

ISpec: Towards Practical and Sound Interface Specifications

Hans B.M. Jonkers

Philips Research Laboratories Eindhoven,
Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands
hans.jonkers@philips.com

Abstract. This paper introduces the ISpec approach to interface specification. ISpec supports the development of interface specifications at various levels of formality and detail in a way compatible with object-oriented modelling techniques (UML). The incremental nature of the levels and the underlying formal framework of ISpec allow informal interface specifications to be made formal in steps. The body of the paper consists of a discussion of the main characteristics of ISpec, which reflect the important decisions taken in the design of ISpec. The idea of component-based specifications and specification plug-ins for constructing heterogeneous specifications is discussed and a small example showing the various levels of specification supported by ISpec is presented.

1 Introduction

1.1 The Role of Interfaces

Interfaces play a role of increasing importance in modern software development. One reason for this is the current trend towards the use of *open systems*. The SEI definition of open system [15] clearly indicates why interfaces, and in particular interface *specifications*, are essential in open systems:

“An open system is a collection of interacting software, hardware, and human components

- designed to satisfy stated needs
- with interface specifications of its components that are
 - fully defined
 - available to the public
 - maintained according to group consensus
- in which the implementations of the components conform to the interface specifications”

Another reason for the increasing importance of interfaces is the growing use of component-based software development techniques [19]. This trend is supported by the availability of standard component models such as COM, CORBA and Java Beans. Components use interfaces to make their context-dependencies explicit. This use of interfaces as a decoupling mechanism can be seen as the crux of component-based software development.

1.2 Objectives of ISpec

ISpec is an interface specification approach developed by Philips Research with the aim of providing a systematic approach to interface specification that

1. supports *multiple levels of formality and detail*;
2. fits in with main-stream *object-oriented modelling techniques* (UML);
3. has a solid footing in *formal techniques*;
4. provides a basis for *systematic testing*.

These objectives are based on our past and recent experience with the use of formal and informal specification techniques in several product divisions of Philips; see e.g. [5,13]. As many formal methodists have learned the hard way, formal techniques are considered a problem rather than a solution by the average software developer in industry. We therefore aim at an approach that can also be used informally and with varying degrees of detail, rather than an all-or-nothing formal approach (objective 1).

Despite the aversion to formal techniques displayed by many industrial software developers, there is a general feeling that better techniques for controlling the complexity of software are required. The growing popularity of object-oriented modelling techniques, and in particular UML, can be seen as a sign of this. We consider it important to fit in with this trend (objective 2), not just because it is opportune to do so but above all because the idea of object-oriented modelling and the formal-methods idea of ‘abstract modelling’ are closely related (see Section 2.4).

The unbridled use of informal techniques can easily lead to unstructured and inconsistent specifications. This can be avoided by choosing an appropriate formal framework as the basis of the approach and structuring specifications in such a way that the various parts of a specification, whether informal or formal, can be related unambiguously to parts of that framework (objective 3). Hence, *if necessary*, an ISpec interface specification can be made ‘completely’ formal. We put ‘completely’ between quotes because there are always aspects of interfaces that can only be expressed informally.

Another reason why it is important to have a formal basis is to be able to provide semantic tool support. The intention is to support the automatic generation of black-box tests from an ISpec specification (objective 4). The price to be paid for this is that the specification, or at least part of it, has to be made formal. Though this price may be high, there are potentially large benefits to be gained in the testing phase. We do not consider the practical use of proof tools a viable option yet, though there could be special situations in which it is.

ISpec focusses on the specification of interfaces as encountered in component technologies such as COM and CORBA. Compared with classical APIs, such as Win32, OpenGL, etc. which are monolithic and static, these interfaces are typically small and dynamic. Furthermore, we generally have to deal with a group of mutually related interfaces, a so-called *interface suite*, rather than a single interface. This focus on component interfaces is not really a restriction since a classical API can be seen as a special case of an interface suite (with one element).

2 Main Features of ISpec

2.1 Contractual View of Interface Specifications

The idea of using a contractual paradigm in software development has been advocated by many researchers, in particular by Meyer [12]. The contractual paradigm is fundamental to the ISpec approach. An interface specification in ISpec is viewed as an *unsigned multi-party contract* between *providers* of services and *users* of services, as will be further explained below. It contains, among other things:

- An identification of the *services*, *interfaces* and *parties* involved in the contract. This serves primarily to introduce *names* and *types* for the services, interfaces and parties, leaving the entities themselves as yet undefined.
- A definition of the relevant *terms* and *concepts*. This can be seen as an *information model* defining the necessary vocabulary for the contract. In practice, the definitions may be drawn from a more comprehensive *domain model*.
- A definition of the *rights* and *duties* of each party. In specifications these rights and duties are formulated using terms such as: *requires* and *provides*, *relies* and *guarantees*, *assumes* and *ensures*, *precondition* and *postcondition*, etc.

The contract is *multi-party* because several parties may be involved in the contract, and not just one or two. This is due to the fact that component interfaces, unlike monolithic APIs, are generally small and dedicated to a particular aspect of a service. The overall service is defined by a *collection* of interfaces, also called an *interface suite*. Because of mutual dependencies, the interfaces in a suite cannot be separately specified. Since each interface has at least a provider and a user, this leads to multiple parties in the contract. An example of an interface suite is the collection of ‘connection point’ interfaces from Microsoft COM, defining a generic notification service. The interfaces and parties, and their main dependencies, are indicated in the UML class diagram in Figure 1.

The contract is *unsigned* in the sense that the parties identified in an interface specification are *abstract*, i.e. they are *roles* and not physical entities. In ISpec specifications these parties are represented as *abstract object classes*. In reality, the roles are played by *concrete* parties that *implement* (their part of) the contract. These concrete parties are usually defined as object classes in programming languages such as Java and C++. In order to determine whether an interface specification as a whole has been correctly implemented, we must first associate a concrete party with each abstract party identified in the contract. That is, the contract must be *signed*; see Figure 2.

Note that a concrete party can play *several roles*, as a provider as well as a user of interfaces, and that it can be involved in *several contracts*. Note also that the provider and the user of an interface are treated in the same way in ISpec in the sense that *both* may be subject to specification constraints, and not just the provider of the interface.

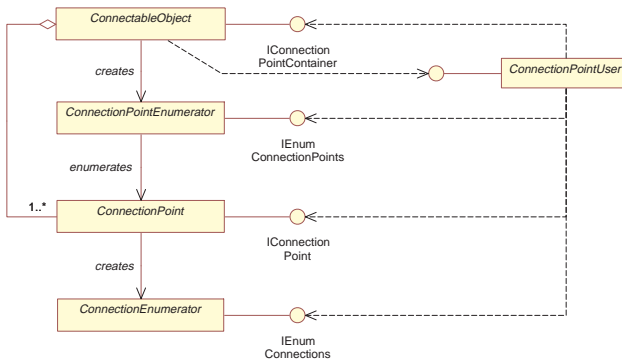


Fig. 1. Microsoft COM's Connection Point Interface Suite

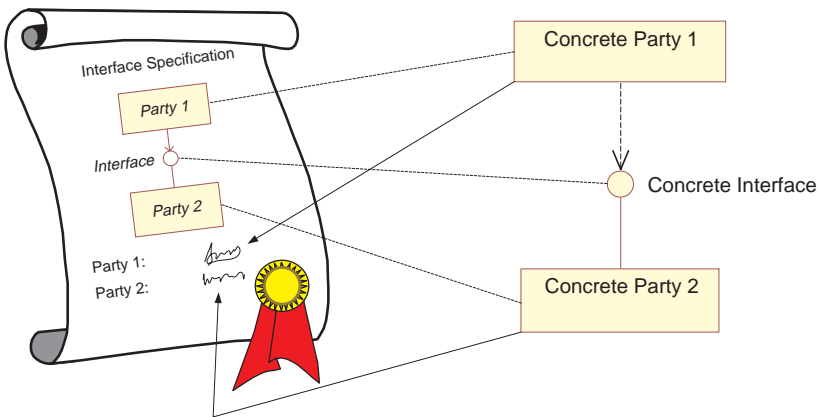


Fig. 2. Signing the Contract

2.2 Incremental Levels of Detail and Formality

One of the explicit objectives of ISpec is to support interface specifications at different levels of detail and formality. In order to enable a smooth transition from one level to the next more detailed or formal level, these levels have been chosen so that they are *incremental*. This implies that a level $n + 1$ specification is obtained from a level n specification by *adding* new sections to the specification. The existing sections retain their function, though their contents may be adjusted to the newly added detail. For example, adding a UML class diagram to an informal specification can enable an informal requirement to be made more precise by phrasing it in terms of the entities and attributes occurring in the diagram.

In ISpec six specification levels are identified, as characterised below:

1. *IDL level*: Defines the *signature* of the interfaces in terms of some IDL (Interface Definition Language). Though most IDLs have (very) limited ways of expressing semantic attributes of interfaces, they basically define the syntax of interfaces only.
2. *Summary level*: Adds *operation summaries* to the interfaces, informally and succinctly describing the effect of the operations in the interfaces. This style of specification corresponds more or less to the current state of practice.
3. *Model level*: Adds an *object-oriented model* to the specification, typically in UML, with abstract object classes representing the parties in the contract. The effect of the operations in an interface is described informally in terms of the attributes of the abstract class providing the interface.
4. *Action level*: Adds *action clauses* to the operations, defining the effect of the operations in a pseudo-algorithmic way. This corresponds to an operational style of specification similar to pseudo-code.
5. *Pre & Post level*: Adds *preconditions* and *postconditions* and other declarative elements such as invariants to the operation specifications. The action clauses are still used at this level, but they are usually more abstract than at the action level (see Section 2.5).
6. *Formal level*: Adds *formal text* and turns the interface specification into a complete formal specification. This level of formality is not normally used in practice, unless formal verification or automatic test generation is required.

The above gives a rough description of each level only. A simple example demonstrating the above levels will be presented in Section 4.

2.3 Template-Based Approach

Rather than providing a concrete syntax for interface specifications, ISpec defines a *document structure* which is essentially a tree structure of *templates*, such as templates for specifying data types, classes, operations, etc. These templates are similar to the kinds of templates used in object-oriented methods such as Catalysis [17]. The levels of detail discussed in Section 2.2 correspond to incremental levels in the tree structure.

The ISpec document structure identifies *logical* components of specification documents. The corresponding *physical* documents will generally have a project-dependent structure and will contain additional components not defined by ISpec. These documents may appear in various representations such as MS Word or Rational Rose format. A part of the logical document structure is shown in Figure 3 using UML class diagrams as a kind of abstract syntax.

Informal and formal text occurs at the leaves of the document tree structure only. This raises the immediate question what language to use for the formal text and what the semantics of a specification structured like this is. This issue will be discussed in Section 3.

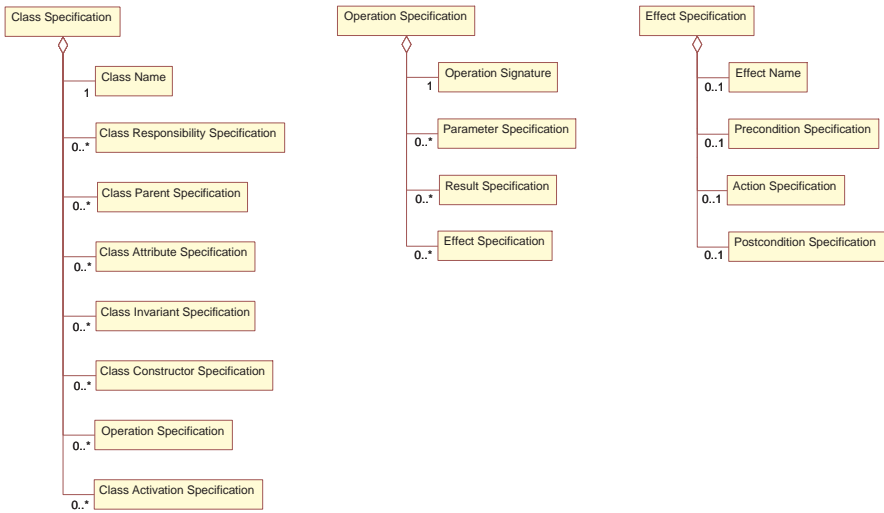


Fig. 3. Part of the ISpec Document Structure

2.4 Abstract Modelling as a Basis

Specifying a system by means of an *abstract model*, also called *abstract modelling* or *model-oriented specification*, amounts to defining an ‘abstract implementation’ of the system. A concrete implementation is said to *conform to* such a specification if its observable behaviour is consistent with that of the abstract implementation.

The abstract model differs from a real implementation in the use of programming variables with abstract types such as sets, relations, functions, etc., and the use of declarative constructs such as pre- and postconditions rather than code to specify operations. One of the first formal methods supporting model-oriented specification was VDM [10].

ISpec uses abstract modelling as the basis of interface specifications. The first reason for this choice is that it constitutes a good compromise between providing a high level of abstraction and providing a high level of intuition. For software developers it is generally easier to think in terms of constructive models than in terms of non-constructive properties, even though, or maybe because, the use of models may introduce implementation bias in specifications. The second reason is that the technique fits in quite well with object-oriented modelling techniques, making it easier to bridge the gap between formal techniques and UML.

An abstract model is defined in an ISpec interface specification by representing each party in the contract by an abstract object class and associating each interface type with the object class representing the provider of the interface. This implies, among other things, that each object class may have zero, one, or more interfaces and that in a real implementation two different abstract ob-

ject classes may correspond to the same concrete class, playing two roles at the same time. For each abstract object class an internal representation is chosen in terms of abstract state variables, and the required and provided behaviour of the parties is specified in terms of the abstract state variables.

The approach described above is very similar to the way interface behaviour would be modelled in UML, but there are some differences. The first is that the current version of ISpec does not use UML's assertion language OCL, mainly because OCL lacks formality and expressivity. Instead, the current version of ISpec uses the assertion language and type mechanism of Z [18], in a way similar to Object-Z [14]. The intention is to make ISpec independent of the assertion language used; see Section 3.

The second difference has to do with the 'impressionistic' way UML is used in practice. UML models are typically defined by drawing lots of diagrams highlighting various aspects of the model without making the model itself explicit; see Figure 4. ISpec is based on an 'expressionistic' way of modelling in the sense that the UML diagrams are used as an *expression of* an explicit model, thus making it a lot easier to guarantee consistency. This raises the question exactly what that model is, an issue that will be addressed in Section 2.6. The above is not to say that the impressionistic approach has no value; both styles of modelling are useful but they serve different purposes (see Section 2.7).

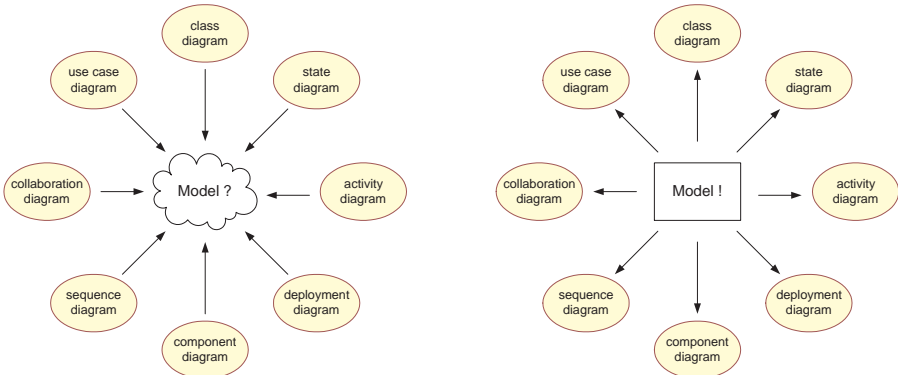


Fig. 4. Impressionistic versus Expressionistic Modelling

2.5 Mixed Declarative/Operational Style

In specifying the effect of operations ISpec uses an extended form of the pre- and postcondition technique based on the extensions described in [11]. An operation specification is split into *effect specifications* specifying the effect of the operation under different circumstances defined by preconditions. An effect specification can be seen as a *mini-contract* between the caller and the callee of an operation. The structure of an effect specification and its interpretation as a mini-contract is indicated in Figure 5.

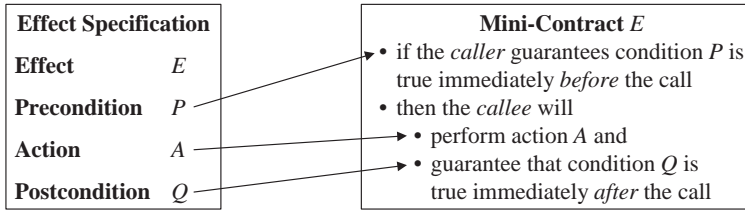


Fig. 5. ISpec Effect Specification Template

The part of the effect specification that is different from a classical pre- and postcondition specification is the *action clause*. The action clause specifies a collection of *allowed* state transitions in an *operational* way, using abstract and often non-deterministic algorithmic constructs. This collection of state transitions acts as a kind of *upper limit* for the nondeterministic behaviour of the operation. The pre- and postcondition clauses are used to further constrain this set of state transitions in a *declarative* way.

The combined use of the action clause and the pre- and postcondition clauses allows an operation to be specified in a mixed declarative/operational style, which is important for a number of reasons. First of all, certain aspects of operations are inherently operational and very hard to specify declaratively. This applies for example to the call-backs performed by an operation. Call-backs are a phenomenon frequently encountered in component-based software. Secondly, some operations can simply be specified more naturally in an abstract algorithmic way than in a purely declarative way. This applies particularly to control-like operations. Finally, the action clause can be generalised in such a way that it allows the specification of non-atomic operations, such as blocking operations and operations with internal interaction points.

The ratio between declarative and operational content in an operation specification can in principle be chosen arbitrarily. One extreme is that the action clause is only used to indicate which variables may be modified by the operation, and the precondition and postcondition are used to define the effect of the operation. This corresponds to the classical use of pre- and postconditions as in VDM-SL [10] or the Larch behavioural interface specification languages [8]. The other extreme is that the action clause is a piece of pure program code and that the precondition and postcondition are omitted. The example below is somewhere in the middle of the spectrum, specifying an operation that increments a counter and notifies the objects in the set *subscribers* after every 100 ticks. Since in this case the operation specification consists of a single effect specification, the effect specification is identified with the operation specification:

```

operation tick()
pre true
action modify counter
    ; if counter = 0 mod 100 then for s : subscribers do s.notify()
post counter = counter' + 1

```

Here the *for*-clause iterates over the elements of a set in an unspecified order. We note that problems with re-entrancy of call-backs, such as `s.notify()` modifying the value of *counter*, can be avoided in ISpec in a simple and natural way. ISpec uses the *closed world assumption* in the sense that (abstract) state variables introduced in an interface specification can be modified by means of the mechanisms identified in that specification only. Any call-back interfaces used by the operations being specified are part of the interface specification and any possible effect of their operations on the state variables should be specified as part of the interface specification. Clearly we do not want `s.notify()` to modify the state variable *counter* of the notifying object, which can be specified in the specification of the call-back interface containing the `notify()` operation.

2.6 Transition Systems as Semantic Model

One of the main objectives of ISpec is to support interface specifications at different levels of formality. In order to maintain consistency between the informal and semi-formal parts of a specification on the one hand, and the formal parts on the other hand, it is essential that both are based on the same semantic model. This means that the semantic model must be both intuitive and formal, which is not an obvious combination. ISpec uses the transition system model [16] as its underlying semantic model, which is believed to satisfy these requirements. This will be further explained and motivated below.

Transition systems have a long-standing reputation as semantic models for concurrent systems. In its basic form a transition system consists of a collection of states, an initial state, a state transition relation defining which state transitions *may* occur, and a set of progress rules defining when state transitions *will* occur. The progress rules can be formulated in various ways, such as very abstractly, based on some notion of *fairness*, or very concretely, based on some notion of real-time *triggering*.

The basic transition system model is very simple and easy to explain. Besides that, it has other advantages, such as:

- it fits in with traditional finite state black-box testing techniques [1];
- it can be used to unify the concepts of callable operation and autonomous transition, leading to a uniform treatment of both;
- it helps in suppressing ‘control-bias’ in specifications by the absence of ‘loci of control’.

The only problem is that the basic transition system model is too austere and unstructured. For specifications of component interfaces we need an object-oriented semantic framework. Fortunately, creating such a framework is mainly a matter of extending the transition system model without affecting the essentials of the model. A short reconstruction of the enhanced transition system model used in ISpec is given below. The enhanced model is obtained by taking the basic transition system model and successively adding the following concepts:

- *Classes*: introduce collections of *objects* and provide a natural way of structuring states, transitions, and progress rules. State is aggregated in the *instance variables* of objects, transitions are aggregated in the *operations* of objects, and progress rules are aggregated in the *activities* of objects.
- *Interfaces*: introduce *information hiding* by defining the external access mechanisms of objects. ISpec interfaces contain operations only and no instance variables, so instance variables are always hidden.
- *Observability*: defines which part of the behaviour of the model is externally observable. This is required to define *conformance*. Observable behaviour is not the same as behaviour observable at the interfaces, because behaviour may also be observable by other means, such as connected hardware represented by certain classes in the model.
- *Factories*: are special classes providing constructors for objects. ISpec classes, like component classes in COM, do not have static methods. Constructors are considered normal operations of objects, creating a chicken-and-egg problem that is solved by factories. Factories are special in only one sense: they do not have explicit constructors.

The concept of abstract model discussed in Section 2.4 should not be confused with that of transition system. Abstract models are *syntactic entities*, like programs, while transition systems are *semantic entities*. The link between the two is that abstract models corresponding to ISpec interface specifications *describe* transition systems. Due to the presence of declarative constructs in abstract models, the transition system described by an abstract model need not be unique. The *meaning* of an ISpec interface specification, therefore, is defined as the *collection* of all transition systems described by the abstract model.

Note that the above requires an adjustment of the notion of conformance. An implementation *conforms* to an ISpec interface specification if its observable behaviour is consistent with the observable behaviour of at least one transition system described by the specification; see Figure 6.

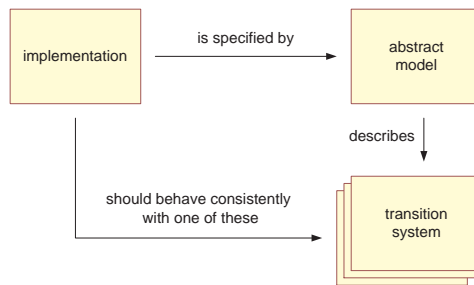


Fig. 6. Conformance to an ISpec Specification

2.7 Separation of Modelling Concerns

ISpec is essentially an approach to developing an interface specification document. Such a document is the result of an interface engineering activity and is usually released at the time the interfaces are released to customers. Besides a contractual function, this document also has an introductory and explanatory function. These two functions require different perspectives in the presentation of the interfaces and are dealt with in separate sections of the document referred to as the *contractual model* and the *conceptual model*.

The purpose of the conceptual model is, among other things, to

- introduce the important *terms* and *concepts* necessary to explain the functionality associated with the interfaces;
- explain the *purpose* and *functionality* of the interfaces;
- provide the link with the interface *requirements*;
- introduce the outlines of the *abstract model*.

In the conceptual model the primary concern is providing the proper *intuition*, rather than accuracy and completeness. The impressionistic style of modelling discussed in Section 2.4 is the most appropriate in providing that intuition, using UML diagrams showing various aspects of the interface functionality and deliberately ‘forgetting’ certain details of the abstract model.

Besides separating the conceptual aspects of interface specifications from the contractual aspects, it also makes sense to separate the *technology-independent* aspects from the *technology-dependent* aspects. By ‘technology’ we mean the programming language and/or component technology in which the interfaces are supposed to be used. The contractual model is therefore split into two parts:

- a technology-independent part referred to as the *specification model*, specifying the interfaces in terms of a neutral Java-like class model;
- a technology-dependent part referred to as the *technical model*, mapping the technology-independent specification to the component technology and/or programming language used. It defines the representation of the interfaces in terms of the technology used and the relation between the technology-independent and the technology-dependent concepts associated with the interfaces.

Note that the technical model can be defined by a generic mapping, requiring only one definition that can be used for all interfaces defined in a project. This is similar to the way some automatic Java to COM interface mappings are defined at the programming language level.

The main advantages of the *technology abstraction* used in the specification model are:

- *Clarity*: Specifying interfaces in terms of technology-oriented concepts clutters up interface specifications, which is now avoided.
- *Reuse*: The same technology-independent specification can be used for various incarnations of the interfaces in, for example, COM, CORBA, Java or C++.

The main document structure of an ISpec interface specification is indicated in Figure 7, showing the separation into conceptual, specification and technical models, as well as other parts of the document structure.

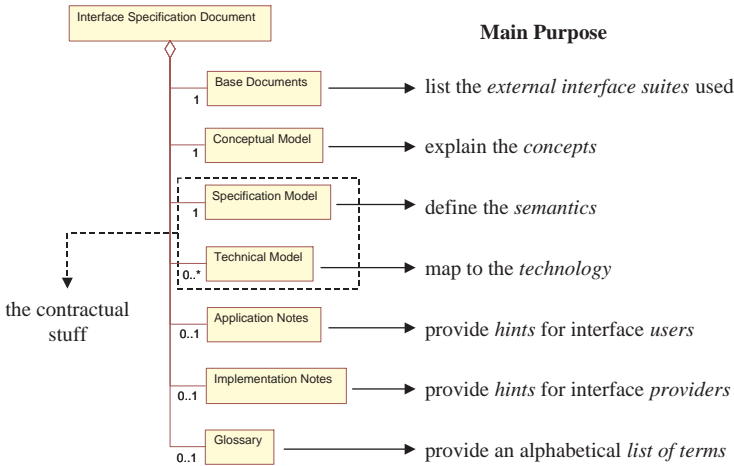


Fig. 7. ISpec Main Document Structure

3 Component-Based Specifications

ISpec is meant to be an *open* approach to the development of interface specifications. This is reflected by, for example, the absence of bias towards a particular component technology or programming language. There is one question, though, that seems inevitable: which specification language should be used for the formal parts of an ISpec interface specification? It is one of the longer term goals of ISpec to turn this question into a non-question and provide openness with respect to the specification language used as well. The key to solving this problem lies in the basic idea of component technology and interfaces itself.

Component technologies with a binary interface standard such as COM are open with respect to the programming language used to implement components. Components can be implemented in any language and which language is used is irrelevant to the users of components. The interfaces shield the details of the implementation language from the users of the components. We can use the same idea at the specification level, as will be explained below.

The issue which specification language to use becomes important at the 'leaves' of the ISpec document structure, at those points where an assertion, expression, etc. is expected. Examples of such leaves are the precondition, action and postcondition clauses discussed in Section 2.5. Ideally we would like to be able to 'plug in' descriptions written in different specification languages at these points;

see Figure 8. Viewing the plug-ins as components and using the analogy with component technology, it follows that this requires the definition of *interfaces* that shield the details of the specification language used in the plug-ins.

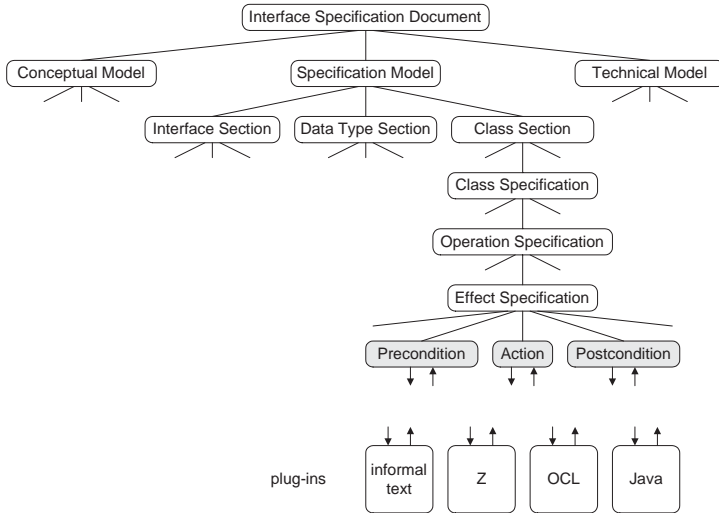


Fig. 8. The Idea of Specification Plug-ins

What should the interfaces of specification plug-ins look like? First of all, the interface of a plug-in should be *bidirectional* since it *provides* information *to* as well as *requires* information *from* the environment into which it is inserted. Since plug-ins are essentially language constructs, such bidirectional interfaces are most conveniently represented by sets of *synthesized* and *inherited attributes* as in attribute grammars.

The attributes in a plug-in interface can be divided into syntactic and semantic attributes. The values of syntactic attributes are syntactic objects such as identifiers, types, etc. The values of semantic attributes are semantic objects such as states, valuations (mappings from names to values), etc.

Each type of plug-in has its own interface specification, defining the names and types of the synthesized and inherited attributes occurring in the interface as well as possible constraints on the values of the attributes. At a point in the document structure at which a plug-in of a particular type is expected, such as a ‘precondition’ or an ‘action clause’, any plug-in inserted at that point should satisfy the interface specification associated with that type.

The component-based specification set-up sketched above leads to a fully compositional approach to the semantics of a multi-language ISpec interface specification. The semantics is defined in terms of the structure of the specification and the semantics of the individual plug-ins occurring in the specification. The semantic definition refers to the attributes in the interfaces of plug-ins only and is thereby fully independent of the specification languages used in the plug-

ins. The semantics of each individual plug-in, on the other hand, is of course determined by the language the plug-in is written in.

The approach described above is still tentative and subject of ongoing research. The current version of ISpec uses a blend of COLD and Z as its formal basis. Essential to the overall approach is the use of a *uniform semantic model* in terms of which expressions from different specification languages can be interpreted and in terms of which the attributes of plug-in interfaces can be defined. The role of this model can be compared with that of the binary standard in COM, although it is of a completely different, semantic, nature. The transition system model discussed in Section 2.6 is the obvious candidate for this model.

4 A Simple Example

4.1 Iterator Interface

In this section we will give a simple example of an interface specification, demonstrating the levels of detail and formality discussed in Section 2.2. We will restrict ourselves to the specification model and (part of) the technical model; see Section 2.7. The example is the specification of the iterator interface `Iiterator`. The service provided through this interface is that of traversing the elements of a finite collection of objects. We distinguish three parties in the contract: *IteratorFactory*, *Iterator* and *IteratorUser*, whose roles are indicated in the UML class diagram shown in Figure 9.

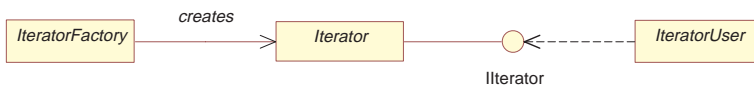


Fig. 9. Iterator Interface and the Parties Involved

4.2 IDL Level

The technology-independent IDL-level specification of the iterator interface and a possible technology-dependent representation of the interface in Microsoft COM IDL are indicated in Figure 10. The former is part of the ‘specification model’ and the latter is part of the ‘technical model’ of the iterator interface. The technical model should, among other things, define what the precise relation between the two representations of the iterator interface is. For a partial answer, resolving the question mark in Figure 10, see [2], Chapter 14. The technical model will be ignored in the rest of the example.

The IDL ‘specification’ of `Iiterator` raises many questions such as

- Should the collection of objects be traversed in a particular order?
- Are duplicates in the collection allowed?

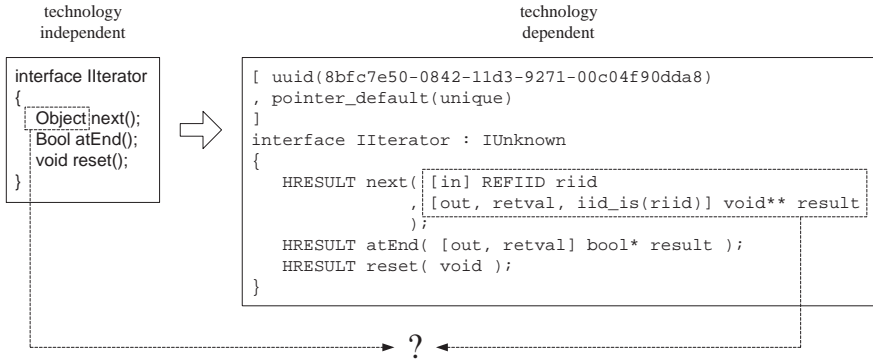


Fig. 10. IDL Level Specification

- What happens when all objects in the collection have been traversed?
- What is the initial state of an iterator?
- How do iterators come into existence?
- What if the collection of objects is modified during the iteration?

It is clear that these questions can only be answered by adding more semantic detail to the specification.

4.3 Summary Level

The summary level introduces a short informal description for each operation. We will not give the complete specification, but restrict ourselves to the operation specifications indicated in Figure 11.

Signature Object next() Summary Returns the next object in the collection.	Signature Bool atEnd() Summary Returns true if all objects in the collection have been traversed, otherwise false.	Signature void reset() Summary Resets the iterator to its initial state.
---	---	---

Fig. 11. Summary Level Specification

Note that the initial state of an iterator, though not specified in the operation summaries, could be specified in another part of the interface specification.

4.4 Model Level

At the model level a UML model is added; see Figure 12. Note that the UML model includes a factory for iterators. It has a single operation that is not part of any interface but is used only to model the creation of iterators and specify their

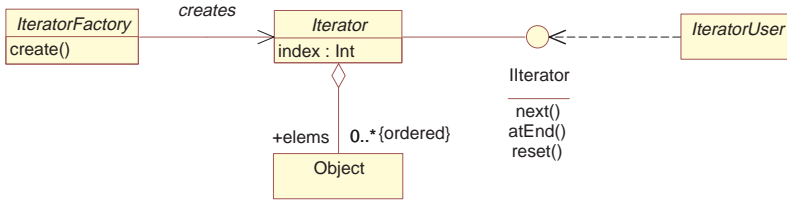


Fig. 12. Iterator Model

required initial state. Whoever creates an iterator should meet the constraints specified for the `create` operation.

The UML model allows the operations to be described in more accurate terms; see Figure 13. The UML ‘role name’ *elems* in the UML model is interpreted as a sequence of objects in the Z sense. The notations $elems(i)$ and $\#elems$ are used as in Z to denote the i -th element of *elems* and the length of *elems*, respectively (sequence indices start at 1).

<p>Signature Iterator create()</p> <p>Summary Returns a new iterator whose <i>index</i> is 0.</p>	<p>Signature Object next()</p> <p>Summary Increments <i>index</i> and returns $elems(index)$ if <i>index</i> is less than the length of <i>elems</i>.</p>	<p>Signature Bool atEnd()</p> <p>Summary Returns <i>true</i> if <i>index</i> is equal to the length of <i>elems</i>, otherwise <i>false</i>.</p>	<p>Signature void reset()</p> <p>Summary Sets the value of <i>index</i> to 0.</p>
---	--	--	---

Fig. 13. Model Level Specification

Note that the summary of `create` does not specify what the value of *elems* is for the newly created object. This is deliberate, because the value of *elems* is determined by the creator of the iterator.

4.5 Action Level

At the action level the effect of operations is specified in ‘structured English’ using constructs such as ‘Let ...’, ‘Set ...’, ‘Return ...’, ‘If ... then ... else ...’, ‘While ... do ...’, etc. Of course, in this simple example there is not much algorithmic detail and no control structures are used; see Figure 14. Bullets and indentation are used to make action clauses more readable. The bullets in action clauses amount to sequential composition and indentation is typically used in control structures to reduce the number of parentheses.

4.6 Pre & Post Level

The pre & post level introduces pre- and postconditions while reducing the role of action clauses. The action clauses are typically used to specify the raw side-effects of operations only; see Figure 15. Bullets and indentation are used in pre-

<p>Signature Iterator create()</p> <p>Summary Returns a new iterator.</p> <p>Action</p> <ul style="list-style-type: none"> • Let <i>it</i> == new Iterator • Set <i>it.index</i> to 0 • Return <i>it</i> 	<p>Signature Object next()</p> <p>Summary Returns the next object in the collection.</p> <p>Action</p> <ul style="list-style-type: none"> • Assert $index < \#elems$ • Increment <i>index</i> • Return <i>elems(index)</i> 	<p>Signature Bool atEnd()</p> <p>Summary Returns <i>true</i> if all objects in the collection have been traversed, otherwise <i>false</i>.</p> <p>Action</p> <ul style="list-style-type: none"> • Return $index = \#elems$ 	<p>Signature void reset()</p> <p>Summary Resets the iterator to its initial state.</p> <p>Action</p> <ul style="list-style-type: none"> • Set <i>index</i> to 0
--	--	---	---

Fig. 14. Action Level Specification

and postconditions in the same way as in action clauses to improve readability, except that a bullet in a pre- or postcondition amounts to conjunction. The standard keyword *result* is used to denote the result returned by an operation. Quotes are used in postconditions to refer to the old values of variables.

<p>Signature Iterator create()</p> <p>Summary Returns a new iterator.</p> <p>Precondition</p> <ul style="list-style-type: none"> • <i>true</i> <p>Action</p> <ul style="list-style-type: none"> • Let <i>it</i> == new Iterator <p>Postcondition</p> <ul style="list-style-type: none"> • $it.index = 0$ • $result = it$ 	<p>Signature Object next()</p> <p>Summary Returns the next object in the collection.</p> <p>Precondition</p> <ul style="list-style-type: none"> • $index < \#elems$ <p>Action</p> <ul style="list-style-type: none"> • Modify <i>index</i> <p>Postcondition</p> <ul style="list-style-type: none"> • $index = index' + 1$ • $result = elems(index)$ 	<p>Signature Bool atEnd()</p> <p>Summary Returns <i>true</i> if all objects in the collection have been traversed, otherwise <i>false</i>.</p> <p>Precondition</p> <ul style="list-style-type: none"> • <i>true</i> <p>Action</p> <p>None</p> <p>Postcondition</p> <ul style="list-style-type: none"> • $result = (index = \#elems)$ 	<p>Signature void reset()</p> <p>Summary Resets the iterator to its initial state.</p> <p>Precondition</p> <ul style="list-style-type: none"> • <i>true</i> <p>Action</p> <ul style="list-style-type: none"> • Modify <i>index</i> <p>Postcondition</p> <ul style="list-style-type: none"> • $index = 0$
---	--	--	---

Fig. 15. Pre & Post Level Specification

Note that object names introduced in a precondition, action or postcondition clause using the ‘Let’ construct, such as *it*, may be used anywhere in the text following the introduction of the name. This way of using object names, discussed in [11], also improves readability by avoiding nesting of expressions. The traditional use of ‘let’ operators, as in VDM or Z, often leads to nested expressions (in order to keep names in scope).

4.7 Formal Level

In turning an ISpec specification into a formal specification we should distinguish between the *structure* of an ISpec specification and the *contents* of that structure, i.e. between the ‘branches’ and the ‘leaves’ of the tree structure defined by the specification. The structure of an ISpec specification is itself already formal. Starting from a pre & post level specification such as the one in Figure 15 there should be no reason to change that structure. The contents of the structure, on the other hand, may be informal or semi-formal. Some parts are meant to stay

informal, such as the summaries of operations, but other parts have to be made formal to turn the specification as a whole into a formal specification with a well-defined semantics.

As discussed in Section 3, our intention is to ultimately support different formal languages at the leaves of an ISpec specification. For now, we will restrict ourselves to the COLD/Z mix used in the prototype ISpec implementation. The transformation to a formal specification is not very interesting in this simple case, but for completeness we give the formal specifications of the operations in ASCII ISpec syntax anyway; see Figure 16. These specifications assume that (in another part of the interface specification) the instance variables of objects of type *Iterator* have been formally declared like this:

```
elems : seq Object
index : Nat
```

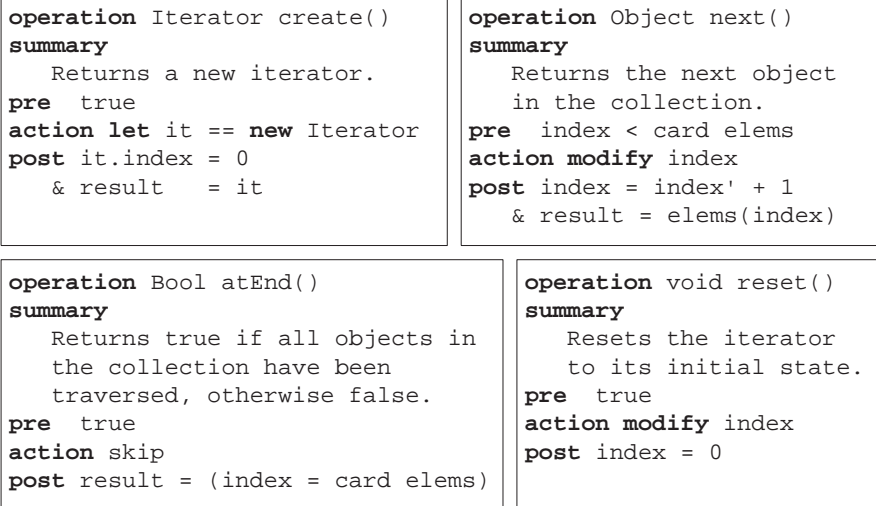


Fig. 16. Formal Level Specification

5 Related Work

ISpec is an eclectic approach. It can be seen as an attempt to put together a number of well-known and proven ideas and techniques to obtain a practical and consistent framework for interface specification. Some of the key inputs to ISpec and related approaches are mentioned below.

ISpec is permeated with the contractual view to interface specification. This view is closely related to the idea of ‘design by contract’ [12], with two main differences. The first is that ISpec does not focus on design but on the contract itself. An ISpec specification *is* a contract; see Section 2.1. The second is that

ISpec contracts are inherently *multi-party* and not just contracts between a caller and a callee (called *mini-contracts* in ISpec), making them very similar to (*design*) *patterns* (cf. [9]).

The multiple levels of formality and detail in ISpec are inspired by the multiple levels of abstraction provided by wide-spectrum languages such as VDM-SL [10] and COLD [4]. For its expression and assertion language, the current version of ISpec uses a combination of language elements from COLD and Z [18].

The separation of modelling concerns as incorporated in ISpec goes back to the ‘modelling perspectives’ introduced in [3]. Following the terminology used in [6], the conceptual and specification perspectives correspond to the conceptual and specification models of ISpec; see Section 2.7. The implementation perspective corresponds partly to the technical model of ISpec because the latter deals only with the representation of interfaces in the implementation technology used.

The approach that comes closest to the model-oriented way interfaces are specified in ISpec is probably Catalysis [17], though Catalysis is a complete software development method, which ISpec is certainly not. The notion of *collaboration framework* in Catalysis corresponds more or less to an interface specification in ISpec, and Catalysis ‘types’ correspond to the abstract object classes associated with interfaces in ISpec. The main difference is that in ISpec the collaborations follow from the specification of the types, while in Catalysis they lead a life of their own. (Referring to the terminology introduced in Section 2.4, Catalysis uses an *impressionistic* style of modelling collaborations while ISpec uses an *expressionistic* style.)

6 Conclusion

In this paper we have given a survey of the main features of ISpec. Several more detailed technical features have not been discussed. They include features dealing with aspects of interface specifications such as exception handling, object lifetime, asynchronous behaviour, non-atomic operations and multi-threading.

The ISpec approach, as indicated by the title of this paper, is intended to be both *practical* and *sound*. The practicality of ISpec is being validated in software development projects at Philips, where the approach is used in combination with standard CASE tools. The focus in these projects is on the more informal usage levels of ISpec. This is in line with the idea of the approach to provide an easy entry to the development of better and more rigorous interface specifications. An ISpec course is part of the regular technical training program at Philips.

The formal part of ISpec is currently being used in experiments with automatic black-box test generation techniques. This is done by compiling sufficiently constrained formal ISpec specifications to transition systems encoded in μ CRL [7] and using the μ CRL tool-set to derive test traces. These traces are then used to test implementations of the interfaces against their corresponding transition systems.

As to the ‘soundness’ of ISpec, further research is necessary to develop the uniform semantic model that can act as the basis of a language-independent ver-

sion of the approach. Other issues that deserve attention are a better integration of ISpec and CASE tools, support for timing aspects in interface specifications, more guidance with respect to the *design* of interfaces, and CSpec, the complementary approach to *component specification*. A number of these issues are currently being addressed in a recently started research project at the Eindhoven University of Technology in cooperation with Philips Research.

References

1. Beizer, B., *Black-Box Testing*, Wiley & Sons (1995).
2. Box, D, Brown, K., Ewald, T., Sells, C., *Effective COM*, Addison-Wesley (1999).
3. Cook, S., Daniels, J., *Designing Object Systems: Object-Oriented Modeling with Syn-tropy*, Prentice Hall (1994).
4. Feijs, L.M.G., Jonkers, H.B.M., Middelburg, C.A., *Notations for Software Design*, Springer Verlag (1994).
5. Feijs, L.M.G., Jonkers, H.B.M., *History, Principles and Application of the SPRINT Method*, Journal of Systems and Software 41 (1998), 199–219.
6. Fowler, M., *UML Distilled*, Addison-Wesley (1997).
7. Groote, J.F., Ponse, A., *The syntax and semantics of μ CRL*. In: Ponse, A., Verhoef, C., van Vlijmen, S.F.M. (Eds.), Algebra of Communicating Processes, Workshops in Computing, Springer Verlag (1994), 26–62.
8. Guttag, J.V., Horning, J.J., *Larch: Languages and Tools for Formal Specification*, Texts and Monographs in Computer Science, Springer-Verlag (1993).
9. Jezequel, J., Train, M., Mingins, C., *Design Patterns and Contracts*, Addison-Wesley (1999).
10. Jones, C.B., *Systematic Software Development using VDM*, Second Edition, Prentice Hall (1989).
11. Jonkers, H.B.M., *Upgrading the Pre- and Postcondition Technique*, In: Prehn, S., Toetenel, W.J. (Eds.), VDM'91, Formal Software Development Methods Vol. 1, LNCS 551, Springer Verlag, (1991), 428–456.
12. Meyer, B., *Object-Oriented Software Construction*, Second Edition, Prentice Hall (1997).
13. Moonen, J.R., Romijn, J.M.T., Sies, O., Springintveld, J.G., Feijs, L.M.G., Koymans, R.L.C., *A two-level approach to automated conformance testing of VHDL designs*, In: Kim, M., Kang, S., Hong, K. (Eds.), IFIP TC6 International Workshop on Testing of Communicating Systems, Chapman & Hall (1997), 432–447.
14. Rose, G., *Object-Z*. In: Stepney, S., Barden, R., Cooper, D. (Eds.), Object Orientation in Z, Workshops in Computing, Springer-Verlag (1992), 59–77.
15. Software Engineering Institute, Carnegie Mellon, *SEI Open Systems Glossary*, www.sei.cmu.edu/opensystems/glossary.html.
16. Shankar, A.U., *An Introduction to Assertional Reasoning for Concurrent Systems*, ACM Computing Surveys, Vol. 25, No. 3. (1993), 225–262.
17. D'Souza, D., Wills, A., *Objects, Components, and Frameworks with UML, The Catalysis Approach*, Addison-Wesley (1998).
18. Spivey, J.M., *The Z Notation: A Reference Manual*, Second Edition, Prentice Hall (1992).
19. Szyperski, C., *Component Software*, Addison-Wesley (1998).