for Computer Science, March 1999. http://www.tucs.fi.

[36] Carroll Morgan. *Programming from Specifications*. Prentice Hall, second edition, 1994.

[37] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall Series in Innovative Technology, 1991.

[38] Oberon microsystems, Inc. BlackBox Component Builder, 1997. http://www.oberon.ch/.

[39] Oberon microsystems, Inc. Component Pascal, 1997. http://www.oberon.ch/docu/component_pascal.html.

[40] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[41] David Lorge Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, May 1972.

[42] The RAISE Language Group. *The RAISE Specification Language*. Prentice Hall, 1992.

[43] Trygve Reenskaug, Per Wold, and Odd Arild Lehne. *Working with Objects: The OOram Software Engineering Method*. Manning, 1995.

[44] Dale Rogerson. *Inside COM*. Microsoft Press, 1996. See also http://www.microsoft.com/com/.

[45] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[46] J.M. Spivey. *The Z Notation*. Prentice Hall, second edition, 1992.

[47] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, third edition, 1997.

[48] Petri Suni. Grey-box specification seems to work — a case study. LuK-tutkielma, University of Turku, Department of Computer Science, 1999.

[49] Clemens A. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1997.

[50] S. Tucker Taft and Robert A. Duff, editors. *Ada 95 Reference Manual: Language and Standard Libraries (International Standard ISO/IEC 8652:1995(E))*. LNCS 1246, Springer Verlag, 1997.

[51] Niklaus Wirth. *Programming in Modula-2*. Springer Verlag, 1982.

[52] Niklaus Wirth. The programming language Oberon. *Software – Practice and Experience*, 18(7):671–690, 1988.

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

http://www.tucs.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**