Chapter 1

A VIEW-BASED APPROACH TO SYSTEM MODELING BASED ON OPEN GRAPH TRANSFORMATION SYSTEMS

R. HECKEL, G. ENGELS
Universität GH Paderborn,
FB 17 Informatik, Warburger Str. 100, D-33098 Paderborn
{engels, reiko}@uni-paderborn.de

H. EHRRIG, G. TAENTZER
Technische Universität Berlin,
FB 13 Informatik, Franklinstrasse 28/29, D-10587 Berlin, Germany
{ehrig, gabi}@cs.tu-berlin.de

The idea of a combined reference model- and view-based specification approach has been proposed recently in the software engineering community. In this chapter we present a specification technique based on open graph transformation systems which supports such a development approach. Open graph transformation systems extend graph transformation systems (in the double-pushout approach) by a new loose semantics for rule-based systems, which allows to model the interaction between different views, and by explicit frame conditions which restrict these interactions to an interface of open types. On this background, formal notions of view and view relation are developed and the behavior of views is described by the loose semantics. Based on the assumption that dependencies between different views are faithfully described by a common reference model, a construction is developed for the automatic integration of views. The views and the reference model are kept consistent manually, which is the task of a model manager. All concepts and results are illustrated at the well-known example of a banking system.

1.1 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.
One 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 specializing integrated networks of classes, so-called frameworks [1]. Thus, a major research as well as development field is currently the definition of frameworks for various application domains.

Another important 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 realization of a software system. For instance, during the requirements specification phase, a team of application engineers with different skills and backgrounds is split 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, modularization 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 modularization 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. [2]).

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. [3]). The specifications of different viewpoints arise during the specification phase when different application engineers define requirements on different aspects of a system. 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. Thus, in addition and in contrast to an application framework, the reference model also determines the dependencies between different views.

In this chapter we present a specification technique based on graph transformations which supports such a development approach. 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 modeling of static and dynamic aspects of software systems are presented in Section 3. In Section 4 we give a definition for views of graph transformation systems. Since all different views are required to be based on a common reference model, we are able to present in Section 5 a general construction for view integration. This can be considered as an automatic view integration. Finally in Section 6 we summarize the main ideas and discuss some remaining open problems.

The presentation in this chapter is based on [4,5] and the PhD thesis of the first author [6].

1.2 Concept of Views and View Integration

A view is an incomplete specification of a system focusing on a particular aspect or subsystem. As such it 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 system model.

The overall approach can be sketched as follows (cf. Figure 1.1). 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 speaking, a (partial) specification is called a view on another specification, if a renamed version of the first can be embedded into the second. The integration of views is done in two steps. First, new dependencies between views (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 reference model. Hence, such new dependencies are mainly problem-specific. 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 current approaches in the object-oriented world mentioned in the Introduction, 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 specialized concurrently by several views.

Following such a view-based specification approach, various forms of possible inconsistencies can be distinguished. Here, 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
1.2. CONCEPT OF VIEWS AND VIEW INTEGRATION

solved during scheme integration (cf. [7]). 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 behavior. In this case, the two views have to be synchronized to achieve a consistent system’s behavior specification.

Different solutions for (iii) can be distinguished. The viewpoint approach (cf. [3]) 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. [8]), follow a descriptive approach, where the application engineer has to integrate different view specifications in an overall specification by additional inter-view constraints. In this paper, 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.
The situation becomes 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 allows, 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 1.2 sketches this situation. A more detailed discussion follows at the end of this chapter.

1.3 Graph Transformation for System Modeling

Graph grammars and transformations have been introduced as a generalization 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., [9]) which has a number of applications to system modeling and software engineering (see [10, 11, 12, 13] as well as [14, 15, 16] in
1.3. GRAPH TRANSFORMATION FOR SYSTEM MODELING

Figure 1.3: Example of type and instance graphs

this volume) based on concrete specification languages and supporting tools (see [17,18,19] and the attached CDROM).

The main idea of graph transformation as a specification technique is to model object structures by graphs and modifying operations by graph transformations. This is in line with the object-oriented paradigm where static and dynamic aspects of a system are specified in an integrated way. Typed graph transformation systems [20] allow to define a set of graphs by a type graph together with type-consistent operations on these graphs.

For defining the concept of view, typed graph transformation systems are provided with a loose semantics and corresponding syntactic features for restricting it in certain cases [21]. The resulting specifications are called open graph transformation systems [6].

In this section, we explain in more detail 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.

Graphs Graphs and diagrams are often used in software engineering for visualizing complex structures. We only mention Entity-Relationship (ER) diagrams and instances in data modeling 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 modeling 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 their 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. The idea of schema and instance graphs can be described more generally by the concept of typed graphs [22,20] where a fixed type graph \( TG \) serves as a schema, and instances are graphs equipped with a mapping to the type graph. Moreover, it is common that (type and instance) graphs contain textual or numerical information like object and relationship names or attribute values, that are associated with vertices and edges. In this case we speak of attributed graphs.

Example 1.1
A sample pair of type and instance graphs is shown in Figure 1.3. The type 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 authorized access and, of course, a balance. Transactions are requests for transferring money between accounts. On the right side of Figure 1.3, an instance of this type graph is shown. It represents a toy state of the banking system where a customer holds two accounts with an ongoing transaction.

Rule-Based Graph Transformation. State changing operations on graphs are modeled by graph transformations which are specified by graph transformation rules (or productions) \( p : L \rightarrow R \). They consist of a rule name \( p \) 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. We assume that the intersection \( L \cap R \) is a graph called the interface of \( p \). It contains those items that are read but not deleted by the operation.

Example 1.2
The upper left rule in Figure 1.4 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
1.3. GRAPH TRANSFORMATION FOR SYSTEM MODELING

Figure 1.4: Graph transformation rules for opening a new account, getting the balance, and starting a transfer transaction.

state. The rule doTransaction below starts a transfer transaction (that has to be completed later).

A derivation step \( G \xrightarrow{r} H \) from \( G \) to \( H \) using a rule \( p : L \rightarrow R \) 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. The rule application is only permitted if after the deletion step, the resulting structure is a graph again, i.e., implicit effects like the automatic deletion of dangling edges do not occur. We denote by delete(\( p \)) the part \( L \setminus R \) and by add(\( p \)) 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 liberal 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 behavior of views, we introduce the notion of graph transition by dropping the above mentioned frame condition. Like a
derivation step, a graph transition \( G \xrightarrow{r} H \) from \( G \) to \( H \) via \( r \) requires that \( L \) occurs in \( G \). Then, \textit{at least delete}(\( r \)) is removed from \( G \) and \textit{at least add}(\( r \)) is added, but there may be unspecified deletion and addition as well.

However, not all unspecified effects are always desirable. In fact, dropping entirely the implicit frame condition yields a behavior which is too loose in many cases: If no assumptions are made for the interaction with the enclosing system, any interference is possible. Thus, an intermediate level is needed which permits \textit{some} unspecified effects but allows to specify \textit{explicit} frame conditions in order to protect particular types of the graphs from the loose semantics.

Such frame conditions can be specified in a positive or negative way. The positive specification lists all the \textit{open types} where loose effects are permitted. In the negative presentation instead, all the \textit{protected types} are given. The two representations are equivalent, since all types that are not open are assumed to be protected, and vice versa. In the following we use the positive representation.

The specification of open types is further refined by distinguishing open types for deletion and for creation. An explicit frame condition \( O = (O^{-}, O^{+}) \) consists therefore of two sets \( O^{-} \) and \( O^{+} \) of types from the type graph \( TG \) containing, respectively, the types which are open for unspecified deletion and creation.

**Open Graph Transformation Systems** An open graph transformation system \( O = (TG, P, \pi, O) \) consists of a type graph \( TG \), a set of rule (or production) names \( P \), a mapping \( \pi \) providing for each rule name \( p \) a rule \( L \rightarrow R \) where \( L \) and \( R \) are instance graphs of \( TG \) (their nodes and edges are typed over \( TG \)), and an explicit frame condition \( O \).

The rules of a graph transformation system can be applied sequentially or in parallel. The parallel application of rules \( p_1 \) to \( p_n \) for \( n > 0 \) is described by applying the so-called \textit{parallel rule} \( p_1 + \cdots + p_n : L_1 + \cdots + L_n \rightarrow R_1 + \cdots + R_n \), constructed as (componentwise) disjoint union of the given rules. The occurrences of \( L_i \) and \( L_j \) for \( i \neq j \) in the given graph may also overlap, but only in items that are not deleted by the rules.

A particular role is played by the \textit{empty rule} \( \epsilon : \emptyset \rightarrow \emptyset \) which is formally obtained by the above construction in the case that \( n = 0 \). This rule can be used for modeling state transitions that are entirely caused by the environment.

It specifies no effect, that is, \( \text{delete}(\epsilon) = \text{add}(\epsilon) = \emptyset \). A transition via \( \epsilon \) allows any change to the current state.

**Example 1.3**
An open graph transformation system modeling a banking system from the customer’s point of view is depicted in Figure 1.5. The rules are shown in
design view: customer

<table>
<thead>
<tr>
<th>types:</th>
<th>opns:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer name</td>
<td>openAccount</td>
</tr>
<tr>
<td>Transaction [-] amount</td>
<td>(in string s,</td>
</tr>
<tr>
<td>From [-]</td>
<td>in int k,</td>
</tr>
<tr>
<td>To [-]</td>
<td>out int n)</td>
</tr>
<tr>
<td>Account number balance [-]</td>
<td>getBalance</td>
</tr>
<tr>
<td>key no</td>
<td>(in int n,</td>
</tr>
<tr>
<td></td>
<td>in int k1,</td>
</tr>
<tr>
<td></td>
<td>out real b)</td>
</tr>
</tbody>
</table>

Figure 1.5: An open graph transformation system modeling the customer’s view of a bank.

Figure 1.4. The open types for deletion and addition are indicated by “−” and/or “+” markers in square brackets following type and attribute names. Attributes may be created and deleted along with their carrier objects. Thus, the permission to delete, e.g., Transaction nodes implies the permission to delete, as a consequence, the corresponding amount attribute. If an attribute like the balance of an account shall be modified independently of its carrier, this has to be declared directly like in balance [-/+]. (Notice that changing the value of an attribute means to delete the old value and to create the new one. This requires both add and delete permissions.)

All the other types of the view are controlled by the customer.

Figure 1.6 shows a sample transition sequence modeling 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 modeled 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”

---

This notation is inspired by the recent PROGRES package concept [23] where markers for mutable types are introduced in order to specify the permission to modify instances of these types in a client package.
Figure 1.6: Transition sequences in the customer's view.
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).

Notice how the frame condition is observed in this sequence, that is, deletion of Transaction orders and changes of balances of accounts are the only unspecified effects.

Remark:
Our notions of graph, transformation rule, derivation step, derivation, etc. are the standard ones in the double-pushout (DPO) approach to graph transformation [24] but for the operational definition of a single rule application which is given for the restricted case where the left-hand side of a rule is isomorphic to a subgraph of the given graph (injective matching). For parallel rule applications, realized by a parallel rule, this restriction has to be relaxed allowing for non-disjoint overlapping between matches of different rules. The local matches themselves, however, are still required to be injective.

Graph transitions are introduced in [21.6]. Technically speaking, they are based on a double-pullback (DPB) construction, unlike direct derivations that are classically defined using a double-pushout construction. Open graph transformation systems are introduced in [6] by equipping typed graph transformation systems [20] with a loose semantics and open types.

1.4 Views of Open 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 type graphs and rules. After the integration, similar view relations are established between the views 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 \( \mathcal{O} \xleftrightarrow{\text{ren}} \mathcal{O}' \) 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 of two open graph transformation systems \( \mathcal{O} \) and \( \mathcal{O}' \). If \( x \) is an item (a type, a rule name, etc.) of \( \mathcal{O} \) and \( x' \) the corresponding item in \( \mathcal{O}' \) we write \( x \leftrightarrow{\text{ren}} x' \).
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 modeled by the subrule relation. The rule \( r_0 : L_0 \rightarrow R_0 \) is a subrule of \( r_1 : L_1 \rightarrow R_1 \), 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), \( \text{delete}(r_0) \subseteq \text{delete}(r_1) \) (more is deleted by \( r_1 \)), and \( \text{add}(r_0) \subseteq \text{add}(r_1) \) (more is added by \( r_1 \)).

An open graph transformation system \( \mathcal{O}_1 \) extends another one \( \mathcal{O}_0 \), written \( \mathcal{O}_0 \subseteq \mathcal{O}_1 \), if type graph and rule names of \( \mathcal{O}_0 \) are extended, i.e., \( TG_0 \subseteq TG_1 \) and \( P_0 \subseteq P_1 \), and for each rule name \( p \in P_0 \) the associated rule in \( \mathcal{O}_0 \) is a subrule of the one in \( \mathcal{O}_1 \), i.e., \( \pi_0(p) \subseteq \pi_1(p) \). In order to ensure that an extension gives rise to a projection of both states and behavior from \( \mathcal{O}_1 \) to \( \mathcal{O}_0 \), the following compatibility conditions for the open types \( O_0 = \langle O_0^-, O_0^+ \rangle \) of \( \mathcal{O}_0 \) have to be obeyed.

**new rules:** For each new rule \( r_1 \in P_1 \), the nodes and edges of \( \text{delete}(r_1)|_{TG_0} \) (all those which are deleted and whose type is in \( TG_0 \)) have to be instances of open types in \( O_0^- \) and analogously for \( O_0^+ \).

**extension of rules:** For each rule \( r_1 \in P_1 \) extending \( r_0 \in P_0 \), the nodes and edges in \( \text{delete}(r_1)|_{TG_0} \setminus \text{delete}(r_0) \) have to be instances of open types in \( O_0^- \). An analogous condition has to be satisfied for \( O_0^+ \).

**open types:** No additional types of \( TG_0 \) are declared open in \( \mathcal{O}_1 \), i.e., \( O_1^- \cap TG_0 \subseteq O_0^- \) and \( O_1^+ \cap TG_0 \subseteq O_0^+ \).

The first two conditions ensure that all derivations using (extended or newly introduced) rules of \( \mathcal{O}_1 \) can be seen as transitions in \( \mathcal{O}_0 \) which verify the frame conditions of \( \mathcal{O}_0 \). The last condition states that this also holds for the valid transitions in \( \mathcal{O}_0 \) since the frame conditions of \( \mathcal{O}_1 \) are more restrictive than the ones of \( \mathcal{O}_0 \).

**View Relation**  In order to specify the connection between a view and the system model, a view relation has to be specified. A view relation \( \nu = (\mathcal{O}_0 \leftrightarrow \mathcal{O}_1) \) from \( \mathcal{O}_0 \) to \( \mathcal{O}_1 \) is a renaming of \( \mathcal{O}_0 \) such that \( \mathcal{O}_1 \) is an extension of the renamed system \( \mathcal{O}_0^1 \). More abstractly, we write \( \nu : \mathcal{O}_0 \rightarrow \mathcal{O}_1 \) and say that \( \mathcal{O}_0 \) is a view of \( \mathcal{O}_1 \). View relations may be composed by composing the underlying renamings and extensions in a suitable way. This makes it possible to regard a view \( \nu : \mathcal{O}_0 \rightarrow \mathcal{O}_1 \) on a view \( \nu : \mathcal{O}_1 \rightarrow \mathcal{O}_2 \) as a view \( \nu \circ \nu : \mathcal{O}_0 \rightarrow \mathcal{O}_2 \).

Let’s discuss in more detail the relationship between a graph transformation system \( \mathcal{O}_1 \) and its view \( \mathcal{O}_0 \).
1.4. VIEWS OF OPEN GRAPH TRANSFORMATION SYSTEMS

![Diagram of graph transformation system]

Figure 1.7: A graph transformation system modeling the customer’s view of a bank with a view relation from an account printer’s view.

- A name $x_0$ of $\mathcal{O}_0$ may change to $x_1$ in $\mathcal{O}_1$. In order to represent this relationship, a dictionary $\mathcal{O}_0 \leftrightarrow \mathcal{O}_1$ is used containing the entry $x_0 \leftrightarrow x_1$ (and $x \leftrightarrow x$ for all unchanged names of $\mathcal{O}_0$).

- $\mathcal{O}_0$ may be extended by $\mathcal{O}_1$ by introducing new types or rule names. An item is new in $\mathcal{O}_1$ if it is not listed in the dictionary $ren$. The new types and rules of $\mathcal{O}_1$ are not visible to $\mathcal{O}_0$, i.e., they do not belong to this view.

- A rule $p$ defined in $\mathcal{O}_0$ may be extended in $\mathcal{O}_1$. On the left- and/or right-hand side of $p$ new vertices and edges may be added, while the name of the rule might be the same. In order to respect the frame conditions of $\mathcal{O}_0$ these new items have to be either instances of newly introduced types or of open types of $\mathcal{O}_0$. In general, the effects of applying $p$ are extended (additional items are deleted and/or added) while the pre- and post-conditions are strengthened.

**Example 1.4**

Figure 1.7 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 printer’s view. The printer’s view does
Figure 1.8: The projection of the customer’s transition sequences in the printer’s view.
not “own” any type, that is, all types are open both for unspecified deletion
and creation.

The printer’s view of the sample transition sequence in Figure 1.6 is shown
in Figure 1.8. 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 is seen as an $\epsilon$-transition. The first $\text{getBalance}$-transition in
the printer’s view results from the parallel derivation step using $\text{doTransaction}$
and $\text{getBalance}$ in the customer’s view. The $\text{doTransaction}$ rule is hidden in
the printer’s view but its effects are still visible. △

Hence, a view relation $v : \mathcal{O}_0 \rightarrow \mathcal{O}_1$ describes not only a projection of the
state graphs of $\mathcal{O}_1$ to $\mathcal{O}_0$ but also a more abstract view of the behavior of $\mathcal{O}_1$.

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 $\mathcal{O}_1$ may be a transition sequence in $\mathcal{O}_0$.

Remark:
In [6] view relations are formally described as morphisms between open graph
transformation systems. Here we present such morphism as decomposed into
an isomorphism and an inclusion (called renaming and extension, respectively).

All morphisms expressible in this way are injective. Based on the definition
of morphisms of open graph transformation systems in [6], however, we could also
represent a notion of view relation where different items of the more abstract
view are identified in the more concrete view. □

1.5 Integration of Views

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

Each development starts with the reference model as domain-specific fram-
ework. The reference model of the banking system is shown in Figure 1.12. In
the beginning it contains the basic object and relationship types of the bank-
ing system (i.e., without the operation $\text{newAccount}$ that shall be added later
as extension of the reference model). The views $\text{Customer}$ and $\text{Clerk}$ are
derived independently of each other from the reference model. This results in
a situation where, technically speaking, the reference model forms itself a view
on the two specifications $\text{Customer}$ and $\text{Clerk}$, 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.
If 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. A situation of two design views \( \mathcal{O}_1 \) and \( \mathcal{O}_2 \) based on a
reference model \( \mathcal{O}_0 \) by view relations \( v_1 \) and \( v_2 \) is shown in Figure 1.9.

**Definedness of the View Integration** Denote by \( \text{affected}^-(v_i) \) resp. \( \text{affected}^+(v_i) \) the sets of affected types, i.e., nodes and/or edges of \( TG_0 \) whose
instances are deleted or created by a new rule in \( \mathcal{O}_i \) or by an extension in \( \mathcal{O}_i \) of
a rule in \( \mathcal{O}_0 \). By the definition of view relations, such affected types are open,
that is, \( \text{affected}^-(v_i) \subseteq O_0^- \) and \( \text{affected}^+(v_i) \subseteq O_0^+ \).
The construction of view integration is defined if the open types of the reference
model that are affected by new or extended rules in one of the design views
are kept open in the other view, formally

\[
\text{ren}_2(\text{affected}^-(v_1)) \subseteq O_2^- \quad \text{and} \quad \text{ren}_1(\text{affected}^-(v_2)) \subseteq O_1^-
\]

and similarly for \( O_1^+ \) and \( O_2^+ \).

In this case, the integration of \( \mathcal{O}_1 \) and \( \mathcal{O}_2 \) over \( \mathcal{O}_0 \) is done in two steps. First,
we have to rename the design views so that the renamed views \( \mathcal{O}_1' \) and \( \mathcal{O}_2' \) share
a name if and only if there is a common origin in the reference model \( \mathcal{O}_0 \). Then
the construction of the integrated system view can be done by componentwise
set-theoretical union of \( \mathcal{O}_1' \) and \( \mathcal{O}_2' \). We assume that the names in the design
views \( \mathcal{O}_1 \) and \( \mathcal{O}_2 \) are disjoint which can be ensured by qualification with the
name of the view (like \( \mathcal{O}_i.name \)). The view relations \( v_i : \mathcal{O}_0 \rightarrow \mathcal{O}_i \) are given
by \( \mathcal{O}_0 \xrightarrow{\text{ren}_i} \mathcal{O}_i' \subseteq \mathcal{O}_i \) in Figure 1.9.
1.5. INTEGRATION OF VIEWS

Figure 1.9: Integration of the design views $O_1$ and $O_2$ to the system model $O_3$.

![Diagram showing the integration of $O_1$ and $O_2$ to $O_3$.]

clerk doTransaction

![Diagram showing the rule for a transaction.

Renaming of Views The system $O'_1$ and the renaming $O_1 \xrightarrow{ren'_1} O'_1$ are obtained by extending the renaming $O'_0 \xrightarrow{ren'_0} O_0$ to $O_1$. More precisely, $ren'_1$ agrees with $ren'_0$ on $O'_0$, i.e., $ren'_1|_{O'_0} = ren'_0$, and is minimal in the sense that nothing else is renamed, i.e., the renaming is the identity on $O_0 \setminus O'_0$. The renamed system $O'_1$ becomes an extension of $O_0$. In a similar way we obtain $O'_2$ with $O_0 \subseteq O'_2$ and $O_2 \xrightarrow{ren'_2} O'_2$ by extending the renaming $ren'_2$ to $O_2$. This renaming is always possible and can be done automatically.

Now, the integrated view $O'_3$ can be constructed as union of the renamed views $O'_1$ and $O'_2$ in Figure 1.9.

Construction of Integrated View The open graph transformation system $O'_3 = (TG'_3, P'_3, \pi'_3, O_3)$ is obtained by forming the union of the type graphs $TG'_3 = TG'_1 \cup TG'_2$ and of the sets of rule names $P'_3 = P'_1 \cup P'_2$ of $O'_1$ and $O'_2$, respectively. For the rule $\pi'_3(p')$ associated with a rule name $p' \in P'_3$ we distinguish three cases:

- If $p' \in P'_1 \setminus P'_2$ then $\pi'_3(p') = \pi'_1(p')$, i.e., the rule of $O'_1$ is inherited
- If $p' \in P'_2 \setminus P'_1$ then $\pi'_3(p') = \pi'_2(p')$, i.e., the rule of $O'_2$ is inherited
Figure 1.11: Integration of the rules `customer.openAccount` and `clerk.makeAccount` to `common.newAccount`
If \( y' \in P'_3 \cap P'_4 \) then \( \pi'_3(y') = L'_1 \cup L'_2 \rightarrow R'_1 \cup R'_2 \) is an integrated rule obtained by componentwise (non-disjoint) union of the left- and right-hand sides of \( \pi'_i(y') = L'_i \rightarrow R'_i \) for \( i = 1, 2 \).

The open types \( O'_3 = (O'_3^-, O'_3^+) \) are obtained as intersection of the open types of \( O_1 \) and \( O_2 \), that is, \( O'_3^- = O_1^- \cap O_2^- \) and \( O'_3^+ = O_1^+ \cap O_2^+ \).

Then, the integrated view \( O'_3 \) may be renamed to \( O_3 \) via \( \text{ren} \). The view relation \( v'_3 \) is obtained by composing the view relation \( O_1 \underset{\text{ren}}{\leftrightarrow} O_3 \subset O'_3 \) with the renaming \( \text{ren} \) (which is a special view relation as well). In a similar way we obtain view relation \( v'_2 \).

**Example 1.5**

The integrated rule \texttt{common.newAccount} is constructed in Figure 1.11 as the union of the rules \texttt{openAccount} and \texttt{makeAccount} of the customer’s and the clerk’s view. It synchronizes 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 \texttt{Customer} and the \texttt{Bank} object. An application of this rule shows the combined effects of its constituents, where the action of the common subrule \texttt{common.newAccount}, the creation of the \texttt{Account} object, is performed only once.

The rule \texttt{clerk.doTransaction} shown in Figure 1.10 describes the completion of a transfer transaction. The integrated system model \texttt{Bank} in Figure 1.12 also contains the other rules of the customer’s and the clerk’s view, which are not synchronized. Note that not only the customer’s view contains a rule \texttt{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 1.12.) On the other hand, the rules \texttt{openAccount} and \texttt{makeAccount} represent the same operation (despite their different names) since they both stem from the same rule \texttt{common.newAccount}. They are both renamed to \texttt{common.newAccount} in the renaming step of the construction.

The integration of the open type declarations is shown in the same figure. Notice that, in this example, all types are protected in the system model, that is, the specification is considered as complete. In general, one may think of situations where this is not the case, e.g., if some interaction with other banks shall be modeled.

In order to understand better the constraints for the definedness of the construction, suppose that the \texttt{Transaction} type is protected in the customer’s view. In this case, it would be impossible to extend this view by a rule like
Figure 1.12: Integration of the customer's and the clerk's view.
1.5. INTEGRATION OF VIEWS

doTransaction from the clerk’s view since the resulting relation would violate
the condition for new rules in the definition of extensions.

More generally, we may have the following situations:

- It may be the case that “semantically the same” concept is described
  in \(O_1\) and \(O_2\) using different names (like openAccount and makeAccount
  above). This relationship between \(O_1\) and \(O_2\) is only understood (and
  may be taken into account by the integration) if both names have a
  common source in the common view \(O_0\) (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 illus-
  trates also the difference between a synchronized 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 synchronized 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 \(O_1\) and \(O_2\) in order
  to describe “semantically different” concepts (e.g., doTransaction in the
  customer’s and the clerk’s view). This does not cause any problem in
  our approach since we assume that the names are qualified, i.e., Cust-
  omer.doTransaction and Clerk.doTransaction

- In order to represent shared knowledge of \(O_1\) and \(O_2\), 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 propa-
  gated 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 in-
  stead which is also based on the reference model and specifies the sharing
  between \(O_1\) and \(O_2\). The result is a hierarchy of views. Scenarios of more
  than two views are discussed at the end of this section.

- A similar observation holds if a rule of \(O_0\) is extended in \(O_1\) and \(O_2\) 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.
Figure 1.13: Derivation sequence in the system model of the bank
1.5. INTEGRATION OF VIEWS

Example 1.6
Figure 1.13 shows a derivation sequence that models the same operations of Figure 1.6 from the bank’s point of view. The common new Account operation is a synchronized action of a customer and the clerk. The parallel Customer do Transaction and Customer get Balance operation is performed, while the clerk has an idle step. The second Customer do Transaction and Customer get Balance are complemented by two Clerk do Transaction 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.

Notice that due to the constructed frame conditions, the semantics of the system model is restricted to the classical, closed behavior.

Integration of Multiple 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 generalized 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 → 3 → 23 compose to 0 → 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 $3'$ may be introduced, that extends 0 by the commonly used concept such that the view relations 0 → 1 and 0 → 2 are “redirected” over $3'$. 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, generalize the construction to three (or more) views and more complex kinds of diagrams.

**Remark:**

The construction of view integration generalizes a construction of parallel composition of graph grammars [25] towards open graph transformation systems with loose semantics. In [8] this construction is described as a pushout of two morphisms of open graph transformation systems corresponding to the given view relations. The gluing of two rules over a common subrule is formalized by the well-known construction of amalgamation [26]. Finite colimits in the category of open graph transformation systems (if they exist) model the integration of the views of an arbitrary (finite) diagram.

1.6 Conclusion

In this section we presented a view-oriented approach to the modeling of concurrent and reactive systems which is based on the concept of open graph transformation systems.

**Summary**

The view-oriented approach is described by the following characteristic features:
1.6. **CONCLUSION**

- Separate views of a system sharing a common reference model are represented by open graph transformation systems.

- The operational semantics of a view is a loose one, in contrast to the semantics of the complete system model. It may be constrained by explicit frame conditions in the form of open types.

- The views are kept consistent by a model manager extending the reference model whenever new dependencies occur in the development process. In case of more than two views, additional abstract views may have to be introduced which leads to a hierarchy of views.

- Using the reference model, consistent views can be integrated automatically.

The concepts of view and view integration are formalized in [6] using notions and results from the theory of typed graph transformation systems.

**Outlook** In [6], this approach is extended by a temporal logic for graph transformation which is used for expressing both control conditions in the form of regular expressions and safety and liveness properties of views. The control conditions can be interpreted as a basic concept of programmed graph transformation. This allows to derive semantical conditions for the extension and integration of complex behaviors, which is not yet sufficiently studied in the graph transformation context. Moreover it is shown in [6] that temporal properties can be verified in a compositional way by analyzing separately the views of a system and deriving the properties of the complete system model from the local properties.

In order to prove the applicability of the approach to more realistic examples, our view-oriented technique has to be implemented by a specification language based on graph transformation systems and corresponding tools. Good candidates for this are PROGRES [17] where a related construction is already used for merging different versions of a document [27], but also the more recent AGG [19] and Fujaba [18].

Finally let us notice that the basic concepts of this chapter like loose semantics, view relation, etc. are interesting also independently of the view-based specification approach. In the next chapter [28] we argue that similar ingredients can be used to build graph transformation modules where, e.g., the exported features form a view of their implementation in the body.
References


1.6. CONCLUSION

15. J. Jahnke and A. Zündorf. Using graph grammars for building the VAR-LET database reverse engineering environment. In this volume.
17. A. Schürr, A.J. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In this volume.
Index

abstract view, 6, 25
construction of view integration, 15
definedness of view integration, 18, 21
derivation step, 10
design view, 2
extension, 13
inconsistencies between views, 4
multiple view integration, 6, 25
open graph transformation system, 10
open types, 10
parallel rule, 11
reference model, 4
renaming, 13, 18
rule, 9
transition, 10
typed graphs, 8
user view, 2

view, 3
view integration, 5, 17, 21, 26
view relation, 14, 17