Interface suites as contracts : composition of contracts in UML

Interface Suites as Contracts. Composition of Contracts in UML

Ella E. Roubtsova, Ruurd Kuiper, H.B.M.Jonkers
Eindhoven Embedded System Institute (EESI),
Faculty of Mathematics and Computing Science, TU Eindhoven,
Den Dolech 2, P.O.Box 513, 5600 MB Eindhoven The Netherlands
E.Roubtsova@tue.nl, phone +31 040 2478219, fax +31 040 2472078
Philips Research Laboratories Eindhoven, Prof. Holstlaan 4,
5656AA Eindhoven, The Netherlands. hans.jonkers@philips.com

Abstract— **We present a tool for composition of component specifications. The tool uses UML diagrams to model the composition. A fixed component specification is a contract between some roles communicating via interfaces. A contract has a specific underlying model that we present. Composition of contracts is a contract that extends contracts of composites, but does not change them. To guarantee this feature we built some rules of the composition into our tool. Our tool is an ADD-IN of the Rational Rose that allows to produce consistent interface suite specifications in form of UML diagrams and documentation and to reuse such specifications in system design by composition.**

I. Introduction

Component technologies modified modern system design. System designers now do not study component implementation, they study specification of components and adapt components or compose them. After that, designers produce a new component specification for system customers and for the design of more complex systems. System customers also do not study the component implementation. They use the specification to understand the functionality. Customers write their requirements to designers on the basis of the specification. Since even a cursory survey of component technologies shows that both system designers and customers spend most of their time reading and writing component and system specifications - supporting that task by a tool is desirable.

To realize such a tool, there is the Unified Modelling Language [6] which is a standard of software design. This language is understandable both for component customers and for producers. The problem is that there is no commonly accepted definition of a component specification. Instead there are several different definitions [2], [9] and practically used specification methodologies based on those definitions [1], [2]. All of these methodologies are complex to follow without any tool. There is only one way to have feedback from users and to understand how useful the methodology is: it is to support the methodology by a convenient tool.

Supported by PROGRESS grant EES5141

For example, over the past four years the ISpec methodology has been developed and used at Philips [5]. The popularization of the methodology was organized by courses and manuals. Nevertheless, only a narrow group of trained people can take advantage of the methodology.

This is a pity, because the ISpec is a very promising approach closely related with a COM definition of component specification.

- The ISpec considers a component specification as an interaction pattern, possibly involving more then one component and abstracting from components. Because of the abstraction, the ISpec defines a pattern in terms of roles, communicating via interfaces (services) provided by these roles. Such a pattern is termed an interface suite.
- An interface suite is closed in the sense that the environment is specified inside of the suite. This feature makes it possible to use an interface suite as a component specification.
- The ISpec uses a set of templates and UML views to specify an interface suite. The idea of the ISpec is to make those views and templates consistent with a specification model of an interface suite. This specification model is a formal definition of an interface suite. We presented this model in [8].
- The ISpec defines rules for interface suite composition that provides the opportunity to reuse component specification.

Existing UML-based tools are not directly suitable to support the ISpec, because those tools allow different definitions of interfaces, do not guarantee consistency of UML diagrams and do not support the composition of interface suites.

In this paper, we present a tool for composition of component specifications developed using the ISpec composition methodology [4]. The tool generates UML diagrams as models of the composition. A fixed component specification is a contract between some roles communicating via interfaces. A contract has a specific underlying model that we present. Composition of contracts is a contract that must not change the contracts of components. To guarantee this feature, composing a system from suites and modifying existing interface suites should follow specific rules and checks. We built such rules and checks into our tool that is realized as an extension of the Rational Rose [7]. Our tool enables to produce consistent interface suite specifications in the form of UML diagrams and documentation and to reuse such specifications in system design by composition.

Section II of this paper illustrates the problem with interface suite composition by example. In Section III, we identify two levels of abstraction in interface suite definition. Section IV precisely defines the composition of interface suites at the higher level of abstraction. In Section V we describe, following this definition, the tool support for the composition of interface suites at the high level of abstraction. We also show how the organized specification at the high level simplifies the specification at the level of behaviour. Section VI observes related work and provides some conclusions.

II. An example of interface-suite composition

Consider an *Modem-Client* component specified as an interface suite (Fig. 1). It consists of a *Modem* pool and a *Client*. The *Modem* pool provides an interface *IConnect* that contains operations *connect(address)* and *disconnect(address)*. There could be several active clients every moment. A *Client* calls operations of interface *IConnect* .

Consider a *Security* component specified as an interfaces suite. It has roles of *Security Controller* and *User*. The *User* provides interface *IPassword*. The *Security Controller* requests the user password via this interface, recognizes or denies it (Fig. 1).

Let us construct an internet provider which realizes the described modem-client relations together with the security control. There are several ways to combine the functionality.

We could take as the basis the *Modem-Client*

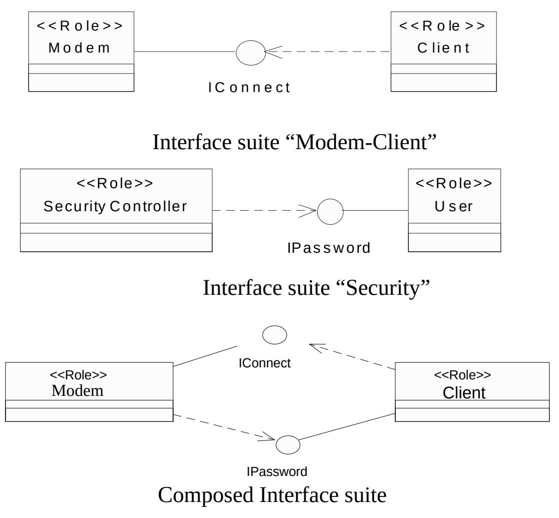

Fig. 1.

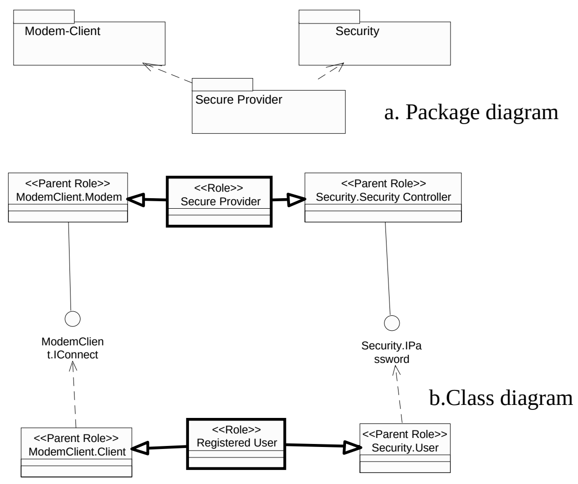

Fig. 2.

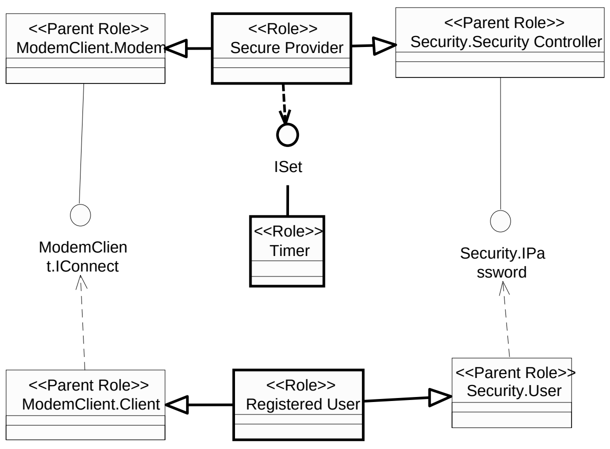

Fig. 3.

component and change it. For example, we could add interface *IPassword* from the *Security* Component (Fig. 1). The specification then loses the names of roles from the *Security* suite - but those names represent relationships! Thus only the client-modem relation is visible. So, combining functionality in this way we can lose information about relations that explain the functionally. If there are more then two kinds of functionality to combine, the specification becomes difficult to read, because all kinds of functionality get combined in one mixed description. Moreover, this way of composition does not help to specify behaviour. As the initial suites are not visible any more, the behaviour of the composition can contain all possible sequences of the interface used.

There is a second way to combine functionality. We exploit the fact that our new role *Secure Provider* plays two roles: the role of *Modem* from the first suite and the role of *Security Controller* from the second suite. In this approach we know the rights and obligations of our *Secure Provider* completely. Our *Registered User* plays both the role of *Client* from the first *Modem-Client* component and the role of *User* from the *Security* component (Fig. 2). In this approach the specifications of the components helps to construct and understand the functionality of a composed component. Furthermore, the behaviour of the initial components should be inherited, so the possible behaviour of the composed component is restricted. It makes it possible to simplify and control the behavioural specification.

It could be the case that we have a new interface or a new role, that was not involved in any components before. Those new roles and interfaces can be specified both in the first and in the second approaches to extend functionality. In the first case this decreases the readability of specification involving a new interface to the specification. In the second way we stress the new interface is allowed to be used only if it does not break the functionality of other components. For example, a role *Timer* can be defined for our *Secure Provider* to control how long a registered user works with the internet source (Fig. 3).

In this paper we advocate the second way of combining functionality. We show how our approach helps in specification of composed systems and how it simplifies reading and understanding of this specification. The construction of composed specifications and corresponding diagrams are supported by the tool that we present and explain in this paper.

III. Three levels in interface suite definition

An interface suite specification defines communications of a finite set of roles via interfaces provided by the roles. A UML class specifies a role. Interfaces provided by a role are disjoint subsets of the operations. Note, that only later in the development process suites are distributed over real components.

An interface suite can be considered at three different levels of abstraction.

We here provide a formal definition of interface suite at the highest level of abstraction, because this is where the present version of the tool operates on: the rigorous definition of composition is based on this definition.

• *At the interface-role level* of abstraction, an interface suite is a graph

$$(R, I, PI, RI)$$

with two kinds of nodes:
– R is a finite set of roles depicted by boxes; $R \neq \emptyset$.
– I is a finite set of interfaces depicted by circles, I can be empty;
and two kinds of relations
– $PI(R, I)$ defines interfaces provided by roles:

$$PI(R, I) = \{(r, i) | r \in R, i \in I\}.$$

The relation is depicted by a line between a role and an interface (Fig. 1).
– $RI(R, PI(R, I))$ defines interfaces required by roles.

$$RI(R, PI(R, I)) = \{(r, (r, i)) | r \in R, i \in I, (r, i) \in PI(R, I)\}.$$

The relation is drawn by a dashed line with an arrow connecting a role and a provided interface. The arrow is directed to the interface (Fig. 1).

Changing any of the tuple elements produces a new graph. We make a distinction between an *interface-role graph* of type (R, I, PI, RI) that can be changed and an *interface suite contract that can not be changed.* So, the set of roles, provided and required interfaces of an interface-suite contract can not be changed.

• *At the operational level* of abstraction, an interface set I is specified via set of operations Op, their parameters P and results R :

$$I = Op \times P \times R.$$

This level narrows the space for the behavioural specification.

• *At the behavioural level*, an interface suite is a set of traces of operation calls and returns as we have defined it in [8] .

So, at the interface-role level of abstraction we abstract from detailed definition of interfaces and behaviour. This abstraction allows to define the composition of interface suites in such a way that the composition allows saving and reusing the operational and behavioural specification of composed interface suites in the result of the composition without mentioning these.

IV. Composition of interface suites

The idea of the definition we give below is to save the specification of composed contracts and the structure of the composition in the result of the composition. Such an approach helps to read and understand the composed specification. The approach allows also to correct mistakes in parents independently from children and distribute correct parent-contracts through all children-contracts.

The structure of the composition can be represented at two diagram levels: UML package diagrams and interface role diagrams which are expressed as UML class diagrams, where roles are represented by classes.

For the package diagrams, when a new contract is constructed by inheritance this is represented by just dependency arrows $- - - - >$ Fig.2(a).

A UML class diagram allows to illustrate some more details of composition. Fig.2(b) and Fig.3 show (fixed) contracts, indicated by grey roles and interfaces. It also shows changeable interface role graphs, drawn in black lines. A black graph can not be considered a fixed contract. For example in Fig. 3 for Secure Provider only one interface is drawn, but because of inheritance during the specialization, Secure Provider has additionally two provided interfaces, IConnect and IPassword.

Informally, any role-child in the interface-role graph always inherits all provided interfaces of its parent-roles.

The inheritance of required interfaces is different. Role x_* which specializes role x from an old contract is an element a new contract. It is possible that in this new contract role x_* requires other interfaces. Therefore, the role does not automatically inherit the required interfaces from role x. So, there is the option to define new required interfaces. However, if role x_* requires the same interfaces as role x requires from role r, then some role r_* should exist in the new contract to provide those interfaces. This role r_* can be supplied through inheritance from the role r of the old contract. In the example, because *Registered User* has required interfaces *IConnect* and *IPassword*, there has to be *Secure Provider* that inherits from roles *Modem* and *Security Controller* from the old contracts.

A simple case of composition is where two contracts are combined to a new one and the two subsystems and no specialization occurs. Intuitively, this means that no communication between roles of those contracts occurs. This we name parallel composition. At the level of package diagram, it cannot be seen yet which kind of composition is used; only at the interface role level the distinction between parallel composition and specialization becomes visible.

These ideas are formalized in the following
Definition:
1. Let two interface suite contracts C_1, C_2 be given:

$$C_1 = (R_1, I_1, PI_1(R_1, I_1), RI_1(R_1, PI_1(R_1, I_1)))$$

$$C_2 = (R_2, I_2, PI_2(R_2, I_2), RI_2(R_2, PI_2(R_2, I_2))),$$
$$R_1 \cap R_2 = \emptyset, I_1 \cap I_2 = \emptyset.$$

Parallel composition of given interface suite contracts is an interface suite contract:

$$C = C_1 \parallel C_2,\ C = (R, I, PI, RI):$$

- $R = R_1 \cup R_2$,
- $I = I_1 \cup I_2$,
- $PI(R, I) = PI_1(R_1, I_1) \cup PI_2(R_2, I_2)$,
- $RI(R, I) = RI_1(R_1, PI_1(R_1, I_1)) \cup RI_2(R_2, PI(R_2, I_2))$.

2. Let an interface suite contract C and an interface suite graph G be given:

$$C = (R_c, I_c, PI_c(R_c, I_c), RI_c(R_c, PI_c))$$
$$G = (R_g, I_g, PI_g(R_g, I_g), RI(R_g, PI_g))$$
$$R_c \cap R_g = \emptyset, I_c \cap I_g = \emptyset.$$

Specialization of interface suite contract C_1 **by interface-suite-graph** G is an interface suite contract:

$$S = G \rhd C,\ S = (R_s, I_s, PI_s, RI_s), where$$

- $R_s = R_c \cup R_g$,
- $I_s = I_c \cup I_g$,
- $PI_s(R_s, I_s) = PI_c(R_c, I_c) \cup PI_g(R_g, I_g) \cup PI_*(R_*, I_*)$, where

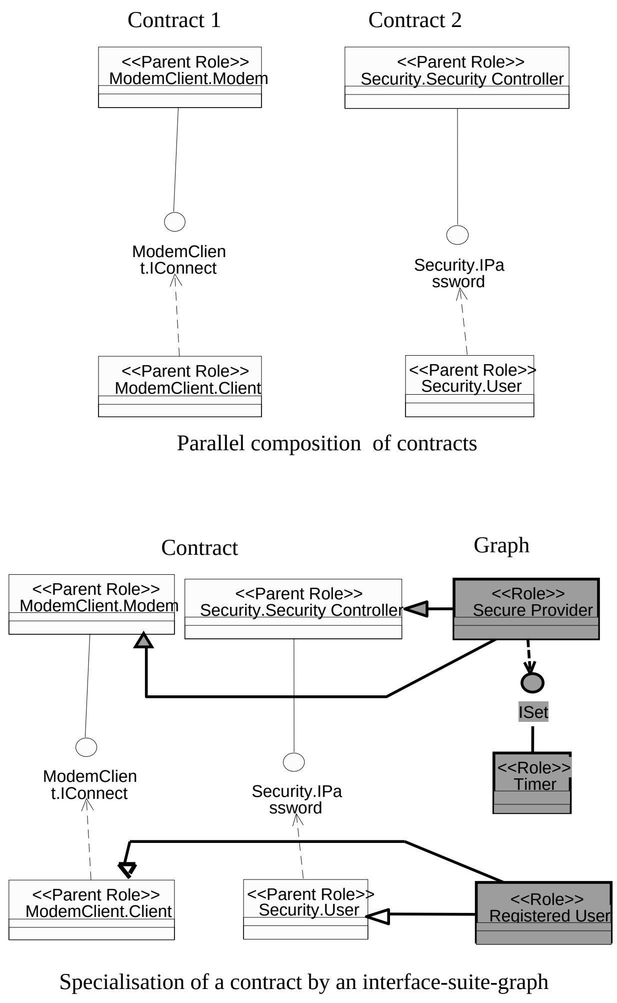
Contract 1
Contract 2
<<Parent Role>> ModemClient.Modem
<<Parent Role>> Security.Security Controller
ModemClient.IConnect
Security.IPassword
<<Parent Role>> ModemClient.Client
<<Parent Role>> Security.User
Parallel composition of contracts
Contract
Graph
<<Parent Role>> ModemClient.Modem
<<Parent Role>> Security.Security Controller
<<Role>> Secure Provider
ISet
<<Role>> Timer
ModemClient.IConnect
Security.IPassword
<<Parent Role>> ModemClient.Client
<<Parent Role>> Security.User
<<Role>> Registered User
Specialisation of a contract by an interface-suite-graph

Fig. 4. Composition of interface-suite contracts

$R_* \subseteq R_g$, $I_* \subseteq I_c$,

$$PI_*(R_*, I_*) = \{(r_*, i) \mid \quad r_* \in R_*, \quad i \in I_c, \; \exists r \in R_c, \; (\text{ such that } r_* \rhd r, \quad \text{and} \; (r,i) \in PI_c(R_c, I_c))\}.$$

- $RI_s\,(R_s, I_s) = RI_c(R_c, PI_c(R_c, I_c) \cup RI_g(R_g, PI_g(R_g, I_g)) \cup RI_*(R_*, PI_*(R_*, I_*)$.

$R_* \subseteq R_g$, $I_* \subseteq I_c$,
$RI_*(R_*, PI_*(R_*, I_*) = \{(x_*, (r_*, i)) \quad |$
$r_*, x_* \in R_* \quad i \in I_c$,
$\exists r \in R_c$, such that
$r_* \rhd r$, $\exists x \in R_c$, such that $x_* \rhd x$, such that
$((r,i) \in PI_c(R_c, I_c) \quad$ and $(x, (r,i) \in RI_c(R_c, PI_c))\}$.
3. There are no other interface suite contracts.

This definition guarantees that a role can not specialize itself and parent-roles can not specialize child-roles. The definition covers both cases of specialization mentioned in section II. Both specialized roles and completely new roles belong to the changeable part, an interface suite graph.

V. Tool support for composition of interface suites

In this section we show how the composition of interface suites defined in section IV is supported by our tool and how the composition simplifies specification of behaviour.

The main ideas of our tool are the following:
1. to collect the information for an interface suite contract and interface suite graph in forms,
2. to adapt the Rational Rose model for an interface suite specification,
3. to represent the composed interface suite by UML diagrams which are automatically drawn by the tool using information from forms,
4. to direct a specifier through the specification process allowing reuse of specifications and resulting on a consistent specification.

We further explain these ideas.

1. To collect the information by forms we define our reaction on Rose events and customize the Rose specification for Rose classes representing roles.

2. Adapting the Rose model, we represent an interface suite graph by one Rose model and save it as an *mdl*-file. An interface suite graph is created as an interface role diagram with an additional data-type diagram. If we decide to use the interface suite as a contract, we save it as a file with extension *sui* (this is a Rose category-file). The Rose model which represents a composed interface suite in our tool is a unidirectional tree. A node of this tree is a link to a package. The root of this tree is the link to a new interface suite graph as a model. There is a slight difference between a model and a package in Rose. A package is always a part of a model. However, a package can be opened in Rose as a model, therefore we have a model as the root. Packages in our tool are interface suite contracts placed in separate files with extension *sui.* Interface suites are implemented as Rose Controlled Units.

3. There are three diagrams that belong to the Rose model for an interface suite inour tool: a package diagram, an interface-role diagram, a data-type diagram. All diagrams are variants of the Rose class diagram. A package diagram consists of packages only (Fig. 2(a)). An interface-role diagram represents roles by classes and shows interfaces provided and required by roles (Fig. 2(b)). A data-type diagram depicts data types by Rose classes which have no operations and interfaces [8].

4. To direct a specifier through the specification process we introduce an order by opening forms and collecting information according to the definition of interface suite composition and the interface suite model:

• interface suite graphs can be defined as interface role diagrams with the corresponding data type diagram;
• interface suite graphs can be saved as interface-suite contracts for reuse;
• composition of interface suite contracts is performed using forms, a package diagram and an interface role diagram;
• specialization of interface suite contracts is defined at the corresponding interface-role diagram;
• the results of composition can be fixed as an interface-suite contract.

A. An Interface suite as the package diagram

The structure of a composed interface suite is represented in our tool by a package diagram. The diagram belongs to the model, it can be opened and seen in the Rose Logical View. The Rose model file of an interface suite links only the first level of controlled units. So, if a controlled unit from the first level also holds the link to other unit, our tool draws this unit at the package diagram.

The supplier-client relations between nodes in this package diagram (Fig. 2(a)) are drawn by dependency arrow $supplier\, package --- >$ *client package*. This arrow means in our tool that the client inherits from the supplier and has a link to the specification of the supplier to visualize it at the interface role diagram level. We do not allow other dependencies between packages. Any change of a supplier package inside this model is forbidden. The supplier-suite specification can be modified only if it is opened as a self-standing model. In this way the information in the package diagram is used by the tool to restrict modification at other levels.

A.1 Composition of suites by users at the package level.

Our tool provides a form to name a new interface suite and to choose parent suites from the set of suifiles. All chosen suites and the new one are drawn at the package diagram automatically. The result of filling in this form is reflected also at the Logical View of Rose. The first level of the Logical View represents the new interface suite, at the second level you can see parent suites. We have lists of parent roles with the specification from this composition in the internal model.

For example, a client suite *Secure Provider* inherits from supplier-suites *Modem-Client* and *Security. Modem-Client* and *Security* can not be modified in the model *Secure Provider* . However, we can open and modify each of the suppliers as a Rose model.

The result of the composition is represented also at the interface-role diagram: all roles and interfaces are depicted. The possible changes of the new interface suite being now the composition of suites are restricted by definition of a new interface suite graph and by specialization of roles of parent suite by roles of the graph. Those changes are reflected at the interface-role diagram and the supporting customized specification forms.

B. An Interface suite at the interface-role diagram level

B.1 Specialization of an interface suite contract by an interface suite graph.

To define a new specialized role, the user puts a new role on the interface-role diagram, opens the customized specification to give a name to this role, say *Registered User.* Using tab *Specialization* (Fig.5) he/she chooses a role from the list of parent roles, say *Client* from *Modem-Client* suite and *User* from the *Security* suite. The specialization relation is drawn automatically at the class diagram. The new roles appear in the Logical View at the second level of packages. The specifications of all interfaces provided by *Client* are inherited in *Registered User. Registered User* requires all interfaces that are required by *User.*

Specialization of a role is supplemented by definition of new elements of an interface-role graph:
1. defining new provided interfaces that were not defined in parent suites; the interfaces can be composed only from new operations;
2. requiring new interfaces from the new provided interfaces that were not defined in parent suites;
3. creating new roles and new interfaces.

1. New provide interfaces are defined by a special tab in the new role specification. In accordance with the definition of a contract, new interfaces can not be defined for parent roles.

2. The required interfaces are chosen from the set of provided interfaces on a special tab in the new role specification. There is no such possibility for parent roles because they belong to a contract. In this way the tool prevents mistakes of a designer.

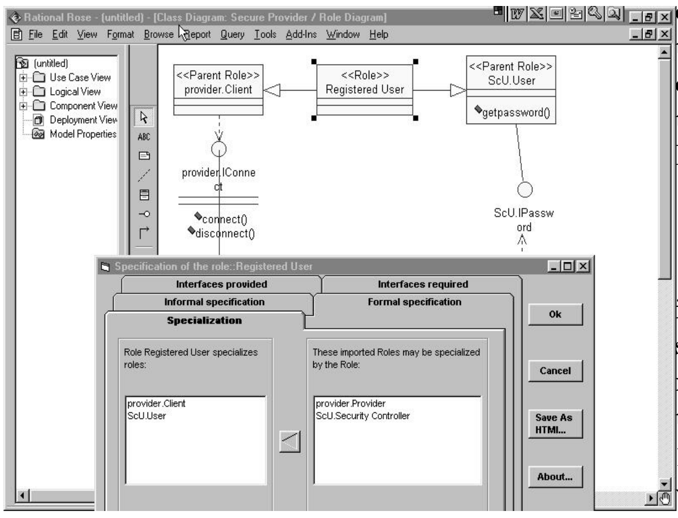

Fig. 5.

3. If a new role does not specialize any existing one from parent suites, than the user does not use the specialization tab. For example, a new role *Timer* is defined without any specialization. Using tab *Provided Interfaces* a new interface can be specified, for example the *ISet* interface of the role *Timer*. Another new role, *Secure Provider* requires the new interface. This relation is defined by tab *Required Interfaces* in the specification the role *Secure Provider*.

B.2 The result of the composition.

The tool makes impossible the modification of parents inside the current model, both on diagrams and in forms. A modified interface with modified operation is used only in the child suite.

The result of the composition can be saved as a mdl-file to enable its modification and as a sui-file to reuse it in other composition.

The tool generates the corresponding documentation as an HTML-file. This file contains a package diagram and an interface suite diagram, which allow to quickly understand the functionality of a complex suite.

B.3 Modification of parent suites.

If we save the current model as mdl-file, our tool closes the model to the link-structure to make it possible to open the model again and to modify the parent suites independent from children.

The modification of parents should be done carefully. Adding new operations and attributes to a parent is well possible. But we can not delete roles, interfaces, operations and attributes because children can use them. Modification of specification can break the correspondence between specification parents and children. Thus, this correspondence is checked by the tool each time one loads of a child-suite using flags of Rose controlled units.

C. Specification of operations and behaviour

The composition of interface suite contracts defined in this paper and supported by our tool simplifies specification of behaviour. This effect comes from the reuse of parent specifications. Additional changes of the specification is restricted by
1. defining new operations for specialized roles that were not defined in parent suites;
2. adding new attributes to specialized roles;
3. specializing an operation from an inherited interface.

1.2. New operations and new attributes have their own tab forms in the role specification form. New attributes and operations can demand definition of new data types. This is supported by the tool. Completely new data types are defined at the type diagram. Moreover, new data types can be defined by specialization of basic and parent data types at the data type diagram. Then we can choose the necessary data types from the data-type list when we define attributes, parameters and results of operations.

3. The forms enable specification of operations by pre-, postconditions and action clauses [4]. The specialization of an operation from an inherited interface is restricted by
• weakening the precondition of this operation;
• adding code to the action clause. The code can modify new attributes;
• strengthening the postcondition of the operation.
So, the specification of the operation is copied to the specialized role. It is allowed to write new preconditions and postconditions using the set of old attributes. The user compares the old and the new expressions. The action clause can be changed using only new attributes. In this way the tool simplifies the behaviour definition.

VI. Related work and Conclusion

The use of contracts for component specification is accepted now: a component should not be used if its requirements to environment are not satisfied. However, the form of the contract representation in the UML is still not clear. In paper [10] it was proposed to define contracts at the component diagram as "requirements on a component environment". A composed contract is introduced as a super-class of interface with the Non Functional Contract subclass. We find useful the composition of contracts at the component diagram (it is similar to our representation of a contract by a package diagram), but we consider it as not sufficient for specializing the contract through inheritance. Namely, important elements of a contract are hidden in this way. A contract has a specific structure: services, providers, clients. A composed contract should be defined in terms of those elements. So, we open the internal structure of a contract and allow contract specification at different levels of abstraction. We have shown what can be done at the interface role level.

Our approach, based on the ISpec methodology, is closely related to the CATALYSIS [2] architectural level of design. An interface suite is a pattern of the CATALYSIS. Both a suite and a pattern are represented by a package. The differences are that we involve interfaces at the architectural level, making visible the structure of contract, and we use another interpretation of package relation. The CATALYSIS uses different relations between packages, such as *refine* and *apply* [3]. Our package can only inherit from other parent-packages. This means the roles of the child-package can inherit from and specialize the roles of parents with the corresponding provided and required interfaces. So, we do not substitute the pattern as in the CATALYSIS, but we inherit and specialize it by other combination of roles. In this way we do not change the initial contracts (patterns) in the new suite and always can read the initial functionality. The restrictions introduced by ISpec give a precise meaning to contract in the UML, to package diagram in the UML and to the composition of contracts at the UML class diagram. The restrictions of ISpec allow also to automatically construct those UML diagrams using specification of contracts by templates and keep the diagrams and templates consistent on the basis of the contract model.

The new tool presented in this paper supports the activities of different groups of professionals involved in the system development cycle: component specifiers, programmers, customers of components, system designers. The component specifiers produce the specification for component programmers, for component customers and for component system designers.

Programmers need a precise and consistent specification to implement a component. Component specifiers can produce such a specification with the help of our tool. In previous work we defined the semantics of a specification as a set of sequences of operation calls and returns [8]. This model also facilitates checking the consistency of different views on a component produced in our tool.

The customers of components want a visual specification to quickly study and understand the component functionality and to compare it with their requirements. Component specifiers can produce that visual specification with the help of our tool in the form of consistent UML diagrams and documentation.

System designers need both the precise and the visual forms of the component specification. The system designers would like to reuse component specifications constructing a system from components. The specification produced with the help of our tool can be reused, because composition is precisely defined and supported by the tool.

Possible extensions of our tool can cover the behavioural views on component specification, for example, UML sequence diagrams. Our specification model is suitable for behavioural views as it is shown in [8]. This also allows connecting our tool with model checkers.

References

[1] Cheesman J., Daniels J. *UML Components. A Simple Process for Specifying Component-Based Software.* Addison-Wesley, 2001.

[2] D'Souza D.F., Wills A.C. *Objects, Components and Framworks with UML. The CATALYSIS Approach.* Addison-Wesley , 1999.

[3] D'Souze D., Mauhg I. Precise Component Architectures,Presented at OOPSLA. *http://www.catalysis.org/omg/*, 1999.

[4] Jonkers H.B.M. ISpec: Towards Practical and Sound Interface Specifications. *Integrated Formal Methods*, LNCS 1945:116–135, 2000.

[5] Jonkers H.B.M. Interface-Centric Architecture Descriptions. *In: Working IEEE/IFIP Conference on Software Architecture*, WICSA 2001:113–124, 2001.

[6] OMG. *Unified Modeling Language Specification v.1.3.* ad/99-06-10, http://www.rational.com/uml/resources/documentation/index.jsp, June 1999.

[7] Rational Rose 2000. *Rose Extensibility Reference 2000.*

[8] 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.

[9] Szyperski C. *Component Software Beyond Object-Oriented Programming.* ADDISON-WESLEY, New-York, 1998.

[10] Weis T., Becker C., Geihs K., Plouzeau N. A UML Metamodel for Contract Aware Components. *UML'01*, LNCS 2185:442–456, 2001.