11 Modelling the CoCoME with DisCComp

André Appel, Sebastian Herold, Holger Klus, and Andreas Rausch

Clausthal University of Technology
Department of Informatics - Software Systems Engineering
Julius-Albert-Str. 4
38678 Clausthal-Zellerfeld, Germany

11.1 Introduction

Most large-scaled software systems are logically structured in subsystems resp. components to cope with complexity. These components are deployed and executed within an distributed system infrastructure. Consequently, for many reasons, like for instance multi-user support or performance issues, the components are to some extent concurrently executed within an distributed environement. Note, this also holds for the *Co*mmon *Co*mponent *M*odelling *E*xample (CoCoME).

11.1.1 Goals and Scope of the Component Model

The first main goal of DisCComp is to provide a *sound formal semantic component model* that is powerful enough to handle distributed concurrent components but also realistic enough to provide a foundation for component technologies actually in use, like for instance CORBA, J2EE, and .NET [1,2,3]. Thereby we claim to close the gap between formal component models and existing programming models (see Section 11.2.1). Hence the semantic component model of DisCComp contains all concepts well known from component programming models like for instance, dynamically changing structures, a shared global state and asynchronous message communication, as well as synchronous and concurrent message calls.

The second main goal of DisCComp is to provide proper *UML-based description techniques* to describe the structural and behavioural aspects of component-based systems (see Section 11.2.2). These description techniques have a clear semantics as they are mapped on the formal semantic component model of DisCComp. The formal component model of DisCComp contains an operational semantics for distributed concurrent component-based systems the UML-based descriptons of DisCComp can be directly executed resp. interpreted.

This leeds us to the third main goal of DisCComp: With DisCComp we claim to support system and component architects and designers in modelling component-based systems, simulate and execute these models, test and validate the functional correctness of those models, and finally generate the code for the finaly system out of the models. Therefore DisCComp provides a set of tools, like for instance plug-ins for modelling tools, simulation environments for DisCComp specifications, and code generators.

A. Rausch et al. (Eds.): Common Component Modeling Example, LNCS 5153, pp. 267–296, 2008.

11.1.2 Modeled Cutout of CoCoME

In the Secion 11.3 we present a cutout of the CoCoME modelled using our proposed description technique. In order to focus on specific aspects of our modelling approach and for the sake of brevity, we will illustrate the parts of the system which are relevant for the use case *Change Price* only.

11.1.3 Benefit of the Modeling

The DisCComp approach provides several benefits: As the DisCComp component model is close to existing programming models no paradigm gap exists between the DisCComp and the predominant programming approaches. Hence programmers, designers, and architects use the same paradigm and share a common understanding of the models. Thus software engineers can easily learn and apply the DisCComp approach as it is close to models they are familiar with.

As the description techniques of DisCComp have a clear semantics the resulting models and specifications are not ambiguous but precise. Thus, they can be directly simulated and executed. Hence the feedback loop and the iteration cycles are extremely shortened. Models and specifications can directly be tested and verified in advance without the need of coding them in the target component technique, which is more laborious and time-consuming due to the complex technologies programmers have to cope with.

11.1.4 Effort and Lessons Learned

To model our cutout of CoCoME with DisCComp an effort of 4 person month were used. During the contest we have learned the following lessons: The formal model of DisCComp is valid and fits well for real software systems like CoCoME. The provided specification techniques can be used to describe real software systems like CoCoME. However, the effort to elaborate these specifications is too high. The reason for this is, that currently our description techniques are too low-level. They are closer to the coding level than to a more abstract specification level. Hence we have to improve our approach and provide more abstract specification techniques.

11.2 Component Model

This section elaborates the basic concepts of the proposed formal model for distributed concurrent component-based software systems. Such a model incorporates two levels: The *instance level* and the *description level* [4]. The *description level* - described in Section 11.2.2 - contains a normalized abstract description of a subset of common instance level elements with similar properties. The *instance level* - described in the Section 11.2.1 - is the reliable semantic foundation of the description level. It provides an operational semantics for distributed concurrent components - it is an abstraction of existing programming models like CORBA, J2EE, and .NET [1,2,3]. Thereby, it defines the universe of all possible software systems that may be specified at the description level and implemented using the mentioned programming models.

11.2.1 The DisCComp System Model

The instance level of our proposed formal model for distributed concurrent components must be powerful enough to handle the most difficult behavioural aspects:

- dynamically changing structures,
- shared global state,
- asynchronous message communication, and
- concurrent method calls.

Figure 1 summarizes these behavioural aspects of the formal model for distributed concurrent components at the instance level on an abstract level. Thereby, software systems consist of a set of disjoint instances during run-time: system, component, interface, attribute, connection, message, thread, and value. In order to uniquely address these basic elements of the instance level we introduce the infinite set $\mathsf{INSTANCE}$ of all instances:
$\mathsf{INSTANCE} =_{def} \{ \mathsf{SYSTEM} \cup \mathsf{COMPONENT} \cup \mathsf{INTERFACE} \cup \mathsf{ATTRIBUTE} \cup \mathsf{CONNECTION} \cup \mathsf{MESSAGE} \cup \mathsf{THREAD} \cup \mathsf{VALUE} \}$

The presented four behavioural aspects of distributed concurrent component-based systems are described in the following.

Structural Behaviour. A system may change its structure dynamically. Some instances may be created or deleted (ALIVE). New attributes resp. interfaces may be assigned to interfaces resp. components ($\mathsf{ALLOCATION}$ resp. $\mathsf{ASSIGNMENT}$). Interfaces and components may have a directed connection to interfaces ($\mathsf{CONNECTS}$). Note, the target of a connection can only be an interface.:

$\mathsf{ALIVE} =_{def} \mathsf{INSTANCE} \rightarrow \mathsf{BOOLEAN}$
$\mathsf{ASSIGNMENT} =_{def} \mathsf{INTERFACE} \rightarrow \mathsf{COMPONENT}$
$\mathsf{ALLOCATION} =_{def} \mathsf{ATTRIBUTE} \rightarrow \mathsf{INTERFACE}$
$\mathsf{CONNECTS} =_{def} \mathsf{CONNECTION} \rightarrow \{\{ \text{from}, \text{to} \} \mid \text{from} \in \mathsf{COMPONENT} \cup \mathsf{INTERFACE}, \text{to} \in \mathsf{INTERFACE}\}$

Valuation Behaviour. A system's state space is not only determined by its current structure but also by the values of the component's attributes. Mappings of attributes or parameters to values of appropriate type are covered by the following definition:

$\mathsf{VALUATION} =_{def} \mathsf{ATTRIBUTE} \rightarrow \mathsf{VALUE}$

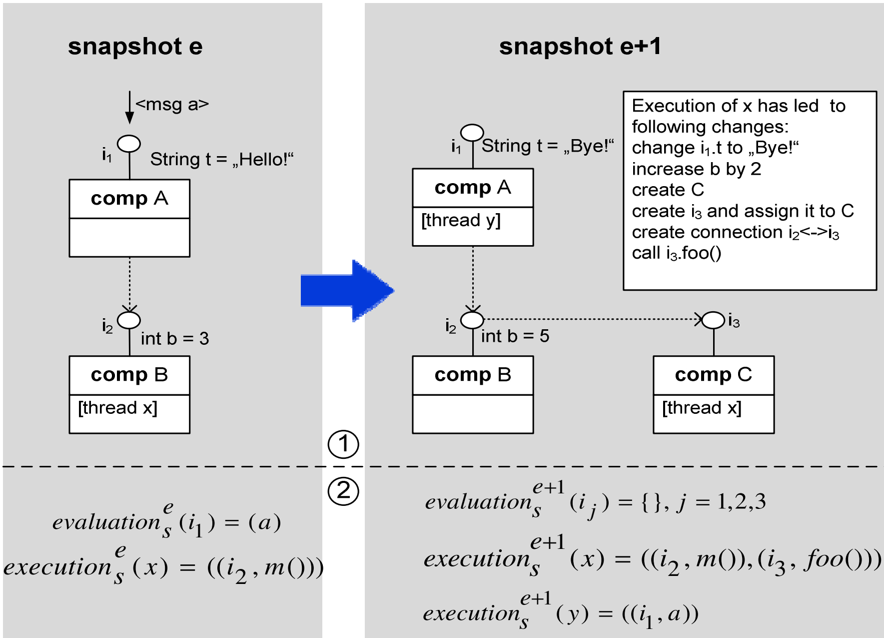

Fig. 1. Instance level of concurrent components

Communication Behaviour. Sequences of asynchronous messages represent the fundamental units of asynchronous communication. Therefore we distinguish the set $\mathsf{MESSAGE}$ in two non disjoint subsets: $\mathsf{MESSAGE} =_{def} \mathsf{ASYNC\_MESSAGE} \cup \mathsf{CALL\_MESSAGE}$ In order to model message-based asynchronous communication, we denote the set of arbitrary finite asynchronous message sequences with $\mathsf{ASYNC\_MESSAGE}^*$. Within each observation point components process message sequences arriving at their interfaces and send message sequences to other interfaces:

$$\mathsf{EVALUATION} =_{def} \mathsf{INTERFACE} \rightarrow \mathsf{ASYNC\_MESSAGE}^*$$

Execution Behaviour. Besides asynchronous communication, synchronous method calls ($\mathsf{CALL\_MESSAGE}$) performed by concurrent executed threads is the predominant execution mechanism in contemporary software systems. Each method is called at a certain interface ($\mathsf{INTERFACE}$). Hence, to model a thread's call stack, we denote the set of arbitrary finite method call sequences with $(\mathsf{INTERFACE} \times \mathsf{CALL\_MESSAGE})^*$. Each thread has its own method call history - its call stack ($\mathsf{EXECUTION}$). Note that threads may change the hosting component in case of a method call at an interface belonging to another component:

$$\mathsf{EXECUTION} =_{def} \mathsf{THREAD} \rightarrow (\mathsf{INTERFACE} \times \mathsf{CALL\_MESSAGE})^*$$

System Snapshot. Based on the former definitions, we are now able to characterize a snapshot of a software system. Such a snapshot captures the current structure, variable valuation, actual received messages, and current method calls. Let $\mathsf{SNAPSHOT}$ denote the type of all possible system snapshots:

$$\mathsf{SNAPSHOT} =_{def} \mathsf{ALIVE} \times \mathsf{ASSIGNMENT} \times \mathsf{ALLOCATION} \times \mathsf{CONNECTS} \times \mathsf{VALUATION} \times \mathsf{EVALUATION} \times \mathsf{EXECUTION}$$

System Behaviour. In contrast to related approaches like [5], we do not focus on timed streams but on execution streams. We regard observation points as an infinite chain of execution intervals of various lengths. Whenever a thread's call stack changes - in case of a new method call or a method return - a new observation point is reached. We use the set of natural numbers N as an abstract axis of those observation points, and denote it by E for clarity.

Furthermore, we assume an observation synchronous model because of the resulting simplicity and generality. This means that there is a global order of all observation points and thereby of all method calls and returns. Note that this is not a critical constraint. Existing distributed component environments like CORBA, J2EE, and .NET control and manage all method calls and returns. Such a component environment may transparently force a global order of all method calls and returns.

We use execution streams, i.e. finite or infinite sequences of elements from a given domain, to represent histories of conceptual entities that change over observation points. An execution stream - more precisely, a stream with discrete execution interval - of elements from the set X is an element of the type

$$X^E =_{def} N^+ \to X, \text{ where } N^+ =_{def} N \setminus \{0\}$$

Thus, an execution stream maps each observation point to an element of X. The notation x^e is used to denote the element of the valuation $x \in X^E$ at the observation point $e \in E$ with $x^e = x(e)$.

Execution streams may be used to model the behaviour of software systems. Accordingly, $\mathsf{SNAPSHOT}^E$ is the type of all system snapshot histories or simply the type of the behaviour relation of all possible software systems:

$$\mathsf{SNAPSHOT}^E =_{def} \mathsf{ALIVE}^E \times \mathsf{ASSIGNMENT}^E \times \mathsf{ALLOCATION}^E \times \mathsf{CONNECTS}^E \times \mathsf{VALUATION}^E \times \mathsf{EVALUATION}^E \times \mathsf{EXECUTION}^E$$

Let $\mathsf{Snapshot}_s^E \subseteq \mathsf{SNAPSHOT}^E$ be the behaviour relation of an arbitrary system $s \in \mathsf{SYSTEM}$[1]. A given snapshot history $\mathsf{snapshot}_s \in \mathsf{Snapshot}_s^E$ is an execution stream of tuples that capture the changing snapshots $\mathsf{snapshot}_s^e$ over observation points $e \in$ E.

Obviously, a couple of consistency conditions can be defined on a formal behaviour $\mathsf{Snapshot}_s^E \subseteq \mathsf{SNAPSHOT}^E$. For instance, it may be required that all

[1] In the remainder of this paper we will use this shortcut. Whenever we want to assign a relation X (element x) to a system $s \in \mathsf{SYSTEM}$ we say $X_s(x_s)$.

attributes obtain the same activation state as the interface they belong to: $\forall a \in \mathsf{Attribute}_s, i \in \mathsf{Interface}_s, e \in \mathrm{E}.\mathsf{allocation}_s^e(a) = i \Rightarrow \mathsf{alive}_s^e(a) = \mathsf{alive}_s^e(i)$ Or furthermore, instances that are deleted are not allowed to be reactivated: $\forall i \in \mathsf{Instance}_s, e, n, m \in \mathrm{E}.\ e < n < m \wedge\ \mathsf{alive}_s^e(i) \wedge \neg\mathsf{alive}_s^n(i) \Rightarrow \neg\mathsf{alive}_s^m(i)$

We can imagine plenty 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 [6,7].

Thread Behaviour. A system's observable behaviour is a result of the composition of all thread behaviours. These threads are executed concurrently and are potentially distributed. To compute the system behaviour from the parallel executed threads, the thread's execution results have to be integrated taking possible inconsistencies due to parallelism into account.

To compose the system behaviour out of the results of the parallel executed threads DisCComp provides a simple but powerful abstraction. Therefore we introduce the notion of an *atomic unit of execution* of threads. In DisCComp this atomic unit of execution is given by the execution of method calls and returns. Whenever a thread executes a method call or a method return the current atomic unit of execution is finished and the next one starts. Hence the execution of method calls resp. method returns define the atomic execution results of threads, which have to be integrate by the run-time environment into the system-wide snapshot and thus composing the system's observable behaviour.

To describe these atomic units of execution of each thread we define a relation between a system-wide snapshot and the thread's wished changes on the system-wide snapshot after performing a method call or return. The run-time environment integrates these wished changes into the syste-wide snapshot and thereby it calculates the the system-wide successor snapshot:

$$\mathsf{BEHAVIOUR} =_{def} \mathsf{SNAPSHOT} \rightarrow \mathsf{SNAPSHOT}$$

Let $\mathsf{behaviour}_t \subseteq \mathsf{BEHAVIOUR}$ be the behaviour of a thread $t \in \mathsf{THREAD}_s$ in the system $s \in \mathsf{SYSTEM}$. The informal meaning of the thread behaviour is as follows: A thread executes the program code (and therefore has a program counter, which is given by its call stack $\mathsf{EXECUTION}$). Each transition relation $\mathsf{transition} \in \mathsf{behaviour}_t$ represents an atomic unit of execution. Intuitively it can be be seen as the execution result (second part of the transition relation) of the atomic unit of execution. Whereas the atomic unit of execution has started with the system-wide snapshot given by the first part of the transition relation.

Each thread performs a sequence of those atomic units of execution represented by transition relations. Each atomic unit of execution resp. transition relation $\mathsf{transition} \in \mathsf{behaviour}_t$ can intuitively be seen as the interpretation of an atomic piece of program code by the thread, which has the following schema:

1. The thread evaluates the system-wide snapshot given by the first snapshot of the tuple $\mathsf{transition}$. If the relevant parts of the system-wide snapshot fits to the conditions contained in the atomic piece of program code to execute (e.g. conditions in if statements).

2. The thread requests the corresponding set of changes on the system-wide snapshot described in the atomic piece of program code like for instance changing the value of an attribute. These changes on the system-wide snapshot are described by the second snapshot of the tuple $\mathsf{transition}$.
3. Finally, the thread – following the atomic piece of program code to execute – has to perform a new method call or return. Again this is given by a call-stack change described in the function $\mathsf{execution}_t \subseteq \mathsf{EXECUTION}$, which is part of the second snapshot in the tuple $\mathsf{transition}$.

Note that the behaviour relation of threads neither left-unique nor right-unique. Moreover the relation has not to be total. Hence, thread behaviour is not non-deterministic as it describes a concrete execution trace of a thread. However the thread behaviour is partial as a thread may not terminate. This is not a general restriction of the proposed approach it just reflects reality.

Behaviour Composition. Consequently, we need some specialized run-time system that asks all threads - one by one - if one wants to perform a new method call or return from a method call. Whenever a thread wants to perform a new method call or return, which means that its behaviour relation fires, the run-time system composes a new well-defined system-wide successor snapshot based on the thread's requested changes and the current system-wide snapshot.

Hence, such a run-time system is similar to a virtual machine. It observes and manages the execution of all threads. Again, this is not a critical constraint even in a concurrent and distributed environment. Existing distributed component environments like CORBA, J2EE, and .NET control and manage all executed components within the environment.

To sum up, the main task of such a run-time system is to determine the next system snapshot $\mathsf{snapshot}_s^{e+1}$ from the current snapshot $\mathsf{snapshot}_s^e \in \mathsf{Snapshot}_s^E$. In essence, we can provide formulae to calculate the system behaviour from the initial configuration $\mathsf{snapshot}_s^0$, the behaviour relations $\{\mathsf{behaviour}_{t_1}, ..., \mathsf{behaviour}_{t_n}\}$ of all threads $t_1, ..., t_n \in \mathsf{THREAD}_s$, $n \in N$, and external stimulations via asynchronous messages and synchronous method calls at free interfaces. Note that free interfaces are interfaces that are not connected with other interfaces and thus can be stimulated from the environment.

Before we can come up with the final formulae to specify the run-time system, 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 equal[2]:

$X \lhd Y =_{def} \{a | a \in Y \vee (a \in X \wedge \pi_1(\{a\}) \cap \pi_1(Y) = \emptyset)\}$

We are now able to provide the complete formulae to determine the next system snapshot $\mathsf{snapshot}_s^{e+1}$:

---

[2] Note that the "standard" notation $\pi_{i_1,...,i_n}(R)$ denotes the set of n-tuples with $n \in N \wedge n \leq r$ as a result of the projection on the relation R. Whereas in each tuple in $\pi_{i_1,...,i_n}(R)$ contains the elements at the position $i_1, ..., i_n$ of the corresponding tuple from R with $1 \leq i_k \leq r$, with $k \in \{1, ..., n\} \subseteq N$.

$$\text{next-snapshot: SNAPSHOT} \longrightarrow \text{SNAPSHOT}$$

$$\text{next-snapshot}(\text{snapshot}_s^e) =_{def} \text{snapshot}_s^{e+1} =$$
$$= (\text{alive}_s^{e+1}, \text{assignment}_s^{e+1}, \text{allocation}_s^{e+1}, \text{connects}_s^{e+1}, \text{valuation}_s^{e+1}, \text{evaluation}_s^{e+1}, \text{execution}_s^{e+1}) :$$
$$\text{alive}_s^{e+1} = \text{alive}_s^e \lhd \pi_1(\text{behaviour}_{\textit{next\_thread}}(\text{snapshot}_s^e)) \lhd \pi_1(\text{message\_execution}(\text{snapshot}_s^e)) \wedge$$
$$\text{assignment}_s^{e+1} = \text{assignment}_s^e \lhd \pi_2(\text{behaviour}_{\textit{next\_thread}}(\text{snapshot}_s^e)) \wedge$$
$$\text{allocation}_s^{e+1} = \text{allocation}_s^e \lhd \pi_3(\text{behaviour}_{\textit{next\_thread}}(\text{snapshot}_s^e)) \wedge$$
$$\text{connects}_s^{e+1} = \text{connects}_s^e \lhd \pi_4(\text{behaviour}_{\textit{next\_thread}}(\text{snapshot}_s^e)) \wedge$$
$$\text{valuation}_s^{e+1} = \text{valuation}_s^e \lhd \pi_5(\text{behaviour}_{\textit{next\_thread}}(\text{snapshot}_s^e)) \wedge$$
$$\text{evaluation}_s^{e+1} = \pi_6(\text{behaviour}_{\textit{next\_thread}}(\text{snapshot}_s^e)) \wedge$$
$$\text{execution}_s^{e+1} = \text{execution}_s^e \lhd \pi_7(\text{behaviour}_{\textit{next\_thread}}(\text{snapshot}_s^e)) \lhd \pi_7(\text{message\_execution}(\text{snapshot}_s^e))$$

Intuitively spoken, the next system snapshot snapshot_s^{e+1} is a tuple. Each element of this tuple, for instance $\text{assignment}_s^{e+1}$, is a function that is determined simply by merging the former function assignment_s^e and the 'delta-function' of $\pi_2(\text{behaviour}_{\textit{next\_thread}}(\text{snapshot}_s^e))$. This 'delta-function' includes all 'wishes' of the next relevant thread determined by the function next_thread.

This intuitive understanding does not completely hold for alive_s^{e+1}, $\text{evaluation}_s^{e+1}$ and execution_s^{e+1} . In alive_s^{e+1} and execution_s^{e+1}, not only the wishes of thread next_thread have to be included. These wishes must contain the thread's actual method call or return. Additionally they may contain new parallel threads created by the current thread.

Moreover, alive_s^{e+1} and execution_s^{e+1} also contain the result of the application of the function $\text{message\_execution}(\text{snapshot}_s^e)$. This function includes new threads created to process the asynchronous messages. Thereby, for each asynchronous message - given by evaluation_s^e which is included in snapshot_s^e - a new thread is created in alive_s^{e+1} to execute the corresponding request in execution_s^{e+1}. message_execution is defined as follows:

$$\text{message\_execution: SNAPSHOT} \rightarrow \text{SNAPSHOT}$$
$$\text{message\_execution}(\text{snapshot}_s^e) =_{def} \text{snapshot'} = (\text{alive'}, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \text{execution'}).$$
$$\forall i \in \text{Interface}_s, m \in \text{Message}_s . m \in \text{evaluation}_s^e(i) \Leftrightarrow$$
$$\Leftrightarrow \exists t' \in \text{Thread}_s . \neg \text{alive}_s^e(t') \wedge \text{alive'}(t') \wedge \text{execution'}(t') = \{(i, m)\}$$

Intuitively spoken for each asynchronous message a new thread is activated and the corresponding call stack is initialized. As all asynchronous messages are with each observation point transformed to corresponding concurrently executed threads, the new system snapshot has only to contain the new asynchronous messages, as denoted by $\text{evaluation}_s^{e+1} = \pi_6(\text{behaviour}_t(\text{snapshot}_s^e))$.

Note that thereby, the delivery of asynchronous message takes some time, exactly one observation point. To model network latency or network failure one would have to provide a more sophisticated function message_execution. Thus, not only delay and loss of asynchronous messages could be integrated but also network related failures in executing method calls.

Moreover note, the function next-snapshot currently does not include the case that a thread terminates as the last return statement has been executed by the thread (empty call stack). This trivial case could be straight forward added. As it would enlarge the formulae we have omitted this case in the paper.

To complete the formal model, the function next_thread has to be defined:
$\mathsf{next\_thread} : \rightarrow \mathsf{THREAD}$

This function returns the next thread to be visited by the run-time system. To provide a simple but general model we propose a round-robin model. Therefore, a given strict order of all active system's threads is required. next_thread follows this given order and provides the next relevant thread to be visited and integrated into the system-wide snapshot by the run-time system.

Note that one can integrate additional features into the model providing other implementations of the function next_thread, like for instance non-determinism and priority-based thread scheduling. Non-determinism could be used to model an unsure execution order or to support under-specification.

Whenever concurrent threads or components are executed, inconsistency or deadlocks may occur. A deadlock concerning elements explicitly modeled in the semantics - like for instance two threads each locking an attribute and waiting to get the lock on the other's thread attribute - cannot occur in this model as all threads are visited one by one and each thread has to release all blocked resources after it has been visited. However, deadlocks on a higher level, like for instance one thread waits for a given condition to become true and another thread waits for this thread to make another condition true, can not be detected in advance.

The model does not suppress inconsistent situations but it helps to detect them. In order to ensure that the next system snapshot $\mathsf{snapshot}_s^{e+1}$ is well-defined, a single basic condition must be satisfied: all elements in the wished successor snapshot given by $\mathsf{behaviour}_{next\_thread}(\mathsf{snapshot}_s^e)$ that cause a change in the resulting next system snapshot must not be changed after the thread next_thread has made his last method call or method return.

For instance, assume that a thread performs a method call. The value of an attribute is 5 as the thread has started the method call execution and the thread wants to change the value to 7 as it returns from this method call. At the observation point where the thread returns from the method call the value of the attribute is already 6, as another thread has changed the value in the meantime. Hence, a possible inconsistency caused by concurrent thread execution occurs.

A run-time system implementing the function next_snapshot has to calculate the next system snapshot. Thereby it can observe this consistency predicate and verify whether such a possible inconsistency situation occurs or not. If the run-time system detects such a possible inconsistent situation it may stop the system execution for reliability reasons. Note that this formal consistency concept for concurrent threads is similar to optimistic locking techniques in databases.

11.2.2 The DisCComp Description Technique

In the previous section the instance level has been introduced. These instances are the semantic domain of the description level. In other words, the description level provides the notations and description techniques to describe the run-time instances in an uniform and precise manner.

Along with the initial DisCComp system model from [7], a description technique for components was developed which consists of a graphical, UML-based notation for the statical structure of systems and a textual description of their behaviours. Further extensions aimed at extending this description technique by specifying behavioural aspects by UML activity diagrams (s. [8]). The specification technique proposed here is essentially based upon that approach and extends it by following features:

- Consideration of system model extensions: The possibility of synchronous communication, which has been added on the system model in [9], has massive impact on the modularity of a component's specification. If a method of an interface which is assigned to a component A calls a method of an interface assigned to a component B, the behaviour of component A can only be fully specified if B's behaviour is known as well. The original DisCComp system model allows only asynchronous communication [7]. So called "island specifications" with explicit specifications for required interfaces are necessary to obtain modularity.
- Usability: The explicit specification of required interfaces as shown in [7] increases the effort of specifying the behaviour of interfaces because behaviour is additionally specified along with the component which assures that interface. In our approach, the behaviour of required interfaces is described in form of contracts [10], whereas an operational language is used to describe methods provided by assured interfaces. These descriptions can be analysed to check whether wiring two components is possible, meaning whether the provided interface is a refinement of the required interface.
- Application of UML 2.0 features: Since version 2.0 [11], UML as a broadly accepted modelling language provides some additional concepts for composite structures, which were not yet available for the original description technique. In contrast, we will focus on a textual specification language for behavioural aspects due to the maturity of graphical approach in the context of DisCComp.

Fig. 2 shows the structure of a component's specification which can be conceptually split into three parts.

The static structure part of a component's specification is described by a UML component diagram and an arbitrary number of class diagrams. The UML terms (components, interfaces, attributes, etc.) used in this context can be mapped one-to-one to the elements of the DisCComp system model. The component diagram gives an overview of the interfaces provided (or assured) and required by a component, whereas the class diagrams visualize the attributes and methods of interfaces as well as relationships between interfaces. These specify the structure and permitted connections between interface instances. The set of instances of a required interface that are assigned to a component is modelled as a *port* of the component. Since a port is a specialisation of the meta-model element *Feature*, it is a quantifiable part of a *Classifier* (see [11] and [12]). By this means, we

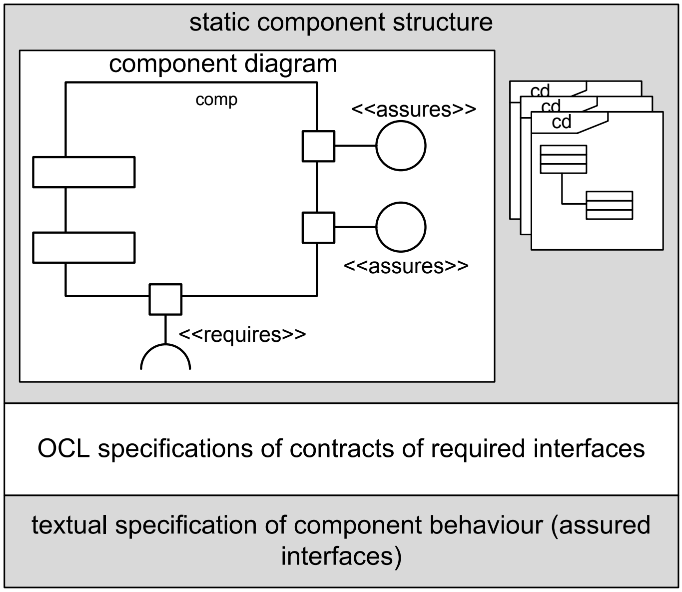

Fig. 2. Conceptual Structure of Component Specifications

are able to reflect the assignment of interface instances to components from the DisCComp system model. We also specify the set of instances of required interfaces by ports to refer to the set of provided interface instances of serving components.

The second part specifies the required interfaces' behaviours by terms of contracts ([10], [13]). We make use of OCL's means to specify pre- and post-conditions of methods and invariants that have to hold for required interfaces. When wiring components and building larger, more complex components, we map the required interfaces of a client component to provided interfaces of serving components, whereas the behaviour of provided interfaces has to "match" the required interfaces of the client component.

The third part consists of a set of textual behaviour specifications which describe the behaviour of interfaces assured by the component. Syntax and semantics are similar to the original approach in [7]. When wiring components, it will be checked by program analysis techniques whether the provided interface fulfills the contract specified by the client interface.

Static specification. The static specification of a component basically consists of an ordinary UML component diagram and additional class diagrams. The component diagram gives an overview of the interfaces that are provided by the component[3]. There are two levels which have to be considered when specifying

[3] We additionally define new stereotypes "requires" and "assures" to be conform to the usual DisCComp terms.

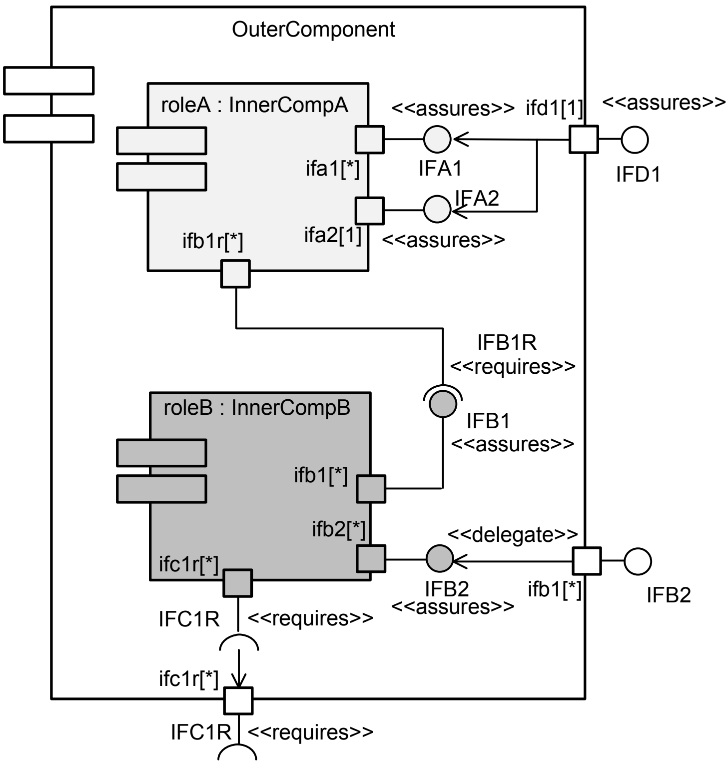

Fig. 3. Wiring and Building Complex Components

the interrelation between components and interfaces. First, on the type level, we define by dependencies with stereotypes <<requires>> or <<assures>> to specify interface types a component provides for use, and which interface types it requires for operation. Secondly, by assigning ports to components we define specifiers for the set of instances of an interface type which is assigned to the component.

Complex, hierarchical components are specified as a white box (see Fig. 3). Inner sub-components are used as black boxes which can only accessed by the interface types and ports they define. Only these specifiers and specifiers which are defined by the outer surrounding component itself can be used inside its specification. Beside this, we use the same means of UML component diagrams to specify hierarchical components as mentioned above.

The outer component can use assured interfaces of inner components to realize its own functionality which is modelled as a dependency between the corresponding ports in the component diagram, as between the interface IFD1 in Fig. 3 and the interfaces IFA1 and IFA2. This means semantically that interfaces instances of IFD1 can connect to instances of IFA1 and IFA2 to call methods and send messages. In contrast, dependencies with the stereotype <<delegate>>

indicate interfaces of inner components which are accessible from and connectable to clients of the specified hierarchical component (see IFB2 in Fig. 3). The instances are assigned to the inner component.

Interfaces labeled as required by inner components can similarly be specified as being required by the surrounding component (IFC1R in Fig. 3). The inner component, which requires such an interface, and interfaces assigned to it can connect to instances of an interface type which is mapped to the required interface.

After describing the specification of the static structure of components, we will illustrate how their behaviours and initialization are specified. That includes an abstract, contract-based specification of required interfaces and an operational description language for assured interfaces.

Abstract behaviour specification. We call the description of required interface behaviour *abstract behaviour specification* because it describes interfaces and their behaviours as abstract as possible, resulting in a specification of what "is needed at least". A component that provides an interface to another component has to specify a behaviour that is a valid refinement of the required behaviour - otherwise wiring the components would not be possible.

The textual description of abstract behaviours is formulated by contracts of the methods an interface is required to have, and by interface invariants. We use OCL [14] to specify such additional constraints which integrates seamlessly with the UML component and class diagrams of the static view. OCL supports pre- and post-conditions as well as invariants. By pre- and post-conditions, we specify which condition for the component's state is going to hold after the method's return if the pre-conditon holds before its invocation[4].

For syntactical details on OCL, please refer to [14]. If necessary, constructs will be introduced in the following component specifications.

Operational behaviour specification. The proposed behavioural specification language is a textual language based on [7]. It is similar to imperative programming languages but is syntactically and semantically designed to match the DisCComp system model. In this section, we will focus on language constructs that reflect the peculiarities of the system model and will omit syntax and semantics for well known constructs like control structures, assignments, etc., for the sake of brevity. The component specifications for the modelling example in the following sections will be explained in more detail.

A behavioural component description consists of an initialization block and a set of interface specifications which again consist of method specifications. The initialization block is executed when a component instance is created. It contains a specifier mapping to map the required interface types of sub-components to assured interfaces of other components. Furthermore, the initial wiring is done, specified by a set of instructions to create an initial set of interfaces and connections. These instructions define the initial port assignment and connection

[4] This is similar for asynchronous messages but the post-condition has to hold after sending, not processing the message.

among components and interfaces. We will see examples of initialization blocks in the following sections.

The DisCComp system model is able to realize asynchronous as well as synchronous communication. The signatures of messages, which can be processed asynchronously by an interface, and synchronous methods, which can be invoked at it, are specified in the UML class diagrams of the static specifications as mentioned above. Behavioural descriptions are described textually in `MESSAGE` or `METHOD` blocks respectively[5].

The most important possibility to change the structural state of the system inside methods is to create instances of interfaces and connections (see Table 1). Interfaces have to be connected to communicate with each other. For that reason, the creation of an interface instance can optionally be coupled with creating a connection to that instance for direct use.

Table 1. Creation of Instances

Syntax	Informal Semantics
`ifInst : IfType = NEW INTERFACE IfType [CONNECT BY ConnType]`	A new interface instance of type `IfType` is created. The instance is assigned to the same component as the calling interface. A new connection is optionally established between the newly created and the existing, calling interface. The connection type is a valid association name. `IfType` must be assured by the component.
`connInst : ConnType = NEW CONNECTION ConnType TO ifInst`	A new connection is established between the calling interface and the interface `ifInst`. `ifInst` must be of a type that is conform to the static specification of the according association and its ends.
`connInst : ConnType = NEW CONNECTION ConnType BETWEEN ifInst1, ifInst2`	Creates a connection of type `ConnType` between the interface instances `ifInst1` and `ifInst2`. Only available in initialization blocks.
`compInst : CompType = NEW COMPONENT CompType`	A new component of type `CompType` is created.

However, this just allows us to create interface instances that are assigned to the same component as the calling interface. To connect to interfaces that are provided by different components, we have to call appropriate methods (or send appropriate messages) of those components which create instances and connect them to the calling or sending interfaces. This is comparable to passing return values of methods to the requesting client. Table 2 shows the according language constructs. The upper construct can be compared to returning a reference to an object which someone else (the serving component) is responsible for. The second

[5] For details regarding the execution of synchronous methods in DisCComp, please refer to [9].

Table 2. Returning Interface Instances

Syntax	Informal Semantics
CONNECT ifInst TO CALLER	ifInst is connected with the calling interface. Assignment of ifInst is not changed.
CONNECT ifInst TO CALLER AND REASSIGN	ifInst is connected with the calling interface and assigned to the same component.

statement is comparable to returning a copy of an object, but instead of actually copying the object, the object itself is separated from the creating component and assigned to the calling component.

After processing a message or method call, locally created instances are disposed.

The keywords and specification blocks will be used and explained in more detail in the example component specification in the following sections.

11.3 Modelling the CoCoME

In order to present our modelling techniques, we decided to take a certain cutout of the *Trading System*. The following sections will describe our approach to model the use case *ChangePrice* (UC 7). This use case requires us to specify many components on different layers, reaching from the GUI to the application layer and down to the data layer and is therefore representative for our modelling approach.

11.3.1 Static View

We decided to model the use case *ChangePrice*. Therefore, we first of all will have a look at the components which are relevant in order to implement the use case. In Fig. 4, the component diagram of the component :TradingSystem::Inventory is given. All components relevant for use case *ChangePrice* are subcomponents of :TradingSystem::Inventory.

The purpose of the use case is to let the manager change the price of a product. Therefore, the manager chooses a stock item from a list of available stock items in his store. The component :TradingSystem::Inventory::GUI is responsible for showing the dialog and forward the information to the application layer. The component therefore uses the interface StoreIf which defines among others a method for changing a price of a stock item. The component :TradingSystem::Inventory::Application implements the application layer of the classical three-layer-architecture (see [15], for example) and decouples the graphical user interface from the data access layer. The application layer communicates with the data layer, which is implemented by :TradingSystem::Inventory::Data, by using the interface StoreQueryIf.

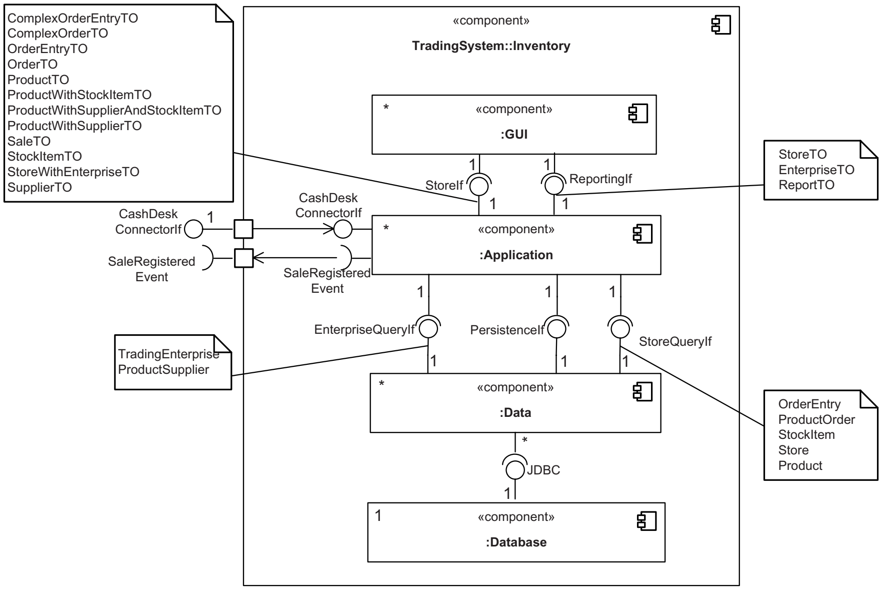

Fig. 4. Overview of the component :TradingSystem::Inventory

11.3.2 Behavioural View

To present the behavioural view on the use case we consider, we use a sequence diagram to visualize how the components presented before interact. In Fig. 5 you can see the Sequence Diagram which depicts how the components interact and how they are involved while realizing the use case.

After the manager has choosen a product item, a so called *transfer object* of type StockItemTO is created and sent to the component *:TradingSystem::Inventory::Application::Store*. After this component has established a new transaction, the price of the product item is changed by first querying the corresponding persistent object in the database and, secondly, change the price by calling *setSalesPrice()* at this object. The new price is then sent back to the GUI component in form of a new transfer object of type *ProductWithStockItemTO*. In the next sections we will show how we model these components and their behaviours using our modelling approach and description technique.

11.3.3 Component Specifications

Taking a closer look at the sequence diagram, you can see that we need to specify the components :GUI::Store, :Application::Store, :Data::Persistence and :Data::Store. Furthermore we need to describe the operational behaviour of interfaces like StockItemTO, StoreIf, etc. So we start with a static view of single components and then switch over to complex, hierarchical components. All these are described by ordinary UML component diagrams extended by "assures" and

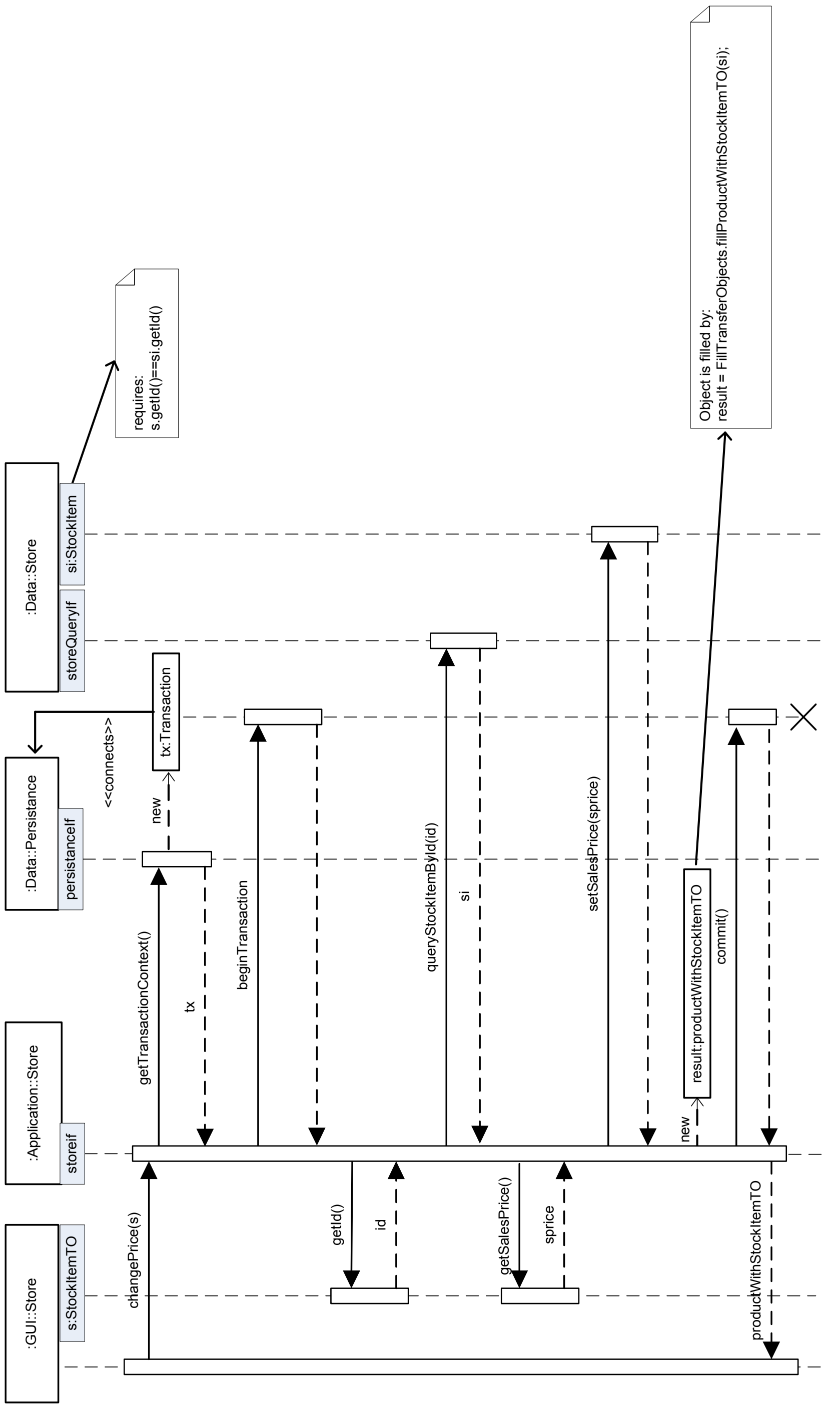

Fig. 5. Sequence Diagram for Use Case 7

"requires" at the according interfaces. An "assures" defines, that this component can provide these interfaces for other components which require it. On the other hand, "requires" provides the counterpart to "assures" and defines that this component needs the "required" interface from another component. The numbers at the port names show whether the corresponding interface is required more than once or whether it can be assured by the component more than once. Afterwards the operational behaviour for the assured interfaces of the component is specified by using a textual description according to the DisCComp system model. The operational behaviour for the required interfaces is specified by OCL code that is embedded into that textual description. It is structured into four consecutive sections:

1. *Specifier Mapping Section:* It defines the mapping between required and assured interfaces components inside the specified component.
2. *Initialization Section:* Here, the initial instantiation of interfaces is specified.
3. *Assured Interfaces Section:* This section contains the behavioural specification of assured interfaces.
4. *Required Interfaces Section:* This section contains the contracts of required interfaces.

We will illustrate this structure by the following components of the modelled cutout of the CoCoME.

11.3.4 Specification of Component Inventory::Application::Store

According to Fig. 6 this component assures the interfaces ProductWithStockItemTO and StoreIf. Furthermore it "requires" several interfaces to work correctly, namely PersistenceIfR, StoreQueryIfR, StockItemR, TransactionContextR and PersistenceContextR. Omitted multiplicities indicate single instances (multiplicity of one).

Since an atomic component is not wired in order to form some more complex and surrounding component, a specifier mapping section in its textual specification would be empty. As a consequence, we can omit the specifier mapping for this component. Listings 11.1 and 11.2 describe the remaining sections.

```
COMPONENT Inventory::Application::Store

INITIALIZATION
  storeIf := NEW INTERFACE StoreIf;

ASSURES
   INTERFACE StockItemTO
      METHOD getId():long
        RETURN VALUE OF self.id TO CALLER;
      END METHOD
      //... further methods omitted here
   END INTERFACE

```

```
    INTERFACE ProductWithStockItemTO
       METHOD getId():long
         RETURN VALUE OF self.id TO CALLER;
       END METHOD

       METHOD getSalesPrice(): double
         RETURN VALUE OF self.salesPrice TO CALLER;
       END METHOD
       //... further methods omitted here
    END INTERFACE

    INTERFACE StoreIf
       METHOD changePrice(StockItemTO stockItemTO):
           ProductWithStockItemTO
         result: ProductWithStockItemTO := NEW INTERFACE
             ProductWithStockItemTO;
         pctx: PersistenceContextR := persistenceIfR.
             getPersistenceContext();
         tx: TransactionContextR := pctx.
             getTransactionContext();
         tx.beginTransaction();
         si: StockItemR := storequery.queryStockItemById(
             stockItemTO.getId())
         IF (si != NULL) THEN
           si.setSalesPrice(stockItemTO.getSalesPrice());
           //copy data to result transfer object
         ELSE
           result := NULL;
         ENDIF
         tx.commit();
         pctx.close();
         CONNECT result TO CALLER AND REASSIGN;
       END METHOD
       //... further methods omitted here
    END INTERFACE

END ASSURES
```

Listing 11.1. Operational Behaviour of Assured Interfaces of Inventory::Application::Store

The section of assured interfaces starts with `ASSURES` and contains the interfaces being assured by the component. Single interface blocks start with `INTERFACE ifName` and end with `END INTERFACE` while each method provided by the interface starts with `METHOD methodname (PARAMETERS):RETURNTYPE` and ends with `END METHOD`. Then the `ASSURES` block is closed by `END ASSURES` (see Listing 11.1).

Identifiers from the component diagram can be used here as, for example, in `changePrice(...)` of the interface `StoreIf`. The instance `persistenceIfR` is used to get the persistence and transaction contexts. After that, the method queries

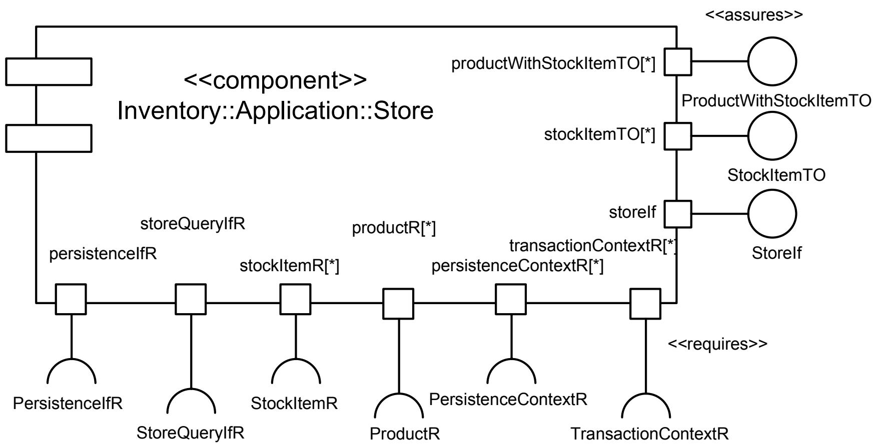

Fig. 6. Static View of Component Inventory::Application::Store

the data layer for the stock item and changes its price. The result is returned and reassigned. Thus, the according instance of ProductWithStockItemTO, which was created during the method call, will be passed to the calling component.

The `ASSURES` block is followed by the `REQUIRES` block for this component (see Listing 11.2), which also encapsulates interface and method blocks. Instead of operational method bodies, the method blocks contain pre- and/or postconditions in OCL. For example, the postcondition for `queryStockItemById(long)` states that if there is a stock item with the given id, it will be returned, otherwise the result is null[6].

The initialization section specifies the creation of a single `StoreIf` instance during the creation of a component instance.

The end of the component description is declared by `END COMPONENT`.

In the following, for the sake of brevity, we will show incomplete specifications of the modelled components, interfaces, and methods that are nevertheless sufficient to cover UC 7.

```

REQUIRES

   INTERFACE PersistenceIfR
     METHOD getPersistenceContext():PersistenceContext
       Post: result!=NULL
     END METHOD
   END INTERFACE

   INTERFACE StockItemR
     METHOD getId():long
       Post: result = self.getId()@pre
```

[6] The postcondition assumes an invariant which states that the ID of a stock item is unique.

```
      END METHOD

      METHOD setSalesPriceR(real salesPrice):void
        Pre: salesPrice >0
        Post: self.getSalesPrice()=salesPrice
      END METHOD
    END INTERFACE

    INTERFACE StoreQueryIfR
       METHOD queryStockItemById(long sId): StockItem
         Pre: sId >= 0
         Post: let queriedItems : Set(StockItemR) =
             stockItemR->select(s|s.getId()=sId) in
           if queriedItems->notEmpty then
           result = queriedItems->first();
         else
           result = NULL
         endif
       END METHOD
    END INTERFACE

    INTERFACE TransactionContextR
      METHOD beginTransaction(): void
        Post: // Transaction is started
      END METHOD

        METHOD commitTransaction(): void
        Post: // Transaction is commited
      END METHOD

      METHOD rollback():void
        Post: //Rollback executed
      END METHOD
    END INTERFACE

    INTERFACE PersistenceContextR
      METHOD getTransactionContext(): TransactionContext
        Post: result != NULL
      END METHOD

      METHOD close():void
        POST: //Close the Persistence context
        END METHOD
      END INTERFACE
END REQUIRES

END COMPONENT
```

Listing 11.2. Operational Behaviour of Required Interfaces of Inventory::Application::Store

11.3.5 Specification of Component Inventory::Data::Persistence

The component Inventory::Data::Persistence provides services that deal with persisting and storing objects into a database as well as managing transactions. It assures a single instance interface PersistenceIf which provides methods for clients to get the persistence context. This interface again enables us to create transaction contexts which contains methods to create and commit transactions.

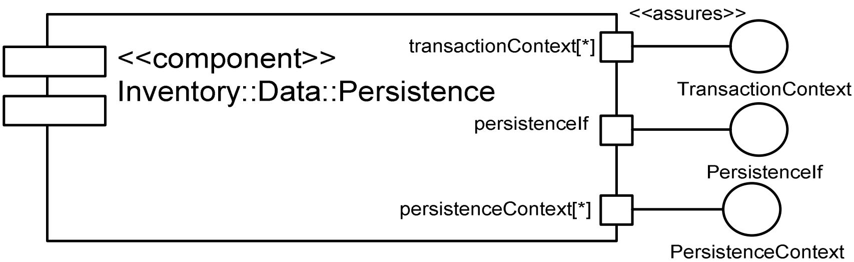

Fig. 7. Static View of Component Inventory::Data::Persistence

Since this component is a rather technical one and externally provided in larger parts, we were not able to model the behaviour properly by analyzing the CoCoME code. We simplify its behaviour by only specifying those methods that create instances of the corresponding context interface instances.

```

COMPONENT Inventory::Data::Persistence

INITIALIZATION
  persistenceRole.persistenceIf := NEW INTERFACE
      PersistenceIf;

ASSURES

   INTERFACE TransactionContext
     METHOD beginTransaction() : void
       //external behaviour, no operational description here
     END METHOD

     METHOD commit() : void
       //external behaviour, no operational description here
     END METHOD

     METHOD rollback() : void
       //external behaviour, no operational description here
     END METHOD

     METHOD isActive() : Boolean
       //external behaviour, no operational description here
```

```
      END METHOD
    END INTERFACE


    INTERFACE PersistenceIf
      METHOD getPersistenceContext(): PersistenceContext
          result : PersistenceContext := NEW INTERFACE
              PersistenceContext;
          CONNECT result TO CALLER AND REASSIGN;
      END METHOD
    END INTERFACE

    INTERFACE PersistenceContext
      METHOD getTransactionContext():TransactionContext
        result : TransactionContext := NEW INTERFACE
            TransactionContext;
        CONNECT result TO CALLER AND REASSIGN;
      END METHOD

          //further methods omitted here
    END INTERFACE
END ASSURES

END COMPONENT
```

Listing 11.3. Operational behaviour of assured Interfaces of Inventory::Data::Persistence

11.3.6 Specification of Component Inventory::Data::Store

The component Inventory::Data::Store provides the interfaces StoreQueryIf and StockItem (see fig. 8). The methods of StoreQueryIf encapsulate methods for querying persistent objects represented by interfaces like StockItem, Product, etc. These interfaces provide methods for retrieving and changing their attribute values. For this reason, it requires technical services like PersistenceIfR, TransitionContextR and PersistenceContextR.

```
COMPONENT Inventory::Data::Store

ASSURES

    INTERFACE StoreQueryIf
      METHOD queryStockItemById(long stockId): StockItem
        StockItem result;
        //calls to persistence framework which were not
            modelled explicitly
        //the StockItem instance is retrieved from the
            database if it exists, otherwise result is set to
             NULL
```

```
      CONNECT result to CALLER;
    END METHOD
  END INTERFACE


  INTERFACE StockItem
    METHOD setSalesPrice(SalesPrice salesPrice): void
      self.salesPrice:=salesPrice;
    END METHOD
  END INTERFACE

END ASSURES



REQUIRES

  INTERFACE PersistenceIfR
     METHOD getPersistenceContextR(): PersistenceContext
       Post: result != NULL
     END METHOD
  END INTERFACE

  INTERFACE TransactionContextR
     METHOD beginTransaction(): void
       Post: //Transaction must be started!
     END METHOD

     METHOD commit(): void
       Post: //Transaction committed
     END METHOD
  END INTERFACE

  INTERFACE PersistenceContextR
     METHOD getTransactionContext():TransactionContext
      Post: result!=NULL
     END METHOD
  END INTERFACE

END REQUIRES
END COMPONENT
```

Listing 11.4. Operational behaviour of assured and required Interfaces of Inventory::Data::Store

11.3.7 Specification View of Component Inventory::GUI::Store

We have not reconstructed the inner structure of the GUI from the code for reasons of simplification, thus Inventory::Gui::Store is quite trivial. It only requires a StoreIf to invoke the changePrice() method.

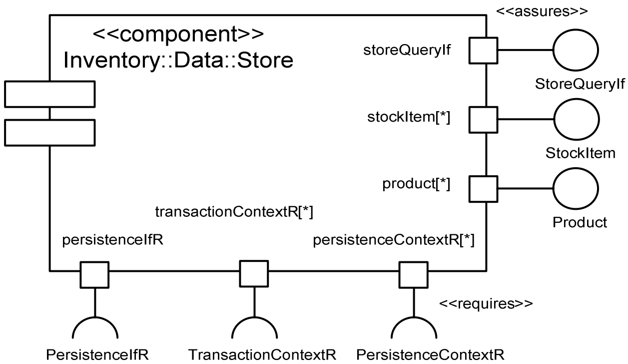

Fig. 8. Static View of Component Inventory::Data::Store

11.3.8 Specification of Component Inventory::Data

Inventory::Data represents the data layer (ref. to Fig. 10) of the inventory system. It consists of Inventory::Data::Persistence and Inventory::Data::Store. The required interfaces of the inner component Inventory::Data::Store are provided by Inventory::Data::Persistence. All assured interfaces are delegated to inner components to allow access from the environment, for example from the application layer of the inventory system.

The specifier mapping for this hierarchical component is described in Listing 11.5.

```
COMPONENT Inventory::Data

SPECIFIER MAPPING
  persistenceRole::PersistenceIfR <-> storeRole::
      PersistenceIf
  persistenceRole::TransactionContextR <-> storeRole::
      TransactionContext
  persistenceRole::PersistenceContextR <-> storeRole::
      PersistenceContext
```

Listing 11.5. Operational behaviour of required Interfaces of Inventory::Data

The specifier mapping simply maps all required interfaces of one component to its assured pendants at another component, namely the interfaces which assure what the other one requires, e.g. `Data::Persistence::PersistenceIfR` will be assured by the interface `PersistenceIf` of the component `Data::Store`.

11.3.9 Specification of Component Inventory::Application

This hierarchical component (ref. to Fig. 11) consists of Inventory::Application::Store and Inventory::Application::Reporting and thereby specifies the application

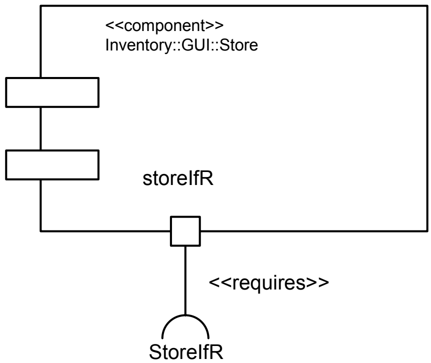

Fig. 9. Static View of Component Inventory::GUI::Store

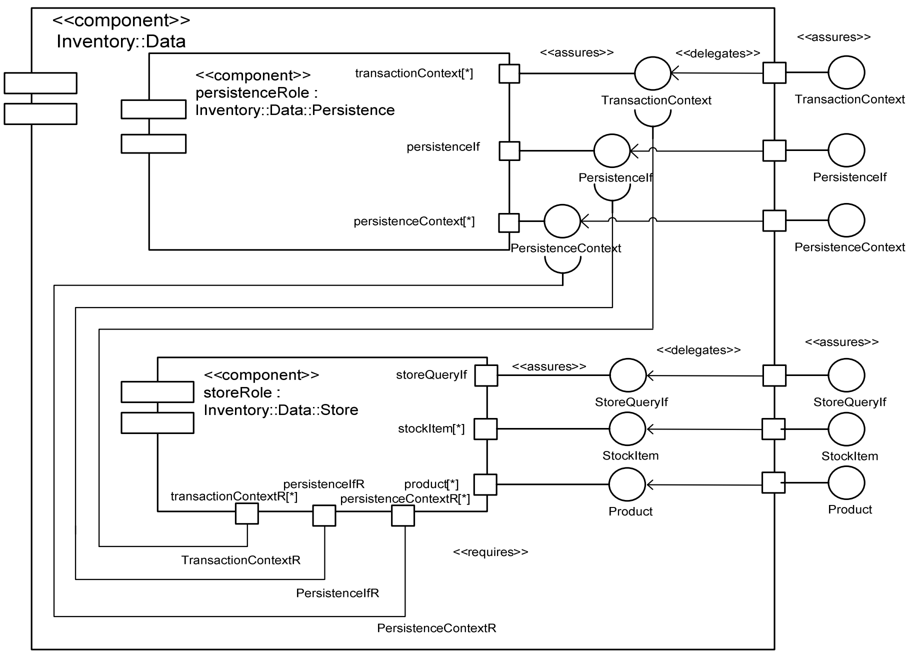

Fig. 10. Static View of Component Inventory::Data

layer. We have not modelled Application::Reporting since it is not required by UC7, so everything it assures or requires is left out.

Once again all assured interfaces of the outer component are delegated to interfaces of inner components, whereas the new required interfaces may not contradict or tighten the existing requirements.

11.3.10 Specification of Component Inventory

This is the hierarchical component which describes the whole system. We left out Inventory::GUI for graphical reasons, so the diagram in Fig. 12 shows only the two components Inventory::Application and Inventory::Data. The wiring between the layers basically reflects the wiring between the components of different

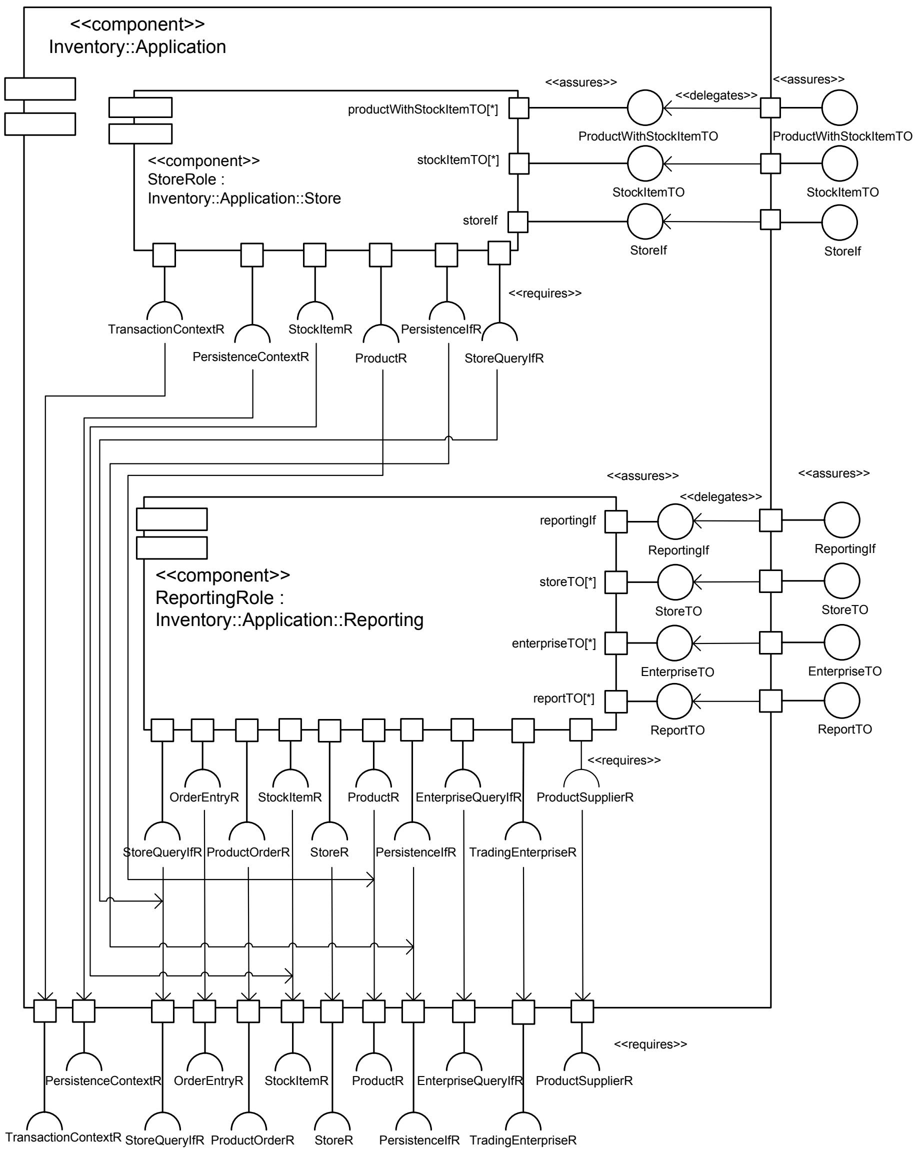

Fig. 11. Static View of Component Inventory::Application

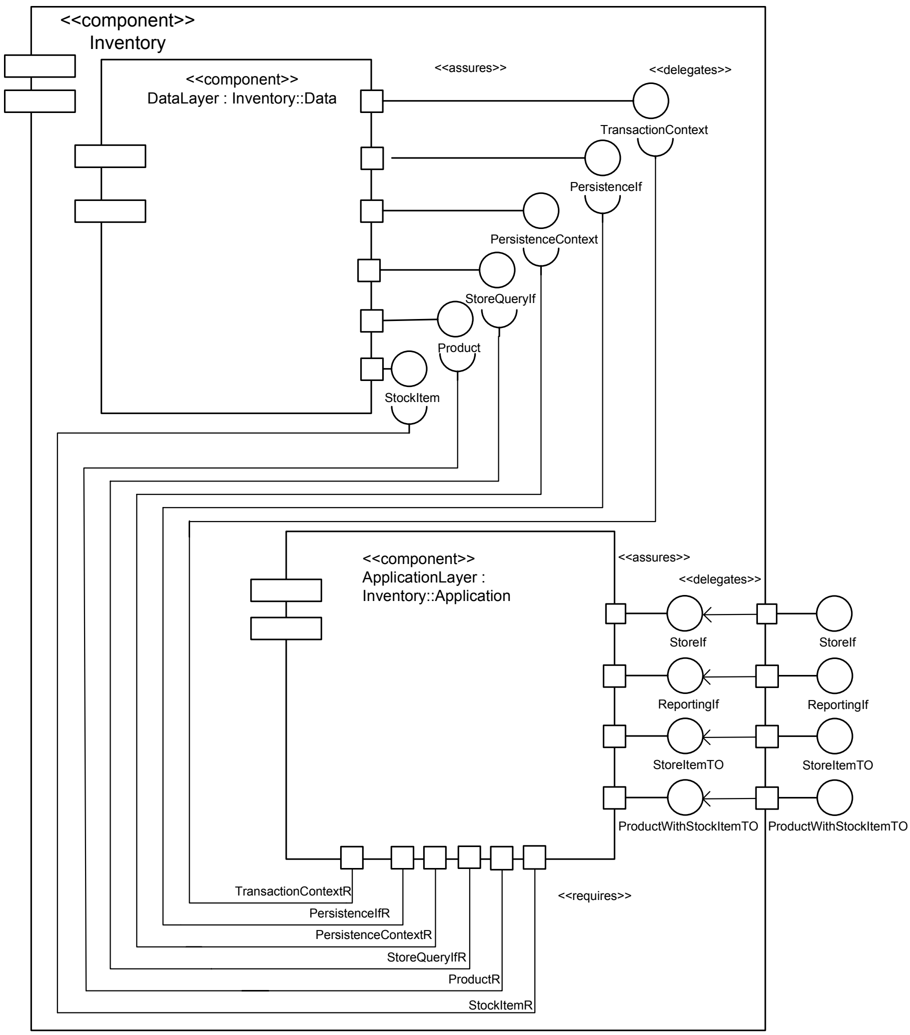

Fig. 12. Static View of Component Inventory

layers. For instance, the component Inventory::Application::Store requires interfaces of Inventory::Data::Store or Inventory::Data::Persistence (s. Fig. 6). Only the interfaces of the application layer are delegated to the environment, since the application layer will control access by different applications to the data layer.

```
COMPONENT Inventory

SPECIFIER MAPPING
  Application::PersistenceIfR <-> Data::PersistenceIf
```

```
Application::TransactionContextR <-> Data::
    TransactionContext
Application::PersistenceContextR <-> Data::
    PersistenceContext
Application::StoreQueryIfR <-> Data::StoreQuery
Application::StockItemR <-> Data::StockItem
Application::ProductR <-> Data::Product
```

Listing 11.6. Specifier Mapping and Initialization of Inventory

11.4 Transformations

During research work regarding the DisCComp approach, the tool *DesignIt* was developed which includes a generator for generating executable code from XML-based representations of specifications as described in [7]. It is controlled by templates which enable it to generate code for different target languages. A detailed description of the code generator is included in [16] and [17]. Model-to-model transformations are not yet considered.

11.5 Analysis

As mentioned above, we aim at a modelling approach which allows us to specify components in a modular way. To compose components, we have to check whether the wiring of components is correct, meaning whether the specified contracts are fulfilled, at design time. For this purpose, the operational behaviour specifications of assured interfaces are analyzed and used to generate some representation which can be compared to the abstract behaviour specifications of the corresponding required interfaces. By using a more intuitive operational description technique and automatic generation, we hope to avoid the disadvantage of specifying the abstract contracts of the wired interfaces twice.

11.6 Tools

As mentioned above, the existing tool DesignIt is based upon the original specification technique of [7]. Tool support for that approach exists in the form of modelling tools, consistency checking, code generation, and runtime and testing environments. Most of this support has to be adapted to the specification techniques we have proposed here.

11.7 Summary

In this chapter we presented the DisCComp model for modelling distributed and concurrent component-based systems. It is separated into the system model and a description techniques which enables us to specify such systems. The proposed description technique differs greatly from the original proposals in the

DisCComp context (s. [7]). Although it is still young, we can summarize some lessons learned by applying it to the common modelling example.

First, the imperative specification of assured interfaces has reduced the effort of specifying both, assured and required interfaces. The imperative way of specifying them seems to be more intuitive. But secondly, it still causes some overhead in comparison to specifications without contracts which seems to be unavoidable to get modular specifications.

The overall approach still lacks of tool support which is part of the future work. Especially program analysis and the checking of component wirings will have to be theoretically founded and embedded into the approach and realized by tools.

References

1. Aleksy, M., Korthaus, A., Schader, M.: Implementing Distributed Systems with Java and CORBA. Springer, Heidelberg (2005)
2. Juric, M.B.: Professional J2EE EAI. Wrox Press (2002)
3. Beer, W., Birngruber, D., Mössenböck, H., Prähofer, H., Wöß, A.: Die .NET-Technologie. dpunkt.verlag (2006)
4. Harel, D., Rumpe, B.: What's the semantics of semantics. IEEE Computer 37(10), 64–72 (2004)
5. Broy, M., Len, K.S.: Specification and Development of Interactive Systems. Springer, Heidelberg (2001)
6. Bergner, K., Broy, M., Rausch, A., Sihling, M., Vilbig, A.: A formal model for componentware. In: Foundations of Component-Based Systems. Cambridge University Press, Cambridge (2000)
7. Rausch, A.: Componentware: Methodik des evolutionären Architekturentwurfs. PhD thesis, Technische Universität München (2001)
8. Seiler, M.: UML-basierte Spezifikation und Simulation verteilter komponentenbasierter Anwendungen. Master's thesis, Technische Universität Kaiserslautern (2005)
9. Rausch, A.: DisCComp - A Formal Model for Distributed Concurrent Components. In: Workshop Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA 2006) (2006)
10. Meyer, B.: Applying 'design by contract'. IEEE Computer 25(10) (1992)
11. Object Management Group (OMG): Unified Modeling Language (UML) 2.0 Superstructure Specification (2005), http://www.omg.org/docs/formal/05-07-04.pdf
12. Object Management Group (OMG): Meta Object Facility Core Specification Version 2.0 (2006), http://www.omg.org/docs/formal/06-01-01.pdf
13. Rausch, A.: Design by Contract + Componentware = Design by Signed Contract. Journal of Object Technology 1(3) (2002)
14. Object Management Group (OMG): Object Constraint Language (OCL) Specification (2006), http://www.omg.org/docs/formal/06-05-01.pdf
15. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P.: Pattern-Oriented Software Architecture. A System of Patterns, vol. 1. John Wiley, Chichester (1996)
16. Kivlehan, D.: DesignIt: Code Generator based on XML and Code Templates. Master's thesis, The Queen's University of Belfast (2000)
17. Rausch, A.: A Proposal for a Code Generator based on XML and Code Templates. In: Proceedings of the Workshop of Generative Techniques for Product Lines, 23rd International Conference on Software Engineering (2001)