Tutorial Introduction to Graph Transformation: A Software Engineering Perspective

Luciano Baresi[1] and Reiko Heckel[2]

[1] Politecnico di Milano, Italy, baresi@elet.polimi.it
[2] University of Paderborn, Germany, reiko@upb.de

Abstract. We give an introduction to graph transformation, not only for researchers in software engineering, but based on applications of graph transformation in this domain. In particular, we demonstrate the use of graph transformation to model object- and component-based systems and to specify syntax and semantics of diagram languages. Along the way we introduce the basic concepts, discuss different approaches, and mention relevant theory and tools.

1 Introduction

Graphs and diagrams are a very useful means to describe complex structures and systems and to model concepts and ideas in a direct and intuitive way. In particular, they provide a simple and powerful approach to a variety of problems that are typical to software engineering [41]. For example, bubbles and arrows are often the first means to reason on a new project, but also the structure of an object-oriented system or the execution flow of a program can be seen as a *graph*.

The artefacts produced in order to conceptualize a system are called *models*, and *diagrams* are used to visualize their complex structures in a natural and intuitive way. In fact, for almost each activity of the software process, a variety of visual diagram notations has been proposed. We can mention, for example, state diagrams, UML, Structured Analysis, control flow graphs, architectural languages, function block diagrams, and several others. Besides having plenty of general-purpose notations, we should also take into account the many domain-specific customizations that provide both dedicated notation elements and special-purpose interpretations.

If graphs define the structure of these models, graph transformation can be exploited to specify both how they should be built and how they can evolve. Although applications of graphs and graph transformations abound and established foundations are available, the knowledge of the formal and conceptual underpinnings is not widely spread among software engineers. Frequently, this leads to ad-hoc solutions to problems that are already well understood in a more general context.

The definition and implementation of visual modeling techniques poses new problems, when compared to programming or formal specification languages:

As a first problem, most established techniques for language definition, like the denotational [88] or the operational [77] approach are intrinsically based on terms (abstract syntax trees) as representation of the structure of the language. This results from the use of context-free grammars (e.g., in Backus-Naur form [3]) for defining their textual syntax. However, as most diagram languages have a *graph-like structure*, such techniques are not readily applicable.

A second problem is the number and diversity of modeling languages and dialects that are currently in use, and the rate in which new ones are proposed to fit the particular needs of certain problem areas. To provide language definitions and implementations for these notations within reasonable resource constraints, meta-level techniques and tools are required which allow, e.g., to generate implementations of languages, like model editors, compilers, or analysis tools, from high-level specifications.

The third problem is the consistency of models consisting of interrelated submodels for different aspects and at different levels of abstractions [38]. Generally speaking, an inconsistent model does not have any correct implementation because the requirements expressed by different submodels are in conflict. To deal with such problems in a systematic way, it is essential to understand the relationships of submodels at a semantic level. Besides techniques and tools to define semantics, this requires semantic domains which are able to capture the essentials of today's systems including aspects like object-orientation, software architecture, concurrency, distribution, mobility, etc. Also here, a general tendency is the shift from hierarchical, tree-like to graph-like structures which are, moreover, dynamic to reflect, for example, the behavior of mobile systems or architectural reconfiguration.

In this tutorial paper, we demonstrate how graph transformation techniques can contribute to solve the above problems. We start (Section 2) with an informal introduction to the basic concepts of graph transformation (like graph, rule, transformation, etc.), we discuss semantic choices (like which notion of graph to use; how to put labels, attributes, or types; or what to do with dangling links during rewriting, etc.) and mention the different ways for formalizing the basic concepts. After introducing the theory, we exemplify (Section 3) what can be done through some examples. To this end, we distinguish between the use of graph transformation as:

- *semantic domain* to supply a specification language and semantic model for reasoning on particular problems: for example, the consistency between functional requirements and software architecture models in concurrent and distributed systems.
- *meta language* to supply a means to formally specify the syntax, semantics, and manipulation rules of visual diagrammatic languages.

We then continue (Section 4) with the review of the main branches of the theory of graph transformation relevant to the applications discussed. The last step, to let possible users really exploit graph transformation to reason on and solve their problems, is a brief summary (Section 5) of available tools to clearly

explain the support offered to (fully) automate proposed and foreseen solutions. We conclude the paper (Section 6) by identifying future research directions with the hope that the graph transformation community can more and more serve as technology provider to other communities that need this formal basis to improve their current practice.

2 Foundations of Graph Transformation

This section is intended as a first introduction to graph transformation, its basic notions, different approaches, and more advances concepts. Graph transformation has evolved in reaction to shortcomings in expressiveness of classical approaches to rewriting, like Chomsky grammars and term rewriting, to deal with non-linear structures. The first proposals appeared in the late sixties and early seventies [76, 70, 83, 79, 95]. They were concerned with rule-based image recognition, translation of diagram languages, or efficient implementation of λ-reduction, based on graph-like structures.

Fundamental approaches that are still popular today include the algebraic or double-pushout (DPO) approach [31, 17], the node-label controlled (NLC) [57, 32], the monadic second-order (MSO) logic of graphs [19, 20], and the PROGRES approach [86, 87] which represents the first major application of graph transformation to software engineering [33, 73].

Below we introduce a simple form of graph transformation which shall serve as a basis for further discussion, i.e., a set-theoretic presentation of the double-pushout approach [31]. Then, we will discuss alternatives and extensions to this approach.

2.1 A basic formalism

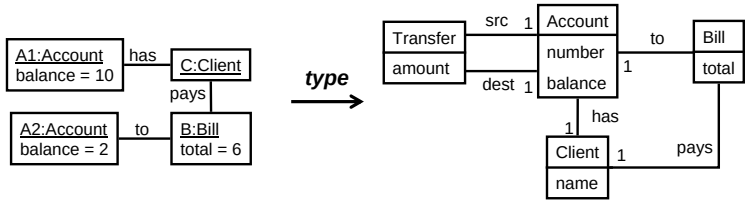


Fig. 1. Object diagram (left) typed over class diagram (right)

Graphs. 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. In object-oriented modeling graphs occur at two levels: the type level (given by a class diagram) and the instance level (given by all valid object diagrams). This

idea can be described more generally by the concept of *typed graphs* [16], where a fixed *type graph* TG serves as abstract representation of the class diagram. Its object diagrams are graphs equipped with a structure-preserving mapping to the type graph, formally expressed as a *graph homomorphism*.

Figure 1 shows examples of an object and a class diagram in UML notation [75] modeling some data objects of a banking application. The (instance graph representing the) object diagram on the left can be mapped to the (type graph representing the) class diagram by defining $type(o) = C$ for each instance $o : C$ in the diagram. Extending this to links, preservation of structure means that, for example, a link between objects o_1 and o_2 must be mapped to an association in the class diagram between $type(o_1)$ and $type(o_2)$. By the same mechanism of structural compatibility we ensure that an attribute of an object is declared in the corresponding class, etc.

Rules and transformations. A *graph transformation rule* $p : L \to R$ consists of a pair of TG-typed instance graphs L, R such that the union $L \cup R$ is defined. (This means that, e.g., edges that appear in both L and R are connected to the same vertices in both graphs, or that vertices with the same name have to have the same type, etc.) The left-hand side L represents the pre-conditions of the rule while the right-hand side R describes the post-conditions.

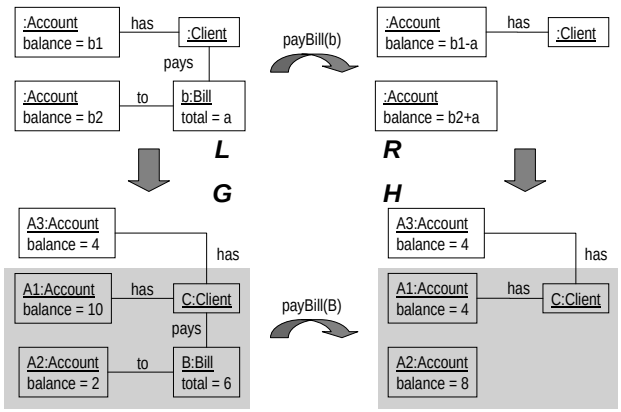


Fig. 2. A sample transformation step using rule payBill

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

- $o(L) \subseteq G$ and $o(R) \subseteq H$, i.e., the left-hand side of the rule is embedded into the pre-state and the right-hand side into the post-state, and

– $o(L \setminus R) = G \setminus H$ and $o(R \setminus L) = H \setminus G$, i.e., precisely that part of G is deleted which is matched by elements of L not belonging to R and, symmetrically, that part of H is added which is matched by elements new in R.

Figure 2 shows an application of the graph transformation rule payBill modeling the payment of a bill by transferring the required amount from the account of the client. Operationally, the application is performed in three steps. First, find an occurrence $o|_L$ of the left-hand side L in the current object graph G. Second, remove all the vertices and edges from G which are matched by $L \setminus R$. Make sure that the remaining structure $D := G \setminus o(L \setminus R)$ is still a legal graph, i.e., that no edges are left dangling because of the deletion of their source or target vertices. This is made sure by the *dangling condition* [31][1] which is checked for a given occurrence before the application of the rule. If the condition is violated, the application is prohibited. Third, glue D with $R \setminus L$ to obtain the derived graph H. In Fig. 2 the occurrence of the rule is designated by the shaded objects and links in the transformation.

2.2 Variants and extensions

Graphs. The graphs introduced above are often referred to as *multi-graphs* because they allow for multiple parallel edges of the same type. An (untyped) multi-graph can be regarded as a two-sorted algebraic structure. Therefore, formalisms dealing with such graphs are known as *algebraic approaches* [17, 28].

The more common alternative is to consider graphs as *relational structures* $G = (V, E)$ with $E \subseteq V \times V$. In this case, there exists at most one edge between a given pair of vertices, which represents a restriction of the algebraic notion. Other variations include, for example, *undirected graphs* which are formalized as directed graphs closed under symmetry of edges, or hypergraphs, where each edge may have a sequence of source and/or target vertices. *Hypergraphs* are sometimes encoded as bipartite graphs.

Typed graphs as above combine the association of labels to nodes and edges with structural constraints expressed by the graph structure on the label sets. In *labelled graphs*, the label sets do not have any additional structure. For example, if vertices and edges are labelled over separate label alphabets L_V, L_E, the relational variant is given by (V, E, lv) with $E \subseteq V \times L_E \times V$ and $lv : V \to L_V$. Attributed graphs are graphs labelled over pre-defined abstract data types, like strings or natural numbers [65]. An example is given in Fig. 1 above where, e.g., the vertex B:Bill has an attribute total = 6.

It is interesting to note that many of the notions and constructions in the theory of graph transformation can be (and have been) described for a wide range of the above structures. Formally, this is reflected in approaches based on category theory like [27, 30, 10], which can be instantiated to a variety of

[1] Indeed, the notion of graph transformation introduced here is a set-theoretic presentation of the categorical double-pushout (DPO) approach [31], which owes its name to the fact that a transformation step may be characterized as a pair of gluing diagrams (pushouts) of graphs.

different graph models. In the following we stick to our simple model of typed and attributed graphs.

Rules and transformations. Co-related with the two notions of graphs, the algebraic and the relational one, are two fundamentally different approaches to rewriting which have been referred to as the *gluing* and the *connecting* approach. They differ for the mechanism used to embed the right-hand side R of the rule in the context D (i.e., the structure left over from G after deletion). In a gluing approach like [31, 64], graph H is formed by gluing R with the context along common vertices. In a connecting approach like NLC [57], the embedding is realized by a disjoint union, with additional edges connecting graph R with the context. To establish these connections, *embedding rules* are specified as part of the transformation rule to determine the connections of the right-hand side from those of the left-hand side in the given graph.

Most practical approaches use combinations of gluing and connection [86, 39]. The first is more efficient in implementations since it allows items in the left-hand side to be preserved by the rule. Otherwise, this option, which is obviously desirable, has to be simulated by deletion and re-generation of the items concerned. On the other hand, the connection approach based on embedding rules is more expressive because it allows to deal with unknown context. In fact, operations like "turning all outgoing edges of a certain vertex into ingoing ones" can only be specified in a gluing approach if the number of such edges is known in advance or specific control structures are used. Instead, such operations are typical to connecting approaches like node-label controlled (NLC) graph grammars [57]. A well-known example of the limitation of gluing approaches has been discussed above: Deletion of a vertex is prohibited by the dangling condition if an edge in the context is attached to it. In the algebraic approach, this problem has been solved in [64] by implicitly deleting all edges whose source or target vertex is deleted.

The most common restriction imposed on graph transformation rules is *context-freeness* in the context of graph grammars: A rule is context-free if it has only one vertex or edge in its left-hand side. Similarly to context-free Chomsky grammars, context-free graph grammars are interesting because of their simplicity, combined with sufficient expressive power to describe many interesting graph languages [22, 32].

Control structures and constraints. Besides approaches that increase (or restrict) the expressiveness of individual rules, a major concern of a rule-based approach is to control the application of rules. This includes programming language-like control structures [85, 61] and application conditions [90, 26, 45] restricting the admissible occurrences for a given rule.

Moreover, different kinds of constraints have been proposed, which restrict the class of graphs and can therefore be used to control the transformation process implicitly by ruling out transformations leaving this class. In their simplest form, such constraints can be compared to cardinality restrictions in class diagrams like in Fig. 1 where, e.g., the association to connects every Bill object

to exactly one Account object. More complex constraints deal with the (non-) existence of certain patterns, including paths, cycles, etc. Constraints on graphs are formalized in terms of first- or higher-order logic [85, 20], by means of linear inequations [40], or as graphical constraints [51, 11].

3 Modeling and Meta Modeling

As evident from the title, we distinguish between the application of graph transformation to the modeling of individual systems, and to the specification of syntax and semantics of visual modeling languages. However, in both cases, similar notations and tools may be employed, and we refer to the latter activity as meta modeling in order to stress the fact that a language may be defined by the same techniques that are also used to model a system.

3.1 Modeling with graph transformation

This section illustrates the use of graph transformation for modeling functional requirements and dynamic change of software architectures. Then, these two aspects of a model are related by means of a common meta model. Along the way, some more advanced concepts and constructions based on graph transformation systems are introduced.

Object dynamics. Functional requirements are often presented in terms of use cases, i.e., services a system shall provide to its users. Every such use case provides a more detailed description of its pre- and post-conditions and of the interactions required to perform the respective service. In [47] typed graph transformation systems have been proposed as a way to specify use cases in a visual, yet formal way with the additional benefit of an executable specification. An example for such a specification has been given in Fig. 2 with the rule payBill and its application.

The idea of a rule-based modeling of operations by means of graphical pre- and post-conditions can be traced back to several sources. PROGRES [87] provides a database-oriented programming language and environment based on graph transformation. In the object-oriented FUSION method [14] actions are specified by snapshots of the object configuration before and after the operation. CATALYSIS [23] advocates the use of UML collaboration diagrams for this purpose, an approach which has also been adopted and implemented in the FUJABA method and tool [59]. A formal relation between collaboration diagrams and graph transformation has been established in [50].

Graph transformation rules like payBill provide a high-level specification of functional requirements in terms of pre and post conditions, abstracting from intermediate actions and states. If the more fine-grained structure of an operation is of interest, a rule may be decomposed into more elementary steps. For example, the rule payBill(b) may be decomposed sequentially as payBill(b) =

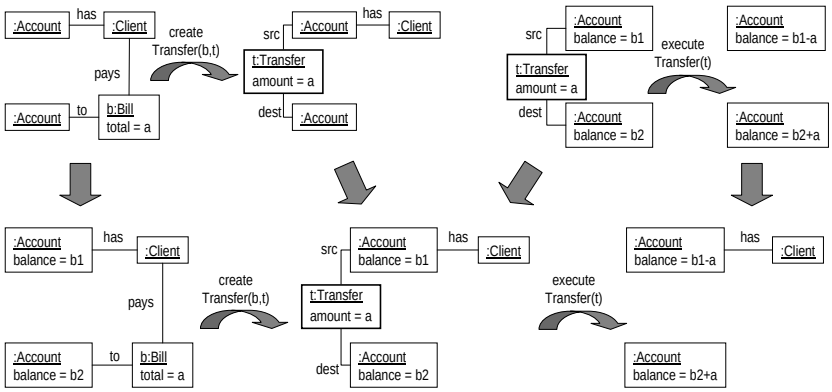


Fig. 3. Rule payBill(b) derived from createTransfer(b,t); executeTransfer(t)

createTransfer(b,t); executeTransfer(t), where the relation between the two elementary rules is specified by the abstract parameters b: Bill and t: Transfer. This decomposition is shown in Fig. 3 where createTransfer(b,t) is applied to the left-hand side of payBill(b) and executeTransfer(t) is applied to the resulting graph, the result being the right-hand side of the original rule.

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

$$G \overset{\mathsf{payBill(B)}}{\Longrightarrow} H$$

if and only if there are a graph X and transformations

$$G \overset{\mathsf{createTransfer(B,T)}}{\Longrightarrow} X \overset{\mathsf{executeTransfer(T)}}{\Longrightarrow} H.$$

Thus, we may think of the composed rule payBill as a two-step transaction. The desired semantic condition can be checked by composing the elementary rules as it is shown in Fig. 3. First, createTransfer(b,t) is applied to the left-hand side of payBill(b) at the occurrence determined by the parameter b: Bill. Then, executeTransfer(t) is applied to the resulting graph, this time determining the occurrence through the parameter t:Transfer. The result of this second step has to be isomorphic to the right-hand side of the original rule. In this case, it can be shown that the desired consistency condition holds [17].

Architectural change. A quite different interpretation of nodes and edges in a graph is in terms of components and connectors of a software architecture model, see e.g., [89, 60]. Here, instance and type graphs model, respectively, configurations and architectural styles, while graph transformation rules specify the reconfiguration of architectures.

An example of a type graph representing an architectural style for distributed banking applications is given in Fig. 4 on the left. The style consists of compo-

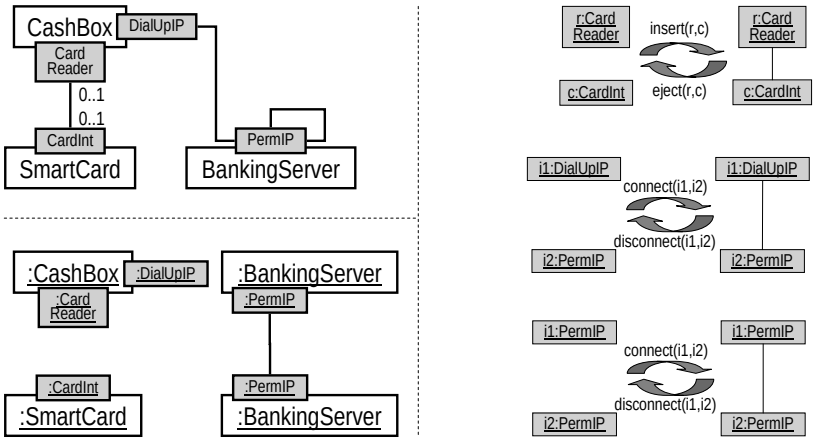


Fig. 4. Architecture of a distributed banking system: a configuration (lower left) typed over an architectural style (upper left) and reconfiguration rules (right).

nents like electronic cashboxes, smartcards, and banking servers—all communicating through the Internet, telephone lines, or card readers. A sample configuration of this style is shown in the lower left of Fig. 4. It consists of two instances of the banking server component, and one instance each of cashbox and smart card, where only the two banking servers are currently connected.

Beside this declarative approach to the specification of architectural styles, constructive approaches based on graph grammars have been proposed. Here the idea is to generate the set of all eligible configurations from a given initial one by means of (often context-free) production rules [63, 52]. Context-freeness, i.e., the restriction to one component (vertex or (hyper)-edge) in the left-hand side of rules, corresponds to the recursive refinement of components by configurations of sub-components.

The main benefit of graph transformation for describing software architectures is the ability to model dynamic reconfigurations in an abstract and visual way. Some approaches [42, 63, 96, 93] assume a global point of view when describing reconfiguration steps which, in a real system, would correspond to the perspective of a centralized configuration management. In a distributed system, the existence of such centralized services cannot be taken for granted. Therefore, [53, 52] model reconfiguration from the point of view of individual components which synchronize to achieve non-local effects. Here, locality corresponds to context-freeness, that is, a rule is local if it accesses only one component (or connector) and their immediate neighborhood. Synchronization of rules is expressed in the style of process calculi like CCS [67] or CSP [54], see [53] for a more detailed discussion.

The rules in Fig. 4 do not require any synchronization. In the upper left, the transformation from left to right establishes a connection between the interface of a smart card and a card reader, modeling the insertion of the card into the reader.

The inverse transformation models the disconnection (ejection) of the card from the reader. In a similar way, the two lower rules model the establishment and release of Internet connections.

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

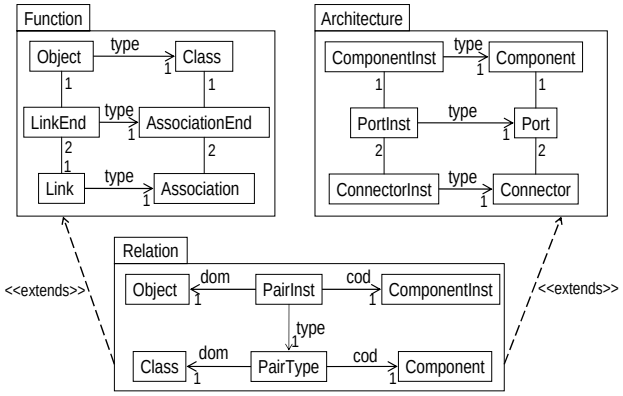


Fig. 5. Meta model relating the functional and architectural view

The static/syntactic integration is achieved by defining the abstract syntax of both the functional and the architectural view by means of a *meta model*. Generally speaking, a meta model is a model for a modeling language expressed within a simple subset of the language itself. This technique has become popular with the UML whose abstract syntax is specified by a subset of UML class diagrams [75]. This subset is determined by the meta object facility (MOF) specification [74] which is also referred to as a meta-meta model because it has meta models as instances whose instances, in turn, are models.

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

Consider, for example, the meta model in Fig. 5. It consists of three packages, whose top left one specifies the functional view of the system by means of meta classes Class and Association as well as Object and Link, etc. whose relation is

given by the meta association type. Therefore, every instance of this meta model represents a pair of a class diagram and an associated object diagram.

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

The meta model allows a uniform representation where elements of different submodels are represented as vertices of the same *abstract syntax graph* [2], i.e., an instance of the meta model. Based on this uniform representation, the different submodels can be related by extending the meta model in Fig. 5 with a package extending the other two packages to define a relation between the functional and the architectural view [58]. Such a relation consists of pairs of, respectively, objects and component instances or classes and components, where instance-level pairs are associated to type-level pairs by a meta association type: To relate an object to a component instance, a corresponding relation between the respective class and component is required.

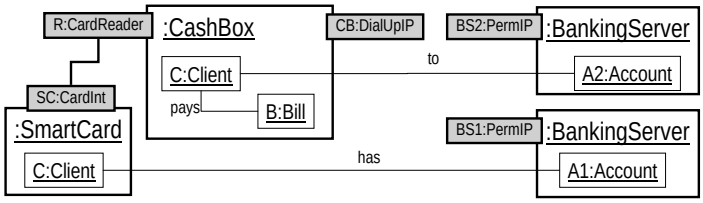


Fig. 6. Sample configuration after transmission of client data

In our example, the relation between classes and components shall be given by { (SmartCard, Client), (CashBox,Transfer), (CashBox,Bill), (CashBox,Client), (BankingServer,Account), (BankingServer,Transfer) }. This information could be given in diagrammatic form, like in UML component diagrams where the relation is expressed by containment or dependencies. Here we simply list it as a set of pairs. A diagrammatic presentation of the object-component instance relation is given in Fig. 6 by means of containment, combining information from all three meta model packages.

Based on the uniform representation of object structures and architectures as meta model instances we may now present the rules specifying functional requirements and architectural reconfigurations as graph transformation rules typed over the corresponding packages of the meta model. However, this is nothing more than a disjoint union of models, turning two distinct models into yet unrelated submodels of the same overall model. To map the functional requirements on a given architecture, we assign responsibilities for operations to components (or sets of components) by specifying the location of the objects in the cor-

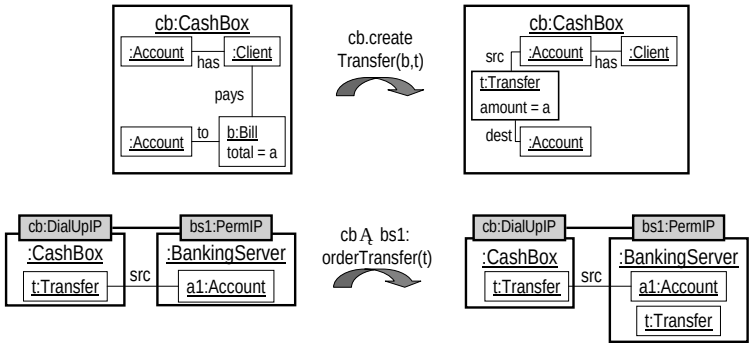


Fig. 7. Operation createTransfer executed on a CashBox component (top) and the effect of transmitting a Transfer object (bottom)

responding rules. Figure 7 shows this step for the rule createTransfer which is associated with the CashBox component.

An example of a communication rule is shown in the bottom of Fig. 7. It models the transmission of a message orderTransfer(t) from a cashbox holding an object t:Transfer to the banking server holding the corresponding source account. The effect of this transmission is the replication of the object. We denote by using the same object identity t that the object is logically shared between the two components, that is, the banking server has access to all references and attributes.

Related work. The integration of functional and architectural models is implicit in many works on graph transformation for specifying software architectures. In [93] an approach based on two-layered distributed graphs [91, 92] is presented. The upper layer represents the network graph of a distributed system whose nodes are attributed with object graphs on the lower layer. Two-level rules are then used to manipulate this structure. This approach is conceptually close to ours, but it does not foresee any means for functional decomposition of rules.

Another approach integrating functional and architectural aspects [96] uses the coordination and programming language CommUnity to specify computations and graph transformation rules to model architectural reconfiguration. The use of a programming notation is appropriate at the later design or implementation stages, but for requirements specification and analysis we prefer visual notations.

Recently, the transformation of hierarchical graphs, of which our approach presents a special case, has received much attention, see e.g. [80, 36, 21, 13, 68]. In particular [13, 68] focus on the separation of *connection* from *hierarchy*. That is, hierarchical graphs can be seen as graphs with two kinds of structures, for expressing links between objects or components, and for modeling hierarchy.

3.2 Meta modeling with graph transformation

In Section 3.1, a meta model has been used to specify the abstract syntax of static diagrams, like class and object diagrams, and graph transformation rules were employed to model, e.g., computations on object structures. More generally, meta modeling techniques may be used to define the concrete syntax, abstract syntax, or semantics of any modeling notation, be it dedicated to static or dynamic aspects [58]. The key idea is that these structures can be specified by means of class diagrams. Then, the crucial question is, how to relate the three levels in order to define syntax and semantics of a language. In this section, we discuss the use of graph transformation systems to specify mappings between concrete and abstract syntax and between abstract syntax and semantics, as well as for defining operational semantics based on a direct interpretation of models at the level of abstract syntax.

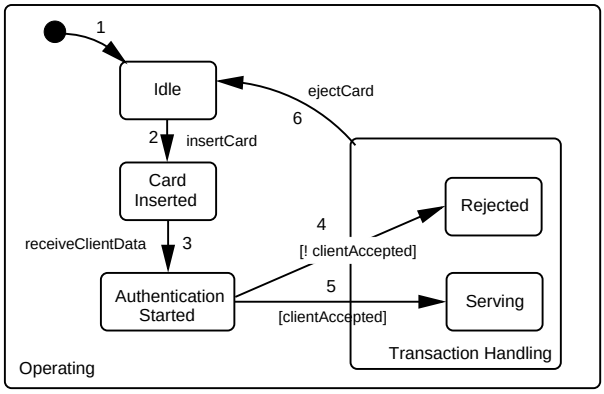


Fig. 8. The CardReader view of the CashBox behavior as a protocol statechart

To make the discussion more concrete, we use a simple example of a protocol statechart[2] that models the behavior of the CashBox component as observed through the CardReader interface, both introduced in Section 3.1. If the CashBox is switched on, from the initial state (the black bullet) we enter state Operating, moving to its default state Idle. As soon as the first card is inserted (i.e., event insertCard occurs), the component enters the state Card Inserted and when it receives the client data from the card (event receiveClientData), it moves to state Authentication Started. Here, a choice occurs depending on the data received. If the client is accepted, the component enters the state Serving otherwise the state Rejected. In both cases – this is why we have the OR-state Transaction Handling comprising the two substates – after processing the transaction, the component returns to state Idle as soon as the card is ejected (event cardEjected).

[2] Readers are referred to [46] for an in-depth presentation of the notation.

This diagram, together with its intuitive semantic interpretation is enough to touch all aspects that define a visual notation. More precisely, we can organize it around the three different layers of language definition as shown in Fig. 9. In the remainder of this section, we describe in some detail the transformations on and between these layers represented by the arrows in Fig. 9.

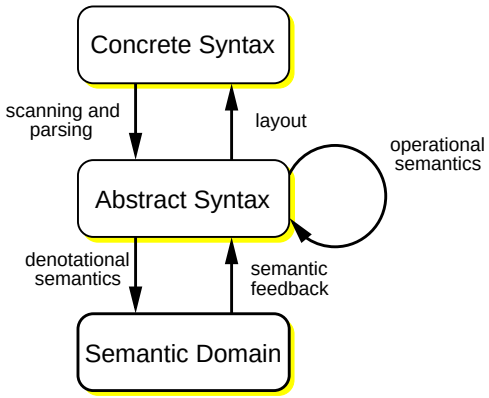


Fig. 9. A layered view of a visual modeling language

Scanning and Parsing. Moving top-down, each language has a *concrete syntax*, which defines how users perceive the different modeling elements supplied by the notation. Since we are thinking of diagrammatic languages, the concrete syntax predicates in terms of bubbles, rectangles, lines, arrows, etc. Each notation element is concretely rendered in terms of the geometrical elements that define its appearance: For example in Fig. 8, states are represented through rectangles with rounded corners, initial states with back bubbles, and state transitions with directed edges. Relationships among these elements can be: a line *connects* two rectangles, a rectangle *contains* other rectangles, or an element *is on the left/right* of another element.

At this level, a graph grammar defines the concrete syntax of the language. It is this grammar that should be employed to *scan* user models. (Notice that we do not yet assign any semantic interpretation to the graphical symbols.) From a practical point of view, the grammar could define also the correct steps that can be done by a user that uses a syntax-directed editor for the supported notation. For example, Minas in [55] proposes a complete hyperedge grammar for *editing* well-formed statechart diagrams, to be fed into the DiaGen tool (see Section 5) for automatically generating a graphical statechart editor. However, the grammar can also be used as a stand-alone definition of the concrete syntax of statecharts.

Formally, the statechart grammar defines all correct Spatial Relationship Hypergraphs (SRHG), that is, hypergraphs with edges like *label*, *rectangle*, *edge*, etc.

and nodes representing the points where the hyperedges are connected. SRHGs through a further set of transformation rules become Reduced Hypergraph Model (HGM). These graphs represent the abstract syntax of the example statechart, i.e., the next layer in our hierarchy of Fig. 9. The *abstract syntax* defines the modeling elements supplied by the notation, without the concrete "sugar", and the relationships among them. Models at this level can be *parsed* to check if they are syntactically correct. Tokens at this level are related to the semantic interpretation, that is, we think of the example of Fig. 8 in terms of states (initial, AND-decomposed, and OR-decomposed) transitions, events, and so on.

This level is comparable to the representation of UML models as instances of the meta model [75]. The main difference is the declarative style of specification in the meta model, which defines well-formedness by means of logic constraints, as opposed to the constructive style of using a grammar to generate models.

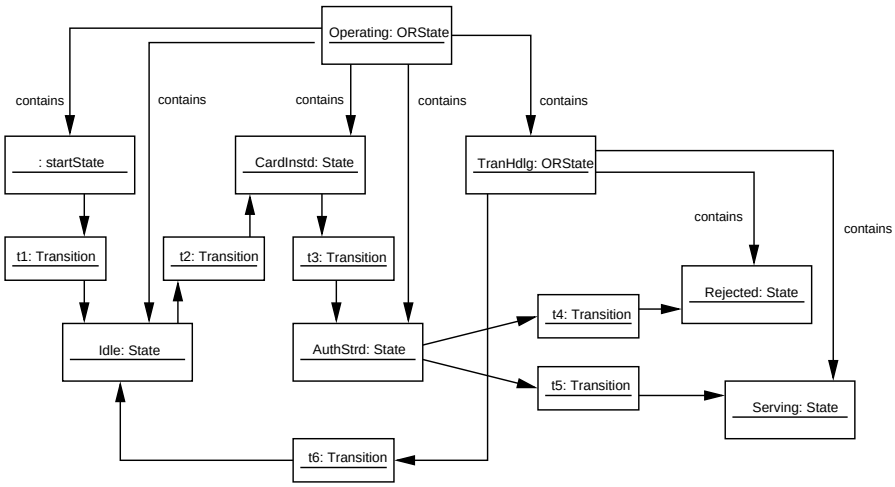


Fig. 10. Abstract syntax graph for the statecharts of Fig. 8

Figure 10 shows a simplified abstract syntax graph for the statechart of Fig. 8. Nodes are instances of a simple metamodel that comprises: startStates, ORStates, States, and Transitions (further details are omitted for the sake of clarity). Edges connect the nodes to render the connections between states and transitions in the statechart diagram. This simple model could easily be constructed through a special-purpose grammar by scanning user-supplied diagrams. In turn, these abstract syntax graphs can be parsed to check if they are syntactically well-formed.

Vice versa, mapping abstract to concrete syntax means that we define the concrete *layout* of models. The grammar in this case defines how abstract concepts should be rendered at the concrete level, but also the correct positioning of each element on the canvas. Special-purpose algorithm for defining the layout of

user models can be implemented using graph grammar productions and textual attributes to compute the coordinates of each graphical symbol.

Operational semantics. Visual models can be given a formal semantics in several different ways. In the case of a class diagram the semantics is given by a set of object diagrams. For behavioral diagrams like statecharts, an operational semantics can be given directly on the abstract syntax of the language, through yet another grammar. In our example of Fig. 8, a formal semantics is desirable to specify precisely the execution of a model. If we think of performing a simple transition, the meaning is clear enough: It moves the *current state* for the source to the target state. But when we think of dealing with multiple events, AND- and OR-decomposed states, histories, etc., then the meaning is not clear anymore. In fact, there are several different statechart dialects, all using essentially the same syntax, but with different rules to execute a model [94].

There are essentially two different ways to define operational semantics by graph transformations. First, graph transformation rules can specify an abstract interpreter for the entire language as proposed, for example, in [34]. Second, each model can be "compiled" into a set of rules. Following the second approach, [62] exploits *structured graph transformation* as means to ascribe UML statecharts with formal dynamic semantics. Roughly, statecharts are translated to structured graph transformation systems that satisfy the well-formedness rules imposed by the notation. Active states are represented as *state configurations* which are (isomorphic to) subgraphs of state hierarchies, that is, tree-like object diagrams that represent instances of the metamodel for the statecharts. Active states change by means of transition firing, specified as graph transformation rules: the left-hand side identifies the current configuration, i.e., the current active state(s), the right-hand side defines the configuration reached by applying the rule (firing the transition), that is, the new set of active states. Transitions for *syntactically correct* statecharts can be generated by applying the rules of the transformation unit *term(S)* presented in [62]: Figure 11 shows the result for some of the transitions of Fig. 11.

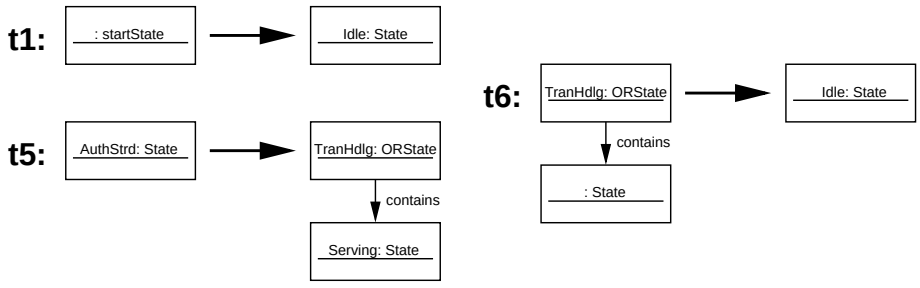


Fig. 11. Some transitions of Fig. 8 as graph transformation rules

Presented rules are self-explaining, but: $t1$ moves the current state from the start state to Idle; $t2$ and $t3$ are similar and are not presented here. $t5$ moves the current state from Authentication Started to the hierarchy Transaction Handling / Serving. Again $t4$ would be similar. $t6$ moves the current state from the hierarchy Transaction Handling / any contained state back to Idle.

Denotational semantics. The last layer of Fig. 9 introduces semantic domains. Semantics defined as a mapping from the abstract syntax into a semantic domain is referred to as *denotational*. In our example, to define the dynamic semantics of statecharts, the semantic domain itself has to provide an operational model—thus operational and denotational semantics occur in combination.

The overall approach is as follows: We choose a *semantic domain*, i.e., a usually simpler formal method whose execution rules are well-established, and define the "behavior" of each abstract syntax element through a suitable *mapping* onto the semantic domain. In this case, the role played by graph transformation depends on the chosen formal method. If it is a textual one, the productions of the grammar that defines the abstract syntax can be augmented with textual annotations to build the semantic representation. More generally, the productions can be paired with those of the textual grammar that specify the semantic models, and the application of a production of the abstract syntax grammar would automatically trigger the application of the paired textual production [79].

For example, if we keep thinking of statecharts and want to define its dynamic semantics through CSP (Communicating Sequential Processes [54]), we can mention [35] where the left-hand side of each rule is how UML-like metamodel instances can be built for statecharts (graph grammar productions); the right-hand side part codes how the corresponding CSP specification must be modified accordingly (textual grammar production). A similar approach is described in [9] where high-level timed Petri nets are used as semantic domain and the rules are pairs of graph grammar productions: The first production defines how to modify the abstract syntax representation, while the second production states the corresponding changes on the functionally equivalent Petri nets.

For example, using the *transformation rules* defined in [9], we could ascribe formal dynamic semantics to the statechart of Fig. 8 through the Petri net of Fig. 12[3].

The hypothesis used to transform the statechart into a functionally equivalent Petri net are: States are directly mapped onto places and state transitions into Petri net transitions. Start states are rendered with marked places. OR-decomposed places imply that their transitions be mapped onto a set of Petri net transitions.

[3] Attributes associated with places and transitions are not shown here since the are not needed to understand the approach, but they would simply make the Petri net more complex.

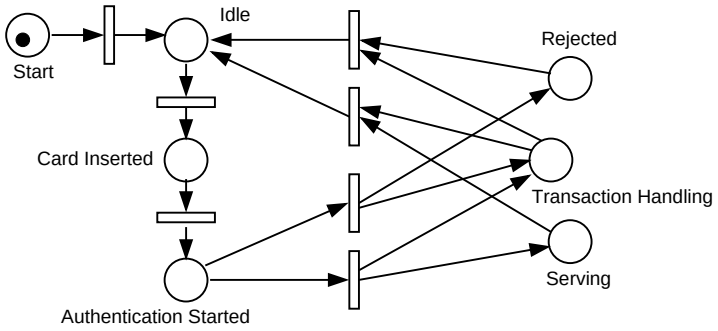
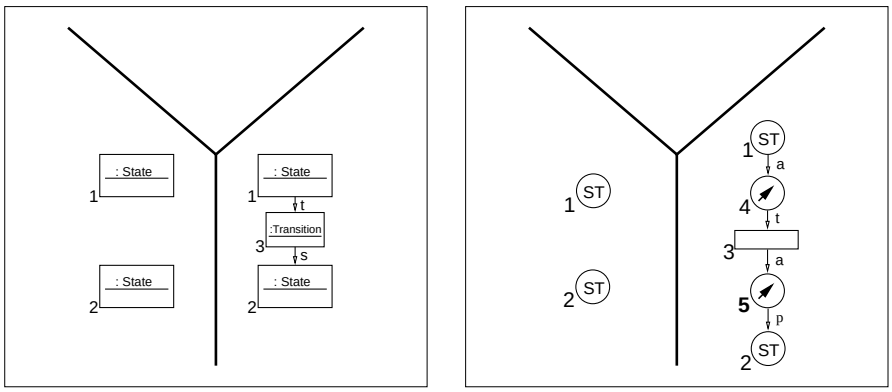


Fig. 12. 8 in terms of Petri nets



(a) Abstract Syntax Model (b) Semantic Model

Fig. 13. A simple transformation rule taken from [9]

A simple transformation rule is presented in Fig. 13. It shows how to connect two SC[4] states through an SC transition. The left production modifies the abstract syntax graph by adding the new SG transition, while the right production modifies the Petri net by adding a PN transition, together with two PN arcs between the two places that correspond to the SC states.

Notice that even if Petri nets are formal and simpler than statechart, the resulting net is more complex than the example diagram. Thus, such a transformation is important to let users ascribe formal semantics to their models, but usually produces formal representations that are more complex and difficult for the reader. More sophisticated mappings could be implemented using triple graph grammars [84], that is, besides the two grammars that define the abstract syntax and the corresponding modifications of the semantic domain, a third grammar states the mapping between the two paired productions explicitly.

Moving from the semantic domain back to the abstract syntax layer (Figure 9), the results produced by executing the semantic representations can be mapped onto the abstract syntax elements. For example, the firing of a PN transition that corresponds to an SC transition could be transformed into a suitable animation of the interested states. Also these transformations, mapping of execution results, can be specified through a suitable grammar. Interested readers can refer to [9] for a couple of interesting examples.

4 Theory

As mentioned in Section 2, graph transformation systems generalize other notions of rewriting, like Chomsky grammars, Petri nets, and term rewriting. Such connections have inspired the development of the theory of graph languages, concurrency, and term graph rewriting, respectively. In this section we briefly discuss these developments and give some references to relevant literature.

4.1 Graph languages

Chomsky grammars have inspired the notion of *graph grammar*, i.e., a graph transformation system with a start graph that is meant to describe graph languages. In particular, a theory of context-free graph grammars [22, 32] has been developed, as well as parsing algorithms for graph languages [81, 69, 12] that are relevant to the application of graph grammars to describe the syntax of visual languages [66].

For example, an important (negative) result of graph grammar theory states that, unlike in the case of programming languages, context-free graph grammars are not sufficient to generate even approximations of many interesting diagram

[4] SC and PN are used here to clarify if we are referring to statechart or Petri net elements. The event associated with the SC transition is not considered here for the sake of simplicity.

languages. As an example, it can be shown that the simple language of statecharts is not generated by any context-free graph grammar.[5] The same holds for other popular diagram languages used in software development, like object or class diagrams. For this reason, approaches to visual language definitions, like [81, 69], have to resort to non-context-free grammars.

4.2 Concurrency

Place-transition Petri nets are essentially rewriting systems on multisets. In this sense, they are like graph transformation systems without edges, and without context [15] if we consider a gluing approach. This view has inspired a variety of developments along the lines of the concurrency theory of nets, like processes and unfoldings, event structures, as well as the notion of typed graphs that we have used in Section 3. These and other concepts in concurrency theory are surveyed in [4] for the gluing approach and [56] for the connecting approach.

The related aim of generalizing to graph transformation systems certain notions of morphisms of Petri nets has influenced the development of structuring and refinement concepts for graph transformation systems. For example, the relation between abstract functional requirements as exemplified by the rule payBill in Fig. 2 and the overall system model, including the architectural aspect, can be described as a refinement relation according to [48, 44] combining an extension of type graphs and a (sequential) decomposition of rules.

4.3 Term graph rewriting

To support the efficient implementation of functional languages, term graph rewriting generalizes term rewriting by replacing trees by rooted DAGS [7, 18, 78, 8]. This has given rise to theoretical questions concerning confluence and termination of (term) graph rewriting to ensure functional behavior [78]. In view of the previous section, such questions are relevant to the use of graph transformation for translating models into formal specifications in terms of Petri nets or CSP, which serve as semantic domains. In fact, in order to be well-defined, a semantic mapping should be a total function. This is also the main motivation of [49] studying confluence of attributed graph transformation.

[5] For hyperedge replacement graph grammars, it is sufficient to observe that the graph language of state diagrams has unbounded connectivity. It follows from the Pumping Lemma (Thm. 2.4.5 in [22]) that such a language cannot be generated by hyperedge replacement. For node replacement grammars, observe that the language includes all finite *square grids*, i.e., all graphs whose nodes can be organized into rows and columns of the same length, and connected by horizontal and vertical edges. Prop. 1.4.8 of [32] states that a language containing infinitely many square grids cannot be generated by a confluent edNCE grammar, the most general form of context-free graph grammar known in the literature. In fact, confluence is a crucial feature of context-free grammars. Approaches like [72, 45] which increase the expressive power of (otherwise) context-free rules by path expressions or application conditions do not fall into this category.

More recently, research on graph transformation approaches has been influenced by the use of term rewriting in the semantics of process calculi [71, 68]. Here terms representing processes, e.g., in the π or ambient calculus, are replaced by process graphs which allow a more direct representation of their structure.

5 Tools

This section briefly describes the main tools that are available to "work" with graphs and graph transformation. They all are good research prototypes that, according to the taxonomy already described in Section 3, can be divided in two main groups: general-purpose environments for modeling graph-centric problems and environments for specifying visual languages. The main representatives of these classes of tools are introduced in the next two sections: Interested readers can refer to [6] for more complete descriptions.

5.1 Graph transformation tools

The first example of graph-oriented modeling environment is PROGRES (*PROgrammed Graph REwriting Systems* [86, 87]). Conceptually, it supplies a graphical/textual language to specify attributed graph structures and graph transformations, parameterized graph queries, graph rewrite rules with complex and negative application conditions, and non-deterministic and imperative programming of composite graph transformations (with built-in backtracking). Pragmatically, the PROGRES environment offers a set of integrated tools to let users models their artifacts. Specifications are produced through a mixed textual/graphical syntax-directed editor (with an incrementally working table-driven pretty printer), but they can also be edited in a fully textual way through an emacs-like text editor. Incrementally, a type-checker detects all inconsistencies with respect to the language's static semantics. Besides being checked, user specifications can be translated into intermediate code and then executed. The intermediate code can be cross-compiled into Modula 2, C, or Java code, to produce "independent" graph manipulation components, and a user interface generator can produce Tcl/Tk code to support rapid prototyping of graph manipulation tools. Tcl/Tk is used also to supply a graph browser to manipulate host graphs and let users apply their PROGRES specifications to particular start graphs.

The tool support offered by AGG [37], which is a rule-based visual language supporting an algebraic approach to graph transformation, is similar. With AGG, the behavior of a system is defined through graph rules, which can have negative conditions, "annotated" with Java code. This means the types of rules' nodes are defined through Java classes and standard Java libraries can be exploited to compute their attributes. The tool environment provides editors for specifying graphs, rules, and the associated Java code, allowing users to play what-if simulations, by specifying the starting graph and the order rules should be applied, but supports also *efficient graph parsing*. This problem is undecidable in general, but it can be solved for restricted classes of graph grammars.

For parsing, AGG requires that users define a so-called *parse grammar* that contains reducing parsing rules, that is, roughly rules whose left-hand and right-hand sides are swapped, and a stop graph. The parsing algorithms are based on back-tracking, but since it has exponential time complexity, it is improved by exploiting critical pair analysis [12].

Moreover, being implemented in Java, AGG, can be used as hidden graph transformation engine for all those Java applications that need to manage and transform graphs.

Quite different is the support offered by FUJABA [1], which is an environment for round trip engineering with UML, Java, and design patterns, based on graph transformation. In this case, users are supplied with a standard UML-like CASE tool to design their models, but the tool exploits graph transformation to let them specify the behavioral aspects of their modeling elements. Like other CASE tools, FUJABA generates Java class definitions from UML class diagrams, but it combines UML statechart, activity, and collaboration diagrams to define *story diagrams*, that is, the diagrams that users exploits to model the dynamics of their methods. This way method bodies can automatically be generated and round trip engineering is supported: Users can modify the generated code manually and FUJABA is able to load modified code and to (re)establish the corresponding diagrams.

5.2 Graph transformation-enabled tools

The tools presented in this section provide mainly automatic support to the generation of graphical editors by making users define visual languages through suitable grammars.

For example, DIAGEN supplies two distinct components: a *framework of Java classes*, to provide the generic functionality for editing and analyzing diagrams, and a *generator program*, to produce Java source code for those aspects that depend on the concrete syntax of the language.

Users specify their languages – mainly the concrete syntax, but also the other aspects could be covered – through an hyperedge graph grammar, which allows DIAGEN to recognize the structure and syntactic correctness of diagrams during the editing process. DIAGEN allows both free-hand and syntax-directed editing, with the former directly mapped onto defined grammar productions, and the latter which requires that the user specification be parsed. Thus, hypergraph transformations and grammars provide a flexible syntactic model and allow for efficient parsing of user specifications (i.e., diagrams produced using the editor).

DIAGEN provides also an automatic layout facility, which is based on flexible geometric constraints and relies on an external constraint-solving engine.

Similarly, the GENGED approach (*Generation of Graphical Environments for Design*) and toolset [5] supports the description of visual modeling languages for producing both graphical editors – available in syntax-directed or free-hand editing mode – and simulation environments. To this end, GENGED exploits AGG and graphical constraint solving techniques.

A visual language is defined through a set of grammars: the *syntax grammar* defines the actual syntax of the language and would be enough if the aim is only to implement a syntax-directed editor. To allow for free-hand editing, GenGED requires also a *parse grammar*, which has the same characteristics of the parsing grammars supported by AGG. Moreover, if users want to simulate their models, they must provide a *simulation grammar* to specify how their model behave when fed correctly.

GenGED integrates these grammars, together with defined graphical constraints to properly laying out user models, and supplies an integrated environment for the specified language.

6 Conclusion

It was the aim of this tutorial paper to provide a light-weight introduction to graph transformation, its concepts, applications, theory, and tools, from the point of view of visual modeling techniques in software engineering. To this aim, we have identified in the Introduction three aspects of visual modeling techniques which complicate their definition and implementation: the graphical structure, the number and diversity of different languages, and the view-based nature of models which creates consistency problems.

After introducing a basic graph transformation approach, i.e., a set-theoretic presentation of the double-pushout approach and discussing possible extensions and alternatives, we have presented applications to the modeling of object behavior and architectural dynamics, and we have demonstrated how these two views can be consistently integrated by means of a common meta model. This part shows the adequacy of graph transformation as a semantic domain for today's systems and languages.

Then, applications of graph transformation to the definition of modeling languages have been discussed. In particular, the presentation has focussed on concrete and abstract syntax definitions by means of graph grammars, operational semantics based on rule-based specifications of abstract interpreters, and denotational semantics based on rule-based translations of visual models into semantic domains.

Finally, we have mentioned the most important branches of the theory of graph transformation based on their history in Chomsky grammars, Petri nets, and term rewriting, and we have given a survey of tools supporting, and supported by graph transformation.

For the future, the main short-term goal of the community should be the improvement of the usability of graph transformation for non-experts. Today, such applications require both theoretical background and experience to understand potential benefits and problems in a certain domain, which is not readily accessible outside the community. One key issue, thus, is the transfer of concepts and theory to different application domains. This requires an in-depth study of problems and existing solutions in each domain and a presentation of graph

transformation techniques and tools using the domain language and notation. It includes the preparation of learning material for people with different backgrounds and skills, but also the organization of schools and tutorials for different communities, trying to customize the contents on the actual needs of the hosting community.

In return, each application domain will raise new requirements for the development of theory and tools, and this will result in a partial shift of motivations from theory- to application-oriented. Theoretical problems with practical motivation include the verification of graph transformation systems, the transformation of "graphs with semantics" like Petri nets or UML models, and the evolution and modularity of graph transformation systems with corresponding compositionality results, to name only a few examples.

References

1. From UML to Java and Back Again: The Fujaba homepage. `www.fujaba.de`.
2. M. Andries, G. Engels, and J. Rekers. How to represent a visual specification. In K. Marriott and B. Meyer, editors, *Visual Language Theory*, pages 241–255. Springer-Verlag, 1997.
3. J.W. Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. In *International Conference on Information Processing, Paris*, pages 125–131, 1959.
4. P. Baldan, A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and F. Rossi. Concurrent semantics of algebraic graph transformation. In Ehrig et al. [29], pages 107 – 188.
5. R. Bardohl and H. Ehrig. Conceptual model of the graphical editor GenGed for the visual definition of visual languages. In Ehrig et al. [25], pages 252 – 266.
6. R. Bardohl, G. Taentzer, M. Minas, and A. Schürr. Application of graph transformation to visual languages. In Engels et al. [24], pages 105–180.
7. H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeier, and M. R. Sleep. Term graph rewriting. In *PARLE*, volume 259 of *LNCS*, pages 141–158. Springer-Verlag, 1987.
8. E. Barendsen and S. Smeters. Graph rewriting aspects of functional programming. In Engels et al. [24], pages 63 – 102.
9. L. Baresi. *Formal customization of graphical notations*. PhD thesis, Dipartimento di Elettronica e Informazione – Politecnico di Milano, 1997. In Italian.
10. M. Bauderon and H. Jacquet. Categorical product as a generic graph rewriting mechanism. *Applied Categorical Structures*, 9(1), 2001.
11. P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. A visualization of OCL using collaborations. In Gogolla and Kobryn [43], pages 257–271.
12. P. Bottoni, A. Schürr, and G. Taentzer. Efficient Parsing of Visual Languages based on Critical Pair Analysis and Contextual Layered Graph Transformation. In *Proc. IEEE Symposium on Visual Languages*, September 2000. Long version available as technical report SI-2000-06, University of Rom.
13. G. Busatto, G. Engels, K. Mehner, and A. Wagner. A framework for adding packages to graph transformation approaches. In Ehrig et al. [25], pages 352–367.
14. D. Coleman, P. Arnold, S. Bodof, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremes. *Object Oriented Development, The Fusion Method*. Prentice Hall, 1994.

15. A. Corradini and U. Montanari. Specification of Concurrent Systems: from Petri Nets to Graph Grammars. In *Quality of Communication-Based Systems*, pages 35–52. Kluwer Academic Publishers, 1995.
16. A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3,4):241–266, 1996.
17. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation, Part I: Basic concepts and double pushout approach. In Rozenberg [82], pages 163–245.
18. A. Corradini and F. Rossi. A new term graph rewriting formalism: Hyperedge replacement jungle rewriting. In Sleep M.R., Plasmeijer M.R., and M.C. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 8, pages 101–116. John Wiley & Sons Ltd, 1993.
19. B. Courcelle. The monadic second-order logic of graphs I, recognizable sets of finite graphs. *Information and Computation*, 8521:12–75, 1990.
20. B. Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In Rozenberg [82].
21. F. Drewes, B. Hoffmann, and D. Plump. Hierarchical graph transformation. In J. Tiuryn, editor, *Foundations of Software Science and Computation Structures (FoSSACS'00), Berlin, Germany*, volume 1784 of *LNCS*. Springer-Verlag, March/April 2000.
22. F. Drewes, H.-J. Kreowski, and A. Habel. Hyperedge replacement graph grammars. In Rozenberg [82], pages 95 – 162.
23. D. D'Souza and A. Wills. *Components and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.
24. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages, and Tools*. World Scientific, 1999.
25. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Proc. 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT'98), Paderborn, November 1998*, volume 1764 of *LNCS*. Springer-Verlag, 2000.
26. H. Ehrig and A. Habel. Graph grammars with application conditions. In G. Rozenberg and A. Salomaa, editors, *The Book of L*, pages 87–100. Springer-Verlag, 1985.
27. H. Ehrig, A. Habel, H.-J. Kreowski, and F. Parisi Presicce. Parallelism and concurrency in high-level replacement systems. *Math. Struct. in Comp. Science*, 1:361–404, 1991.
28. H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation, Part II: Single pushout approach and comparison with double pushout approach. In Rozenberg [82], pages 247–312.
29. H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3: Concurrency and Distribution*. World Scientific, 1999.
30. H. Ehrig and M. Löwe. Categorical principles, techniques and results for high-level replacement systems in computer science. *Applied Categorical Structures*, 1(1):21–50, 1993.
31. H. Ehrig, M. Pfender, and H.J. Schneider. Graph grammars: an algebraic approach. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180. IEEE, 1973.
32. J. Engelfriet and G. Rozenberg. Node replacement graph grammars. In Rozenberg [82], pages 1 – 94.
33. G. Engels, R. Gall, M. Nagl, and W. Schäfer. Software specification using graph grammars. *Computing*, 31:317–346, 1983.

34. G. Engels, J.H. Hausmann, R. Heckel, and St. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In A. Evans, S. Kent, and B. Selic, editors, *Proc. UML 2000, York, UK*, volume 1939 of *LNCS*, pages 323–337. Springer-Verlag, 2000.
35. G. Engels, R. Heckel, and J.M. Küster. Rule-based specification of behavioral consistency based on the UML meta model. In Gogolla and Kobryn [43].
36. G. Engels and A. Schürr. Hierarchical graphs, graph types and meta types. In *Proc. of SEGRAGRA'95 "Graph Rewriting and Computation"*, volume 2 of *Electronic Notes in TCS*, 1995.
37. C. Ermel, M. Rudolf, and G. Taentzer. The AGG approach: Language and tool environment. In Engels et al. [24], pages 551 – 601.
38. A. Finkelstein, J. Kramer, B. Nuseibeh, M. Goedicke, and L. Finkelstein. Viewpoints: A framework for integrating multiple perspectives in system development. *Int. Journal of Software Engineering and Knowledge Engineering*, 2(1):31–58, March 1992.
39. T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph transformation language based on UML and Java. In Ehrig et al. [25].
40. P. Fradet, D. Le Métayer, and M. Périn. Consistency checking for multiple view software architectures. In *Proc. Joint European Software Engineering Conference and Symp. on Foundations of Software Engineering, ESEC/FSE'99*, volume 1687 of *LNCS*, pages 410–428, 1999.
41. C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall Int., 1991.
42. M. Goedicke. Paradigms of modular software development. In R. J. Mitchell, editor, *Managing Complexity in Software Engineering*, volume 17 of *IEE Computing Series*. Peter Peregrinus, 1990.
43. M. Gogolla and C. Kobryn, editors. *Proc. UML 2001 – Modeling Language, Concepts and Tools, Toronto, Kanada*, LNCS. Springer-Verlag, 2001.
44. M. Große-Rhode, F. Parisi-Presicce, and M. Simeoni. Refinement of graph transformation systems via rule expressions. In Ehrig et al. [25], pages 368–382.
45. A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3,4):287 – 313, 1996.
46. D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, oct 1996.
47. J.H. Hausmann, R. Heckel, and G. Taentzer. Detecting conflicting functional requirements in a use case driven approach: A static analysis technique based on graph transformation. In *Proc. Int. Conference on Software Engineering (ICSE'2002)*, Orlando, FL, May 2002. ACM/IEEE Computer Society.
48. 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.
49. R. Heckel, J. Küster, and G. Taentzer. Confluence of typed attributed graph transformation systems. In A. Corradini and H.-J. Kreowski, editors, *Proc. 1st Int. Conference on Graph Transformation (ICGT 02), Barcelona, Spain*, LNCS. Springer-Verlag, October 2002. To appear.
50. R. Heckel and St. Sauer. Strengthening UML collaboration diagrams by state transformations. In H. Hußmann, editor, *Proc. Fundamental Approaches to Software Engineering (FASE'2001), Genova, Italy*, volume 2185 of *LNCS*. Springer-Verlag, April 2001.

51. R. Heckel and A. Wagner. Ensuring consistency of conditional graph grammars – a constructive approach. In *Proc. of SEGRAGRA'95 "Graph Rewriting and Computation"*, volume 2 of *Electronic Notes in TCS*, 1995.
52. D. Hirsch, P. Inverardi, and U. Montanari. Modeling software architectures and styles with graph grammars and constraint solving. In *Proceedings of the First Working IFIP Conference on Software Architecture, San Antonio, Texas, E.E.U.U.*, February 1999.
53. D. Hirsch and M. Montanari. Synchronized hyperedge replacement with name mobility. In *Proc. CONCUR 2001, Aarhus, Denmark*, volume 2154 of *LNCS*, pages 121–136. Springer-Verlag, August 2001.
54. C. Hoare. Communicating sequential processes. *Communicat. Associat. Comput. Mach.*, 21(8):666–677, 1978.
55. B. Hoffmann and M. Minas. A generic model for diagram syntax and semantics. In *Proc. ICALP2000 Workshop on Graph Transformation and Visual Modelling Techniques, Geneva, Switzerland*. Carleton Scientific, 2000.
56. D. Janssens. Actor grammars and local actions. In Ehrig et al. [29], pages 57–106.
57. D. Janssens and G. Rozenberg. On the structure of node-label controlled graph grammars. *Information Science*, 20:191–216, 1980.
58. A. Kent and D. Akehurst. A relational approach to defining transformations in a metamodel. In *Proc. UML 2002, Dresden, Germany*, LNCS. Springer-Verlag, 2002. To appear.
59. H.J. Köhler, U. Nickel, J. Niere, and A. Zündorf. Integrating UML diagrams for production control systems. In *Proc. of the* 22^{th} *International Conference on Software Engineering (ICSE), Limerick, Irland*. ACM Press, 2000.
60. J. Kramer and J. Magee. Distributed software architectures. In *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)*, pages 633–634. Springer-Verlag, May 1997.
61. H.-J. Kreowski and S. Kuske. On the interleaving semantics of transformation units - a step into GRACE. In *5th Int. Workshop on Graph Grammars and their Application to Computer Science, Williamsburg '94, LNCS 1073*, pages 89 – 106. Springer-Verlag, 1996.
62. S. Kuske. A formal semantics of UML state machines based on structured graph transformation. In Gogolla and Kobryn [43].
63. Le Métayer, D. Software architecture styles as graph grammars. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 216 of *ACM Software Engineering Notes*, pages 15–23, New York, October 16–18 1996. ACM Press.
64. M. Löwe. Algebraic approach to single-pushout graph transformation. *Theoret. Comput. Sci.*, 109:181–224, 1993.
65. M. Löwe, M. Korff, and A. Wagner. An algebraic framework for the transformation of attributed graphs. In M. R. Sleep, M. J. Plasmeijer, and M.C. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 14, pages 185–199. John Wiley & Sons Ltd, 1993.
66. K. Marriott, B. Meyer, and K.B. Wittenburg. A survey of visual language specification and recognition. In B. Meyer K. Marriott, editor, *Visual Language Theory*, chapter 2, pages 5–85. Springer-Verlag, 1998.
67. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
68. R. Milner. Bigraphical reactive systems. In Kim Guldstrand Larsen and Mogens Nielsen, editors, *Proc. 12th Intl. Conference on Concurrency Theory (CONCUR 2002), Aalborg, Denmark*, volume 2154 of *LNCS*, pages 16–35. Springer-Verlag, August 2001.

69. M. Minas. Hypergraphs as a uniform diagram representation model. In Ehrig et al. [25], pages 281 – 295.
70. U. Montanari. Separable graphs, planar graphs and web grammars. *Information and Control 16*, pages 243–267, 1970.
71. U. Montanari, M. Pistore, and F. Rossi. Modeling concurrent, mobile, and coordinated systems via graph transformation. In Ehrig et al. [29], pages 189 – 268.
72. M. Nagl. *Graph-Grammatiken: Theorie, Implementierung, Anwendungen*. Vieweg, 1979.
73. M. Nagl, editor. *Building Tightly Integrated Software Development Environments: The IPSEN Approach, LNCS 1170*. Springer-Verlag, 1996.
74. Object Management Group. Meta object facility (MOF) specification, September 1999. `http://www.omg.org`.
75. Object Management Group. UML specification version 1.4, 2001. `http://www.celigent.com/omg/umlrtf/`.
76. J. L. Pfaltz and A. Rosenfeld. Web grammars. *Int. Joint Conference on Artificial Intelligence*, pages 609–619, 1969.
77. G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, 1981.
78. D. Plump. Term graph rewriting. In Engels et al. [24], pages 3 – 62.
79. T. W. Pratt. Pair grammars, graph languages and string-to-graph translations. *Journal of Computer and System Sciences*, 5:560–595, 1971.
80. T.W. Pratt. Definition of programming language semantics using grammars for hierarchical graphs. In H. Ehrig, V. Claus, and G. Rozenberg, editors, *1st Int. Workshop on Graph Grammars and their Application to Computer Science and Biology, LNCS 73*, volume 73 of *LNCS*. Springer-Verlag, 1979.
81. J. Rekers and A. Schürr. Defining and parsing visual languages with layered graph grammars. *Journal of Visual Languages and Computing*, 8(1):27 –55, 1997.
82. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific, 1997.
83. H.-J. Schneider. Chomsky-Systeme für partielle Ordnungen. Technical Report 3-3, Universität Erlangen, 1970.
84. A. Schürr. Specification of graph translators with triple graph grammars. In Tinhofer, editor, *Proc. WG'94 Int. Workshop on Graph-Theoretic Concepts in Computer Science*, number 903 in LNCS, pages 151–163. Springer-Verlag, 1994.
85. A. Schürr. Logic based programmed structure rewriting systems. *Fundamenta Informaticae*, 26(3,4):363 – 386, 1996.
86. A. Schürr. Programmed graph replacement systems. In Rozenberg [82], pages 479 – 546.
87. A. Schürr, A.J. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In Engels et al. [24], pages 487–550.
88. D. Scott and C. Strachey. Towards a mathematical semantics for computer languages. In *Computers and Automata*, pages 19–46. Wiley, 1971.
89. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
90. S. H. von Solms. Node-label controlled graph grammars with context conditions. *Intern. J. Computer Math 15*, pages 39–49, 1984.
91. G. Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems*. PhD thesis, TU Berlin, 1996. Shaker Verlag.
92. G. Taentzer, I. Fischer, M. Koch, and V. Volle. Distributed graph transformation with application to visual design of distributed systems. In Ehrig et al. [29].

93. G. Taentzer, M. Goedicke, and T. Meyer. Dynamic change manegement by distributed graph transformation: Towards configurable distributed systems. In *Proceedings TAGT'98*, volume 1764 of *LNCS*, pages 179–193. Springer-Verlag, 2000.
94. M. von der Beek. A comparison of Statecharts variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer LNCS 863, 1994.
95. C.P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, University of Oxford, 1971.
96. M. Wermelinger and J.L. Fiadero. A graph transformation approach to software architecture reconfiguration. In H. Ehrig and G. Taentzer, editors, *Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems (GraTra'2000), Berlin, Germany*, March 2000. `http://tfs.cs.tu-berlin.de/gratra2000/`.