Consistent Specification of Interface Suites in UML*

E.E. Roubtsova[1], L.C.M. van Gool[2], R. Kuiper[2], and H.B.M. Jonkers[3]

[1] Faculty Technology Management, Group of Information and Technology,
TU Eindhoven, Den Dolech 2, P.O. Box 513, 5600MB Eindhoven, The Netherlands.
E.Roubtsova@tue.nl

[2] Faculty of Mathematics and Computing Science, TU Eindhoven,
Den Dolech 2, P.O. Box 513, 5600MB Eindhoven, The Netherlands.
L.v.Gool@tue.nl, r.kuiper@tue.nl

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

Abstract. The paper motivates and describes a model oriented approach for consistent specification of interface suites in UML. An interface suite is a coherent collection of interfaces defining interactions that transcend component boundaries. The specification of interface suites contains diagrammatic views and documentation, but it is extended with templates for structured specifications deriving from the ISpec approach. To guarantee that the specification views, documentation and templates are consistent, a specification model has been constructed. The model contains both structural and behavioural information, represented in the form of sequences of carefully designed tuples. The model provides the underlying structure for the tool supporting the design process. The tool directs the designer to specify all elements of the model in a consistent way. The specification is collected both by customized specification templates and by diagrams. The documentation and the diagram elements - both derived from the template information - are automatically generated. This prevents errors and provides specification consistency.

Keywords: UML, component specification, consistency of several views, specification model.

1 Introduction

Specification becomes an obligatory intermediate result in the component software development process. The Component Object Model (COM, COM+) of Microsoft, JavaBeans technology of Sun and the Common Object Request Broker Architecture (CORBA) of the Object Management Group (OMG) separate component specification from implementation, providing a corresponding division of labor in software development. This gives rise to a new group of professionals, responsible for producing the specification. The result of their work

* Supported by PROGRESS grant EES.5141 and ITEA DESS grant IT990211.

should be both precise to give requirements for programmers and diagrammatic to be understandable for customers. This new group of professionals needs a definition for component specification, a well defined methodology for specification development and tool support for it [3].

One of the main ideas of component technology is to provide the functionality of a piece of software (component) as a well-defined set of interfaces where an interface is a set of operations. Components can interact in a system only through interfaces. Interfaces abstract from implementation details of a component, selecting a part of the component functionality important for interaction. This allows to specify interactions in a system at an early stage when the components have not been chosen. Thus, a collection of interfaces, called an *interface suite*, together with a corresponding set of interactions, becomes a new building block, a component specification. As an interface suite transcends component boundaries, it is not described in terms of components; it is described in terms of roles and interfaces.

There are several existing approaches that specify behavioural blocks [4, 11]. The approaches use different names for a building block: an interface suite, a pattern, a framework. To handle complexity, most of the approaches specify such a building block by different views: different tables, templates and diagrams; they apply different formal notations to represent properties of the block.

Analyzing the approaches we can say that the main problem with specification of behavioural building blocks is the problem of *consistency* of different specification views. The way to solve this problem is to define one model that integrates these views and to build a well-organized specification process based on this model. In this paper we offer such a model, explain our choice and describe the tool support for the model-oriented specification process.

We provide a semantic specification model to enable dealing with consistency of the various UML views during development. This use of one model comes from the ISpec approach [11] developed at Philips, but it can equally well be applied in other approaches.

At the university of München a tool called *Autofocus* has been developed that is based on the same idea of one model for several views [20, 10]. Consistency checks on the several views are not an inherent part of the tool, but can be added manually using some form of predicate logic. The main difference between *Autofocus* and our tool is that we use the idea from ISpec to have one explicit description of the model and consider other views as derived ones.

The paper is organized as follows. Section 2 indicates the consistency problem of UML based approaches to interface suites specification. Section 3 describes the ISpec approach. Section 4 gives the specification model that allows to solve the problem of consistency. Section 5 considers the link between various views and the model. In Section 6 we discuss the specification process and the tool support based on the ISpec approach and the interface suite model.

2 A Case Study with an Interface Suite

The aim of this Section is highlighting the problems with UML based specification approaches that use a set of different UML diagrams. We use for this purpose a case study concerned with the interaction in a system for collision detection in local networks (Carrier Sense, Multiple Access, Collision Detection CSMA/CD [2]). It can be considered as an interaction pattern, i.e. an interface suite. Concrete examples of this suite are a system for urgent telephone calls as described in [1], or a local network where several computers share a printer or a plotter.

We describe this interface suite in term of roles [17] to represent some observable behavioural aspects of components. We use UML classes to describe roles.

Description of the Case Study. There are two main roles in the *CSMA/CD* suite named *Resource* and *Device*. Several devices can be connected to one resource and transfer data to it. Connecting and disconnecting is possible at any moment. Each *Device* checks the *Resource*. If there is another device that transfers data, the *Device* waits some time and checks the *Resource* again. If the *Resource* is free the *Device* begins the *transfer* of some data and checks the *Resource* a second time. If no one has begun to transfer at this check time, the *Device* transfers data. Depending on the number of devices connected to the *Resource*, a maximum time P is defined that is allowed for a transfer between the *Resource* and a *Device*. The time of transfer is controlled by a timer. The transfer can terminate by itself or be stopped by the timer when time is up. However, it is possible that two devices begin to transfer simultaneously. In this case both devices stop transferring and try again later after some random time interval. We can compare the behaviour of devices in the suite with a dialog of well-behaved persons: if two of them begin to speak simultaneously, they stop talking for some time interval and then begin their attempt again, hopefully not simultaneously. In a system for collision detection in local networks, for example, the channel with multiple access plays the role of resource and a local station plays the role of device.

Role View. Self Consistency of a View. The first view on the suite is a UML class diagram where classes represent roles, see Fig. 1.

Role *Resource* provides the interfaces *IConnect* and *ITransfer*. *Device* uses these interfaces. Interface *IConnect* provides operations *connect()* and *disconnect()* devices. Interface *ITransfer* contains operation *check()* to check and return the current state of *Resource*, operation *transfer()* to transfer all data or to signal that *Resource* is occupied and operation *toIdle()* that terminates transferring data when the allowed time is over.

Timer and *Random* are two auxiliary roles in the suite. Role *Timer* contains a clock. *Timer* provides interface *ISet* to reset a clock to zero. After resetting, the value of the clock increases at a constant rate till a certain control value T is reached and operation $onTime()$ via interface $IWhen()$ of *Device* is called. Role *Random* models a generator of random values in a given value interval $(0..B)$.

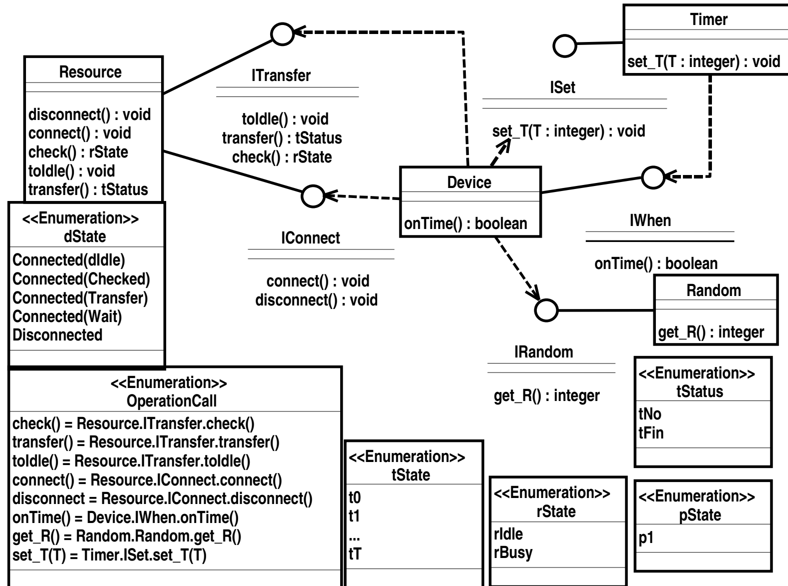


Fig. 1. An abstract class diagram for the role view on CSMA/CD interface suite

The current value of the generator is of type Integer. Role *Random* provides interface *IRandom* with operation $get_R()$.

Even one role view can be inconsistent and unclosed with respect to its own elements. For example, an interface could be defined such that it is not used or provided. The result of an operation can be of an undefined datatype. Such an unclosed role view can't be considered as a specification. To make the role view (such as in Fig. 1) self consistent we have very carefully specified communications only via interfaces and defined all datatypes we use with the *Enumeration* stereotype.

Consistency between Views of One Type. The specification of a role can be done at different levels of abstraction. This gives rise to different views of one type. For example, a role can be specified with and without attributes. If some interfaces are not needed at some level of abstraction a designer does not show them. As a result, the roles of the same name at two role diagrams can have different interfaces or different operation sets where one is not an abstraction of the other, and the operations can have different results. This set of role diagrams can not be interpreted as an unambiguous specification.

More Views. Consistency between Views of Different Types. Even if we have a class diagram which is a self consistent role view we need another

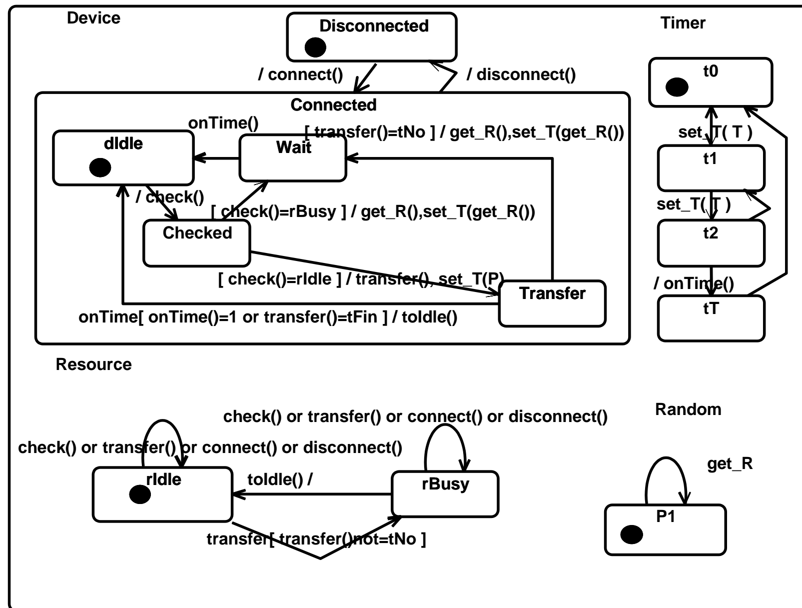


Fig. 2. A statechart diagram of the interface suite

view to specify behaviour. The behaviour of our interface suite is shown by a UML statechart, Fig. 2.

Consistency checking even for two diagrams of different types can take a lot of time and effort. For example, to make the role view (Fig. 1) and the statechart view (Fig. 2) consistent, one would need to check the names of roles and arrow labels that represent operation calls. The list of possible operation calls that is derived from the role view is shown in Fig. 1 as an enumeration type *OperationCall*. An arrow label on the statechart means a call and its return. For the sake of consistency of role and statechart views, arrow labels of the statechart should be taken from *OperationCall*.

We can add state information to the class diagram as attributes. For example attributes of the *dState* and *rState* enumeration types in (Fig. 1) represent the states of *Device* and *Resource* respectively. States in the statechart diagram should be consistent with the class diagram.

If an approach uses more UML diagrams the consistency problem becomes more complicated. For example, relevant traces are often depicted by UML sequence diagrams as presented in Fig. 3 for the $CSMA/CD$ interface suite. All sequence diagrams should also be consistent with other views.

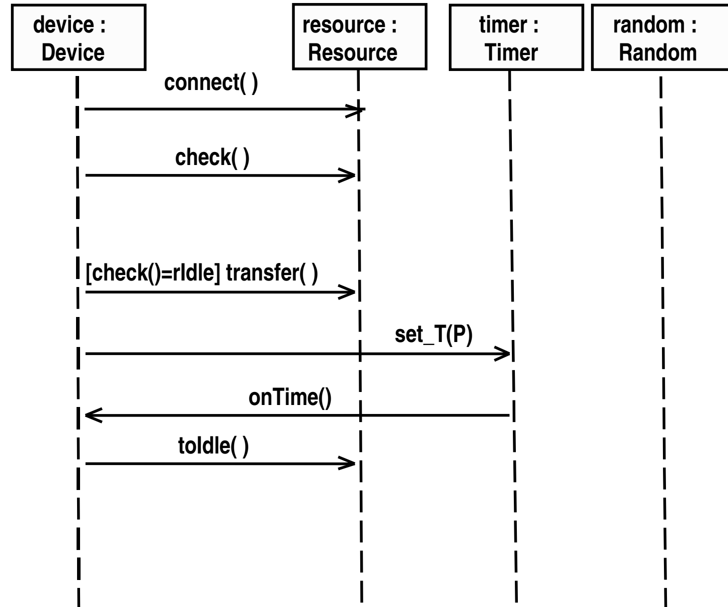


Fig. 3. An interaction diagram of the interface suite

Let's assume that there are four diagrams (Class diagram C, Statechart S, interaction diagrams I_1 and I_2) with precise semantics. Checking pairwise consistency means that we should perform six consistency checks: (C, S), (C, I_1), (C, I_2), (S, I_1),(S, I_2), (I_1, I_2). Generally speaking, if we have n diagrams we should perform $\frac{1}{2}n(n-1)$ consistency checks for diagram pairs. And this pairwise consistency does not even guarantee the consistency of the collection of all four diagrams.

The Problem of View Consistency Is Repeated During Design. The specification given by diagrams can not catch all the designer intuition. That is why designers additionally define properties of the system. They are satisfied with the design result only if the properties hold. In UML these properties can be represented as comments (that we consider as an underestimation of the importance of a property specification) or localized in an aggregation class that represents the suite as a whole. For example, for our case study:

1. Each device that requests a data transfer has access to the resource in a reasonable time interval T.
2. Each moment in time no more than one of the devices really transfers data to *Resource.*

The definition of the properties should be also consistent with the design. Very often we cannot represent properties without some additional efforts. Sometimes we need additional roles to specify properties. For example, an additional timer is necessary for the specification of property 1. Property 2 of our case study suggests a counter k to count the current number of devices that use the *Resource*. So, the specification of properties of our case study gives rise to new variants of diagrams and repeatedly poses the problem of consistency. Moreover, the properties can be specified using different notations [6, 19] that should be consistent with each other.

The problem of consistency should be solved again at the stage in the design process where we adapt the interface suite to some chosen components or to a concrete application field. For example, we can split the role *Resource* into *Connector* and *Resource*. New versions of diagrams extend the set of design views.

Summarizing, we can say that all design approaches that use different UML views face the consistency problem. This problem needs to be repeatedly solved during the design cycle. The way out of this problem is to use one specification model for the specification process. We then have to perform only n consistency checks of type $(View, Model)$. We can even predict some checks via a model oriented specification methodology. In this paper we offer a possible form of this model. It is developed to support the ISpec approach to interface suite specification [11, 12]. On the one hand the model is a basis of a tool supported specification technology that allows to specify consistent views, on the other hand the model facilitates connecting the tool to existing tools like model checkers.

3 The ISpec Approach

In our work we use ideas of the ISpec approach developed by Philips Research [11].

ISpec addresses the problem of specifying the type of interfaces encountered in component technologies such as Microsoft COM, JavaBeans and CORBA. In ISpec an interface specification is perceived as the description of an interaction pattern. It specifies the behaviour of an *interface suite* (i.e. a collection of mutually related interfaces). Interface suites, rather than individual interfaces, are the units of independent specification. An interface specification identifies a number of *roles* that can be seen as abstractions of component object classes [22]. Each interface has a role associated with it and the behavioural aspects of the interfaces including the interactions they are involved in follow from the specification of the roles. ISpec interface specifications are closely related to design patterns [7], role models [16] and contracts [9]. They are referred to as *i-specs*, which can be interpreted as 'interface specifications' as well as 'interaction specifications'.

ISpec supports the construction of i-specs at different levels of detail and formality, ranging from 'signature only' to 'complete and formal' specifications. ISpec is a model-based approach that uses UML in a number of ways. The

	Conceptual view	Specification view	Technical view
Goal	Provide *intuition*	Provide *precision*	Provide *technological details*
Model	*Implicitly* characterized by - set of UML diagrams - explanatory text	*Explicitly* characterized by - interface role diagram - abstract model diagram - (filled) templates	*Explicitly* characterized by - specification view - mapping to technology

Fig. 4. Interface-Role Diagram and Abstract Model Diagram of the Concrete Observer Interface Suite

modelling concerns are separated into three views: conceptual, specification, and technical.

The *conceptual view* is primarily meant to introduce the necessary terms and concepts and explain the purpose and behaviour of the interfaces in an informal and intuitive way. It uses UML diagrams in an 'impressionistic way', sketching various aspects of interface behaviour using state diagrams, sequence diagrams, etc. These diagrams are typically abstractions, simplifications or snapshots of interface behaviour and need not be complete.

The *specification view* is meant to provide a technology-independent description of the behavioural aspects of the interfaces that is sufficiently precise and detailed. The latter normally implies that the description can be used as a basis for applying, implementing and testing the interfaces. The specification view is based on two types of diagrams: an interface-role diagram and an abstract model diagram [12]. The interface-role diagram is a UML class diagram that can be seen as the 'signature' of an interface suite: it identifies the names of the interfaces and roles, defines the 'provides' and 'requires' relations between roles and interfaces, as well as the specialization relations between roles (the latter

relation is not considered in this paper; for more details see [12]). For example, if we have a set of interfaces supporting the concrete observer pattern [7], the interface-role diagram of this interface suite would look something like Fig. 4. The abstract model diagram is a UML class diagram that is a specialization of the interface-role diagram at the later stage of design. It associates an abstract representation with each role in terms of attributes and relations, gives the signatures of all interfaces and may introduce additional artifacts such as auxiliary classes and operations, see Fig. 4. The behavioural aspects of the interfaces are directly specified in terms of this model using templates, attached to the diagrams. A template provides slots where information about behaviour can be put in in a structured manner, e.g. pre- and postconditions. Templates also contain action clauses to capture further information, for example about the interactions of roles. The slots in the templates are pieces of text that may be informal, semi-formal or formal. ISpec is neutral with respect to the language used in the placeholders. The formal language that is currently used is the expression language of Z, but OCL could be used as well.

The templates themselves are hierarchically ordered in a tree-structure, allowing behaviour to be specified at different levels of detail. The specification view uses other UML diagrams as well but only in an 'expressionistic' way, i.e. as 'expressions' of the model defined by the interface-role diagram, abstract model diagram and the templates. For example, state diagrams are derived from the abstract state representation of the roles and the pre- and post-conditions of the operations (similar to [3]). Mathematically speaking, the state representation and the pre- and post-conditions are the axioms and the diagrams are the theorems. The advantage of this approach is that it is much easier to ensure consistency of the various diagrams since they have to relate to a single explicit model.

The *technical view* provides the mapping of the interfaces to their concrete representation in a particular component technology or programming language. This concrete representation of the interfaces is important in connection with implementation and testing, but specifying the interfaces directly in terms of this representation will generally mess up specifications (just have a look at some Microsoft COM IDL). By keeping the technical details separated from the essential behavioural details defined by the specification view, the interface specifications can be kept clean and mappings to different component technologies and programming languages can be supported without the need to change the specification view. The key aspects of the three views used in ISpec are summarized in Fig. 5.

The main idea of ISpec that we adopt is the use of a single model as the basis of all UML diagrams that characterizes an interface suite. In this paper we define a semantic model that can play this role.

	conceptual view	specification view	technical view
goal	provide *intuition*	provide *precision*	provide *technological details*
model	*implicitly* characterized by - set of UML diagrams - explanatory text	*explicitly* characterized by - interface role diagram - abstract model diagram - (filled) templates	*explicitly* characterized by - specification view - mapping to technology

Fig. 5. The Three Modelling Views of an Interface Suite in ISpec

4 The Specification Model for Interface Suites

In this Section we introduce the specification model that is the basis for guaranteeing consistency of views [18].

Informally, our model consists of a number of *objects* that interact by calling operations of each other. Each object has a role attached to it. This models dynamic binding where the behaviour of the operations of an object is dependent on the role of the object. An object is now identified by a pair (r, p) consisting of a *role* name r and a *player* identifier p. The role names allow us to talk about different kinds of objects and the player identifiers allow us to distinguish between objects that play the same role. We talk about "object (r, p)" or "player p of role r".

The operations of an object are grouped into *interfaces*. Operations that have the same name but occur on different interfaces are supposed to be *different* operations. This corresponds to the notion of interface as used in Microsoft COM.

Our model captures the interaction between several objects. More formally, our model is a set of (possibly infinite) sequences of messages where a *message* is either a call of an operation or a return from an operation call.

A *call* is described by

– a *caller role* cr,
– a *caller player* cp,
– a *callee role* rr,
– a *callee player* rp,
– an *interface* i,
– an *operation* o and
– a *parameter* x.

The interpretation of a call $(cr, cp, rr, rp, i, o, x)$ is that player cp of caller role cr calls, with parameter x, operation o of the interface i on player rp of callee role rr.

A *return* is described by

– a *call depth* d and
– a *result* y.

The interpretation of a return (d, y) is that the call identified by d returns with result y. How a call depth d identifies a call is explained next. For each sequence of messages we imagine a stack in which each call that occurs is stored. If a return with call depth d occurs, we interpret this as the return of the call that lies d layers deep in the current stack (if the call depth is 0, this is the call on top of the stack). When a return occurs, the corresponding call is removed from the stack.

Notice that it follows from this interpretation that we don't allow every sequence of messages; we have to make sure that the call depth of each return in a sequence does not exceed the stack size. This means that the call depth of each return in the sequence should be smaller than the number of calls preceding this return minus the number of returns preceding this return. A sequence of messages that adheres to this restriction is called a *trace*.

We now give the formal definition of our specification model.

Definition 1. *The type of all messages Msg is defined by*

$$\begin{aligned} \mathit{Msg} ::= {} & \mathit{call}(\mathit{Role}, \mathit{Player}, \mathit{Role}, \mathit{Player}, \mathit{Interface}, \mathit{Operation}, \mathit{Parameter}) \\ & \mid \mathit{return}(\mathbb{N}, \mathit{Result}) \end{aligned}$$

where

- *Role is some set of role names,*
- *Player is some set of player identifiers,*
- *Interface is some set of interface names,*
- *Operation is some set of operation names,*
- *Parameter is some set of parameter values,*
- $\mathbb{N}$ *is the set of the natural numbers and*
- *Result is some set of result values.*

Definition 2. *A trace is a (possibly infinite) sequence with elements of type Msg where the call depth of each return in a trace is smaller than the number of calls preceding this return minus the number of returns preceding this return.*

Definition 3. *The specification model for interface suites is a set of traces.*

Our traces are a generalization of single threads and actually abstract completely from the threads in the system. A single thread is modeled by a trace in which each call-depth is 0 and the caller object of each call is equal to the callee object of the call that is on top of the imagined stack. The initial stack should have a call on top. The callee of this call is then the object that initiated the trace. If all our traces were single threads then our model could be simplified by omitting the return depths and caller objects. However, the information of who initiated the trace is then lost.

In a trace, each return is associated with exactly one call and each call with at most one return. A crucial concept in connection with our model is the combination of a return and its corresponding call. This will be called a *roundtrip*. The *value* of a roundtrip with call $(cr, cp, rr, rp, i, o, x)$ and return (d, y) is the

tuple $(cr, cp, rr, rp, i, o, x, y)$. Notice that it is possible that a call never returns, so not every call needs to be part of a roundtrip.

Roundtrips are the basic communications in our suites. They generalize the communications of standard message passing systems by allowing a reply to a message that is sent. Usually in a message passing system, a reply is just the sending of another message. Roundtrips make the link between the sending of a message and a reply to that message explicit; they bring us closer to the standard functional theory of I/O.

5 Consistency of Views w.r.t. Our Model

In this Section we will show for several views which models correspond to each view. So, if we have a collection of views, we then know whether or not there exists a model that corresponds to each view in this collection, rendering the collection of views consistent.

Each view is an instance of a certain *view type*: a construction in which some aspects of systems can be described. Role diagrams, statecharts and sequence diagrams are examples of view types. Actually, if we speak about "our model" we mean "our model *type*" because it doesn't describe one specific model (instance) but the mathematical type of all models.

To define for a certain view type which models are associated with a certain view of this type, we need to define a *consistency relation* between models and views of this type that tells which views are consistent with which models. We say that a *collection of views is consistent w.r.t. our model* if there exists a model that is consistent with each view in this collection.

5.1 Role View

We will now show a way to link a class diagram that represents a role view, to our model. There are several ways to interpret these kind of diagrams. The way that is described here, we think, matches the intuition of UML-users. The key idea is that all interactions that are not explicitly forbidden by the diagram can appear anywhere in a trace. The advantage of this approach is in a simple and natural definition of refinement, but this is beyond the scope of this paper. Interested readers are referred to [8]. However, this approach doesn't correspond to the "closed world" interpretation of these kind of diagrams in ISpec where the only interactions that are allowed are the ones that are explicitly mentioned in the diagram. As explained in [12], these are important choices as they influence the feasibility of, for example, a compositional approach of interface suites. Our model enables to accommodate such choices through different interpretations of the diagrams or different notions of refinement. This is the subject of current research.

A role view expresses *static* aspects of a system. We introduce the notion of a *signature* as a formal description of these static aspects. For our kind of

systems, these static aspects are expressed by a set of allowed roundtrip values. Formally a signature is a set

$$S \subseteq \mathit{Role} \times \mathit{Player} \times \mathit{Role} \times \mathit{Player} \times \mathit{Interface} \times \mathit{Operation} \times \mathit{Parameter} \times \mathit{Result}$$

Definition 4. *A model is consistent with a signature S exactly when the value of each roundtrip in the model is an element of S.*

By means of an example, using the role view of Fig. 1, we explain how a role view defines (in the intuitive UML sense) a signature S. This signature consists exactly of all tuples $(cr, cp, rr, rp, i, o, x, y)$ that satisfy the next constraint (the type *void* consists of the single element ()).

$$\begin{array}{ll}
(cr, rr, i, o, x) = (\mathit{Device}, \mathit{Resource}, \mathit{ITransfer}, \mathit{check}, ()) & \Rightarrow y \in \mathit{rState} \\
(cr, rr, i, o, x) = (\mathit{Device}, \mathit{Resource}, \mathit{ITransfer}, \mathit{transfer}, ()) & \Rightarrow y \in \mathit{tStatus} \\
(cr, rr, i, o, x) = (\mathit{Device}, \mathit{Resource}, \mathit{ITransfer}, \mathit{toIdle}, ()) & \Rightarrow y \in \mathit{void} \\
(cr, rr, i, o, x) = (\mathit{Device}, \mathit{Resource}, \mathit{IConnect}, \mathit{connect}, ()) & \Rightarrow y \in \mathit{void} \\
(cr, rr, i, o, x) = (\mathit{Device}, \mathit{Resource}, \mathit{IConnect}, \mathit{disconnect}, ()) & \Rightarrow y \in \mathit{void} \\
(cr, rr, i, o) \quad = (\mathit{Device}, \mathit{Timer}, \mathit{ISet}, \mathit{set_T}) \wedge x \in (T{:}\mathit{integer}) & \Rightarrow y \in \mathit{void} \\
(cr, rr, i, o, x) = (\mathit{Timer}, \mathit{Device}, \mathit{IWhen}, \mathit{onTime}, ()) & \Rightarrow y \in \mathit{boolean} \\
(cr, rr, i, o, x) = (\mathit{Device}, \mathit{Random}, \mathit{IRandom}, \mathit{get_R}, ()) & \Rightarrow y \in \mathit{integer}
\end{array}$$

5.2 Statecharts and Other Interaction Diagrams

UML statecharts can be given a semantics in terms of our model. A statechart corresponding to a role specifies behaviour of each player of the role. An arrow in statechart represents both a call and the corresponding return. Our model considers calls and returns as separate actions by having a stack of calls that have not yet returned. This is similar to the semantics presented in [13].

Assume that we have two devices that can be connected to one resource (Fig. 1, 2). Both of them call the operation *connect*() of role *Resource* via interface $IConnect$. The returns of both connects can come in any order. One of the legal traces, derived from the statechart diagram in our case study (Fig. 2) is the following:

$$\begin{array}{l}
\mathit{call}(\mathit{Device}, 1, \mathit{Resource}, 1, IConnect, \mathit{connect}, ()) \\
\mathit{call}(\mathit{Device}, 2, \mathit{Resource}, 1, IConnect, \mathit{connect}, ()) \\
return(1, ()) \\
return(0, ()) \\
\mathit{call}(\mathit{Device}, 1, \mathit{Resource}, 1, IConnect, \mathit{disconnect}, ()) \\
return(0, ()) \\
\mathit{call}(\mathit{Device}, 2, \mathit{Resource}, 1, IConnect, \mathit{disconnect}, ()) \\
return(0, ()).
\end{array}$$

According to the statechart diagram (Fig. 2) a device cannot be connected twice without disconnecting. Thus, an illegal trace is

$call(Device, 1, Resource, 1, IConnect, connect, ())$
$return(0, ())$
$call(Device, 1, Resource, 1, IConnect, connect, ())$
$return(0, ())$.

Note, that this trace *is* consistent with the signature of the operations. So the expressive power of sets of traces allows to also do consistency checks of *dynamic* behaviour.

Assume that a straightforward trace semantics as indicated at the beginning of this Section, is available.

Definition 5. *A statechart is consistent with exactly one model, being its trace set.*

Here we assume that we have exactly one statechart for the entire system behaviour. However, a statechart could describe only part of the system behaviour. We then also need to give a signature (in the closed world interpretation, see Section 5.1) that describes which messages the statechart talks about. A statechart is then consistent with exactly all models that, after projection to this signature, are equal to the statechart's trace set.

Dynamic diagrams like statecharts, interaction diagrams, activity charts, collaboration diagrams, can be handled in a similar way.

5.3 Extensions of the Model

Our model allows to check the consistency of a collection of views only at a certain level of abstraction. A view could describe aspects that are not captured by this model, like state information or real-time aspects. If we want those aspects to be checked, the model should be extended with these aspects.

Statecharts, for example, talk about state and so do abstract model diagrams of ISpec by declarations of attributes. Of course we want the state information in the statecharts to be consistent with these declarations. ISpec also talks about state (attributes) in the pre-conditions, post-conditions and action clauses of operations.

Our traces can be extended with a state aspect by adding to each call and return an extra entry in the tuple that represents the state at that point. The *pre-condition* of an operation talks about the points where the operation is called. It is formulated in terms of the parameters and the value of the attributes at the point of the call. The *post-condition* talks as well about points where the operation is called as about points where the operation returns. It is formulated in terms of the parameters and the state at the point of a call and the result and the state at the point of the return that corresponds to this call. The *action clause* of an operation is a more program-like description, talking about the order of operation calls and state changes in between the call and the return. A precise description of how the model is extended with state information and how it is constrained by pre-conditions, post-conditions and action clauses is the subject of another paper that is currently under development.

UML diagrams used in ISpec	Consistency demanded by ISpec	Supported by the current (+) model extended (*) model	Supported by the current tool
Class diagram	Self consistency of an interface role view	+	+
Class diagram	Self consistency of an abstract model view	*	+
Documentation	Self consistency of a documentation view	+	+
Sequence diagram	Self consistency of a sequence view	+	-
Statechart diagram	Self consistency of a statechart view	*	-
Class diagram/ class diagram	Consistency between two role views (interface-role view/ abstract model view)	*	+
Documentation/ Documentation	Consistency between two documentation views (HTML files)	*	+
Class diagram/ Documentation	Consistency between an interface-role view and a documentation view	+	+
Class diagram/ Documentation	Consistency between an abstract model view and a documentation view	*	+
Class diagram/ Sequence diagram	Consistency between a role view and a sequence diagram view	+	-
Class diagram/ Statechart	Consistency between a role view and a statechart view	*	-
Sequence diagram/ Statechart	Consistency between a sequence view and a statechart view	+	-

Fig. 6. Model and tool support to solve ISpec consistency tasks

6 Specification Process Supported by Tool

6.1 Current Tool and Its Extensions

In this Section we describe a tool based on our model, that allows to solve consistency problems identified in Section 2 for the ISpec approach presented in Section 3.

Figure 6 shows consistency tasks on ISpec views and model/tool support to solve these tasks. The tool we present allows to produce self consistent interface-role view, self consistent abstract model view and self consistent documentation. This means that all of these views and their union can be considered as a consistent specification of an interface suite because they correspond to one model. Via this model the tool guarantees also the consistency between views of different types, between interface-role and documentation views, between abstract model and documentation views. Figure 6 shows also the tasks that should be solved by extension of the model that we mentioned in Subsection 5.3. Some of these tasks, for example consistency between interface-role view and abstract model view, are supported in advance by tooling. Currently we cover two role views and several documentation views in interface suite specification. Role views are

represented by interface-role type and abstract model type. We use an auxiliary *data type diagram* to specify data types for both role views. The documentation views can be automatically generated by the user. The tool keeps the documentation consistent with the correspondent role view. Further we plan to support the consistency with respect to our model of sequence diagram views and statechart views.

6.2 The Tool and the Model

Our tool offers to a designer specification forms that are constrained and guided by the specification model. The forms are structured according to the model for an interface suite. The main feature of our specification tool is that it does not use UML diagrams for gathering information but for illustrating the specification given by forms corresponding to templates in ISpec.

Let us remind the reader that we model the behaviour of an interface suite by a set of traces of operation calls and returns of the type (Section 4)

$$RoundTrip = Role \times Player \times$$

$$Role \times Player \times Interface \times Operation \times Parameter \times Result.$$

It is this model that underlies consistent specification in the form of diagrams, forms and documentation.

- First, using the data type diagram, just data types for *Parameter* and *Result* specification are defined.
- Second, at the level of interface-role diagram we couple names of parameters and results with predefined data types. This narrows the sets *Parameter* and *Result*. We specify a set of roles *Role*, a set of players *Player* for the role and operations *Operation* provided by roles. From the set of operations of a role we allow to construct disjoint sets defining interfaces provided by roles *Interface*. Then we are able to define the set of provided interfaces.

  $$ProvidedInterface = Role \times Player \times Interface \times Operation \times Parameter \times Result.$$

  This set is saved by our tool for the interface suite. To finish the specification of the *RoundTrip* set we open the form for every role and choose interfaces required by players of this role *RequiredInterface* from the set of saved provided interfaces *ProvidedInterface*

  $$RoundTrip = Role \times Player \times ProvidedInterface.$$

  All these restrictions have been built into the tool as constraints that are followed when filling in the next form. The result of this activity is illustrated automatically on the interface-role diagram. This ensures that the interface-role diagram is self consistent.
  Structured documentation is generated from elements of the *RoundTrip* set being consistent with the interface-role diagram.

– Third, we restrict the set of traces representing the interface suite by coupling names of attributes *Attribute* with predefined data-types and by specifying *pre-, post-, action-clauses* using only names of specified actions of type $RoundTrip$, attributes of type *Attribute* and a restricted set of logical operations. The result is an abstract model diagram illustrating this specification level and the corresponding documentation. This ensures self consistency of the abstract model diagram as well as consistency of the abstract model diagrams with respect to the interface-role diagram and the documentation.

6.3 Details of the Tool Realization. The Consistent Specification of Our Case Study with Support of the Tool

The tool is realized as a Rational Rose Add-In [15], because Rose already solves some consistency problems. Rose guarantees, for example, consistency between two views of one type, interface-role view and abstract model view of ISpec. However, the self consistency of a view and consistency between views of different types are still problems for users of Rose.

To implement the tool support for these tasks we have customized Rose specification according to our model, adapting Rose class diagrams and the following subset of the Rose class model:

$$Class = (Name, Stereotype, Attribute, Operation, Relation).$$

This allows us to use Rose functions for drawing diagrams automatically from our Add-In on the basis of elements of the diagram and their relations specified in forms.

– Data types are specified by $Class$ where $Stereotype = DataType$. The set of basic data types consists of $Integer$, $Real$, $Boolean$, $Enumeration$, $String$, $Array$, $Structure$. All of these types are drawn at the data type diagram by default. Each type has attributes. The user can not define new operations for a data type, those operations are defined in programming languages, so for data types $Operation = \emptyset$, because the operations are standard and should not be specified. There is a *New Type* at the diagram Fig.7. A double click on the *New Type* opens a form. The user can define a new type by specialization of an existing one, changing values of inherited attributes and adding new ones. So, the set of data-type relations is restricted by specialization $Relation = Specialization$.
  Fig. 7 shows the data type diagram of CSMA/CD interface suite (see the case study in Section 2). New data types *rState, rStatus, dState, tState, pStatus* are specified as specialization of the *Enumeration* type. Type $tStatus$, for example, specifies the result of operation *transfer()*. Using data types that are not specified by the data type diagram is forbidden during specification of other diagrams. If a designer tries to use an unknown data type, our tool directs the designer to modify the data type diagram.

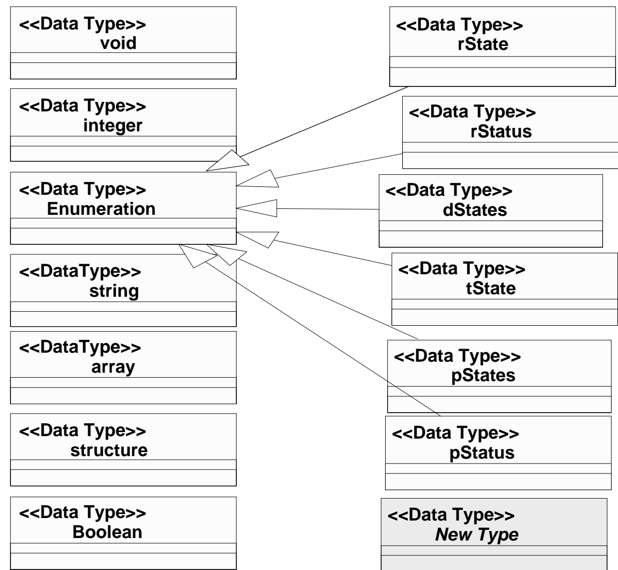


Fig. 7. Data Type diagram of CSMA/CD suite

– At the interface-role diagram a role is specified by a Rose *Class*, where $Stereotype = Role$. In the Rose operation specification

$$Operation = (Name, ReturnType, Argument)$$

we use *ReturnType* to specify the *Result* and *Argument* to collect the set $Parameter$.
Associations of roles are replaced and restricted now by the provide relations and the require relations with interfaces. Provide and require relations are defined as sets of pairs ($role$, $interface$). The relations are represented by the *class. realize* and the *class.depends* collections of Rose correspondingly. The set *Attribute* is empty at the interface-role diagram. The multiplicity of roles in associations is interpreted by our tool as sets of players $Player$ for our model.
The interface-role diagram allows to specify roles and operations of the roles using only lists of predefined data types for parameter and result specification. Forms that belong to the interface-role diagram allow to group existing operations into interfaces. For example, interface *ITransfer* consists of operations *toIdle(), transfer()* and *check()* of role *Resource* (Fig.8,9). Another form offers to choose some interfaces from existing ones as required by some roles. For example, we have chosen interfaces *ITransfer, IConnect, ISet, IRandom* for role *Device* (Fig.9). The independent specification of interfaces is forbid-

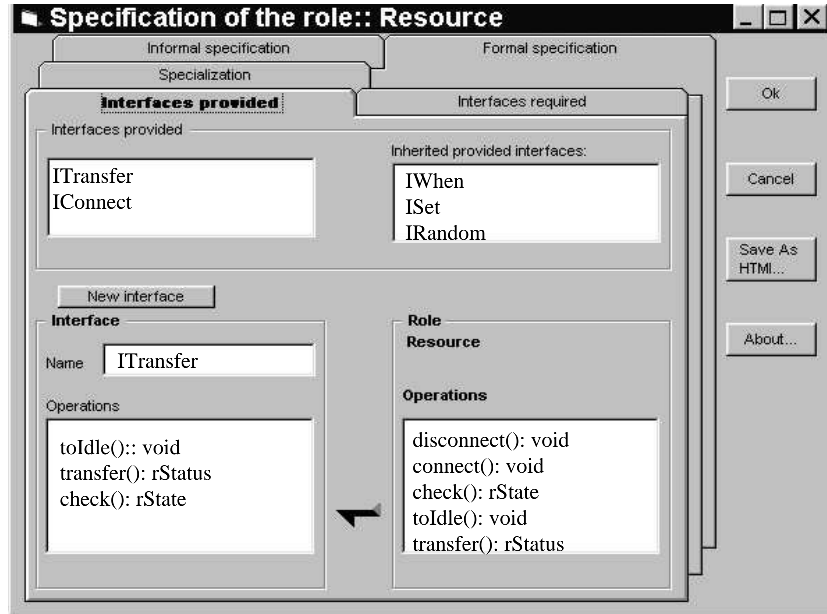


Fig. 8. Interfaces provided by Resource

den to prevent inconsistencies. Interfaces are specified via Role specification template. The tool generates a list of all interfaces provided by the interface suite and uses this list to specify the interfaces that can be required.
At the interface-role diagram level the attributes are not specified. As a result the following diagram is generated for CSMA/CD suite (Fig.9).

– The abstract model diagram is an interface-role diagram extended by attribute specifications and by pre-, post- and action- clauses. The Rose attribute model is adopted without changes of interpretation. For operation specification another subset of the Rose operation model is used

$$Operation = (Name, ReturnType, Argument, Pre, Post, Semantics),$$

where *Semantics* is used to specify action-clauses.
The abstract model diagram allows to define attributes using predefined data types. It offers to specify action-clauses, pre- and post- conditions using predefined operations and attributes and given logical operations. In Fig.10 we have shown the specification on the operation *disconnect()*. This operation decreases the number n of connected devises, that is specified in pre- and post- conditions. By the action clause we take into account that operations

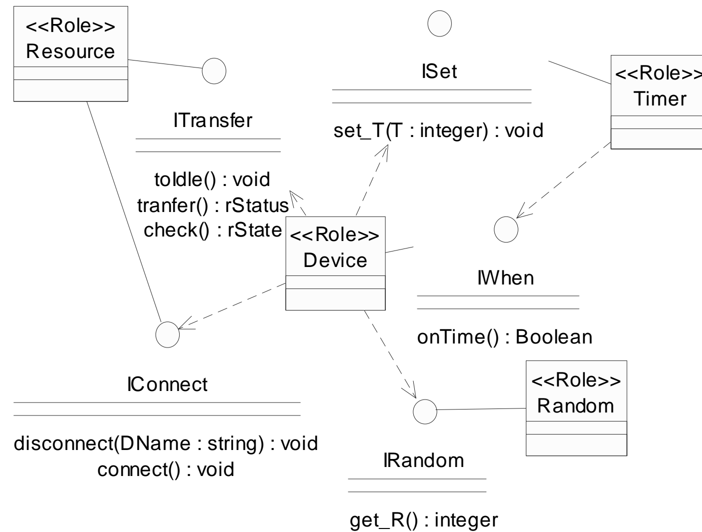
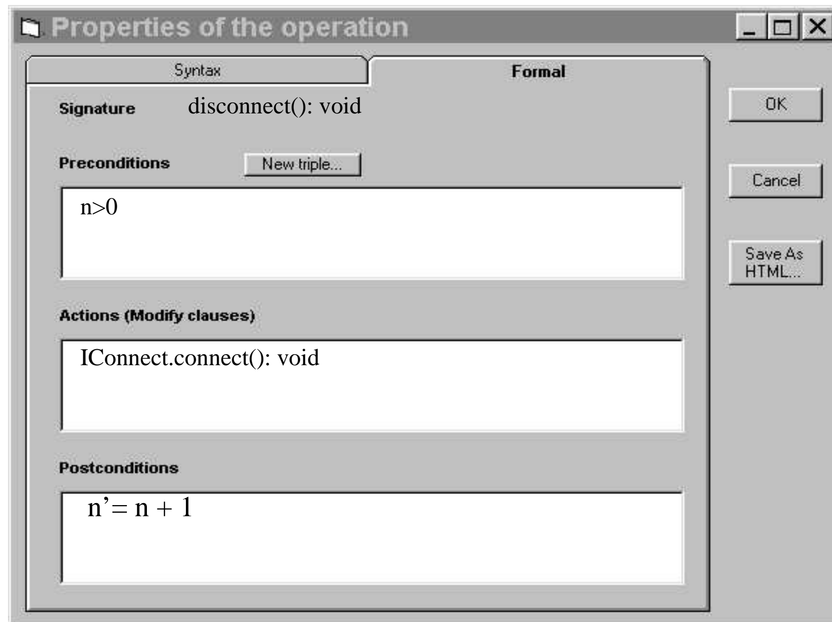


Fig. 9. Interface-role diagram of CSMA/CD suite

connect() and *disconnect()* can be called between the call and return of *disconnect()* and that their pre and post- conditions also should be taken into account. The allowed model elements are restricted by the tool that prevents possible mistakes and fills the specification model. The tool provides support for the choice of attributes of all suite roles and operations of all suite interfaces to use them in *Pre-, Post-* and *Action*-clauses specifications (Fig.10).

A designer can start to specify at any level and the tool will redirect him/her to another level if some elements have not been specified yet.

The specification is distributed via documentation from the main group of designers to groups producing components. So, the documentation plays a key role in information transfer in the "design by contract" approach [14]. To make the documentation self consistent and consistent with different views on the specification model our tool supports the automatic generation of the documentation. Our Add-In provides a special tab "Save as HTML" for creating the documentation on all specification forms. By clicking this tab we open the *Print Form* (Fig.11). The Print form of our tool allows to build the documentation as a puzzle from selected roles, interfaces and diagrams. We have chosen the HTML-format for the documentation to enable browsing it in Word or internet browsers and to deliver the component via internet. The diagrams are saved as files in EMF-format and inserted into the HTML documentation file (Fig.12).



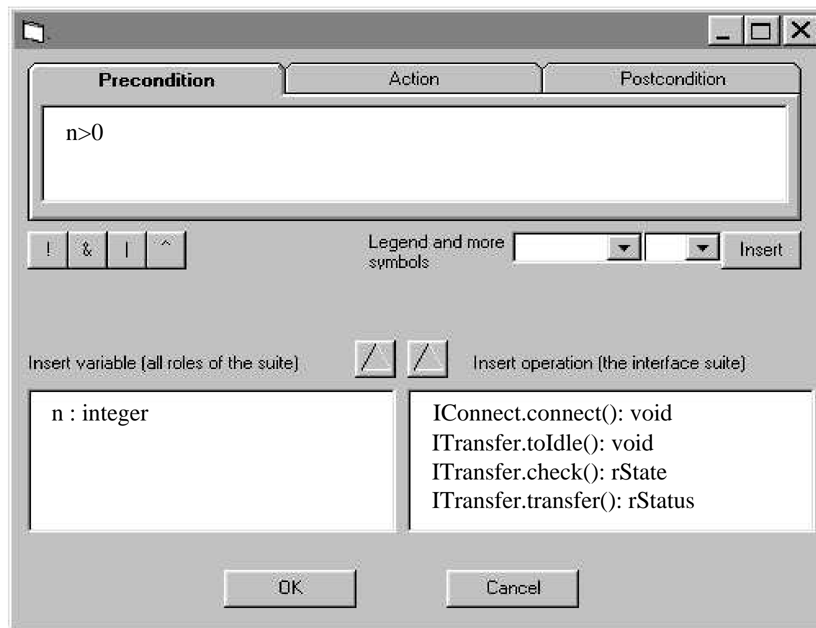


Fig. 10. Specification of pre-, post- conditions and action clauses

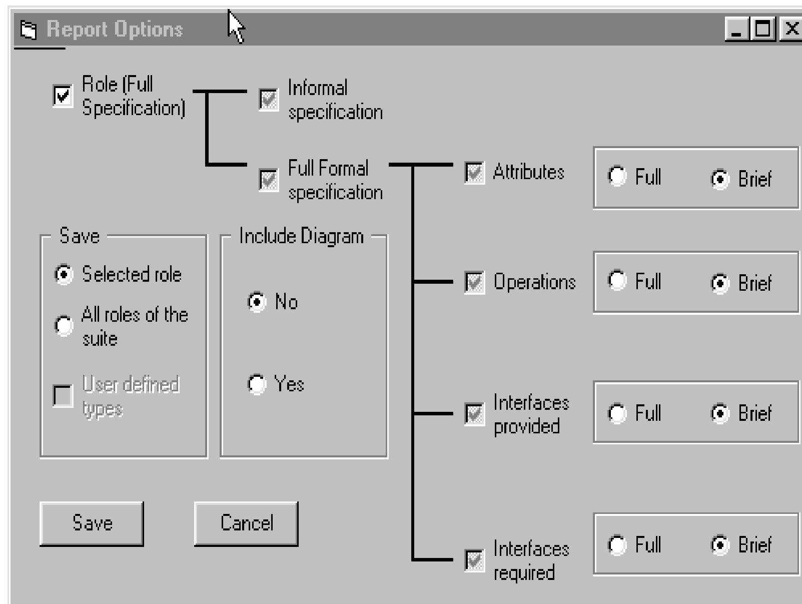


Fig. 11. Print form to compose documentation

The fact that this specification tool is based on a clear model, enables further extensions, especially to support consistency of behavioural views as shown in Fig.6.

7 Conclusion

Component based models including Microsoft's Component Object Model COM+ encompass the idea of an interface suite, a suite of component services integrated with an environment [5]. The models and technologies based on them use interface suites for specification of component systems in earlier stages of design. Specification of interface suites is the first step of component system design technologies. The specification should be consistent and precise because it provides requirements for programmers, it is used for formal method applications, for model checking, verification and testing. On the other hand the specification should be visible and understandable for customers to enable collecting their requirements. The UML in combination with a specification model gives a unique opportunity to realize the combination of these demands.

We have developed a specification model, the interactive specification methodology for interface suite design and tool support for it. Our methodology is based on the ISpec approach used at Philips for component specification [11]. The tool

Role: Resource

Informal specification:

Stereotype: Role

Description:

```
Something that can be shared by several devices.
Concrete examples are a printer in a local network,
a connector in a system for telephone calls.
```

Responsibilities:

```
- connecting and disconnecting of devices;
- setting an order of sharing ;
```

Formal specification:

Attributes of the role:

Operations of the role:

Name: disconnect

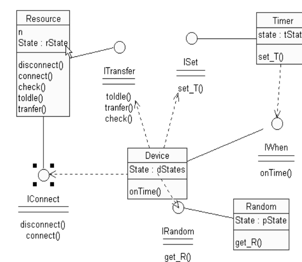


Fig. 12. Documentation file for CSMA/CD suite

support [21] has been realized as an UML-based tool, namely, as a Rational Rose Add-In.

The presented tool supports the activities of new professionals, component specifiers, who should produce consistent UML specifications. The tool offers to a designer specification steps that are constrained and guided by the specification model. It automatically draws diagrams as illustrations of the specification model giving feedback to designer and customer to verify their design requirements. The tool provides automatically the documentation of the design and guarantees consistency of documentation, diagrams and internal model representation.

The model enables both further extensions of the tool, especially to cover behaviour of interface suites, and connecting the tool to existing tools like model checkers.

We are now developing also tool support for composition of interface suites on the basis of the definition of composition given in [12]. The tool support for the composition allows reuse of component specifications [21].

References

1. Booch G., Rumbaugh J., Jacobson I. *The Unified Modeling Language User Guide.* Addison-Wesley, Amsterdam, 1999.

2. Carrier Sense Multiple Access/ Collision Detection (CSMA/CD). *http://webopedia.internet.com/TERM/C/CSMA_CD.html*. 1998.
3. Cheesman J., Daniels J. *UML Components. A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2001.
4. D'Souza D.F., Wills A.C. *Objects, Components and Framworks with UML. The CATALYSIS Approach*. Addison-Wesley, 1999.
5. Eddon G., Eddon H. *Inside COM+ Base Services*. Microsoft Press, 1999.
6. Evans A., France R., Lano K., Rumpe B. The UML as a Formal Modeling Notation. *The Unified Modeling Language. UML'98: Beyond the Notation, Editors: J.Bezivin, P.Muller* , LNCS 1618:336–348, 1998.
7. Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns. Elements of Reusable Object-Oriented Software*. ADDISON-WESLEY, New-York, 1994.
8. Gool L. C. M. van. Cylindrische Componenten Calculus. *Technical University Eindhoven, Department of Computer Science*, 2000.
9. Helm R., Holland I.M., Gangopadhyay D. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. *Proc. ECOOP/OOPSLA '90, ACM*, pages 169–180, 1990.
10. Huber F., Schätz B., Spies K. AutoFocus - Ein Werkzeugkonzept zur Beschreibung verteilter Systeme. *Formale Beschreibungstechniken für verteilte Systeme, Ulrich Herzog, Holger Hermanns (ed.). Universität Erlangen-Nürnberg.*, pages 165–174, 1996.
11. Jonkers H.B.M. ISpec: Towards Practical and Sound Interface Specifications. *Integrated Formal Methods*, LNCS 1945:116–135, 2000.
12. Jonkers H.B.M. Interface-Centric Architecture Descriptions. *In: Working IEEE/IFIP Conference on Software Architecture*, WICSA 2001:113–124, 2001.
13. Lilius J., Paltor I.P. Formalising UML StateMachines for Model Checking. *UML'99. Beyond the Standard, LNCS 1723*, pages 430–445, 1999.
14. Meyer B. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ, 1997.
15. Rational Rose 98i. *Rose Extensibility Reference 2000*. http://www.rational.comwww.se.fh-heilbronn.de/usefulstuff/Rational Rose 98i Documentation.
16. Reenskaug T. *Working with objects*. Manning Publications, 1995.
17. Riehle D. *Framework Design: A Role Modeling Approach. Ph.D. Thesis, No. 13509*. Zrich, Switzerland, ETH Zrich, 2000.
18. Roubtsova E.E., Gool L.C.M.van, Kuiper R., Jonkers H.B.M. A Specification Model For Interface Suites. *UML'01*, LNCS 2185:457–471, 2001.
19. Roubtsova E.E., Katwijk J.van, Toetenel W.J., Pronk C., Rooij R.C.M.de. The Specification of Real-Time Systems in UML. *MCTS2000*, http://www.elsevier.nl/locate/entcs/volume39.html , 2000.
20. Schätz B., Huber F. Integrating Formal Description Techniques. In J. M. Wing, J. Woodcock, and J. Davies, editors, *Proc. World Congress Formal Methods, Vol. 2*, volume 1709 of *Lect. Notes Comp. Sci.*, pages 1206–1225. Springer, Berlin, 1999.
21. SpecTEC project. *Specification tooling for embedded software components. Rational Rose Add-In*. http://www.win.tue.nl/ ella/, June 2002.
22. Szyperski C. *Component Software Beyond Object-Oriented Programming*. Addison-Wesley, New-York, 1998.