A Combined Reference Model- and View-Based Approach to System Specification*

Gregor Engels[1], Reiko Heckel[2], Gabi Taentzer[2], Hartmut Ehrig[2]

[1] Leiden University, Dept. of Computer Science, P.O. Box 9512, NL-2300 RA Leiden, The Netherlands
engels@wi.leidenuniv.nl
[2] Technical University of Berlin, Dept. of Computer Science, Franklinstrasse 28/29, D-10587 Berlin, Germany
{ehrig, reiko, gabi}@cs.tu-berlin.de

Abstract. The idea of a combined reference model- and view-based specification approach has been proposed recently in the software engineering community. In this paper we present a specification technique based on graph transformations which supports such a development approach. The use of graphs and graph transformations supports an intuitive understanding and an integration of static and dynamic aspects on a well-defined semantical base. On this background, formal notions of view and view relation are developed and the behaviour of views is described by a loose semantics. The integration of two views derived from a common reference model is done in two steps. First, dependencies between the views which are not given by the reference model are determined, and the reference model has to be extended appropriately. This is the task of a model manager. If the two views and the reference model are consistent, the actual view integration can be performed automatically. For the case of more than two views more general scenarios are developed and discussed. All concepts and results are illustrated at the well-known example of a banking system.

keywords specification language, view, viewpoint, view integration, software process, graph transformation systems

1 Introduction

The most challenging issue of software engineering still is the question how to master the complexity of the development of large software systems. Currently, a variety of approaches try to solve certain aspects of this problem.

Many of them use graphical representations like entity-relationship diagrams, Statecharts, dataflow diagrams, etc. in order to specify various aspects of the system. In particular, within object-oriented modelling and design such visual techniques form an important part of the overall methodology.

Another important approach is to reuse well-established pieces of specifications, documentations, and/or software, while developing a new system. While in the beginning the reuse idea was restricted to often needed classes, in the meantime it has become clear that reuse should be tackled on a much greater scale by specialising integrated networks of classes, so-called frameworks [LRP+96]. Thus, a major research as well as development field is currently the definition of frameworks for various application domains, like reservation systems, banking systems, or syntax-directed editors.

* Research partially supported by the German Research Council (DFG), the TMR network GETGRATS, and the ESPRIT Basic Research Working Group APPLIGRAPH

A third approach is based on the observation that, due to the size and the diversity of the planned software system, teams of concurrently working application engineers are needed for the realisation of a software system. For instance, during the requirements specification phase, a team of application engineers with different skills and backgrounds is splitted into subgroups. Each subgroup specifies only that aspect of an interactive software system, which is later seen and used by a certain type of user (role). Thus, modularisation concepts are required, which allow to compose a complete and consistent specification out of possibly overlapping pieces.

In the data base world, but also in the field of software and requirements engineering, one way to obtain this modularisation is the concept of views and view integration. In the data base world it is standard to distinguish between a conceptual model and several external models, which are considered to be individual views of the data base. Each view is a restriction of the conceptual model - the total community user view - to just that portion of interest to that particular user (cf. [Dat79]).

Views may be defined in two different ways. First, by using features of a corresponding query language, *user views* are defined on top of an existing database scheme. Instances of a user view are derived from the instances of the complete database. Contrarily, *design views* are developed as a starting point to define different perspectives of a system. In this case, an often tedious integration step is required to resolve inconsistencies between the different design views in order to reach an integrated overall scheme.

In the software engineering field, the view-oriented approach is known by the notion of viewpoints (cf. [FKN+92]). In contrast to the view integration approach, here is no common integrated model intended. The basic idea is to monitor the relationships between different viewpoints, to detect inconsistencies and to resolve them by interactive support of the user. Relationships between different viewpoints are inferred by the use of common names.

This implies that the different application engineers agree on a certain vocabulary for a specific problem domain before they start to develop their own viewpoint. As all notions within a problem domain are somehow related, a more suited starting point than a long list of notions is a so-called reference model for a problem domain, where basic notions and their interrelations are fixed. This idea of a combined reference model- and view-based specification approach was especially proposed by B. Balzer during his keynote speech at SP'96 (Software Process) (cf. [Bal96]).

The goal of this paper is to present an approach, which combines the advantages of the above approaches. The basic idea is to start with a reference model (or framework) for a certain application domain, to refine this reference model in a second step by different design views on the system to be developed, and to integrate these design views to the system model. Using concepts and results from the theory of typed graph transformation systems [CEL+96,HCEL96,Rib96] we give precise definitions for views and view relations and support the integration of views by a general automatic construction.

We explain this approach informally in Section 2. In Section 3, 4, and 5 we present the formal base of our approach together with illustrating examples. The basic notions of graphs and graph transformation for the modelling of static and dynamic aspects of software systems are presented in Section 3. We distinguish between the classical semantics of graph transformation systems defined by derivation sequences and a new kind of semantics, called loose semantics [?], based on transition sequences. While the classical semantics can be considered as the closed behaviour of a fully specified system, the loose semantics formalises the open behaviour of a system that is embedded in a not completely specified environment. This reflects the behaviour of system components corresponding to specific views. In Section 4 we define view relations. They are used in Section 5 for presenting a construction which supports view integration. Finally in Section 6 we summarise the main ideas and discuss some remaining open problems.

2 Concept of Views and View Integration Using Graph Transformations

Graph grammars and transformations have been introduced as a generalisation of Chomsky grammars on one hand and of term rewriting systems on the other hand about 25 years ago. Meanwhile there is a well-established theory of graph transformations (see e.g., [Roz97]) which has a number of applications to system modelling and software engineering (cf. [Nag96,Zam96,AE96,SWZ95]) based on concrete specification languages and supporting tools (cf. [Sch91,LB93]). The main idea of our specification approach is to model object structures and their interrelationships by graphs and modifying operations by graph transformations. In particular our approach is based on typed graph transformation systems [CEL+96,HCEL96,Rib96] which allow to define a set of graphs by a type (scheme) graph together with type-consistent operations on these graphs. Compared to most of the currently popular object-oriented modelling techniques, typed graph transformation systems really support an integrated modelling of static and dynamic aspects, which goes much further than an integration merely based on the use of common names. Nevertheless, it shall be noted that graph transformations in its pure form are not object-oriented. There are, however, class-based extensions (see e.g., [Tae96,Wag97]).

This paper applies typed graph transformation systems for defining the concept of a view that models a certain aspect of the complete system. Thus, a view specifies only partially the structure of the system's state and analogously only partially, what the effect of an operation is. It may be that a view operation, being executed on the system's state, has to be concurrently coupled with operations of other views to ensure a consistent system's state transition. Thus, a view specifies only what at least has to happen on a system's state. In this sense, the semantics of a view can only be a loose one, in contrast to the semantics of the complete model.

The overall specification approach can be sketched as follows (cf. Figure 1). Starting with a common reference model, each application engineer develops his own viewpoint by extending and refining the reference model appropriately. In the case that different names for the same concept have been used, a renaming step has to be executed by the application engineer. We will explain later that technically spoken, a (partial) specification is called a view on another specification, if a renamed version of the first can be embedded into the

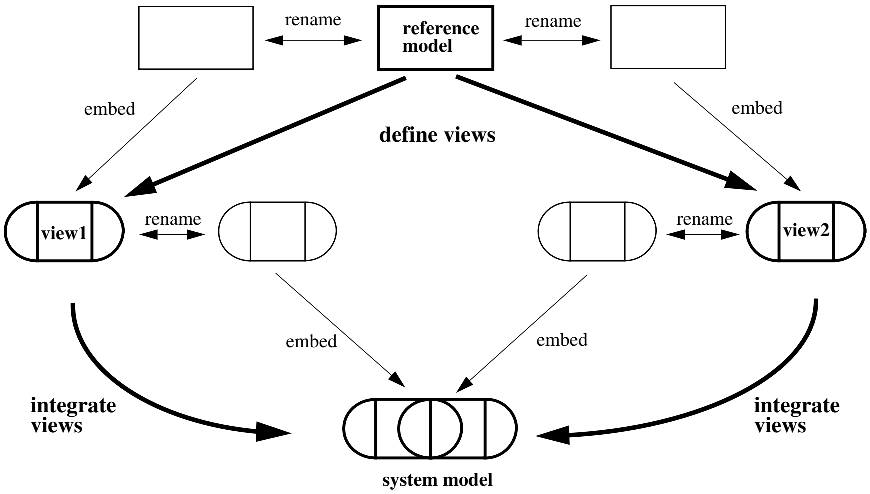


Fig. 1. A view-oriented approach to system modelling.

second. The integration of views is done in two steps. First, new dependencies between views (i.e., those which are not already given by the reference model) have to be determined by a *model manager*, a dedicated developer who is responsible for the consistency of the different views. His task is simplified considerably by the fact that domain-specific notions and operations are already shared through the common reference model. Hence, the new dependencies apply mainly to notions specific to the particular problem. In order to reestablish consistency, the original reference model is extended. In a following step, the actual integration of views can be done automatically.

The assumption of a common reference model is in line with above mentioned current approaches in the object-oriented world, where also reference models in the form of domain-specific frameworks are regarded as the desired starting point for any new software development project. But in addition and in contrast to such a framework-based specification approach, we allow that the framework (or reference model) is specialised concurrently by several views.

Following such a view-based specification approach, various forms of possible inconsistencies can be distinguished. We only discuss two simple examples related to the treatment of names.

(i) The same concept, e.g. operation, is specified in two different views by using different names.
(ii) The same names are used in two different views denoting semantically different concepts.

In particular, the first form (i) of inconsistency has extensively been investigated in database research, as it is one of the problems which have to be solved during scheme integration (cf. [BLN86]). Instead of trying to identify dependencies between different names, we start with a common reference model of names and their interrelations. In the case that different views want to share the same name for the same concept and this name is not yet contained in the reference model, the reference model has to be extended. In this situation, the model manager mediates between the different view designers and extends the reference model appropriately.

In the second case (ii), two solutions are possible: The two names are kept distinct within the overall specification (for instance, by qualifying them with the view name) or the two names are even rejected by the model manager.

While the above explained two forms of inconsistencies relate to static inconsistencies between specification documents, a third form of inconsistency may occur during executing (or enacting) the system.

(iii) Execution of a view operation violates the constraints defined by another view.

This means that two different views overlap in their specification of the desired system's behaviour. In this case, the two views have to be synchronised to achieve a consistent system's behaviour specification.

Different solutions for (iii) can be distinguished. The viewpoint approach (cf. [FKN+92]) follows an algorithmic approach by checking the effect of operations and triggering update operations to end in a consistent result state. Other specification approaches, like e.g. Z (cf. [Jac95]), follow a descriptive approach, where the application engineer has to integrate different view specifications in an overall specification by additional inter-view constraints. In our approach, we follow a constructive approach, where different views are automatically integrated. This means that two operations from different views are merged into one operation in the resulting overall system specification. The common underlying reference model indicates and identifies the overlapping part.

We illustrate our specification approach by the often used example of a banking system. The reference model consists of basic notions within the banking world like customer, account, or transfer, and the typical relationships between them. Two design views are specified, one by an application engineer, who models the functionality as it is seen by a customer, and one by an application engineer, who models what is happening inside the banking system, as it is seen by a clerk. During the specification of these views the model manager extends the reference model by an operation for opening new accounts that represents a joint activity of the customer's and the clerk's view. These two views, the reference model, and the automatic integration are presented in the following sections of this paper.

The situation is becoming more complicated in the case that more than two views are involved in the integration process. Then, additional so-called *abstract views* have to be defined by the model manager. This prevents that an agreement on common names between two views is propagated to all other views. It enables, for instance, that application engineers working on the user interface may agree on their own abstract view, i.e., extended reference model, which differs from the abstract view of application engineers who are designing the database part of a software system. Figure 2 sketches this situation. A more detailed discussion follows at the end of Section 5.

3 Graph Transformation for System Modelling

In this section, we explain how rule-based graph transformations can be used to model the static and dynamic aspects of software systems in a formal and integrated way. The main concepts are illustrated by a small banking example.

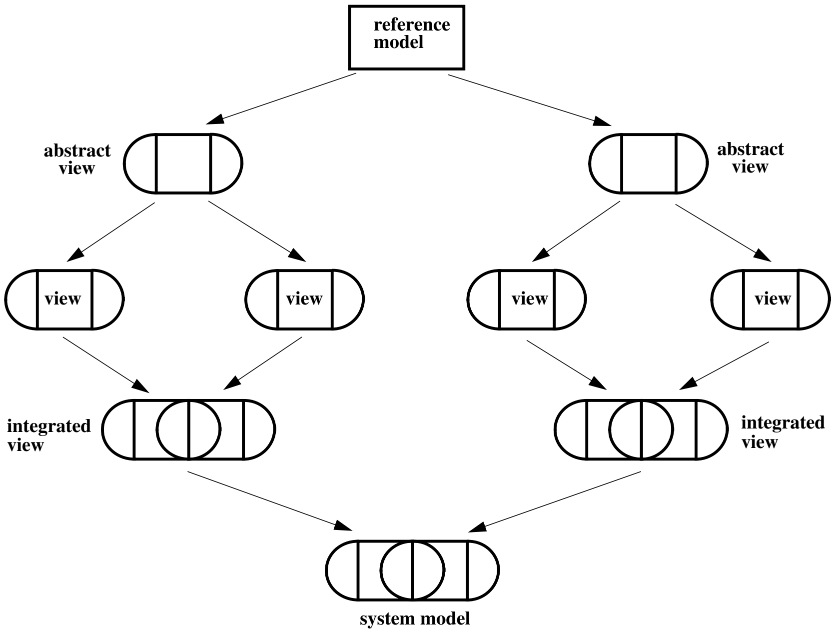


Fig. 2. Additional abstract views in case of multi-view integration.

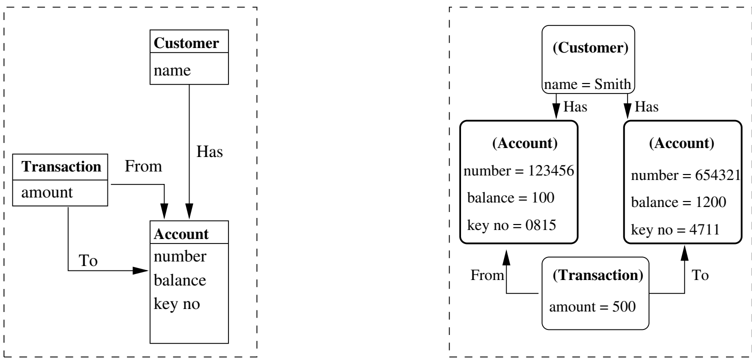


Fig. 3. Example of scheme and instance graphs

Graphs. Graphs and diagrams are often used in software engineering for visualising complex structures. We only mention Entity-Relationship (ER) diagrams and instances in data modelling or class and object diagrams in object-oriented design. 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.

Both in ER modelling and OO design graphs occur on two levels, as *scheme graphs* (ER diagram, class diagram) and their *instance graphs* (ER instance, object diagram). Scheme graphs impose structural constraints on its instances by requiring that each instance can be mapped to its scheme in a structure-preserving way. This mapping also provides vertices and edges of the instance graph with their *types*, i.e., the vertices and edges of the scheme graph. Scheme and instance graphs may contain textual or numerical information like object and relationship names or attribute values associated to vertices and edges.

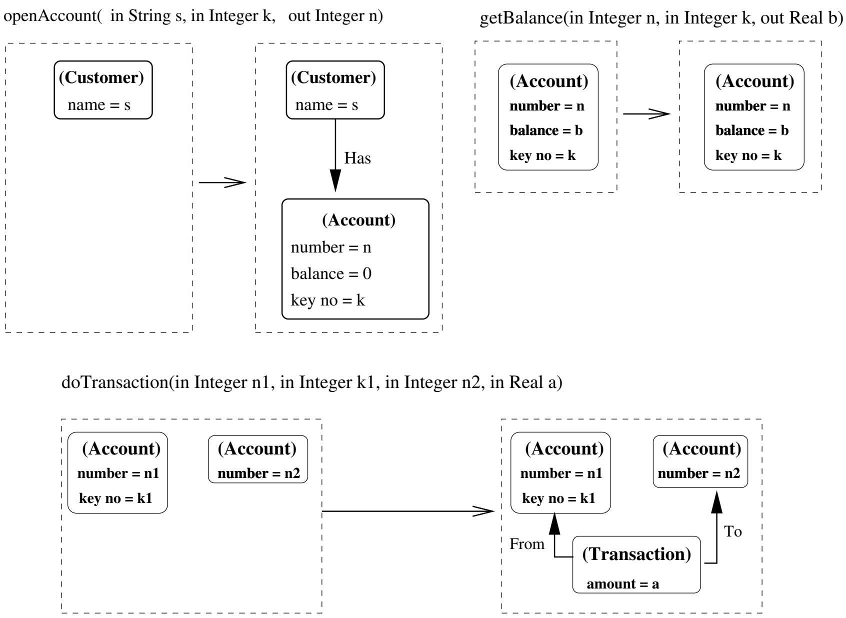


Fig. 4. Graph transformation rules for opening a new account, getting the balance, and starting a transfer transaction.

Example. A sample pair of scheme and instance graphs is shown in Figure 3. The scheme graph on the left contains the main object and relationship types. Object types are *Customer, Account*, and *Transaction*. Customers have a name and are linked by a *Has* relationship to their accounts. Accounts have an account number for identification, a key number for authorised access and, of course, a balance. Transactions are requests for transferring money between accounts. On the right side of Figure 3, an instance of this scheme is shown. It represents a toy state of the banking system where a customer holds two accounts with an ongoing transaction. $\triangle$

Rule-Based Graph Transformation. State changing operations on graphs are modelled by graph transformations which are specified by *graph transformation rules* $r : L \to R$. They consist of a rule name r and two instance graphs L and R, called *left-* and *right-hand side*, which represent a part of the system's state before and after the operation, that is, the pre– and postcondition of the operation. We assume that the intersection $L \cap R$ is a graph, called the *interface* of r. It contains those items that are read but not deleted by the operation.

Example. The upper left rule in Figure 4 specifies the customer's operation of opening a new account. It requires the customer's name and a key number as input. Then, a *Customer* object with this name is selected from the current state, and a new *Account* object is created together with a *Has* relationship. Balance and key number are set, and an account number is chosen and passed to the customer as output. The *getBalance* rule in the right reads the balance of an account, that is, it specifies a query to the graph representing the current state. The rule *doTransaction* below starts a transfer transaction (that has to be completed later). $\triangle$

More generally, a derivation step $G \stackrel{r}{\Longrightarrow} H$ from G to H using a rule $r : L \to R$ requires that (a renaming of) L occurs as a subgraph in G. Then, $L \setminus R$ (which consists of all nodes and edges of L not belonging to R) is removed from G, and $R \setminus L$ is added to the result. This leads to the derived graph H which contains a renaming of R as a subgraph.[1] Hence we denote by $delete(r)$ the part $L \setminus R$ and by $add(r)$ the part $R \setminus L$. The application deletes and creates exactly what is specified by the rule, i.e., there is an implicit *frame condition* stating that everything that is not rewritten explicitly by the rule is left unchanged.

The same rule can also be interpreted in a more loose way. In this case, it specifies only some part or local view of the changes that affect the current state. Since we are interested in the behaviour of views, we introduce the notion of *graph transition* by dropping the above mentioned frame condition. Like a derivation step, a graph transition $G \stackrel{r}{\leadsto} H$ from G to H via r requires that L occurs in G. Then, *at least* $delete(r)$ is removed from G and *at least* $add(r)$ is added, but there may be unspecified deletion and addition as well.

For modelling state transitions that are entirely caused by the environment we introduce ϵ-transitions, i.e., transitions using the empty production $\epsilon : \emptyset \to \emptyset$. Such a production specifies no effect, that is, $delete(\epsilon) = add(\epsilon) = \emptyset$. A transition via ϵ allows any change to the current state.

Graph Transformation Systems. A graph transformation system $\mathbf{G} = \langle SG, N, \rho \rangle$ consists of a scheme graph SG, a set of rule names N and a mapping ρ providing for each rule name r a rule $L \to R$ where L and R are instance graphs of SG (their nodes and edges are typed over SG).

The rules of a graph transformation system can be applied sequentially or in parallel. The parallel application of two rules r_1 and r_2 is described by applying the so-called *parallel rule* $r_1 + r_2 : (L_1 + L_2 \to R_1 + R_2)$ constructed as the disjoint union of the two rules. The occurrences of L_1 and L_2 in the given graph may also overlap, but only in items that are not deleted by the rules. If more than two rules shall be applied in parallel, this construction may be iterated. Finite or infinite *derivation sequences*, i.e., sequences of (sequential or parallel) steps form the classical semantics of a graph transformation system. They correspond to the *closed behaviour* of a non-reactive and fully specified system. The

---

[1] Formally, the occurrences of L in G and R in H are represented as graph homomorphism $m : L \to G$ and $m^* : R \to H$, called *match* and *comatch* respectively, that allow the renaming of L and R. Derivation steps are defined according to the so-called double-pushout approach [?].

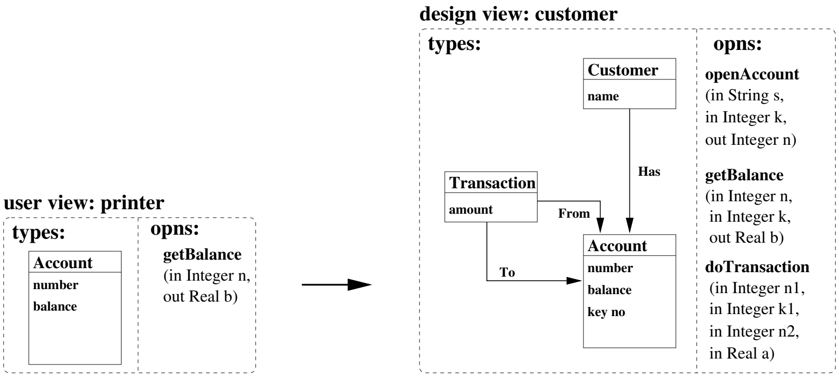


Fig. 5. A graph transformation system modelling the customer's view of a bank with a view relation from an account printer's view.

loose semantics of a graph transformation system is given by all *transition sequences* in **G**. It represents the *open behaviour* of a system that is embedded in a not completely specified environment. Also here we allow parallel transitions defined as transitions via parallel rules.

Example. A sample graph transformation system modelling a banking system from the customer's point of view is depicted in Figure 5 on the right. The rules are shown in Figure 4. The left-hand side of Figure 6 shows a sample transition sequence modelling the customer's view of some banking operations. After opening the new account "234567" (where the number is provided by the bank), a transfer transaction is ordered by customer Smith. At the same time, customer John asks for the balance of his account. This is modelled by two derivation steps (without unspecified changes) where the second one consists of the parallel application of *doTransaction* and *getBalance*. Thereafter, also customer John starts an order, while the first order is executed by the bank and the result becomes visible for the customers. Thus, the first *Transaction* object disappears without actually being deleted by the *doTransaction* rule, and the balances of the accounts "123456" and "234567" change. From the customer's point of view these are (expected, but technically) unspecified effects, i.e., the third step in the sequence is a true transition. Finally, customer Smith asks for the balance of account "123456" while the second transaction order is executed (which again is a true transition). $\triangle$

4 Views of Graph Transformation Systems

In order to integrate two views, their intended correspondences have to be specified by relating a common reference model to each view by a view relation. A view relation allows the renaming and extension of scheme graphs and rules. After the integration, similar view relations are established between a view and the overall system model.

Renaming. In order to allow, for example, the use of different names for the same operation in different views and the reference model, renaming relations are introduced. A *renaming relation* $\mathbf{G} \overset{ren}{\longleftrightarrow} \mathbf{G}'$ can be seen as a kind of dictionary establishing a one-to-one correspondence between the types, the rule names, and the (vertices and edges of the) rules

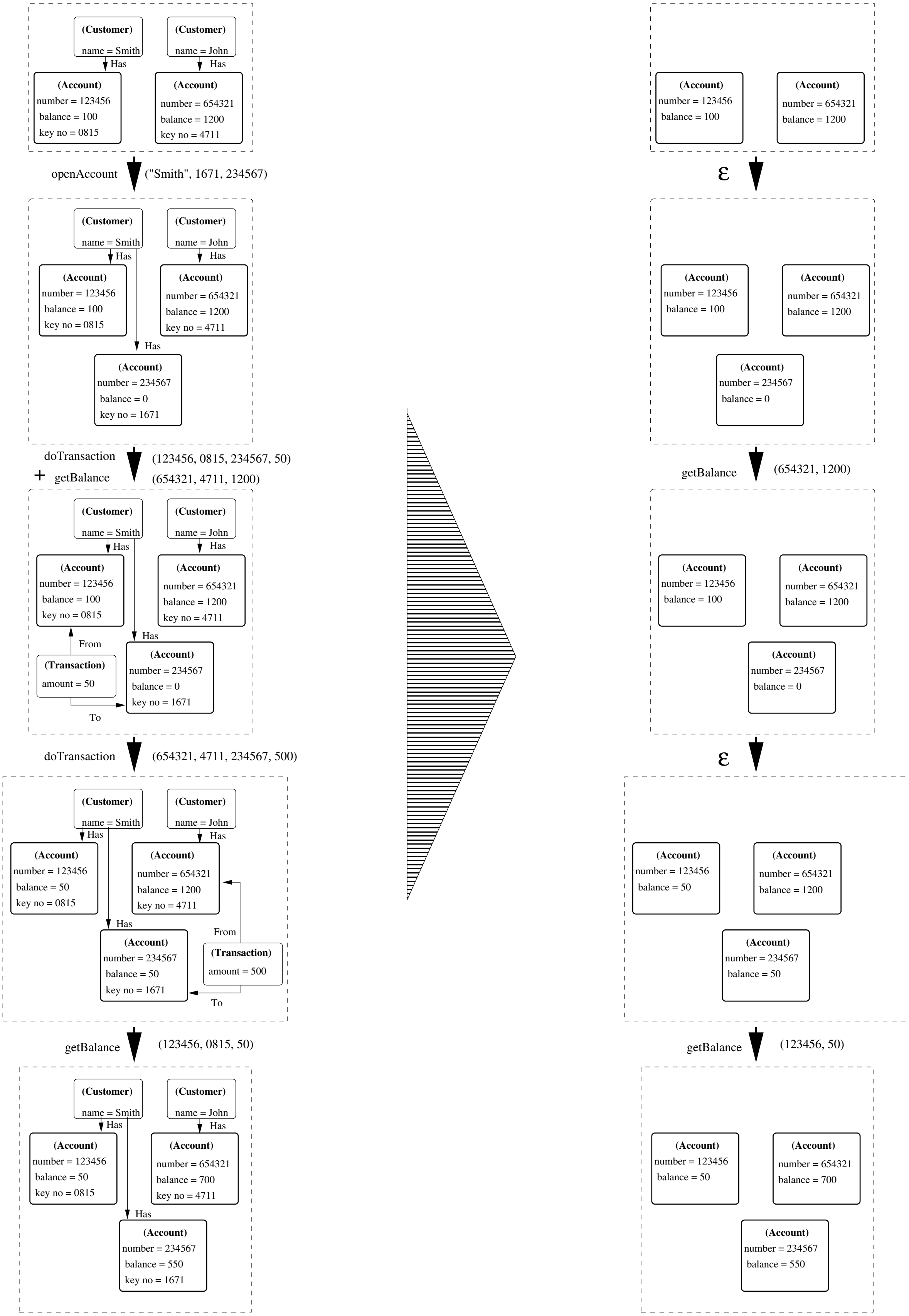


Fig. 6. Transition sequences in the customer's view (left) and in the printer's view (right), where the latter is a projection of the first.

of two graph transformation systems $\mathbf{G}$ and $\mathbf{G}'$. If x is an item (a type, a rule name, etc.) of $\mathbf{G}$ and x' the corresponding item in $\mathbf{G}'$ we write $x \stackrel{ren}{\longleftrightarrow} x'$.

Extension. As anticipated above, a view relation shall be composed of a renaming and an extension. The extension of a rule r_0 by another rule r_1 is modelled by the *subrule relation*. The rule $r_0 : L_0 \to R_0$ is a *subrule* of $r_1 : L_1 \to R_1$, written $r_0 \subseteq r_1$ if the effects of applying the latter extend the effects of applying the first. Formally, this means that $L_0 \subseteq L_1, R_0 \subseteq R_1$ (pre- and post-conditions are extended), $delete(r_0) \subseteq delete(r_1)$ (more is deleted by r_1), and $add(r_0) \subseteq add(r_1)$ (more is added by r_1).

A graph transformation system $\mathbf{G}_1$ *extends* another one $\mathbf{G}_0$, written $\mathbf{G}_0 \subseteq \mathbf{G}_1$, if scheme graph and rule names of $\mathbf{G}_0$ are extended, i.e., $SG_0 \subseteq SG_1$ and $N_0 \subseteq N_1$, and for each rule name $r \in N_0$ the associated rule in $\mathbf{G}_0$ is a subrule of the one in $\mathbf{G}_1$, i.e., $\rho_0(r) \subseteq \rho_1(r)$.

View relation. In order to specify the relation between a view and a system model, a *view relation* has to be specified. A view relation $v = (\mathbf{G}_0 \stackrel{ren}{\longleftrightarrow} \mathbf{G}_0^1 \subseteq \mathbf{G}_1)$ from $\mathbf{G}_0$ to $\mathbf{G}_1$ is a renaming of $\mathbf{G}_0$ such that $\mathbf{G}_1$ is an extension of the renamed system $\mathbf{G}_0^1$. More abstractly, we write $v : \mathbf{G}_0 \to \mathbf{G}_1$ and say that $\mathbf{G}_0$ *is a view of* $\mathbf{G}_1$. View relations may be composed by composing the underlying renamings and extensions in a suitable way.[2] This makes it possible to regard a view $v_0 : \mathbf{G}_0 \to \mathbf{G}_1$ on a view $v_1 : \mathbf{G}_1 \to \mathbf{G}_2$ as a view $v_0; v_1 : \mathbf{G}_0 \to \mathbf{G}_2$.

Let's discuss in more detail the relationship between a graph transformation system $\mathbf{G}_1$ and its view $\mathbf{G}_0$.

- A name x_0 of $\mathbf{G}_0$ may change to x_1 in $\mathbf{G}_1$. In order to represent this relationship, a dictionary $\mathbf{G}_0 \stackrel{ren}{\longleftrightarrow} \mathbf{G}_1$ is used containing the entry $x_0 \stackrel{ren}{\longleftrightarrow} x_1$ (and $x \stackrel{ren}{\longleftrightarrow} x$ for all unchanged names of $\mathbf{G}_0$).
- $\mathbf{G}_0$ may be extended by $\mathbf{G}_1$ by introducing new types or rule names. An item is new in $\mathbf{G}_1$ if it is not listed in the dictionary *ren*. The new types and rules of $\mathbf{G}_1$ are not visible to $\mathbf{G}_0$, i.e., they do not belong to this view.
- A rule r defined in $\mathbf{G}_0$ may be extended in $\mathbf{G}_1$. On the left- and/or right-hand side of r new vertices and edges may be added, while the name of the rule might be the same. This means that the effects of applying r are extended (additional items are deleted and/or added) while the pre- and post-conditions are strengthened.

Example. Figure 5 shows a view *Printer* of the customer's view that shall become a view of the banking system by composition of view relations later on. It models the restricted view of a printer where customers can ask for the balances of their accounts. Such a printer does not know about key numbers and is not able to open new accounts or to order transactions. Hence, the corresponding types and rules are not visible in the printers view. A renaming is not needed.

The printer's view of the sample transition sequence in Figure 6 on the left is shown in the same figure on the right. Recall that, e.g., the step in the user sequence on the left – opening an account – is a derivation step. In the printer's sequence on the right, however, it

[2] Technically, a view relation is an (injective) morphism between typed graph transformation systems. Here we presented such morphisms as decomposed into an isomorphism and an inclusion (called renaming and extension, resp.).

is seen as an ϵ-transition. The first *getBalance*-transition in the printer's view results from the parallel derivation step using *doTransaction* and *getBalance* in the customer's view. The *doTransaction* rule is hidden in the printer's view but its effects are still visible. $\triangle$

Hence, a view relation $v : \mathbf{G}_0 \to \mathbf{G}_1$ describes not only a projection of the state graphs of $\mathbf{G}_1$ to $\mathbf{G}_0$ but also a more abstract view of the behaviour of $\mathbf{G}_1$. Notice that derivation sequences (without unspecified effects) are not viewed as derivation sequences in general but as transition sequences, too: The view of a derivation sequence in $\mathbf{G}_1$ may be a transition sequence in $\mathbf{G}_0$.

5 Integration of Views

In the previous section, view relations were introduced for describing the relationships between views, reference model, and system model. Now, we explain the main technical concept of our approach, the actual integration of views.

All development starts with the reference model as domain-specific framework. The reference model of the banking system is shown in Figure 9. In the beginning it contains the basic object and relationship types of the banking system (i.e., without the operation *newAccount* that shall be added later on as extension of the reference model). The views *Customer* and *Clerk* are derived independently of each other from the reference model. This results in a situation where the reference model itself forms a view on the two specifications *Customer* and *Clerk*, which are so-called design views on the complete system model.

When integrating the design views to the system model, we have to know which items in the two views represent the same types and operations. Rather then relying on the names of these items, this correspondence is specified by the reference model, i.e., two items are assumed to represent the same concept if and only if they have a common origin. In fact, since view relations allow the renaming of items, this means that developers are free to choose the names in their view according to the preferences of the particular user group.

Typically, by extending and refining the design views new dependencies will be created which are not yet specified by the reference model. Hence the reference model has to be kept consistent with the views by extending it each time a new dependency is detected. The task of finding and resolving dependencies is simplified here by the fact that we start with a common reference model. This allows to reuse many general concepts, and every concept which is reused by two views is automatically shared. Moreover, the reference model and its view relations can be updated incrementally after each refinement of the design views.

When the reference model consistently specifies the intended dependencies of the design views, the actual integration of these views to the system model can be done automatically.[3] A situation of two design views $\mathbf{G}_1$ and $\mathbf{G}_2$ based on a reference model $\mathbf{G}_0$ by view relations v_1 and v_2 is shown in Figure 7. The integration of $\mathbf{G}_1$ and $\mathbf{G}_2$ over $\mathbf{G}_0$ will be done in two steps. First, we have to rename the design views so that the renamed views $\mathbf{G}'_1$ and $\mathbf{G}'_2$ share a name if and only if there is a common origin in the reference model $\mathbf{G}_0$. Then the

[3] In terms of category theory this construction can be abstractly described as a pushout of two morphisms of graph transformation systems corresponding to the view relations.

construction of the integrated system view can be done by componentwise set-theoretical union of $\mathbf{G}_1'$ and $\mathbf{G}_2'$. We assume that the names in the design views $\mathbf{G}_1$ and $\mathbf{G}_2$ are disjoint which can be ensured by qualification with the name of the view (like $\mathbf{G}_i.name$). The view relations $v_i : \mathbf{G}_0 \to \mathbf{G}_i$ are given by $\mathbf{G}_0 \overset{ren_i}{\longleftrightarrow} \mathbf{G}_0^i \subseteq \mathbf{G}_i$ in Figure 7.

Renaming of Views. The system $\mathbf{G}_1'$ and the renaming $\mathbf{G}_1 \overset{ren_1'}{\longleftrightarrow} \mathbf{G}_1'$ are obtained by extending the renaming $\mathbf{G}_0^1 \overset{ren_1}{\longleftrightarrow} \mathbf{G}_0$ to $\mathbf{G}_1$. More precisely, ren_1' agrees with ren_1 on $\mathbf{G}_0^1$, i.e., $ren_1'|_{\mathbf{G}_0^1} = ren_1$, and is minimal in the sense that nothing else is renamed, i.e., the renaming is the identity on $\mathbf{G}_0 \setminus \mathbf{G}_0^1$. The renamed system $\mathbf{G}_1'$ becomes an extension of $\mathbf{G}_0$. In a similar way we obtain $\mathbf{G}_2'$ with $\mathbf{G}_0 \subseteq \mathbf{G}_2'$ and $\mathbf{G}_2 \overset{ren_2'}{\longleftrightarrow} \mathbf{G}_2'$ by extending the renaming ren_2 to $\mathbf{G}_2$. This renaming is always possible and can be done automatically. Now, the integrated view $\mathbf{G}_3'$ can be constructed as union of the renamed views $\mathbf{G}_1'$ and $\mathbf{G}_2'$ in Figure 7.

$$\begin{tikzcd}
\mathbf{G}_0^1 \arrow[rr, leftrightarrow, "ren_1"] \arrow[d, phantom, sloped, "\subseteq"] & & \mathbf{G}_0 \arrow[rr, leftrightarrow, "ren_2"] \arrow[dll, "v_1"'] \arrow[drr, "v_2"] \arrow[dl, phantom, sloped, "\subseteq"] \arrow[dr, phantom, sloped, "\subseteq"] & & \mathbf{G}_0^2 \arrow[d, phantom, sloped, "\subseteq"] \\
\mathbf{G}_1 \arrow[r, leftrightarrow, "ren_1'"] \arrow[ddrr, "v_1^*"'] & \mathbf{G}_1' \arrow[dr, phantom, sloped, "\subseteq"] & & \mathbf{G}_2' \arrow[r, leftrightarrow, "ren_2'"] \arrow[dl, phantom, sloped, "\subseteq"] & \mathbf{G}_2 \arrow[ddll, "v_2^*"] \\
& & \mathbf{G}_3' \arrow[d, leftrightarrow, "ren"] & & \\
& & \mathbf{G}_3 & &
\end{tikzcd}$$

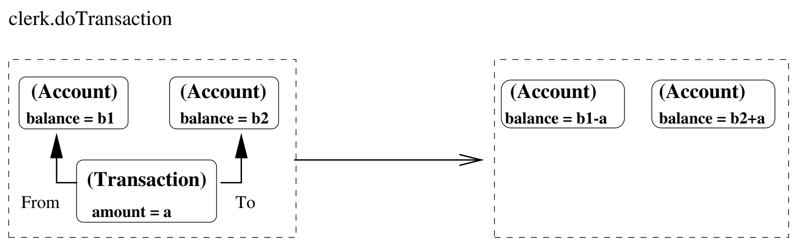
Fig. 7. Integration of the design views $\mathbf{G}_1$ and $\mathbf{G}_2$ to the system model $\mathbf{G}_3$.

Construction of Integrated View. The integrated view $\mathbf{G}_3' = \langle SG_3', N_3', \rho_3' \rangle$ is obtained by forming the union of the scheme graphs $SG_3' = SG_1' \cup SG_2'$ and of the sets of rule names $N_3' = N_1' \cup N_2'$ of $\mathbf{G}_1'$ and $\mathbf{G}_2'$, respectively. For the rule $\rho_3'(r')$ associated with a rule name $r' \in N_3'$ we distinguish three cases:

- If $r' \in N_1' \setminus N_2'$ then $\rho_3'(r') = \rho_1'(r')$, i.e., the rule of $\mathbf{G}_1'$ is inherited
- If $r' \in N_2' \setminus N_1'$ then $\rho_3'(r') = \rho_2'(r')$, i.e., the rule of $\mathbf{G}_2'$ is inherited
- If $r' \in N_2' \cap N_1'$ then $\rho_3'(r') = L_1' \cup L_2' \to R_1' \cup R_2'$ is an integrated rule obtained by componentwise (non-disjoint) union of the left- and right-hand sides of $\rho_i'(r') = L_i' \to R_i'$ for $i = 1, 2$.

Then, the integrated view $\mathbf{G}_3'$ may be renamed to $\mathbf{G}_3$ via ren. The view relation v_1^* is obtained by composing the view relation $\mathbf{G}_1 \overset{ren_1'}{\longleftrightarrow} \mathbf{G}_1' \subseteq \mathbf{G}_3'$ with the renaming ren (which is a special view relation as well). In a similar way we obtain view relation v_2^*.

Example. The integrated rule *common.newAccount* is constructed in Figure 8 as the union of the rules *openAccount* and *makeAccount* of the customer's and the clerk's view. It synchronises the activities that are necessary for creating a new account: The integrated rule has all the pre- and postconditions of the two original rules, i.e., it requires the existence of both, the *Customer* and the *Bank* object. An application of this rule shows the combined



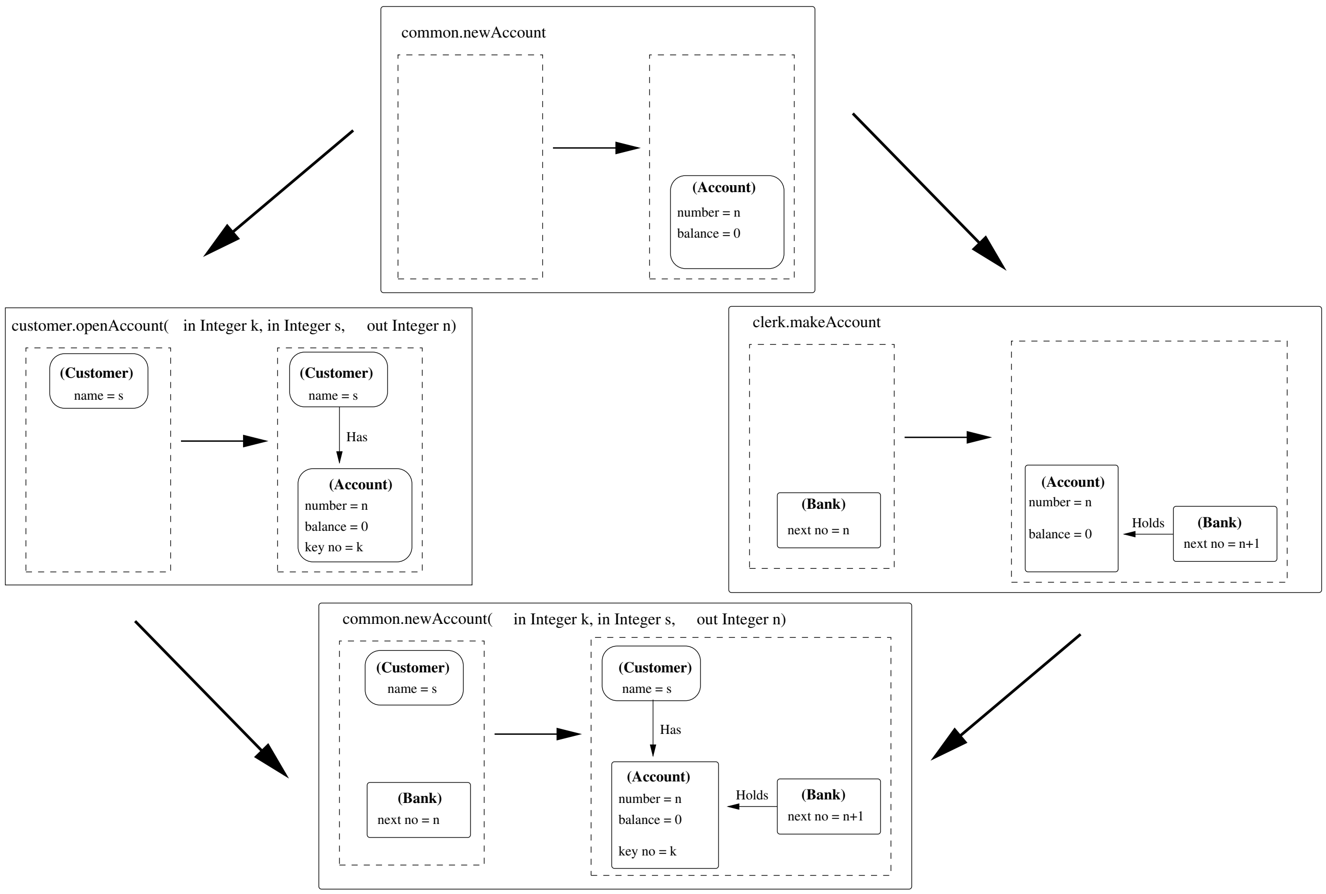


Fig. 8. The *doTransaction* rule of the clerk and the synchronisation of the *openAccount* rules.

effects of its constituents, where the action of the common subrule *common.newAccount*, the creation of the *Account* object, is performed only once.

The rule *clerk.doTransaction* shown in Figure 8 describes the completion of a transfer transaction. The integrated system model *Bank* in Figure 9 also contains the other rules of the customer's and the clerk's view, which are not synchronised. Note that not only the customer's view contains a rule *doTransaction* but also the clerk's view. These two rules are not identified, however, since they have no common source in the reference model. The name conflict is resolved automatically by qualification of the local names with the names of the views. (The qualifications are skipped in the clerk's and customer's view in Figure 9.) On the other hand, the rules *openAccount* and *makeAccount* represent the same operation (despite their different names) since they both stem from the same rule *common.newAccount*. They are both renamed to *common.newAccount* in the renaming step of the construction. △

More generally, we may have the following situations:

- It may be the case that "semantically the same" concept is described in $\mathbf{G}_1$ and $\mathbf{G}_2$ using different names (like *openAccount* and *makeAccount* above). This relationship between $\mathbf{G}_1$ and $\mathbf{G}_2$ is only understood (and may be taken into account by the integration) if both names have a common source in the common view $\mathbf{G}_0$ (like *newAccount*). This has to be defined in the dictionaries ren_1 and ren_2. Then, the concept occurs only once in the integrated model, under the name of the common view. If this relationship is not specified, the two concepts are considered as unrelated and are kept separately in the integrated model. This illustrates also the difference between a synchronised rule and the parallel application of two rules. A parallel application of *Customer.openAccount* and*Clerk.makeAccount* would create two distinct new *Account* nodes. The synchronised rule *common.newAccount* realizes that, as desired, only one new account node is created.
- On the other hand, the same name may be used in $\mathbf{G}_1$ and $\mathbf{G}_2$ in order to describe "semantically different" concepts (e.g., *doTransaction* in the customer's and the clerk's view). This does not causes any problem in our approach since we assume that the names are qualified, i.e., *Customer.doTransaction* and *Clerk.doTransaction*.
- In order to represent shared knowledge of $\mathbf{G}_1$ and $\mathbf{G}_2$, which is not yet expressed by the reference model, it has to be extended. This should be done by the model manager. If the reference model is used by more than two views, however, this means that the extra information is also propagated to all other views as well. If this is not desired the reference model has to be kept unchanged and an *abstract view* has to be introduced instead which is also based on the reference model and specifies the sharing between $\mathbf{G}_1$ and $\mathbf{G}_2$. The result is a hierarchy of views. Scenarios of more than two views are discussed in a subsequent paper.
- A similar observation holds if a rule of $\mathbf{G}_0$ is extended in $\mathbf{G}_1$ and $\mathbf{G}_2$ with the same intended meaning. Also in this case, the model manager may suggest to lift this extension to the reference model, or in case of other views, to specify the extension by an abstract view instead.

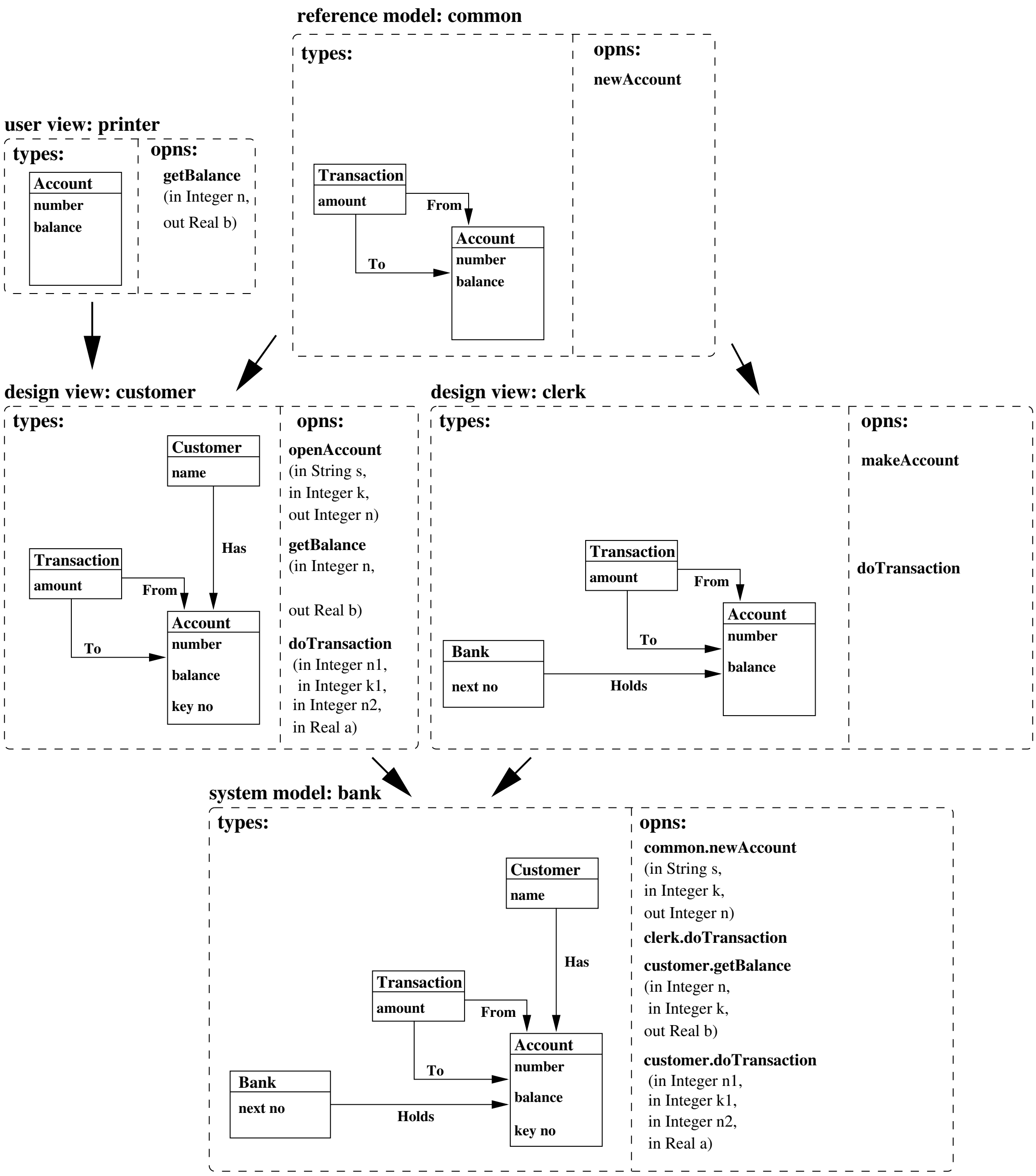


Fig. 9. Integration of the customer's and the clerk's view.

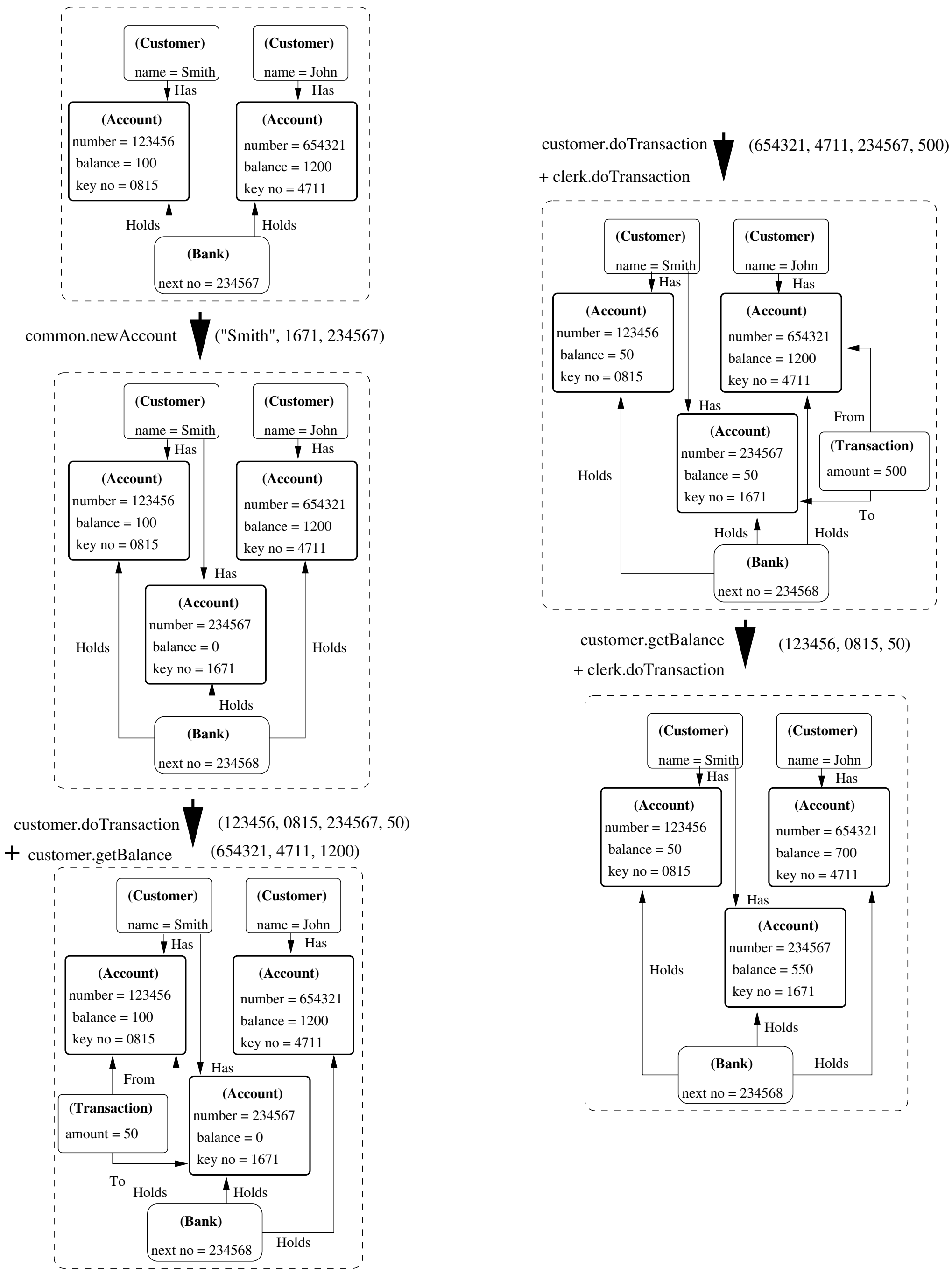
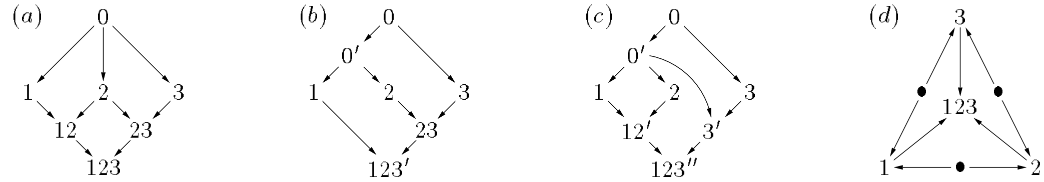


Fig. 10. Derivation sequence in the system model of the bank.

Example. Figure 10 shows a derivation sequence that models the same operations of Figure 6 from the bank's point of view. The *common.newAccount* operation is a synchronised action of a customer and the clerk. The parallel *Customer.doTransaction* and *Customer.getBalance* operation is performed, while the clerk has an idle step. The second *Customer.doTransaction* and *Customer.getBalance* are complemented by two *Clerk.doTransaction* operations, that take over the formerly (in the customer's view) unspecified effects. Hence, the transitions of the system model are obtained in a compositional way by integrating the transitions of the two design views. △

It has been shown by the discussion above that the simple idea of deriving all views from a single reference model is no longer sufficient if more than two views are involved. Firstly, abstract views have to be introduced if two views share a certain concept that shall not be visible to the third view. Secondly, the construction of view integration has to be iterated or generalised in order to obtain one integrated view.

Below, we show and discuss some abstract scenarios that may arise if three views, denoted by $1, 2$ and 3, shall be integrated.



(a) Three views based on the same abstract view (or reference model) 0 are integrated iteratively. First 12 and 23 are obtained by integrating $1, 2$ and $2, 3$ over 0, respectively. Then, 2 is a view of both 12 and 23 so that the global view 123 is obtained by integrating 12 and 23 over 2. Up to a renaming, the resulting view is the same if we first construct 23 as above and then integrate 1 and 23 over 0 using the fact that the view relations $0 \to 3 \to 23$ compose to $0 \to 23$.
(b) Given the situation of (a), assume that the same concept is used in 1 and 2 without being part of the reference model 0. Then, an abstract view $0'$ may be introduced, that extends 0 by the commonly used concept such that the view relations $0 \to 1$ and $0 \to 2$ are "redirected" over $0'$. Now, the new concept of $0'$ is not visible to 3. The construction of the integrated view $123'$ may now be done incrementally by reusing the "intermediate result" 23 of (a), which is then integrated with 1 over $0'$.
(c) Assume that in the situation of (b), the extension of 0 to $0'$ shall be propagated to view 3. Then, we have to integrate $0'$ with 3 over 0. This use of the integration of views' construction is asymmetric in the sense that $0'$ is an abstract view (that is only used to specify the sharing) while 3 is a design view. This should be reflected in the integrated view $3'$ where the names from 3 should in any case have priority over those from 0. After the construction of $3'$, the situation of (a) is recovered, i.e., all three views are based on the same abstract view $0'$.
(d) Finally there may be a situation of "cyclic sharing" where each two views have a common abstract view (denoted by $\bullet$). In this case, the "binary integration" is not sufficient for

constructing an integrated view 123. We can, however, generalise the construction to three (or more) views and more complex kinds of diagrams.[4]

6 Conclusion

In this paper we have presented a view-oriented approach to concurrent system modelling with the following basic features:

- Separate viewpoints of a system are represented by different views sharing a common reference model.
- Views are represented in an intuitive graphical way integrating static and dynamic aspects of the system.
- The views are kept consistent by a model manager by extending the reference model whenever new dependencies occur in the development process.
- Using the reference model, consistent views can be integrated automatically.
- In case of more than two views, additional abstract views may have to be introduced leading to a hierarchy of views.
- The specification approach is based on typed graph transformation systems providing a sound mathematical base and a rich theory.
- In addition to the classical semantics of graph transformation systems based on derivation sequences, a new loose semantics is considered which is able to model the open behaviour of views.

Despite the benefits mentioned above, our view-oriented approach has still to be fully exploited and systematically applied to examples of realistic size. Reference models for different application areas have to be developed. Moreover, views are usually more complex than flat graph transformation systems, i.e., we have to extend our approach by horizontal structuring techniques as considered in [HCEL96] and more powerful operations like rule-based transactions. This presents the additional problem of transaction refinement and coordination of complex behaviours, which is not yet sufficiently studied in the graph transformation context. Also, we would like to consider different kinds of constraints, like temporal logic formulas, in order to specify and verify important (e.g. safety critical) system properties (see [HEWC97]). Last but not least our view-oriented technique must be supported by a specification language based on typed graph transformation systems and corresponding tools. A good candidate for this is PROGRES [Sch91] where a related construction is already used for merging different versions of a document [Wes91]. Applied to carefully prepared examples, this merge operation already delivers the desired result of our construction. The precise relationship to our approach has still to be investigated.

References

[AE96] M. Andries and G. Engels. A hybrid query language for the extended entity relationship model. *Journal of Visual Languages and Computing*, 7(3):321–352, 1996. Special Issue on Visual Query Systems.

[4] That is, the pushout describing the binary integration is generalised here to a colimit of an arbitrairy finite diagram.

[Bal96] R. Balzer. Current state and future perspectives of software process technology. Keynote Speech, Software Process (SP 96), Brighton, 2-6 December 1996.

[BLN86] C. Batini, M. Lenzerini, and S. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, 1986.

[CEL+96] A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and J. Padberg. The category of typed graph grammars and their adjunction with categories of derivations. In *5th Int. Workshop on Graph Grammars and their Application to Computer Science, Williamsburg '94, LNCS 1073*, pages 56–74, 1996.

[Dat79] C.J. Date. *An Introduction to Database Systems.* Addison-Wesley, 1979.

[FKN+92] A. Finkelstein, J. Kramer, B. Nuseibeh, M. Goedicke, and L. Finkelstein. Viewpoints: A framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):31–58, March 1992.

[HCEL96] R. Heckel, A. Corradini, H. Ehrig, and M. Löwe. Horizontal and vertical structuring of typed graph transformation systems. *Math. Struc. in Comp. Science*, 6(6):613–648, 1996. Also techn. report no 96-22, TU Berlin.

[HEWC97] R. Heckel, H. Ehrig, U. Wolter, and A. Corradini. Integrating the specification techniques of graph transformation and temporal logic. In *Proc. of MFCS'97, Bratislava*, 1997. To appear.

[Jac95] D. Jackson. Structuring Z specifications with views. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 4(4):365–389, October 1995.

[LB93] M. Löwe and M. Beyer. AGG — an implementation of algebraic graph rewriting. In *Proc. Fifth Int. Conf. Rewriting Techniques and Applications, LNCS 690*, pages 451–456. Springer Verlag, 1993.

[LRP+96] T. Lewis, L. Rosenstein, W. Pree, A. Weinand, E. Gamma, P. Calder, G. Andert, J. Vlissides, and K. Schmucker. *Object-oriented Application Frameworks.* Prentice-Hall, 1996.

[Nag96] M. Nagl, editor. *Building Tightly Integrated Software Development Environments: The IPSEN Approach.* LNCS 1170, Springer, 1996.

[Rib96] L. Ribeiro. *Parallel Composition and Unfolding Semantics of Graph Grammars.* PhD thesis, TU Berlin, 1996.

[Roz97] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations.* World Scientific, 1997.

[Sch91] A. Schürr. Progress: A vhl-language based on graph grammars. In *4th Int. Workshop on Graph Grammars and their Application to Computer Science, LNCS 532.* Springer, 1991.

[SWZ95] A. Schürr, A.J. Winter, and A. Zündorf. Graph grammar engineering with PROGRES. In W. Schäfer and P. Botella, editors, *5th European Software Engineering Conference (ESEC'95), Sitges*, pages 219–234. Springer LNCS 989, September 1995.

[Tae96] G. Taentzer. Modeling dynamic distributed object structures by graph transformation. *Object Currents*, 1(12), Dec. 1996. http://www.sigs.com/publications/docs/oc/9612/oc9701.f.taentzer.html.

[Wag97] A. Wagner. *A Formal Object Specification Technique Using Rule-Based Transformation of Partial Algebras.* PhD thesis, TU Berlin, 1997.

[Wes91] B. Westfechtel. Structure-oriented merging of revisions of software documents. In P. Feiler, editor, *Proc. 3rd Int. Workshop on Software Configuration Management (SCM3) New York*, pages 68–79. ACM Press, 1991.

[Zam96] A. Zamperoni. GRIDS - graph-based integrated development of software: Integrating different perspectives of software engineering. In *Proc. 18th International Conference on Software Engineering (ICSE)*, pages 48–59. IEEE CS Press, March 25-29 1996.