
Research

Relating functional requirements and software architecture: separation and consistency of concerns



R. Heckel^{*,†} and G. Engels

Department of Mathematics and Computer Science, University of Paderborn, D-33095 Paderborn, Germany

SUMMARY

In the early stages of most software processes, functional and non-functional (such as architectural, performance, or security) requirements are expressed separately in different sub-models. Later these requirements have to be integrated into one overall system design. The integration raises consistency issues between different sub-models, which have to be resolved in the process.

If requirements evolve over time, this leads to changes to the sub-models concerned. Thus, new consistency issues arise between changed and unchanged sub-models. In this case, a clear separation of concerns between different sub-models is required to keep the effect of changes as local as possible.

In this paper, we use a relational approach to couple functional and architectural models while keeping them separated in order to simplify change. The approach uses meta modeling to support the static integration and concepts from the theory of graph transformation to formalize the semantic consistency of the dynamic aspects. Copyright © 2002 John Wiley & Sons, Ltd.

KEY WORDS: typed graph transformation; meta modeling; model evolution

1. INTRODUCTION

Software development can be perceived as the problem of implementing a set of diverse, incomplete, and conflicting requirements expressing the concerns of different stakeholders within a single system.

*Correspondence to: R. Heckel, Department of Mathematics and Computer Science, University of Paderborn, D-33095 Paderborn, Germany.

†E-mail: reiko@upb.de

Contract/grant sponsor: ESPRIT Working Group APPLIGRAPH



This process involves the structured representation of requirements, either diagrammatically or in textual form, as well as their subsequent integration.

Driven by requirements for different viewpoints, such as functionality, architecture, performance, etc., separation of concerns in models is essential to reduce complexity by abstracting from details that are irrelevant for the specific viewpoint. However, since different viewpoints of the same system are related, this raises consistency problems which have to be detected and resolved in the course of development. We distinguish between syntactic and semantic consistency problems.

Syntactic consistency problems are caused by the fact that models are (or should be) syntactically related, e.g. through the use of common names, and that these relations have to obey certain well-formedness rules. For example, objects in a collaboration diagram may have to be instances of classes in a class diagram. Semantic consistency is concerned with conflicts between requirements for the system to be built. An inconsistent model does not have a correct realization. Therefore, semantic inconsistencies which cannot be resolved immediately have to be traced through further development steps in order to be dealt with later [1].

Software evolution is driven by changes of both requirements and technology. Functional requirements change if new features have to be implemented or an existing function needs to be performed in a different way. If technology advances, this frequently leads to evolution of non-functional requirements. For example, the architecture of an application will have to change if a new component infrastructure is adopted. Also, new requirements, e.g. related to safety or performance issues, may be raised at any time in the software lifecycle.

Since requirements are expressed from the viewpoints of different stakeholders, they are unlikely to evolve in a coherent and synchronized way. At the level of models this is reflected by the evolution of certain sub-models while others remain unchanged. This independent evolution causes new consistency problems, both syntactic and semantic.

In this paper, we study this problem in the case of functional requirements and architectural constraints. Functional requirements tell us what the system has to do in order to perform its genuine task. Architectural constraints are concerned with decomposition into components, their physical distribution, and the possible connections between them. The development problem consists in realizing the required functionality on the given architecture.

The discussion will be based on a meta object facility (MOF)-like [2] meta modeling approach for the static/syntactic aspect (cf. the UML specification [3]) combined with graph transformation to model the dynamic/semantic aspect. Meta modeling allows for a uniform representation of models of different concerns as required for their interrelation, while keeping them syntactically separated. Graph transformation provides the concepts and theory required to deal with the interplay of ongoing computation and dynamically changing architecture.

Throughout the paper, we will discuss a simple application scenario of a distributed banking application. This will help us to introduce the basic concepts of graph transformation and meta modeling along with the methodological approach. First, in Section 2 we discuss the modeling of functional behavior and architectural dynamics by graph transformation. Then, in Section 3 we describe how these two aspects are related both syntactically and semantically. Section 4 demonstrates an evolution of the architecture while the functional model is preserved, thus exemplifying the benefit of a clear separation of concerns for the evolution of models. We close with a summary and discussion in Section 5.

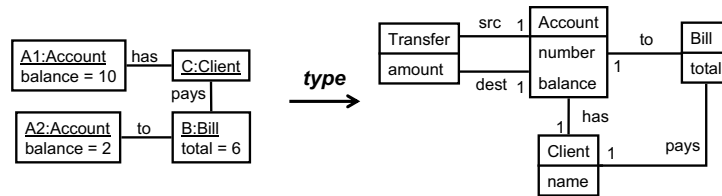


Figure 1. Object diagram (left) typed over class diagram (right).

2. MODELING WITH GRAPH TRANSFORMATION

In this section, the use of graph transformation for modeling functional requirements as well as software architectures and their dynamic changes shall be illustrated by an example. By doing this, we will give an introduction to some basic concepts and constructions from the theory of graph transformation that are needed to relate functional and architectural models in the next section.

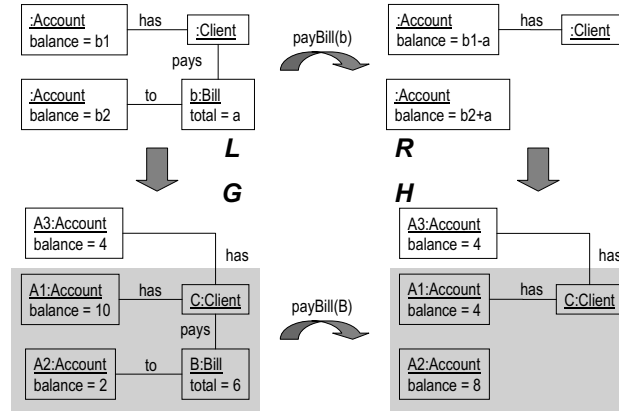
2.1. Functional behavior

Functional requirements are often presented in terms of use-cases, i.e. the main services a system shall provide to its users. Every such use-case provides a more detailed description of its pre- and post-conditions and of the interactions required to perform the respective service. In [4] typed graph transformation systems have been proposed as a way to specify use-cases in a visual, yet formal, way.

Graphs as states. Graphs are often used as abstract representations of diagrams, e.g. in the UML meta model [3]. Formally, a *graph* consists of a set of vertices V and a set of edges E such that each edge e in E has a source and a target vertex $s(e)$ and $t(e)$ in V , respectively. Variations include hypergraphs, where edges can be attached to an arbitrary sequence of vertices, attributed graphs [5], whose vertices and edges are decorated with textual or numerical information, or more complex object-oriented or hierarchical graph models. The constructions described below are largely independent of the notion of graph, which can be chosen to reflect as closely as possible the concepts of the modeling language. In the following we deal with attributed graphs.

In object-oriented modeling graphs occur at two levels: the type level (given by the class diagrams) and the instance level (given by all valid object diagrams). This idea can be described more generally by the concept of *typed graphs* [6], where a fixed *type graph* TG serves as an abstract representation of the class diagram. Its object diagrams are graphs equipped with a structure-preserving mapping to the type graph, formally expressed as a *graph homomorphism*.

Figure 1 shows examples of an object and a class diagram in UML notation [3]. The object diagram on the left can be mapped to the class diagram by defining $type(o) = C$ for each instance $o : C$ in the diagram. Extending this to links, preservation of structure means that, e.g., a link between objects o_1 and o_2 must be mapped to an association in the class diagram between $type(o_1)$ and $type(o_2)$. By the

Figure 2. A sample transformation step using rule payBill .

same mechanism of structural compatibility we ensure that an attribute of an object is declared in the corresponding class, etc.

Rules and transformations. After defining the states of an object system as instances of a type graph, graph transformation rules describe the intended preconditions and effects of operations on states. A *graph transformation rule* $p : L \rightarrow R$ consists of a pair of *TG*-typed instance graphs L, R such that the intersection $L \cap R$ is defined. (This means that, e.g., edges that appear in both L and R are connected to the same vertices in both graphs, or that vertices with the same name have to have the same type, etc.) The left-hand side L represents the pre-conditions of the rule while the right-hand side R describes the post-conditions.

A *graph transformation* from a pre-state G to a post-state H , denoted by $G \xRightarrow{p(o)} H$, is given by a graph homomorphism $o : L \cup R \rightarrow G \cup H$, called *occurrence*, such that:

- $o(L) \subseteq G$ and $o(R) \subseteq H$, i.e. the left-hand side of the rule is embedded into the pre-state and the right-hand side into the post-state; and
- $o(L \setminus R) = G \setminus H$ and $o(R \setminus L) = H \setminus G$, i.e. precisely that part of G is deleted which is matched by elements of L not belonging to R and, symmetrically, that part of H is added which is matched by elements new in R .

Operationally, the application of a graph transformation rule like payBill in Figure 2 is performed in three steps. First, find an occurrence $o|_L$ of the left-hand side L in the current object graph G . Second, remove all the vertices and edges from G which are matched by $L \setminus R$. Make sure that the remaining structure $D := G \setminus o(L \setminus R)$ is still a legal graph, i.e. that no edges are left dangling because of the deletion of their source or target vertices. This is made sure by the *dangling condition* [7] which is checked for a given occurrence before the application of the rule. If the condition is violated, the



application is prohibited. Third, glue D with $R \setminus L$ to obtain the derived graph H . In Figure 2 the occurrence of the rule is designated by the shaded objects and links in the transformation.

Altogether, the static and functional aspects of a model can be formally represented as a *typed graph transformation system* [6] $\mathbf{G} = \langle TG, P, \pi \rangle$ consisting of a type graph TG , a set of rule names P , and a mapping π associating with each rule name $p \in P$ a rule $\pi(p) = L \rightarrow R$ over TG . In this case we write $p : L \rightarrow R \in \mathbf{G}$. The *behavior* of \mathbf{G} is given by the set \mathbf{G}^* of its transformation sequences $G_0 \xRightarrow{p_1(o_1)} \dots \xRightarrow{p_n(o_n)} G_n$, i.e. sequences of consecutive transformation steps in \mathbf{G} .

This idea of a rule-based modeling of behavior by means of graphical pre- and post-conditions can be traced back to several sources. PROGRES [8] provides a database-oriented programming language and environment based on graph transformation. In the object-oriented FUSION method [9] actions are specified by snapshots of the object configuration before and after the operation. CATALYSIS [10] advocates the use of UML collaboration diagrams for this purpose, an approach which has also been adopted and implemented in the FUJABA method and CASE tool [11].

Functional (de)composition. Graph transformation rules like `payBill` in the top of Figure 2 provide a high-level specification of functional requirements in terms of pre- and post-conditions, abstracting from intermediate actions and states. If the more fine-grained structure of an operation is of interest, a rule may be decomposed into more elementary steps. For example, the rule `payBill(b)` may be decomposed sequentially as `payBill(b) = createTransfer(b,t); executeTransfer(t)`, where the relation between the two elementary rules is specified by the abstract parameters `b:Bill` and `t:Transfer`.

The (obvious) semantic consistency condition for this kind of decomposition requires that, for any given graph G , there exists a transformation

$$G \xRightarrow{\text{payBill}(B)} H$$

if and only if there exist a graph X and transformations

$$G \xRightarrow{\text{createTransfer}(B,T)} X \xRightarrow{\text{executeTransfer}(T)} H.$$

Thus, we may think of the composed rule `payBill` as a two-step transaction.

The desired semantic condition can be checked by composing the elementary rules as shown in Figure 3. First, `createTransfer(b,t)` is applied to the left-hand side of `payBill(b)` at the occurrence determined by the parameter `b:Bill`. Then, `executeTransfer(t)` is applied to the resulting graph, this time determining the occurrence through the parameter `t:Transfer`. The result of this second step has to be isomorphic to the right-hand side of the original rule. In this case, it can be shown that the desired consistency condition holds [12].

The decomposition of rules can be used to support a notion of refinement of functional specifications, as in [13] where the relation between module interfaces and their implementations is described with rule expressions similar to the one above. In this paper, it will be crucial to the distribution of functional requirements across different components.

2.2. Architectural dynamics

A quite different interpretation of nodes and edges in a graph is in terms of components and connectors of a software architecture model, see e.g. [14,15]. Here, instance and type graphs model, respectively,

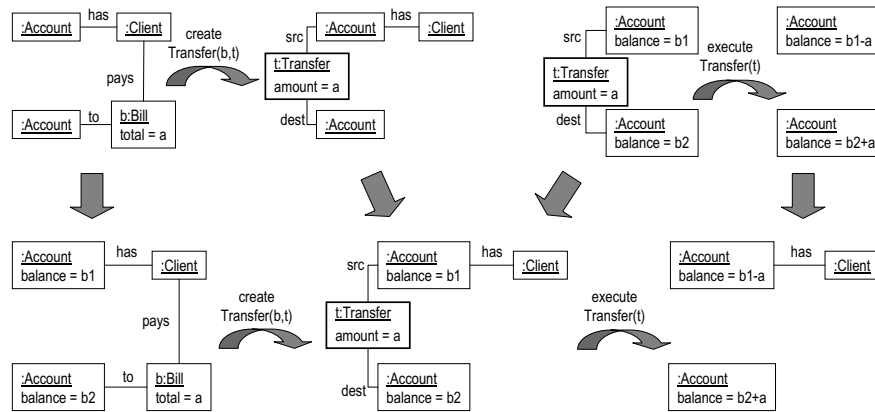


Figure 3. Rule $\text{payBill}(b)$ derived from $\text{createTransfer}(b,t)$; $\text{executeTransfer}(t)$.

configurations and architectural styles, while graph transformation rules specify the reconfiguration of architectures.

Configurations and architectural styles. The idea of using graphs to represent software architectures is already implicit in the box-and-line diagrams drafted by software engineers to visualize the architecture of a system. In order to understand such drawings, it is important to distinguish between the representation of an individual system configuration and a class of such configurations, called an *architectural style* [14].

Representing configurations directly by instance graphs, it is natural to use type graphs (and constraints) as a representation of styles, as exemplified in Figure 4 in the upper left. The example shows an architectural style for distributed banking applications consisting of components such as electronic cashboxes, smartcards, and banking servers communicating through the Internet or card readers. The cardinalities between the **CardReader** and the **CardInt** interfaces model the constraint that, at any given time, only one card can be inserted into a card reader, and a given card can only be in one card reader at a time. A sample configuration of this style is shown in the lower left of Figure 4. It consists of two instances of the banking server component, one cashbox, and one smart card, where only the two banking servers are currently connected.

Besides this declarative approach to the specification of architectural styles, constructive approaches based on graph *grammars* have been proposed. Here the idea is to generate the set of all eligible configurations from a given initial one by means of production rules [16,17].

Reconfiguration. The main benefit of graph transformation for describing software architectures is the ability to model dynamic reconfigurations in an abstract and visual way. Some approaches [16,18,19] assume a global point of view which, in a real system, would correspond to the perspective of a centralized configuration management. Since in a distributed system the existence of such a

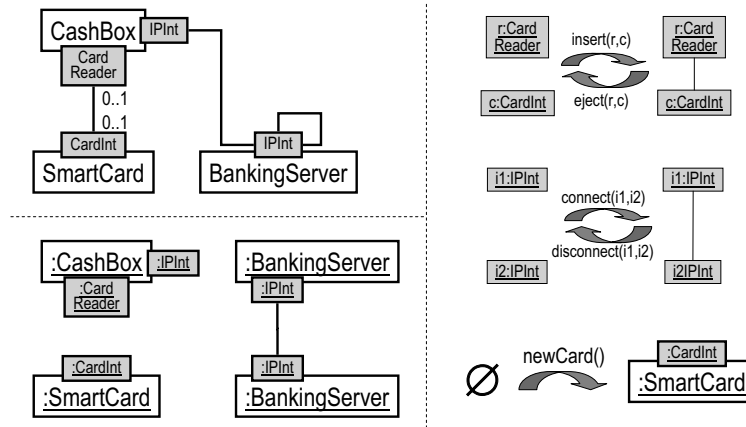


Figure 4. Architecture of a distributed banking system: sample configuration (lower left) typed over an architectural style (upper left) and reconfiguration rules (right).

centralized service cannot be taken for granted, Hirsch *et al.* [17,20] model reconfiguration from the point of view of individual components which synchronize so as to achieve non-local effects.

The rules in the upper right of Figure 4 do not require any synchronization. They model the insertion and ejection of smart cards, the establishment and release of Internet connections, and the release of new cards. Similar rules could have been provided to set up new banking servers or cash boxes.

3. RELATING FUNCTIONAL REQUIREMENTS AND ARCHITECTURAL MODELS

In the previous section, we have shown how to model both functional requirements and architectures by separate graph transformation systems. This section is devoted to the combination of these two views both statically by relating classes to components and dynamically by interleaving computations on objects with reconfiguration steps and communication between components.

3.1. Classes and components

As pointed out in the previous section, the interpretation of vertices and edges is quite different in the functional and the architectural view, i.e. despite using the same formalism the two models are conceptually disjoint. It is the aim of this subsection to make the different interpretations explicit, thus providing the intended separation of concerns and, subsequently, to establish a relation between the two views to describe, e.g., the deployment of classes at components.

The first aim is achieved by defining the abstract syntax of both the functional and the architectural view by means of a *meta model*. Generally speaking, a meta model is a model for a modeling language expressed within a simple subset of the language itself. This technique has become popular with

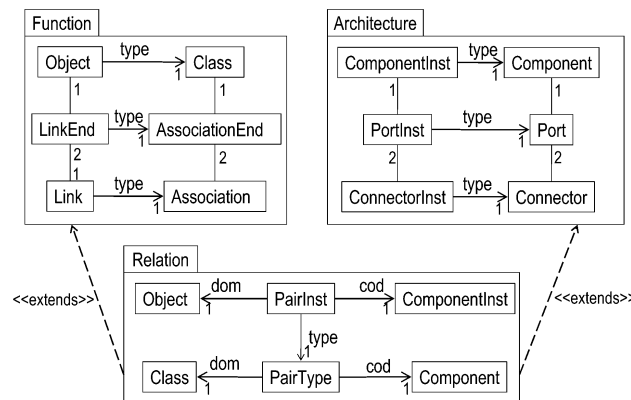


Figure 5. A meta model relating the functional and architectural view.

the UML whose abstract syntax is specified by a subset of UML class diagrams [3]. This subset is determined by the MOF specification [2] which is also referred to as a meta-meta model because it has meta models as instances whose instances, in turn, are models.

In our approach, meta models are type graphs whose instance graphs represent models. That means, the type-instance mapping of typed graphs, which has so far been used to model the relation of objects to their classes and component instances to their components, shall now be reserved for the mapping between a model and its meta model. Therefore, the object-class and component instance-component mappings are defined in the meta model itself.

Consider, for example, the meta model in Figure 5. It consists of three packages, where the top left one specifies the functional view of the system by means of meta classes **Class** and **Association** as well as **Object** and **Link**, etc., whose relation is given by the meta association **type**. Therefore, every instance of this meta model represents a pair of a class diagram and an associated object diagram.

A similar structure is present in the package for the architectural view whose instances represent both a declarative definition of an architectural style (by meta classes **Component**, **Connector**, and **Port** with their meta associations) as well as an individual configuration (by meta classes **ComponentInst**, **ConnectorInst**, and **PortInst** with their meta associations) and their interrelation by the meta association **type**.

This idea of a meta model consisting of different packages for different concerns allows us to separate the sub-models while at the same time retaining a uniform representation where all elements of the same overall model are represented as vertices of the same *abstract syntax graph*, i.e. an instance of the meta model. Based on this uniform representation, the different sub-models can be related by extending the meta model in Figure 5 with a package extending the other two packages to define a relation between the functional and the architectural view. (The same structure could have been used above to define the class-object and the component instance-component relation in a more structured way [21], but this would have complicated the presentation without contributing much to the aim of the

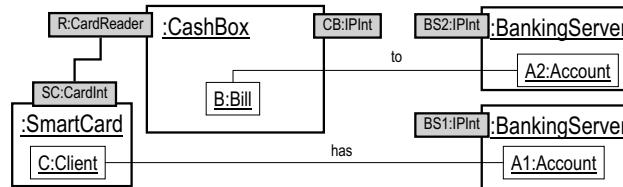


Figure 6. Sample configuration after creation of a bill.

paper.) Such relations consist of pairs of, respectively, objects and component instances or classes and components, where instance-level pairs are associated to type-level pairs by a meta association **type**: to relate an object to a component instance, a corresponding relation between the respective class and component is required.

In our example, the relation between classes and components shall be given by

$\{(\text{Client}, \text{SmartCard}), (\text{Transfer}, \text{CashBox}), (\text{Bill}, \text{CashBox}), (\text{Client}, \text{CashBox}),$
 $(\text{Account}, \text{BankingServer}), (\text{Transfer}, \text{BankingServer})\}$

This information could be given in diagrammatic form, as in UML component diagrams where the relation is expressed by containment or dependencies. Here we simply list the pairs of classes and components. A diagrammatic presentation of the object–component instance relation by means of containment is given in Figure 6, combining information from all three meta model packages in Figure 5.

3.2. Computation and reconfiguration

Based on the uniform representation of object structures and architectures as meta model instances we may now formalize the rules specifying functional requirements and architectural reconfigurations as graph transformation rules typed over the corresponding packages of the meta model. However, this is nothing more than a disjoint union of models, turning two distinct models into yet unrelated sub-models of the same overall model.

To map the functional requirements on a given architecture, we assign responsibilities for operations to components (or sets of components) by specifying the location of the objects in the corresponding rules. Figure 7 shows this step for the `createTransfer` and `executeTransfer` rules. The former is specified to be a local operation of the `CashBox` component. The latter turns out to be an operation which is shared between two instances of the `BankingServer` component, i.e. where the source and the destination accounts of the transfer are located. This operation has to be decomposed into local operations in a further iteration of the model.

The goal is to come up with a model where operations are local, that is, associated with individual components. Such a model is typical at the design stage of development and can be directly related to the implementations of components.

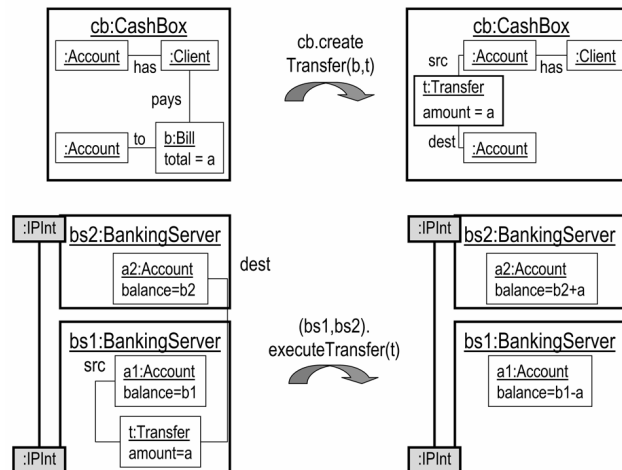


Figure 7. Operation `createTransfer` executed by a `CashBox` component (top) and operation `executeTransfer` jointly performed by two banking servers (bottom).

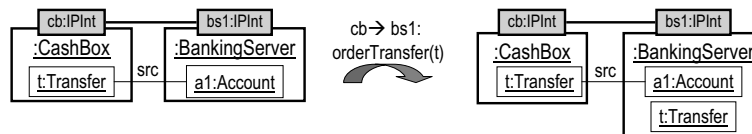


Figure 8. The communication rule modeling the effect of transmitting a `Transfer` object.

A rule specifying a local operation may only access objects located at a single component instance, as in the case of `createTransfer`. To achieve the same overall effect as specified by the pre/post-conditions of a global rule like `payBill`, local operations have to be interleaved with communication rules which have the effect of moving or copying objects between components.

An example of such a rule is shown in Figure 8. It models the transmission of a message `orderTransfer(t)` from a cashbox holding an object `t:Transfer` to the banking server holding the corresponding source account. The effect of this transmission is the replication of the object. We show by using the same object identity `t` that the object is logically shared between the two components.

How to decompose a global rule like `payBill` into a series of local rules, interleaved with communication actions, is by no means straightforward. A useful tool in this task are scenarios expressed by UML sequence diagrams [3] which model the communication between the components required to implement the desired effect. The methodology is similar to a use-case-driven approach where use-cases represent functional requirements in terms of important operations on the system

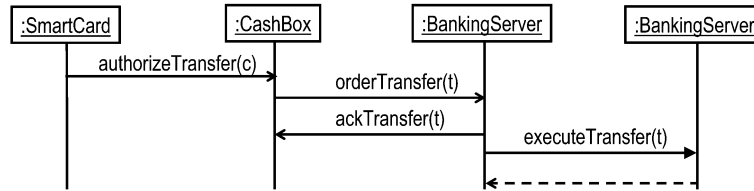


Figure 9. The communication scenario for `payBill`.

level while sequence diagrams refine use-cases by modeling the required (internal) interactions among the system's components [22]. In our example, one possible refinement of the 'use-case' `payBill` could be the scenario in Figure 9.

Sequence diagrams focus on the communication between objects or component instances disregarding local operations and reconfiguration steps. A complete sequence of steps implementing the operation `payBill(B)` on the given architecture is given by

```
payBill(B) = insert(R, SC); authorize(C); eject(R, SC); createTransfer(B, T);
            connect(CB, BS1); orderTransfer(T); ackTransfer(T); disconnect(CB, BS1);
            executeTransfer(T)
```

The sequence diagram in Figure 9 can be obtained from this trace by restricting to *communication steps*, i.e. `authorize(C)`; `orderTransfer(T)`; `ackTransfer(T)`; `executeTransfer(T)`. These are interleaved with *reconfiguration steps* `insert(R, SC)`; `eject(R, SC)`; `connect(CB, BS1)`; `disconnect(CB, BS1)` to establish the required communication links and with *computation steps* `createTransfer(B, T)`; `executeTransfer(T)` providing the actual changes to the objects. Recall that the corresponding rules `createTransfer(b, t)`, `executeTransfer(t)` have been obtained as a decomposition of `payBill(b)`. Their relation with the architectural level is shown in Figure 7. The communication rules `authorizeTransfer(t)`, `orderTransfer(t)`, `ackTransfer(t)` are exemplified by `orderTransfer(t)` in Figure 8 whose purpose is to replicate a `Transfer` object of the cashbox on the banking server.

Notice how the separation between the three meta model packages is reflected in the rules of the model.

- Reconfiguration rules as shown in Figure 4 on the right change the configuration of the system, working entirely on the architectural view.
- Computation rules like `createTransfer` in Figure 7 are responsible for manipulating objects, links, and attribute values. They are typically local (i.e. restricted to one component) and they do not change the architectural view.
- Communication rules as shown in Figure 8 update the relation of objects and component instances without changing the objects themselves.

This separation of concerns allows us to re-map the computations of the system to a modified architectural model, as is demonstrated in the next section, or to map new functional requirements to the same architecture.



Related work. Let us conclude this section by a discussion of related work. The integration of functional and architectural models is implicit in many works on graph transformation for specifying software architectures. In [19] an approach based on two-layered distributed graphs [23,24] is presented. The upper layer represents the network graph of a distributed system whose nodes are attributed with object graphs on the lower layer. Two-level rules are then used to manipulate this structure. This approach is conceptually close to ours, but it does not foresee any means for functional decomposition of rules. This means, it does not account for the development problem inherent in relating functional requirements and architecture, but only for the result of the integration.

Another approach integrating functional and architectural aspects [18] uses the coordination and programming language COMMUNITY to specify computations and graph transformation rules to model architectural reconfiguration. While the use of a programming notation is appropriate at the later design or implementation level, for requirements specification and analysis we prefer visual notation so as to also express the computational aspects.

Both cited approaches are based on a category-theoretic formalization of how the computational and the architectural layer are related. Although mathematically elegant, such a definition is more difficult for non-experts to understand and provides less flexibility, e.g. if another layer shall be added or the conditions for rule applications shall be changed. Here we prefer the meta modeling approach for its conceptual simplicity and flexibility.

3.3. Semantic consistency

The consistency between the different views can be expressed in terms of refinement relations of typed graph transformation systems [25]. The basic concept is a projection of the states of the larger (more refined) system model $\mathbf{G}_2 = \langle TG_2, P_2, \pi_2 \rangle$ to the smaller (more abstract) one $\mathbf{G}_1 = \langle TG_1, P_1, \pi_1 \rangle$, which is induced by an inclusion of type graphs $TG_1 \subseteq TG_2$: given a graph G_2 typed over TG_2 , its projection $G_2|_{TG_1}$ is defined as the largest sub-graph of G_2 typed over TG_1 .

For example, the meta model in Figure 5 defines a projection $G|_{\text{Function}}$ of complete system states G as shown in Figure 6 to the functional view by hiding all component, connector, and port instances, retaining only the objects and links. Similarly, we could define a projection $G|_{\text{Architecture}}$ of G to the architectural view.

A *refinement relation* $\text{ref} : \mathbf{G}_1 \rightarrow \mathbf{G}_2$ requires an inclusion of the type graphs $TG_1 \subseteq TG_2$ and defines, for each rule $p_1 : L_1 \rightarrow L_2 \in \mathbf{G}_1$, a non-empty set $\text{ref}(p_1) \subseteq \mathbf{G}_2^*$ of transformation sequences in \mathbf{G}_2 such that for every such sequence $s_2 : G_2 \Rightarrow H_2$ we have that $s_2|_{TG_1} = K_2|_{TG_1} \rightarrow Q_2|_{TG_1}$ equals $\pi(p_1) = L_1 \rightarrow L_2$. This means, by projecting the given and the derived graph of the sequence s_2 to the smaller type graph TG_1 , we obtain the left- and the right-hand side of the rule p_1 , respectively. In this case, we say that the sequence s_2 *realizes* the rule p_1 . For example, the transformation sequence in Section 3.2 above realizes the rule `payBill`.

In object-oriented modeling it is common to specify the refinement of system-level operations like `payBill` by means of UML sequences or collaboration diagrams [3]. As shown in [26] these *interaction diagrams* can be regarded as (equivalence classes of) graph transformation sequences. In this context, the formal notion of refinement introduced above ensures the *partial correctness* of the interaction diagram with respect to the pre- and post-conditions expressed by the abstract rule: if the specified sequence of operations is applied in a state satisfying the pre-conditions, its final state will satisfy the post-condition [25].



The corresponding notion of *total correctness* requires that the refinement of an abstract rule p_1 is complete in the following sense: for every state G'_2 in the larger model whose projection allows a transformation $G'_2|_{TG_1} \xRightarrow{p_1(o_1)} H'_1$ in \mathbf{G}_1 , there exists a transformation sequence $s'_2 : G'_2 \Rightarrow H'_2$ in \mathbf{G}_2 such that $H'_2|_{TG_1} = H'_1$. This means, whenever the precondition of an abstract operation p_1 is fulfilled, at least one of its realizations is executable and produces the required effect.

By virtue of partial correctness, the last requirement is satisfied if the sequence s'_2 is obtained by executing a sequence $s_2 : G_2 \Rightarrow H_2 \in \text{ref}(p_1)$ in a larger context. The existence of such a transformation sequence s'_2 requires an embedding of G_2 in G'_2 which satisfies the gluing condition [7]. In our application, satisfaction of the gluing condition is ensured if the transformation sequences in $\text{ref}(p_1)$ never delete a component which could have additional connectors in a larger context, because this would prohibit their execution. Such a condition can be checked statically if the architectural style specifies a limit on the number of connectors that may be attached to a component. In a similar way, the existence of the embedding $G_2 \subseteq G'_2$ itself may depend on constraints on configurations.

However, as architectural models in practice are often incomplete sketches serving as documentation and communication means rather than as formal specifications, we may lack the information to decide if the condition of total correctness is indeed satisfied. In order to formalize at least the *degree* to which an abstract rule p_1 is realized in the refined system \mathbf{G}_2 , we define the *support for p_1 in \mathbf{G}_2* as the set of all TG_2 -typed graphs G'_2 which allow for the execution of a realization $s_2 \in \text{ref}(p_1)$.

4. EVOLUTION OF FUNCTIONAL AND ARCHITECTURAL MODELS

As pointed out in the introduction, one benefit of separating architectural and computational aspects as much as possible is the possibility of reusing one of the two models if the other one has to be changed due to an evolution of the requirements. In this section, we shall consider the case where the same functionality has to be achieved on the basis of a new architecture.

Suppose that the business we have been modeling is now offering its service via the Internet as an online shop. In this case, the architecture described in Figure 4 may be replaced by the one given in Figure 10. The essential differences are a **HomeBankingClient** replacing the **SmartCard** and an **OnlineShop** component instead of the **CashBox**.

Reconfiguration rules model the installment of a new **HomeBankingClient** instance as well as the establishment and release of Internet connections which we do not repeat here from Figure 4. Our sample configuration contains two banking servers, one online shop, and one home banking client to start with.

The functional specification remains the same as given by the class diagram in Figure 1, the rule **payBill**, and its decomposition in Figure 3. If we assume the architectural model as given, the evolution problem consists of re-mapping the functional requirements to the modified architecture. The first step in this activity is to detect those elements of the functional model where, due to the removal of components, a new mapping is required.

In the static case, candidate classes for re-deployment are designated by the ‘dangling pairs’ of the relation, i.e. those instances of meta class **PairType** in the **Relation** package that have no **cod** element in the new architectural model. These are the classes **Client**, **Transfer**, and **Bill** previously supported by the components **SmartCard** and **CashBox**. Thus, while preserving the connection with

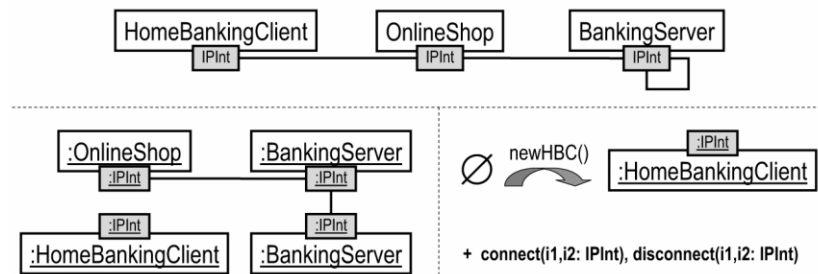


Figure 10. A distributed banking system with online shop: sample configuration (lower left) typed over an architectural style (top) and reconfiguration rules (lower right).

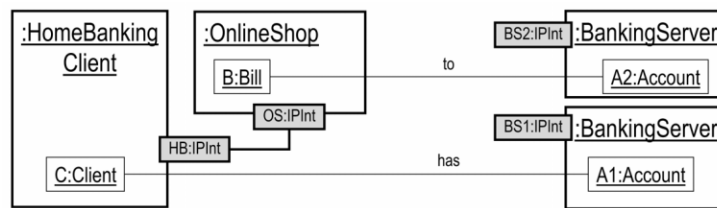


Figure 11. Sample configuration after creation of a bill.



Figure 12. Migration of states from the old to the new model.

BankingServer, the new relation has to provide new pairs for these ‘orphan classes’:

{(Bill, OnlineShop), (Client, OnlineShop), (Account, HomeBankingClient),
 (Bill, HomeBankingClient), (Client, HomeBankingClient), (Transfer, HomeBankingClient),
 (Account, BankingServer), (Transfer, BankingServer)}

A system state for the new integrated model is visualized in Figure 11. It is obtained from the state in Figure 6 by transferring the contents of the CashBox to the OnlineShop and the contents of the SmartCard to the HomeBankingClient. Such systematic transfer of states from an old to a new model can be specified by transformation rules at the meta-level [27,28]. In Figure 12 we exemplify the idea

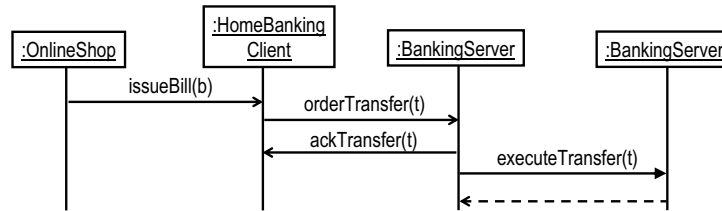


Figure 13. The communication scenario for `payBill` with an online shop.

by a rule replacing `CashBox` by `OnlineShop` instances and transferring their contained objects. In general, we assume a mapping mig from graphs typed over the old model's type graph TG_2 to the new one TG'_2 . In [27] such a rule-based translation of instance graphs is used to support an automatic evolution of rules and transformation sequences. That means, if the evolution consists in local changes to the architecture only, the realizations of functional rules can be automatically transferred.

More often, however, we encounter a situation where changes are driven by new scenarios, like the one in Figure 13, modeling a different realization of the operation `payBill`. This requires the replacement of some of the rules of the original system entirely. Here, the idea of 'dangling pairs' can be applied again to find out the rules of the old model which have to be replaced in order to repair syntactic inconsistencies caused by the evolution of the architectural model. In our example, all operations except for `executeTransfer`, `connect`, and `disconnect` are effected by the change.

The new realization of `payBill` is given by the sequence `connect(OS,HB)`; `issueBill(B)`; `disconnect(OS,HB)`; `createTransfer(B,T)`; `connect(HB,BS1)`; `orderTransfer(T)`; `ackTransfer(T)`; `disconnect(HB,BS1)`; `executeTransfer(T)`. The corresponding interaction between the components is modeled by the scenario in Figure 13. Again, the sequence diagram is obtained from the trace by projection to communication steps. The communication rule `issueBill` is similar to `orderTransfer` in Figure 8.

Summarizing in terms of the meta model, the evolution was driven by a change in the instances of the `Architecture` package. The new model reuses the instances of the `Functional` package, while the instances of the `Relation` package are partly updated as a consequence of the change. Which parts of the package are updated is dictated by the syntactic inconsistencies caused by the removal of instances of meta classes of the `Architecture` package.

Beside the syntactic conditions represented by the existence of a well-defined refinement relation, we may impose semantic requirements on the new model. In relation to the old model, it is usual to require that the new model is an improvement with respect to the support for a given abstract operation p_1 ; formally, if S_2 denotes the support for p_1 in G_2 and S'_2 is the support for p_1 in G'_2 , the support is improved if $mig(S_2) \subseteq S'_2$. (Here we use the migration function from TG_2 to TG'_2 assumed above.) Such a semantic requirement could be validated by testing the new model with test cases generated by executing the old model.

Naturally, there is no guarantee that a functional specification can be implemented on a given architecture. Therefore, the complete reuse of the functional requirement specification in the above



example might be considered a lucky case. Yet, to find out whether a given architecture is suitable or not to support a certain function is one important task of such models. That is, if the above approach fails in a real system model, either the architecture or the functional requirements have to be revised.

5. CONCLUSION

The contribution of this paper can be summarized as follows. Building on existing approaches to the modeling of functional requirements and software architectures by means of graph transformation, we have shown how to relate these two views in order to arrive at a consistent overall model. Syntactically, this integration is achieved by a meta model providing separate packages for the functional and the architectural view and a third package specifying the relation between these views.

The behavioral consistency of these views is achieved by decomposing the rules specifying global functional requirements into more elementary ones associated with individual components, and interleaving these rules with communication and reconfiguration rules to implement the required overall effect. Due to the separation of functional and architectural concerns into related, yet disjoint views, evolution is supported because changes in one view do not necessarily imply changes in the other view. On the behavioral side, this is reflected by the strict separation between computation, reconfiguration, and communication. This was demonstrated by means of an example.

One important concern which has been left out of the discussion is the coordination or control aspect. This could be specified by statechart diagrams associated to components or connectors which guide the application of computation, communication, and/or reconfiguration operations.

The overall presentation has not been tailored towards a specific component model or communication infrastructure to make it applicable to a wide range of systems. However, the approach can be specialized by defining additional constraints, e.g. on the relation of objects and component instances, to reflect more closely the properties of a particular platform.

REFERENCES

1. Finkelstein A, Kramer J, Nuseibeh B, Goedicke M, Finkelstein L. Viewpoints: A framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering* 1992; **2**(1):31–58.
2. Object Management Group. Meta object facility (MOF) specification. <http://www.omg.org> [September 1999].
3. Object Management Group. UML specification version 1.4. <http://www.celigent.com/omg/umlrtf/> [2001].
4. Hausmann JH, Heckel R, Taentzer G. Detecting conflicting functional requirements in a use case driven approach: A static analysis technique based on graph transformation. *Proceedings of the International Conference on Software Engineering (ICSE'2002)*. ACM/IEEE Computer Society Press, 2002.
5. Löwe M, Korff M, Wagner A. An algebraic framework for the transformation of attributed graphs. *Term Graph Rewriting: Theory and Practice*, ch. 14, Sleep MR, Plasmeijer MJ, van Eekelen MC (eds.). Wiley: Chichester, 1993; 185–199.
6. Corradini A, Montanari U, Rossi F. Graph processes. *Fundamenta Informaticae* 1996; **26**(3,4):241–266.
7. Ehrig H, Pfender M, Schneider HJ. Graph grammars: An algebraic approach. *14th Annual IEEE Symposium on Switching and Automata Theory*. IEEE Press, 1973; 167–180.
8. Schürr A, Winter AJ, Zündorf A. The PROGRES approach: Language and environment. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages, and Tools*, Ehrig H, Engels G, Kreowski H-J, Rozenberg G (eds.). World Scientific: Singapore, 1999; 487–550.
9. Coleman D, Arnold P, Bodof S, Dollin C, Gilchrist H, Hayes F, Jeremes P. *Object Oriented Development, The Fusion Method*. Prentice-Hall: Englewood Cliffs NJ, 1994.



10. D'Souza D, Wills A. *Components and Frameworks with UML: The Catalysis Approach*. Addison-Wesley: Reading MA, 1998.
11. Köhler HJ, Nickel U, Niere J, Zündorf A. Integrating UML diagrams for production control systems. *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*. ACM Press, 2000.
12. Corradini A, Montanari U, Rossi F, Ehrig H, Heckel R, Löwe M. Algebraic approaches to graph transformation, Part I: Basic concepts and double pushout approach. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, Rozenberg G (ed.). World Scientific: Singapore, 1997; 163–245.
13. Große-Rhode S, Parisi-Presicce F, Simeoni M. Refinement of graph transformation systems via rule expressions. *Proceedings of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT'98)*, Paderborn, November 1998 (*Lecture Notes in Computer Science*, vol. 1764), Ehrig H, Engels G, Kreowski H-J, Rozenberg G (eds.). Springer: Berlin, 2000; 368–382.
14. Shaw M, Garlan D. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall: Englewood Cliffs NJ, 1996.
15. Kramer J, Magee J. Pulling Together. *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)*. ACM Press, 1997.
16. Le Métayer D. Software architecture styles as graph grammars. *ACM Software Engineering Notes*, 1996; 15–23.
17. Hirsch D, Inverardi P, Montanari U. Modeling software architectures and styles with graph grammars and constraint solving. *Proceedings of the First Working IFIP Conference on Software Architecture*, San Antonio TX, E.E.U.U., February 1999.
18. Wermelinger M, Fiadero JL. A graph transformation approach to software architecture reconfiguration. *Science of Computer Programming* 2002; **44**:133–155.
19. Taentzer G, Goedicke M, Meyer T. Dynamic change management by distributed graph transformation: Towards configurable distributed systems. *Proceedings TAGT'98 (Lecture Notes in Computer Science*, vol. 1764), Ehrig H, Engels G, Kreowski H-J, Rozenberg G (eds.). Springer: Berlin, 2000; 177–190.
20. Hirsch D, Montanari M. Synchronized hyperedge replacement with name mobility. *Proceedings of CONCUR 2001*, Aarhus, Denmark (*Lecture Notes in Computer Science*, vol. 2154). Springer: Berlin, 2001; 121–136.
21. Kent A, Akehurst D. A relational approach to defining transformations in a metamodel. *Proceedings of UML 2002*, Dresden, Germany (*Lecture Notes in Computer Science*). Springer: Berlin, 2002. To appear.
22. Jacobson I, Booch G, Rumbaugh J. *The Unified Software Development Process*. Addison-Wesley: Reading MA, 1999.
23. Taentzer G. Parallel and distributed graph transformation: Formal description and application to communication-based systems. *PhD Thesis*, TU Berlin, 1996.
24. Taentzer G, Fischer I, Koch M, Volle V. Distributed graph transformation with application to visual design of distributed systems. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3: Concurrency and Distribution*, Ehrig H, Kreowski H-J, Montanari U, Rozenberg G (eds.). World Scientific: Singapore, 1999.
25. Heckel R, Corradini A, Ehrig H, Löwe M. Horizontal and vertical structuring of typed graph transformation systems. *Mathematical Structures in Computer Science* 1996; **6**(6):613–648.
26. Heckel R, Sauer St. Strengthening UML collaboration diagrams by state transformations. *Proceedings of Fundamental Approaches to Software Engineering (FASE'2001)*, Genova, Italy (*Lecture Notes in Computer Science*, vol. 2185), Hußmann H (ed.). Springer: Berlin, 2001.
27. Löwe M. Evolution patterns. *Postdoctoral Thesis, Technical Report 98-4*, Department of Computer Science, Technical University of Berlin, 1997.
28. Jahnke J, Zündorf A. Using graph grammars for building the VARLET database reverse engineering environment. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages, and Tools*, Ehrig H, Engels G, Kreowski H-J, Rozenberg G (eds.). World Scientific: Singapore, 1999; 267–286.



AUTHORS' BIOGRAPHIES

Reiko Heckel has been assistant professor at the University of Paderborn, Germany, since 1998. He studied Computer Science at the Technical Universities of Dresden and Berlin and received his PhD from the Technical University of Berlin in 1998. His research interests are the use of graph transformation in software engineering, including the development of relevant theory like, structuring and modularity concepts, concurrency theory, graph-based temporal logic, etc., and the application of this theory to the modeling of object-oriented and agent-based systems, and to the semantics of visual modeling languages.



Gregor Engels has been a full professor for Data Base and Information Systems at the University of Paderborn, Germany, since 1998. He studied Computer Science and Mathematics at Dortmund University and received his PhD from the University of Osnabrück in 1986. Between 1991 and 1997 he held the Chair of Software Engineering and Information Systems at Leiden University, The Netherlands. Gregor Engels is active in the field of object-oriented modeling techniques and applications thereof to the development of multi-media, real-time, and embedded systems and has considerable expertise in graph transformation as well as its application to visual languages and system modeling. As of 2002 he is the coordinator of the European Research Training Network *SegraVis* on Syntactic and Semantic Integration of Visual Modeling Techniques.