

Software Evolution in Componentware using Requirements/Assurances Contracts*

Andreas Rausch

Technische Universität München
Institut für Informatik
Arcisstrasse 21
80290 Munich, Germany
+49 89 289 28362
rausch@in.tum.de

ABSTRACT

In practice, pure top-down and refinement-based development processes are not sufficient. Usually, an iterative and incremental approach is applied instead. Existing methodologies, however, do not support such *evolutionary development processes* very well. In this paper, we present the basic concepts of an overall methodology based on componentware and software evolution. The foundation of our methodology is a novel, well-founded model for component-based systems. This model is sufficiently powerful to handle the fundamental structural and behavioral aspects of componentware and object-orientation. Based on the model, we are able to provide a clear definition of a software evolution step.

During development, each evolution step implies changes of an appropriate set of development documents. In order to model and track the dependencies between these documents, we introduce the concept of *Requirements/Assurances Contracts*. These contracts can be rechecked whenever the specification of a component evolves, enabling us to determine the impacts of the respective evolution step. Based on the proposed approach, developers are able to track and manage the software evolution process and to recognize and avoid failures due to software evolution. A short example shows the usefulness of the presented concepts and introduces a practical description technique for *Requirements/Assurances Contracts*.

Keywords

Software evolution, componentware, formal methods, contracts, description techniques, software architecture, object-orientation.

1 INTRODUCTION

Most of today's software engineering methodologies are

based on a top-down development process, e.g., Object Modeling Technique (OMT) [27], Objectory Process [15], or Rational Unified Process (RUP) [14]. All these methodologies share a common basic idea: During system development a model of the system is built and stepwise refined. A refinement step adds additional properties of the desired system to the model. At last the model is a sufficiently fine, consistent, and correct representation of the system under consideration. It may be implemented by programmers or even partly generated. Surely, all of these processes support local iterations, for instance the RUP allows iterations during analysis, design or implementation. However, the overall process is still based on refinement steps to improve the specification model and finally end with the desired system. In formal approaches, like ROOM [3] or FOCUS [4] the concept of refinement is even more strict.

These kinds of process models involve some severe drawbacks: Initially, the customer often does not know all relevant requirements, cannot state them adequately, or even states inconsistent requirements. Consequently, many delivered systems do not meet the customer's expectations. In addition, top-down development leads to systems that are very brittle with respect to changing requirements, because the system architecture and the involved components are specifically adjusted to the initial set of requirements. This is in sharp contrast to the idea of building a system from truly reusable components, as the process does not take already existing components into account. Beyond this, software maintenance and life-cycle are not supported. This is extreme critical as, for instance, nowadays maintenance takes about 80 percent of the IT budget of Europe's companies in the average, and 20 percent of the user requirements are obsolete within one year [21].

However, software evolution as a basic concept is currently not well supported. In our opinion, this is partly due to the lack of a suitable overall componentware methodology with respect to software evolution. Such a methodology should at least incorporate the following parts [26]:

- The common **system model** provides a well-defined conceptual framework for componentware and software evolution is required as a reliable foundation.
- Based on the system model a set of **description techniques** for componentware are needed. Devel-

*This paper originates from the research in the project A1 "Methods for Component-Based Software Engineering" at the chair of Prof. Dr. Manfred Broy, Institut für Informatik, Technische Universität München. A1 is part of "Bayerischer Forschungsverbund Software-Engineering" (FORSOFT) and supported by Siemens AG, Department ZT.

opers need to model and document the evolution of a single component or a whole system.

- Development should be organized according to a **software evolution process**. This includes guidelines for the usage of the description techniques as well as reasonable evolution steps.
- To minimize the costs of software evolution, systems should be based on **evolution-resistant architectures**. Such architectures contain a common basic infrastructure for components, like DCOM [2], CORBA [22], or Java Enterprise Beans [16]. But even more important are business-oriented standard architectures, that are evolution-resistant.
- At last, all former aspects should be supported by **tools**.

The contribution of this work can be seen from two different perspectives. From the viewpoint of specification methods, it constitutes a sophisticated basic system model as solid foundation for new techniques in the areas of software architectures, componentware, and object-orientation. From a software engineering perspective, it provides a clear understanding of software evolution steps in an evolutionary development process. Moreover, it offers a new description technique, called *Requirements/Assurances Contracts*. These contracts can be rechecked whenever the specification of an component evolves. This allows us to determine the impacts of the respective evolutionary step.

The paper is structured as follows. Section 2 provides the basic definitions to model dynamics in a component-based system. In the next section, Section 3, we specify the observable behavior of an entire component-based system based on former definitions. In Section 4, we provide a composition technique that enables us to determine the behavior of the system from the behavior of its components. Section 5 will complete the formal model with a simple concept of types. These types are described by development documents. Section 6 introduces our view of development documents and evolution steps on those documents. In Section 7 we present the concept of *Requirements/Assurances Contracts* to model explicitly the dependencies between development documents. Section 8 provides a small example to show the usefulness of the proposed concepts in case of software evolution. A short conclusion ends the paper.

2 BASIC CONCEPTS

This section elaborates the basic concepts and notions of our formal model for component-based systems. The system model incorporates two levels: The *instance-level* represents the individual operational units of a component-based system that determine its overall behavior. We distinguish between component, interface, connection, and variable instances. We define a number of relations and conditions that model properties of those instances. The *type-level* contains a normalized abstract description of a subset of common instances with similar properties.

Although some models for component-based and object-

oriented systems exist, we need to improve them for an evolutionary approach. Formal models, like for instance FOCUS [4] or temporal logic [17], are strongly connected with refinement concepts (cf. Section 1). Furthermore, these methods do not contain well elaborated type concepts or sophisticated description techniques, that are needed to discuss the issues of software evolution (as in case of evolution the types and the descriptions are usually evolved). Moreover, in practice formal methods are not applicable, since formal models are too abstract and do not provide a realistic view on today's component-based systems.

Architectural description languages, like MILs, Rapide, Aesop, UniCon, are other, less formal approaches. As summarized in [5] they introduce the concepts of components and communication between them via connectors, but do not consider all behavior-related aspects of a component system. In a component-based system behavior is not limited to the communication between pairs of components, but also includes changes to the overall connection structure, the creation and destruction of instances, and even the introduction of new types at runtime. In the context of componentware and software evolution, these aspects are essential because dynamic changes of a system may happen both during its construction at design-time as well as during its execution at runtime, either under control of the system itself or initiated by human developers.

Other approaches, like pre/post specifications cannot specify mandatory external calls that components must make. This restriction also applies to Meyer's design by contract [20] and the Java Modeling Language (JML) [18], although they are especially targeted at component-based development.

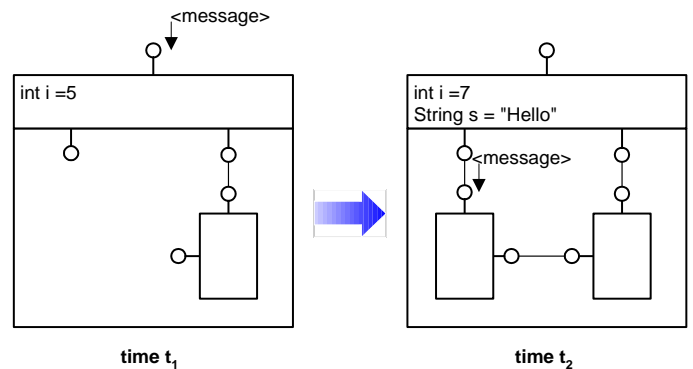


Figure 1: A Component System: Behavioral Aspects

For that reason, we elaborated a novel, more realistic model. We claim, that the presented formal model is powerful enough to handle the most difficult aspects of component-based systems (cf. Figure 1): dynamically changing structures, a shared global state, and at last mandatory call-backs. Thus, we separate the behavior of component-based systems into these three essential parts:

- **Structural behavior** captures the changes in the system structure, including the creation or deletion of instances and changes in the connection as well as aggregation structure.
- **Variable valuations** represent the local and global data space of the system. This enables us to model a shared global state.
- **Component communication** describes message-based asynchronous interaction between components. Thus, we can specify mandatory call-backs without problems.

In the following sections we first come up with definitions for these three separate aspects of behavior in component-based systems.

STRUCTURAL BEHAVIOR

Components are the basic building blocks of a component-based system. Each component possesses a set of local attributes, a set of sub-components, and a set of interfaces. Interfaces may be connected to other interface via connections. During runtime some of these basic building blocks are created and deleted.

In order to uniquely address the basic elements of a component-based system, we introduce the disjoint sets: $\text{COMPONENT} \cup \text{INTERFACE} \cup \text{CONNECTION} \cup \text{VARIABLES} \subseteq \text{ID}$.

As Figure 1 shows, a component-based system may change its structure dynamically. Some of these basic elements may be created or deleted (**ALIVE**). New interfaces may be assigned to components (**ASSIGNED**). Interfaces may be connected to or de-connected from other interfaces (**CONNECTED**). New Subcomponents may be aggregated by existing parent-components (**PARENT**). The following definitions cover the structural behavior of component-based systems:

$$\begin{aligned}
\text{ALIVE} &=_{\text{def}} \text{ID} \rightarrow \text{BOOLEAN} \\
\text{ASSIGNED} &=_{\text{def}} \text{INTERFACE} \rightarrow \text{COMPONENT} \\
\text{CONNECTED} &=_{\text{def}} \text{CONNECTION} \rightarrow \\
&\quad \{\{i, j\} \mid i, j \in \text{INTERFACE} \wedge i \neq j\} \\
\text{PARENT} &=_{\text{def}} \text{COMPONENT} \rightarrow \text{COMPONENT}
\end{aligned}$$

Note, that this approach is strong enough to handle not only dynamic changing connections structures in systems but also mobile systems as, for instance it covers mobile components that migrate from one parent component to another (**PARENT**).

VARIABLE VALUATIONS

Usually, the state space of a component-based system is not only determined by its current structure but also by the values of the component's attributes (cf. Figure 1). With **VALUES** the set of all possible valuations for attributes and parameters are denoted. They are in essence mappings of variables (attributes, parameters, etc.) to values of appropriate type (**VALUATION**). These variables belong to components, characterizing the state of the component (**ALLOCATION**). The following definitions cover the variable valuations of

component-based systems:

$$\begin{aligned}
\text{ALLOCATION} &=_{\text{def}} \text{VARIABLES} \rightarrow \text{COMPONENT} \\
\text{VALUATION} &=_{\text{def}} \text{VARIABLES} \rightarrow \text{VALUES}
\end{aligned}$$

Later on we will allow components to change the values of other component's variables (cf. Section 4). Thus, we can model shared global states as well-known from object-oriented systems. Note, we do not elaborate on the underlying type system of the variables and values here, but assume an appropriate one to be given.

COMPONENT COMMUNICATION

Based on existing formal system models, e.g. **FOCUS** [4], sequences of messages represent the fundamental units of communication. In order to model message-based communication, we denote the set of all possible messages with M , and the set of arbitrary finite message sequences with M^* . Within each time interval components resp. interfaces receive message sequences arriving at their interfaces resp. connections and send message sequences to their respective environment, as given by the following definition (cf. Figure 1):

$$\text{EVALUATION} =_{\text{def}} \text{INTERFACE} \rightarrow M^*$$

The used message-based communication is asynchronous, like **CORBA** one-way calls. Hence, call-backs based on those asynchronous one-way calls can be explicitly specified within our model. But one cannot model "normal" blocking call-backs as usual in object-oriented programming languages. However, our observation shows, call-backs need not to be blocking calls. Often call-backs are used to make systems extensible. In layered system architectures they occur as calls from lower into higher layers in which they are known as up-calls. These up-calls are usually realized by asynchronous events (cf. the *Layers Pattern* in [9]). Another representative application of call-backs as asynchronous events is the *Observer Pattern* [11]. There the observer may be notified via asynchronous events if the observed object has changed. To sum up, we believe call-backs as supported in our model are powerful enough to model real component-based systems under the assumption that a middleware supporting asynchronous message exchange is available.

SYSTEM SNAPSHOT

Based on all former definitions we are now able to characterize a snapshot of a component-based system. Such a snapshot captures the current structure, variable valuation, and actual received messages. Let **SNAPSHOT** denote the type of all possible system snapshots:

$$\begin{aligned}
\text{SNAPSHOT} &=_{\text{def}} \text{ALIVE} \times \text{ASSIGNED} \times \\
&\quad \text{CONNECTED} \times \text{PARENT} \times \text{ALLOCATION} \times \\
&\quad \text{VALUATION} \times \text{EVALUATION}
\end{aligned}$$

Let **SYSTEM** denote the infinite set of all possible systems. A given snapshot $\text{snapshot}_s \in \text{SNAPSHOT}$ of a system $s \in \text{SYSTEM}$ ¹ is tuple that capture the current active sets of components, interfaces, connections,

¹In the remainder of this paper we will use this shortcut. Whenever we want to assign a relation X to a system $s \in \text{SYSTEM}$ (component $c \in \text{COMPONENT}$) we say X_s (X_c).

and variables, the current assignment of interfaces to components, the current connection structure between interfaces, the current super-/sub-component relationship, the current assignment of variables to components, the current values of components, and finally the current messages for the components.

3 TIME AND SYSTEM BEHAVIOR

Similar to related approaches [4], we regard time as an infinite chain of time intervals of equal length. We use \mathbf{N} as an abstract time axis, and denote it by T for clarity. Furthermore, we assume a time synchronous model because of the resulting simplicity and generality. This means that there is a global time scale that is valid for all parts of the modeled system. We use *timed streams*, i.e. finite or infinite sequences of elements from a given domain, to represent histories of conceptual entities that change over time. A *timed stream* (more precisely, a stream with discrete time) of elements from the set X is an element of the type

$$X^T \stackrel{\text{def}}{=} \mathbf{N}^+ \rightarrow X$$

with $\mathbf{N}^+ = \mathbf{N} \setminus \{0\}$. Thus, a timed stream maps each time interval to an element of X . The notation x^t is used to denote the element of the valuation $x \in X^T$ at time $t \in T$.

Streams may be used to model the behavior of systems. Accordingly, SNAPSHOT^T is the type of all system snapshot histories or simply the type of the behavior relation of all possible systems:

$$\begin{aligned} \text{SNAPSHOT}^T &\stackrel{\text{def}}{=} \text{ALIVE}^T \times \text{ASSIGNED}^T \times \\ &\text{CONNECTED}^T \times \text{PARENT}^T \times \text{ALLOCATION}^T \times \\ &\text{VALUATION}^T \times \text{EVALUATION}^T \end{aligned}$$

Let $\text{Snapshot}_s^T \subseteq \text{SNAPSHOT}^T$ be the behavior a system. A given snapshot history $\text{snapshot}_s \in \text{Snapshot}_s^T$ is a timed stream of tuples that capture the changing snapshots snapshot_s^t over time $t \in T$.

Obviously, a couple of consistency conditions can be defined on such a formal behavior specification Snapshot_s^T . For instance, we may require that all assigned interfaces are assigned to an active component:

$$\forall i \in \text{INTERFACE}, t \in T. \text{assigned}_s^t(i) = c \implies \text{alive}_s^t(c) = \text{true}$$

Furthermore, components may only be connected via their interfaces if one component is the parent of the other component or if they both have the same parent component. Connections between interfaces of the same component are also valid:

$$\begin{aligned} \forall a, b \in \text{COMPONENT}, c \in \text{CONNECTION}, \\ i, j \in \text{INTERFACE}, t \in T. \text{connected}_s^t(c) = \{i, j\} \wedge \\ \text{assigned}_s^t(i) = a \wedge \text{assigned}_s^t(j) = b \implies \\ \text{parent}_s^t(a) = b \vee \text{parent}_s^t(b) = a \vee \\ \text{parent}_s^t(a) = \text{parent}_s^t(b) \end{aligned}$$

We can imagine an almost infinite set of those consistency conditions. A full treatment is beyond the scope of this paper, as the resulting formulae are rather lengthy. A deeper discussion of this issue can be found in [1].

4 BEHAVIOR COMPOSITION

In the previous sections we have presented the observable behavior of a component-based system. This behavior is a result of the composition of all component behaviors. To show this coherence we first have to provide behavior descriptions of a single component. In practice are transition-relations an adequate behavior description technique. In our formal model we use a novel kind of transition-relation: In contrast to “normal” transition-relations—a relation between *predecessor state* and *successor state*—the presented transition relation is a relation between a *certain part of the system-wide predecessor state* and a *certain part of the wished system-wide successor state*:

$$\text{BEHAVIOR} \stackrel{\text{def}}{=} \text{SNAPSHOT} \rightarrow \text{SNAPSHOT}$$

Let $\text{behavior}_c \subseteq \text{BEHAVIOR}$ be the behavior of a component $c \in \text{COMPONENT}$. The informal meaning of each tuple in behavior_c is: If the specified part of the system-wide predecessor state fits (given by the first snapshot), the component wants the system to be in the system-wide successor-state in the next step (given by the second snapshot). Consequently we need some specialized runtime system that collects at each time step from all components all wished successor states and composes a new well-defined successor state for the whole system.

The main goal of such a runtime system is to determine the system snapshot snapshot_s^{t+1} from the snapshot snapshot_s^t and the set of behavior relations behavior_c of all components. In essence, we can provide a formulae to calculate the system behavior from the initial configuration snapshot_s^0 , the behavior relations behavior_c , and external stimulations via messages at free interfaces. Note, free interfaces are interfaces that are not connected with other interfaces and thus can be stimulated from the environment.

First we have to calculate all transition-tuples of all active components:

$$\text{behavior}_s^t \stackrel{\text{def}}{=} \bigcup_{c \in \text{COMPONENT}. \text{alive}_s^t = \text{true}} \text{behavior}_c$$

Now, we can calculate all transition-tuples of the active components that fit the actual system state. Let transition_s^t be the set of all those transition-tuples that could fire:

$$\text{transition}_s^t \stackrel{\text{def}}{=} \{(x, y) \mid (x, y) \in \text{behavior}_s^t \wedge x \in \text{snapshot}_s^t\}$$

Before we can come up with the final formulae for the calculation of the system snapshot snapshot_s^{t+1} we need

a new operator on relations. This operator takes a relation X and replaces all tuples of X with tuples of Y if the first element of both tuples is equal²:

$$X \downarrow_Y =_{\text{def}} \{a \mid a \in Y \vee (a \in X \wedge \pi_1(a) \notin \pi_1(Y))\}$$

At last, we are now able to provide the complete formulae to determine the system snapshot $snapshot_s^{t+1}$:

$$\begin{aligned} snapshot_s^{t+1} = & (alive_s^{t+1}, assigned_s^{t+1}, connected_s^{t+1}, parent_s^{t+1}, \\ & allocation_s^{t+1}, valuation_s^{t+1}, evaluation_s^{t+1}). \\ & alive_s^{t+1} = alive_s^t \upharpoonright_{\pi_8(transition_s^t)} \wedge \\ & assigned_s^{t+1} = assigned_s^t \upharpoonright_{\pi_9(transition_s^t)} \wedge \\ & connected_s^{t+1} = connected_s^t \upharpoonright_{\pi_{10}(transition_s^t)} \wedge \\ & parent_s^{t+1} = parent_s^t \upharpoonright_{\pi_{11}(transition_s^t)} \wedge \\ & allocation_s^{t+1} = allocation_s^t \upharpoonright_{\pi_{12}(transition_s^t)} \wedge \\ & valuation_s^{t+1} = valuation_s^t \upharpoonright_{\pi_{13}(transition_s^t)} \wedge \\ & evaluation_s^{t+1} = evaluation_s^t \upharpoonright_{\pi_{14}(transition_s^t)} \end{aligned}$$

Intuitively spoken, the next system snapshot ($snapshot_s^{t+1}$) is a tuple. Each element of this tuple, for instance $alive_s^{t+1}$, is a function, that is determined simply by merging the former function ($alive_s^t$) and the “delta-function” $\pi_8(transition_s^t)$. This “delta-function” includes all “wishes” of all transition-relations that fire.

5 TYPE SYSTEM

The basic concepts and their relations as covered in the previous sections provide mathematical definitions for the constituents of a component-based system at runtime. However, in order to present an adequate model useful for practical development, we introduce the concept of a type. Let $COMPONENT_TYPE \cup INTERFACE_TYPE \cup CONNECTION_TYPE \cup VARIABLES_TYPE \subseteq TYPE$ be the infinite set of all types. A type models all common properties of a set of instance in an abstract way. $TYPE_OF$ assigns to each instance (component, interface, connection, and variables) its corresponding type:

$$TYPE_OF =_{\text{def}} ID \rightarrow TYPE$$

Let $PREDICATE$ be the infinite set of all predicates that might ever exist. Predicates (boolean expressions) on a type are functions from instances of this type to $BOOLEAN$. For instance, for the component $c \in COMPONENT$, $transition \in behavior_c$ is a predicate on the type of c . This is one of the simplest predicate we can imagine. It provides a direct mapping from the type-level to the instance-level. The predicate is true, if the arbitrary $transition$ is part of component the behavior. Now, we can define functions that provide an abstract description for all existing types³:

$$DESCRIPTION =_{\text{def}} TYPE \rightarrow \mathcal{P}(PREDICATE)$$

²The “standard” notation $\pi_{i_1, i_2, \dots, i_m}(R)$ denotes the set of m -tuples as a result of the projection of the relation R of arity r onto the components i_1, i_2, \dots, i_m ($1 \leq i_j \leq r, i_j \neq i_k$ if $j \neq k$).

³ $\mathcal{P}(A)$ denotes the powerset of the set A .

6 SOFTWARE EVOLUTION

Usually, during the development of a system, various development documents are created. These development documents are concrete descriptions, in contrast to the abstract descriptions linked to types as discussed in the last sections. Such a development document is separate unit that describes a certain aspect of, or “view” on the system under development. In componentware we typically have the following kinds of documents:

- **Structural Documents** describe the internal structure of a system or component. The structure of a component consists of its subcomponents and the connections between the subcomponents and with the supercomponent, e.g. aggregation or inheritance in UML Class Diagrams [23] or architecture description languages [5].
- **Interface Documents** describe the interfaces of components. Currently most interface descriptions (e.g. CORBA IDL [24]) only allow one to specify the syntax of component interfaces. Enhanced descriptions that also capture behavioral aspects use pre- and post-conditions, e.g. Eiffel [20] or the Java Modeling Language [18].
- **Protocol Documents** describe the interaction between a set of components. Typical interactions are messages exchange, call hierarchies, or dynamic changes in the connection structure. Examples of protocol descriptions are: Sequence Diagrams in UML [23], Extended Event Traces [6], or Interaction Interfaces [7].
- **Implementation Documents** describe the implementation of a component. Program code is the most popular kind of those descriptions, but we can also use automata, like in [28, 12] or some kind of greybox specifications [8]. Especially in componentware the implementation of a component can be (recursively) described by a set of structural, interface, protocol, and implementation documents.

During development we describe a system—or more exactly the types of the system—by sets of those documents. Let DOC be the infinite set of all possible documents. Each type of a component-based system is described by a set of those development documents:

$$DESCRIBED_BY =_{\text{def}} TYPE \rightarrow \mathcal{P}(DOC)$$

The semantics of a given set of development documents is simply a mapping from this set of documents to a set of predicates. Thus, we can define a semantic function which assigns to a given set of documents a set of properties characterizing the system:

$$SEM =_{\text{def}} \mathcal{P}(DOC) \rightarrow \mathcal{P}(PREDICATE)$$

The semantic mapping from the concrete descriptions of a system ($doc_s \subseteq DOC$) into a set of predicates is correct, if these predicates are equal with the predicates of the abstract description of each $t \in TYPE$. More formally, the semantic mapping is correct if the following

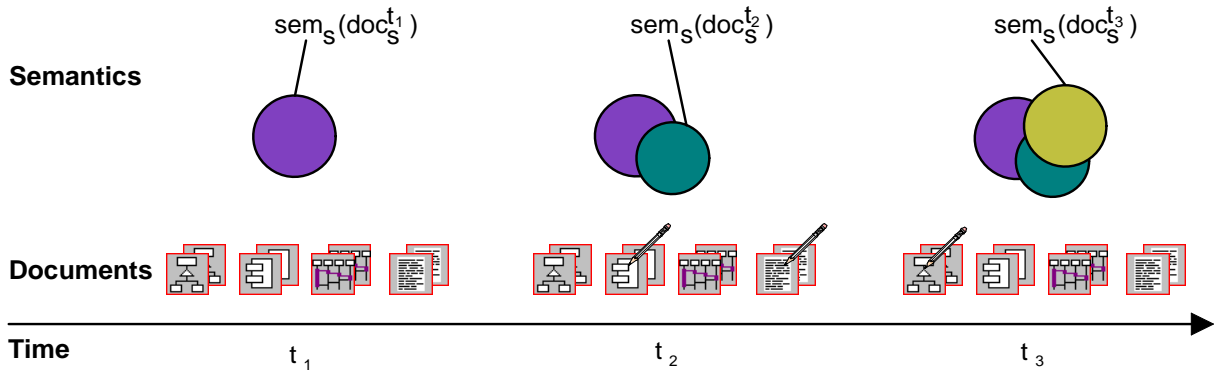


Figure 2: Software Evolution during System Development

condition holds:

$$sem_s(described_by_s(t)) = description_s(t)$$

As already discussed in Section 1, the ability for software to evolve in a controlled manner is one of the most critical areas of software engineering. Developers need support for an evolutionary approach. Based on the semantic function SEM, we are able to formulate the concept of an evolution step. Figure 2 shows three typical evolution steps during system development. An evolution step in our sense causes changes in the set of development documents within a certain time step as given by the functions of type EVOLVE:

$$EVOLVE =_{\text{def}} \mathcal{P}(\text{DOC}) \rightarrow \mathcal{P}(\text{DOC})$$

We call an evolution step of a set of documents $doc_s \subseteq \text{DOC}$

- **refinement**, if the condition $sem_s(evolve_s(doc_s)) \supseteq sem_s(doc_s)$ holds,
- **abstraction**, if the condition $sem_s(doc_s) \supseteq sem_s(evolve_s(doc_s))$ holds,
- **strict evolution**, if the condition $sem_s(doc_s) \not\supseteq sem_s(evolve_s(doc_s)) \wedge sem_s(evolve_s(doc_s)) \not\supseteq sem_s(doc_s) \wedge sem_s(doc_s) \cap sem_s(evolve_s(doc_s)) \neq \emptyset$ holds, or
- **total change**, if the condition $sem_s(doc_s) \cap sem_s(evolve_s(doc_s)) = \emptyset$ holds.

Obviously, we should pay the most attention to the strict evolution. In the remaining paper we use evolution and strict evolution as synonymous, unless if we explicitly distinguish the various kinds of evolution steps. A more detailed discussion about the differences between evolution and refinement steps can be found in [26].

7 REQUIREMENTS/ASSURANCES CONTRACTS

If a document changes via an evolution step, the consequences for documents that rely on the evolved document are not clear at all. Normally, the developer who causes the evolution step has to check whether the

other documents are still correct or not. As the concrete dependencies between the documents are not explicitly formulated, the developer has usually to go into the details of all concerned documents. For that reason we claim that an evolution-based methodology must be able to model and track the dependencies between the various development documents.

To reach this goal we have to make the dependencies between the development documents more explicit. Currently, in description techniques or programming languages dependencies between different documents can only be modeled in an extremely rudimentary fashion. For instance, in UML [23] designers can only specify the relation *uses* between documents or in Java [10] programmers have to use the *import* statement to specify that one document relies on another.

Surely, more sophisticated specification techniques exist, e.g. *Evolving Interoperation Graphs* [25], *Reuse Contracts* [29, 19], or *Interaction Contracts* [13]. *Evolving Interoperation Graphs* provide a framework for change propagation if a single class changes. These graphs take only into account the syntactical interface of classes and the static structure (class hierarchy) of the system, but not the behavioral dependencies.

Reuse Contracts address the problem of changing implementations of a stable abstract specification. There, evolution conflicts in the scope of inheritance are discussed, but not conflicts in component collaborations. This might be helpful to predict the consequences of evolving a single component, but the effects for other components or the entire system are not clear at all.

Finally, *Interaction Contracts* are used to specify the collaborations between objects. Although the basic idea of interaction contracts—to specify the behavioral dependencies between objects—seems to be a quite good suggestion, this approach takes neither evolution nor componentware sufficiently into account. Interaction contracts strongly couple the behavior specification of the component seen as an island and the behavioral dependencies to other components. Hence, the impacts of an evolutionary step can not be determined.

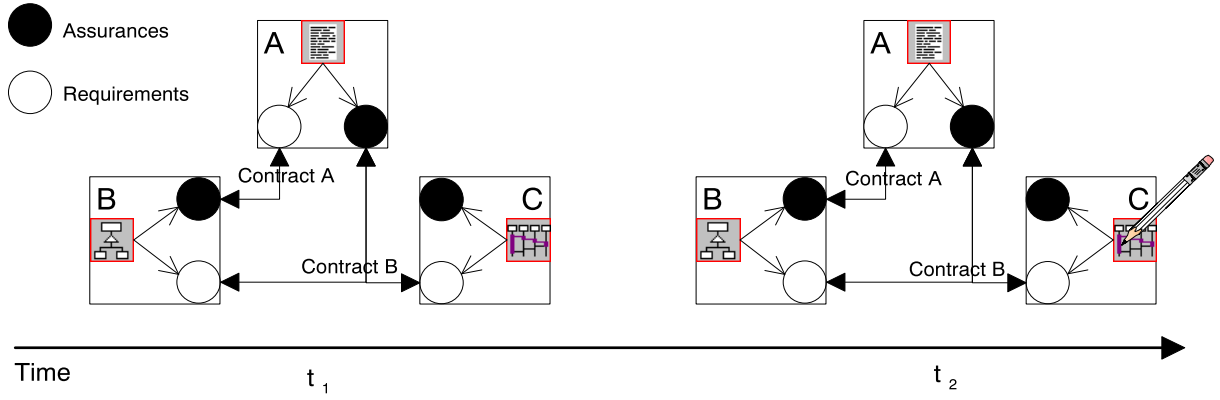


Figure 3: Requirements/Assurances Contracts between Development Documents of Component Types

To avoid these drawbacks and support an evolution-based development process at the best, we propose to decouple the *component island specification* from the *behavioral dependencies specification*. The following two types of functions allow us to determine the behavioral specification of a single component seen as an island:

REQUIRES =_{def}
 $\text{COMPONENT_TYPE} \rightarrow (\mathcal{P}(\text{DOC}) \rightarrow \mathcal{P}(\text{PREDICATE}))$
 ASSURES =_{def}
 $\text{COMPONENT_TYPE} \rightarrow (\mathcal{P}(\text{DOC}) \rightarrow \mathcal{P}(\text{PREDICATE}))$

Intuitively, a function $requires_s \in \text{REQUIRES}$ calculates for a given set of documents $doc_s \in \mathcal{P}(\text{DOC})$ the set of predicates the component type $ct \in \text{COMPONENT_TYPE}$ expects from its environment. The function $assures_s \in \text{ASSURES}$ calculates the set of predicates the component type provides to its environment.

We need specialized description techniques to model the required and assured properties of a certain component explicitly within this development document. Such description techniques must be strongly structured. They should have at least two additional parts capturing the set of required and assured properties (cf. Figure 3):

- **Requirements:** In the requirements part the designer has to specify the properties the component needs from its environment.
- **Assurances:** In the assurances part the designer describes the properties the component assures to its environment, assuming its own requirements are fulfilled.

Once these additional aspects are specified (formally given by the functions $requires_s$ and $assures_s$), the designer can explicitly state the behavioral dependencies between the components by specifying for each component the assurances that guarantee the requirements. We call such explicit formulated dependencies *Requirements/Assurances Contracts* (r/a-contracts). Figure 3 illustrates the usage of those contracts. The three development documents include the additional requirements

(white bubble) and assurances (black bubble) parts. Developers can explicitly model the dependencies between the components by r/a-contracts shown as double arrowed lines. Formally a r/a-contract is a mapping between the required properties of a component and the assured properties of other components:

CONTRACT =_{def} COMPONENT_TYPE \times
 PREDICATE \times COMPONENT_TYPE \times PREDICATE
 FULFILLED =_{def} COMPONENT_TYPE \rightarrow
 $(\mathcal{P}(\text{PREDICATE}) \rightarrow \text{BOOLEAN})$

For a given contract $contract_s \in \text{CONTRACT}$ the predicate $fulfilled_s \in \text{FULFILLED}$ holds, if all required properties of a component are assured by properties of other components:

$$\begin{aligned} fulfilled_s(ct)(requires_s(ct)(described_by_s(ct))) &\iff \\ \{p \mid (ct, r, x, p) \in contract_s \wedge & \\ r \in requires_s(ct)(described_by_s(ct))\} &\subseteq \\ \{q \mid q \in assures_s(x)(described_by_s(x))\} & \end{aligned}$$

In the case of software evolution the designer or a tool has to re-check whether requirements of components, that rely on the assurances of the evolved component, are still guaranteed. Formally the tool has to re-check whether the predicate $fulfilled_s(ct)(requires_s(ct)(evolve_s(described_by_s(ct))))$ still holds.

For instance, in Figure 3 component C has changed over time. The designer has to validate whether Contract B still holds. More exactly, he or she has to check whether the requirements of component C are still satisfied by the assurances of component A or not.

The advantages of r/a-contracts come only fully to validity if we have adequate description techniques to specify the requirements and assurances of components within development documents. In the next section we provide a small sample including some simple description techniques to prove the usefulness of r/a-contracts.

8 EVOLVING A HELP WINDOW

To illustrate the practical relevance of the proposed r/a-contracts we want to discuss a short example. Consider a windows help screen as shown in Figure 4. It contains two components: a *text box* and a *list box* control element. The content of the text box restricts the presented help topics in the list box. Whenever the user changes the content of the text box—simply by adding a single character—the new selection of help topics is immediately presented in the list box.

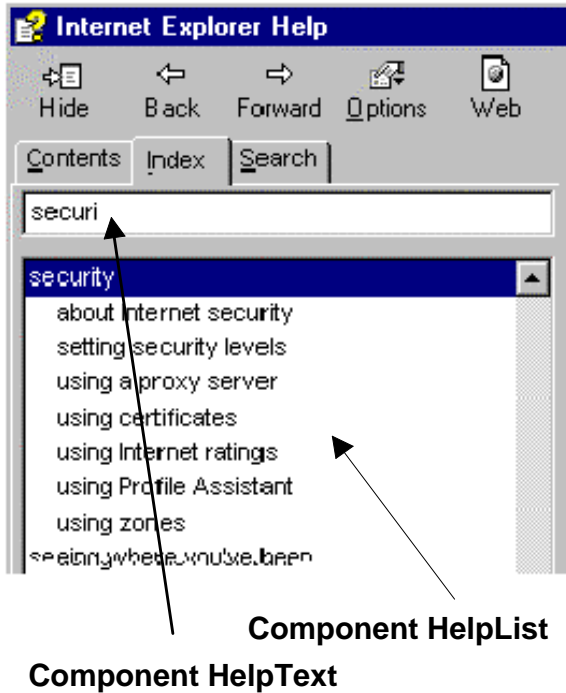


Figure 4: A Short Sample: Windows Help Screen

A simple implementation of such a help screen may contain the two components `HelpText` and `HelpList`. The collaboration between these two components usually follows the *Observer Pattern* [11]. In the case of an “observable” component (`HelpText`) changing parts of its state, all “observing” components (`HelpList`) are notified.

Components in a system often evolve. To make the windows help screen more evolution resistant, one should specify the help screen in a modular fashion. Thus, we use two different kinds of descriptions as proposed in Section 7:

- Descriptions of the behavior of a single component seen as an island start with **COMPONENT** and
- descriptions of the behavioral dependencies between components start with **RA-CONTRACT**.

In the example description technique we use, keywords are written with capital letters. Each component island specification consists of two parts in the specification: The first part is the **REQUIRES** part containing all in-

terfaces the component needs. For each interface the required predicates (syntax and behavior) are explicitly specified. The second part is the **ASSURES** part capturing all interfaces the component provides to its environment. For each interface the assured predicates (again syntax and behavior) are explicitly described.

The notation and semantic within these parts is equal to the one used for the interaction contracts [13]. The language only supports the actions of sending a message M to a component C , denoted by $C \rightarrow M$, and change of a value v , denoted by Δv . The ordering of actions can be explicitly given by the operator “;”, an **IF-THEN-ELSE** construct, or be left unspecified by the operator \parallel . The language also provides the construct $\langle o \ v : c : e \rangle$ for the repetition of an expression e separated by the operator o for all variables v which satisfy c .

Now, we can start out with a textual specification of the requirements and assurances of the two components `HelpText` and `HelpList`—the components island specification:

```
COMPONENT HelpText
  REQUIRES INTERFACE Observer
  WITH METHODS
    update() : void
  ASSURES INTERFACE TextBox
  WITH LOCALS
    observers : Set(Observer)
    text : String
  WITH METHODS
    getText() : String ⇒ return text
    addText(t : String) : void ⇒
      Δ text {text = text + t} ; { || obs :
        obs ∈ observers : obs → update() }
```

The component `HelpText` requires an interface supporting the method `update():void`. Note that, in the context of this specification the required interface is named `Observer`. This represents neither a global name nor a type of the required interface. Later, we can explicitly model the mapping between the various required and assured interface and method names via the proposed r/a-contracts. Additionally, the component `HelpText` assures an interface `TextBox` with the two methods `getText():String` and `addText(t:String):void`. When `addText(t)` is called the method `update()` is invoked for all observers.

Correspondingly, the component `HelpList` requires an interface named `Observable` that includes the method `getText():String`. Moreover, whenever the return value of `getText()` changes, the `update()` method of the component `HelpList` has to be called via the interface `ListBox`. This is the basic behavior requirement the component `HelpList` needs to be assured by its environment.

```
COMPONENT HelpList
  REQUIRES INTERFACE Observable
  WITH METHODS
```

```

    getText() : String
WITH INVARIANTS
    Observable → getText() ≠ Δ( Observable
    → getText() ) ⇒ ListBox → update()
ASSURES INTERFACE ListBox
WITH LOCALS
    observable : Observable
WITH METHODS
    update() : void ⇒ ListBox →
    redisplayTopics(observable → getText())

```

Now, we can specify two r/a-contracts: One to satisfy the requirements of the component `HelpList` and the other for the requirements of component `HelpText`. Such a contract contains two sections: The first section, the `INSTANTIATION`, declares the participants of the contract and their initial configuration. For instance, in the contract `HelpListContract` are two participants `hl:HelpList` and `ht:HelpText` instantiated and the initial connection between both is established. Note, the variables declared in the instantiation section are global identifiers, as one must be able to refer them in the current contract as well as in other contracts..

The second section, the `PREDICATE MAPPING`, maps the required interfaces to assured interfaces of the participants. Additionally, it contains the most important part of the contract: the “proof”. There, the designer has to validate the correctness of the contract, means he or she has to proof whether the syntax and behavior of the requirements/assurance pair fits together. The contract `HelpListContract` includes a proof. It simply starts with conjunction of all assured predicates of the interface `ht.TextBox` and has to end with all required predicates of the interface `hl.Observable`:

```

RA-CONTRACT HelpListContract
INSTANTIATION
    hl : HelpList
    ht : HelpText
    ht.TextBox.observers.add(hl)
    hl.ListBox.observable = ht
PREDICATE MAPPING: REQUIRED hl.Observable
    ASSURED BY ht.TextBox
    ht → getText() ⇒ return text ∧ ht →
    addText(t) ⇒ Δ text {text = text + t} ;
    ⟨ || obs : obs ∈ ht.TextBox.observers :
    obs → update() ⟩ ⇒
    ht → getText() ≠ Δ( ht → getText() ) ⇒
    ⟨ || obs : obs ∈ ht.TextBox.observers :
    obs → update() ⟩ ⇒
    ht → getText() ≠ Δ( ht → getText() ) ⇒
    hl → update()

```

```

RA-CONTRACT HelpTextContract
INSTANTIATION

PREDICATE MAPPING: REQUIRED ht.Observer
    ASSURED BY hl.ListBox
    proof is omitted

```

Once the windows help screen is completely specified and implemented, it usually takes a couple of months until one of the components appears in a new, improved version. In our example, the new version of the component `HelpText` has been evolved. The new version assures an additional method `addChar(c:Char):void`. For performance reasons, this method does not guarantee that the observers are notified if the method is invoked:

```

COMPONENT HelpText
REQUIRES INTERFACE Observer
WITH METHODS
    update() : void
ASSURES INTERFACE TextBox
WITH LOCALS
    observers : Set(Observer)
    text : String
WITH METHODS
    getText() : String ⇒ return text
    addText(t : String) : void ⇒
    Δ text {text = text + t} ; ⟨ || obs :
    obs ∈ observers : obs → update() ⟩
    addChar(c : Char) : void ⇒
    Δ text {text = text + c}

```

The assurances part in the specification of the component `HelpText` has changed. Therefore, the designer or a tool should search for all r/a-contracts where `HelpText` is used to fulfill the requirements of other components. Once, all of these contracts are identified, the corresponding proofs have to be re-done. In our example the contract `HelpListContract` is concerned. The designer has to re-check whether the goal `ht → getText() ≠ Δ(ht → getText()) ⇒ hl → update()` still can be reached. But the premises have been changed.

Obviously the goal cannot be derived, as a call of `addChar(c)` changes the return value of the method `getText()` but does not result in an `update()` for the `HelpList`. Thus, the requirement of the component `HelpList`—whenever the text in `HelpText` changes `update()` is called—are no longer satisfied by the new component `HelpText`. The current design of the system may not longer meet the expectations or the requirements. Now, the designer can decide to keep the former component in use or to realize a workaround in the `HelpList` component. However, this is outside the scope of the discussed concepts.

9 CONCLUSION AND FUTURE WORK

The ability for software to evolve in a controlled manner is one of the most critical areas of software engineering. Therefore, a overall evolution-based development methodology for componentware is needed. In this paper we have outlined a well-founded common system model for componentware that copes with the most difficult behavioral aspects in object-orientation or componentware: dynamical changing structures, a shared global state, and finally mandatory call-backs. The model presented includes the concepts of a type and abstract as well as concrete descriptions for types. During

system development a set of those descriptions are created. Software evolution means that these descriptions are changed over time. Thus, we need techniques to determine the impacts of the respective evolution steps. With the presented requirements/assurances-contracts developers can explicitly model the dependencies between the different components. Whenever a component or the entire system changes the contracts show the consequences for other components. Contracts help the developer to manage the evolution of the complete system.

A number of additional issues remain items of future work: We are currently working on a first prototype runtime environment for the presented system model. We still have to elaborate on the underlying type system. Additionally, we have to provide more sophisticated graphical description techniques based on UML and OCL (structural documents, interface documents, protocol documents, and implementation documents). A complete development example will show these description techniques in practice. For each of those description techniques a clear semantical mapping into the system model has to be defined. Additionally, syntax compatible checkers, theorem prover, and model checker could be included to run the correctness proof for evolution steps semi-automatically or even full-automatically. Finally, we have to develop tool support and provide a set of evolution-resistant architectures based on technical componentware infrastructures like CORBA, DCOM, or Java Enterprise Beans.

ACKNOWLEDGEMENTS

I am grateful to Klaus Bergner, Manfred Broy, Ingrid Krüger, Jan Philipps, Bernhard Rumpe, Bernhard Schätz, Marc Sihling, Oskar Slotosch, Katharina Spies, and Alexander Vilbig for interesting discussions and comments on earlier versions of this paper.

REFERENCES

- [1] K. Bergner, A. Rausch, M. Sihling, A. Vilbig, and M. Broy. A formal model for componentware. In *Formale Beschreibungstechniken für verteilte Systeme FBT'99*. Herbert Utz Verlag, 1999.
- [2] D. Box. *Essential COM*. Object Technology Series. Addison-Wesley, 1998.
- [3] B. Selic, G. Gullekson and P. T. Ward. *Real-Time Object-Oriented Modeling*. Wiley & Sons, 1994.
- [4] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. Gritzner, and R. Weber. The design of distributed systems - an introduction to FOCUS. Technische Universität München, January 1992.
- [5] M. Broy, E. Denert, K. Renzel, and M. Schmidt (eds.). *Software architectures and design patterns in business applications*. Technische Universität München, 1997.
- [6] M. Broy, C. Hofmann, I. Krüger, and M. Schmidt. Using extended event traces to describe communication in software architectures. In *Proceedings of the APSEC '97, Hong Kong*. IEEE Computer Society, 1997.
- [7] M. Broy and I. Krger. Interaction Interfaces - Towards a scientific foundation of a methodological usage of Message Sequence Charts. In *Proceedings of the ICFEM 98*. IEEE Press, 1998.
- [8] M. Büchi and W. Weck. A plea for grey-box components. Technical Report 122, Turku Center for Computer Science, September 1997.
- [9] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture – A System of Patterns*. Wiley & Sons, 1996.
- [10] D. Flanagan. *Java in a Nutshell*. O'Reilly & Associates, Inc., 2nd edition, 1996.
- [11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [12] D. Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–531, May 1988.
- [13] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. *ECOOP/OOPSLA '90 Proceedings*, pages 169–180, Oct. 1990.
- [14] Ivar Jacobson and Grady Booch and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
- [15] I. Jacobson. *Object-Oriented Software Engineering — A Use Case Driven Approach*. Addison-Wesley, 1992.
- [16] JavaSoft. Enterprise JavaBeans website, <<http://java.sun.com/~products/~ejb/>>, 1999.
- [17] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3), 1994.
- [18] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Iowa State University, 1999.
- [19] C. Lucas, P. Steyaert, and K. Mens. Managing software evolution through reuse contracts. Vrije Universiteit Brussel Faculteit Wetenschappen, BELGIUM, 1997.
- [20] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.
- [21] G. Neumann. 500 Europa: Der Club der Innovatoren. *Information Week*, pages 10–12, Jan. 1999.
- [22] OMG. *The Common Object Request Broker: Architecture and Specification*. Object Management Group, February 1998.
- [23] OMG. *OMG Unified Modeling Language Specification*. Version 1.3, Object Management Group, 1999.
- [24] R. Orfali and D. Harkey. *Client/Server Programming with Java and CORBA*. John Wiley & Sons, 1997.
- [25] V. Rajlich. Modeling Software Evolution by Evolving Interoperation Graphs. In *Software Change and Evolution 1999 Workshop Proceedings*, 1999.
- [26] A. Rausch. Executive Summary: Software Evolution in Componentware – A Practical Approach. In *Software Change and Evolution 1999 Workshop Proceedings*, 1999.
- [27] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.
- [28] B. Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. PhD thesis, Technische Universität München, 1996.
- [29] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse Contracts: Managing the Evolution of Reusable Assets. In *OOPSLA 1996 Conference Proceedings*, ACM Sigplan Notices, pages 268–285. ACM Press, 1996.