

From Graph Transformation to Software Engineering and Back

Luciano Baresi¹ and Mauro Pezzè²

¹ Politecnico di Milano,
Dipartimento di Elettronica e Informazione,
Milano, Italy, baresil@elet.polimi.it

² Università degli Studi di Milano-Bicocca,
Dipartimento di Informatica Sistemistica e Comunicazione,
Milano, Italy, pezzed@disco.unimib.it

Abstract. Software engineers usually represent problems and solutions using graph-based notations at different levels of abstractions. These notations are often semi-formal, but the use of graph transformation techniques can support reasoning about graphs in many ways, and thus can largely enhance them.

Recent work indicates many applications of graph transformation to software engineering and opens new research directions. This paper aims primarily at illustrating how graph transformation can help software engineers, but it also discusses how software engineering can ameliorate the practical application of graph transformation technology and its supporting tools.

1 Introduction

Software engineering aims at developing large software systems that meet quality and cost requirements. The development process that moves from the initial problem to the software solution is based on models that describe and support the development during the different phases. Models are key elements of many software engineering methodologies for capturing structural, functional and non-functional aspects. Popular methodologies prescribe various models — with different degrees of formality, flexibility, and analyzability — to solve the different problems. For example, UML proposes some semi-formal diagrammatic languages: class, object, component, and deployment diagrams for modeling structural aspects, and use case, sequence, activity, collaboration, and statechart diagrams for modeling behavioral aspects [24].

The syntax and semantics of these models are defined informally with different degrees of precision. Although users and tools agree on the main syntactic and semantic aspects, important details are often given different — and frequently incompatible — interpretations.

The scientific community agrees on the need to improve current practice by increasing the degree of formality of these notations. Unfortunately, formal methods have not found wide application so far, and cannot be easily used in

conjunction with popular modeling notations [6]. This is where graph transformation (*GT*) provides unique features to strengthen diagrammatic models by adding formality. Similarly to grammars for textual languages, GT can formally describe the concrete and abstract syntaxes of modeling languages, but it can also formalize the semantic aspects, and thus provides a strong basis for reasoning on diagrammatic models at all levels.

Formalizing the concrete and abstract syntaxes eliminates ambiguities and contradictions and supports automatic checks for consistency and correctness. GT allows the designer to describe the semantics both operationally and denotationally. Operational semantics can be given by describing the legal evolutions of models in terms of GT rules. Denotational semantics can be expressed by mapping models onto semantic domains by means of rules that mimic syntactical changes onto the chosen semantic domain [28].

GT is well supported by tools, which are useful for solving many problems and validating new ideas and applications, but often they do not scale well. When the size of the application grows, and requires significant mediation between theory and performance, software engineering principles can provide useful ideas. They can contribute significantly in this direction and help GT experts move towards complex problems and applications.

The main goal of this paper is to frame the opportunities offered by GT to software engineering by illustrating sample cases and proposing additional applications not fully explored yet. The paper also suggests ways for improving GT technology and tools, inspired by well-known software engineering principles, to address practical problems.

The paper is organized as follows. Section 2 discusses the opportunities offered by GT as modeling language by touching the use of GT for modeling and reasoning on particular aspects of software systems. Section 3 illustrates the potentiality of GT for modeling and verifying notations, i.e., for formalizing the concrete and abstract syntaxes as well as the semantics of notations, thus providing a uniform framework for modeling heterogeneous notations, and fostering analysis tools. Section 4 indicates how software engineering can help experiment and introduce graph transformation in current modeling practice. Section 5 proposes some future directions and concludes the paper.

2 Graph transformation for models

GT provides a formal and intuitive way for describing systems and their evolution. For example, GT has been proposed to model software architectures. Architectural styles constrain the connectivity among components to guarantee the regularity of architectural models, and thus improve the maintainability and evolvability of systems [10].

GT provides a natural way for formalizing architectural styles by means of rules that support the creation of models that comply with the style by construction. Le Metayér [19] first and then Hirsch et al. [12] investigated different approaches to model software architectural styles by using GT; Baresi et al. [3]

extend these approaches for service-oriented applications and emphasize the analyzability of such systems.

Type graphs, or metamodels according to the OMG terminology, define the elements that belong to a style. GT uses these definitions to create new models and make existing ones evolve. For example, Fig. 1, taken from [3], shows a UML class diagram — used as type graph — that models a fragment of the service-oriented architectural style. This means that all service-oriented applications must share this type graph and be suitable instantiation of these classes (types). In other words, the node types of the graph behind these architectures must belong to the set of nodes defined in Fig. 1. The type graph says that a *Component* dynamically searches for *Services* that meet given *Requirements* through a *Connect Request*. A *Service* is a special *Component* that satisfies a *Service Specification*. A *Component* knows the *Service Specification* of available *Services* and can *require services for* given *Requirements*, which *could satisfy* the given specification. The *Connect Request* establishes the link.

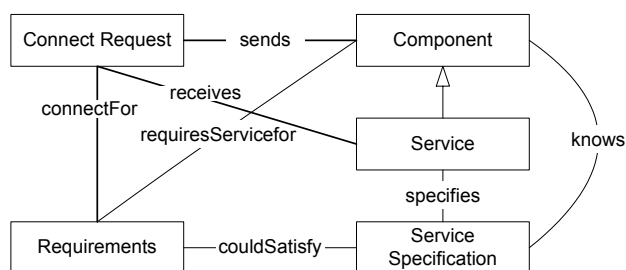


Fig. 1. An excerpt of the type graph for service-oriented architectures [3].

Fig. 2 presents an example GT rule that describes how to establish a new connection. A GT rule comprises two parts: a left- and a right-hand side. The left-hand side describes the conditions for the application of the rule. In this example, the *Component* must *know* a *Service Specification* that *could satisfy* its *Requirements*. The right-hand side describes the effect of the modification. In this case, the *Component* can *send* a *Connect Request*, newly added, for the required services to the *Service*.

The rule applies to all instances of the SoA style to enable generic components to request services according to given specifications. It creates a new *Connect Request*, called *req* to connect the *Requirements*, *Service*, and *Component* accordingly. The rule formally describes one of the steps to build and configure architectural models, without changing either the syntax or semantics. A set of rules can formally describe all possible building steps, thus formalizing completely the semantics of an architectural style, i.e., the semantics of an entire family of models.

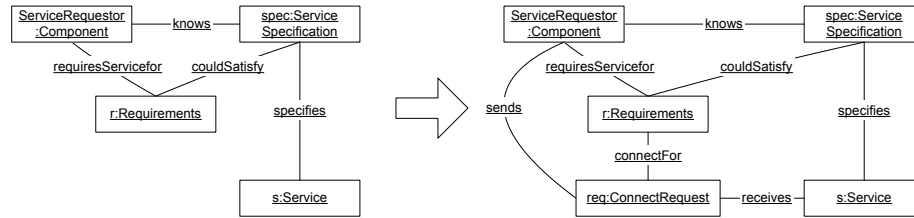


Fig. 2. A GT rule that models a reconfiguration of a service oriented architecture, where a component plays the role of a service requester and sends a request to the service it would like to connect to (The figure is taken from [3]).

Architectures often evolve with systems. Architecture reconfiguration refers to control the evolution of architectures according to given guidelines. The formalization of reconfiguration guidelines and constraints with GT provides a means to automatically check for consistency of architecture reconfigurations, and in perspective also a means for predicting and comparing the effects of different reconfiguration approaches.

Security is another interesting problem that presents variegated aspects that can be modeled with diagrammatic languages and GT. For example, Fig. 3 shows an Alloy model of the type system of a role based access control (RBAC) policy taken from [17]. The figure formalizes the relations among the entities in the system. *Roles* can be organized hierarchically (a role can be a *super-* or a *sub-role*, but each role has a unique super-role); roles' duty are separated (*sod*); a *session belongs* only to *users* whose *roles are activated* for the considered sessions.

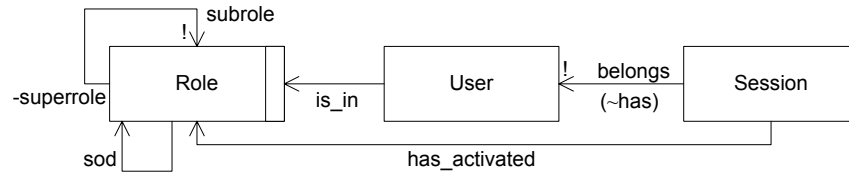


Fig. 3. An Alloy model of a role based access control (RBAC) policy (The figure is taken from [17]).

Policies can be expressed with rules that describe the evolutions of the relations among roles. A rule may for example allow users to activate only a subset of authorized roles when in a session. Such rule can be formally expressed with the GT rule of Fig. 4 taken from [17]. Koch and Parisi-Prsicce show that GT rules allow to model not only types and policy rules, as UML or Alloy, but also policy

constraints, and thus support a larger set of analyses that include constraint checking and conflict solving.

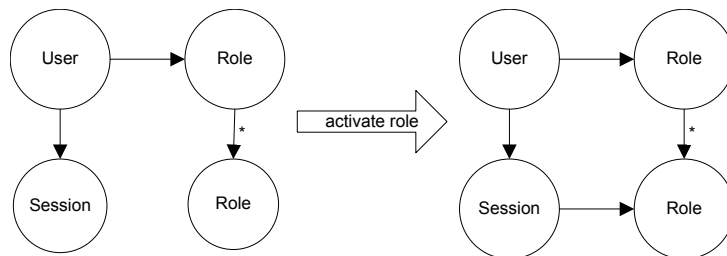


Fig. 4. A GT rule that allows a *User* within a *Session* to activate a *Role* only if it belongs to a specific subset of *Roles* that represent the roles the user is allowed to activate (The figure is taken from [17]).

Also the behavior of mobile systems depends on the current configuration that changes dynamically. Mobile systems can be modeled with graphs that evolve when agents move from node to node, activate or deactivate, and leave the network. Typed graphs can model different configurations, while graph transformation can formalize the rules that describe how configurations evolution, as observed in an early work by Corradini et al. [7].

So far, we introduced graph transformation mainly as a modeling means, but recently we have seen proposals that go a step beyond and address the validation of modeled GT systems. The pairing of graph transformation and model checking techniques is proposed by Varrò [29]. His approach transforms a graph transformation system, along with an initial configuration of the system under analysis, into a Promela specification, which is the representation required by SPIN. The model checker starts from the initial configuration and searches all the possible sequences of rules. This way, we can check if a given sequence of rules is feasible or try to identify the right sequence to obtain a target configuration of the system under analysis.

3 Graph transformation for notations

In the previous section, we have illustrated how GT can be used as modeling language. In this section, we show how GT can be used to define specification languages. Figure 5 uses the well-known MVC (Model-View-Control) pattern to identify the different parts that belong to a diagram notation ([1]). The *graphical elements* are the *view*, i.e., the set of lines, boxes, bubbles, and labels perceived by the user. The *concrete syntax*, the *control*, captures the structure of diagrams in terms of their graphical elements and the relationships among them. The *abstract syntax* stands for the *model* and defines the structure of the diagram in

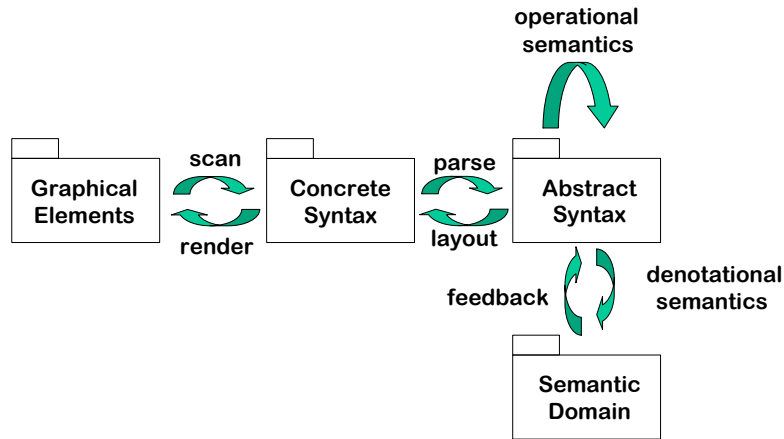


Fig. 5. Layered view of a *diagram language*

terms of the diagram notation itself. Even if we could move directly from the graphical elements to their abstract syntax, it is important that we distinguish between *visualization* and *interpretation*. The concrete syntax deals with visualization and identifies the spatial relationships among the graphical elements. The abstract syntax deals with interpretation and specifies the tokens of the language and how they are composed to obtain a meaningful sentence (i.e., a model). The interpretation may require a *semantic domain*, that is, an external formal model. The meaning of a diagram becomes the result of its translation into the semantic domain.

GT systems can be employed to define both the different views and the relationships among them. This means, for example, that we can use a GT system to formalize the abstract syntax of a diagram language, but we can also imagine a pair of GT systems to specify the relationships between the concrete and abstract syntaxes of a language. More precisely, the pairs of rules define how to *parse* models if we move from concrete to abstract, while they specify the *layout* of models if we move from abstract to concrete.

Let us consider for example the simple Statechart diagram of Fig. 6, taken from [5], as running example. It models the behavior of a *CashBox*. After switching it on, the *CashBox* enters the super-state *Operating* and moves to its default sub-state *Idle*. As soon as a card is inserted (i.e., event *insertCard* occurs), the component enters state *Card Inserted*. When it receives the client data from the card (event *receiveClientData*), it moves to state *Authentication Started*. Here, the choice of the next state depends on the data received. If the client is accepted, the component enters state *Serving*, otherwise it moves to state *Rejected*. In both cases and after processing the transaction, the component returns to state *Idle* as soon as the card is ejected (event *cardEjected*).

The first aspect that characterizes a diagram like this is the set of *graphical elements* that define its structure. These elements are specific to the chosen

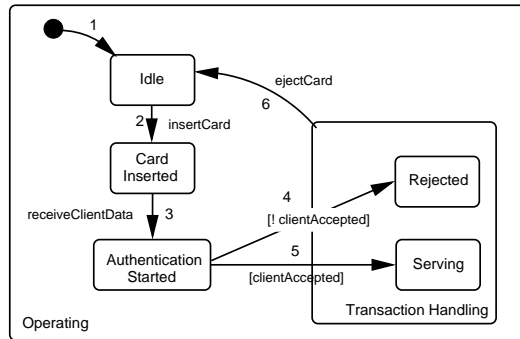


Fig. 6. A simple Statechart model of a CashBox (The figure is taken from [5])

format and concur in defining the *concrete syntax* of the notation. If we chose an XML representation, like Scalable Vector Graphic (SVG) [30], the schema associated with the notation specifies how to identify a line, a bubble, a rectangle, or a label. These representation-specific elements are the starting point to construct and understand the graphical sentences behind a set of pictorial symbols.

The *concrete syntax* abstracts away from the particular representation and identifies the relationships among the graphical shapes. No matter of XML or an object-oriented language, states in Fig. 6 are represented through rectangles with rounded corners, initial states with black 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 GT system defines the concrete syntax of the language in terms of the steps necessary to build a correct model. These rules can be conceived with the idea of scanning an existing graphical representation to produce the concrete syntax model, but it could also be defined with respect to a user that uses a syntax-directed editor for the supported notation. In this latter case, the transformation system defines all the correct user actions on the editor.

For example, Minas in [14] proposes a complete hyperedge grammar for *editing* well-formed Statechart diagrams, to feed the DIAGEN tool for automatically generating a graphical editor for Statechart diagrams. Formally, the 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 Statecharts.

The *abstract syntax* defines the modeling elements supplied by the notation, without the concrete “sugar”, and the relationships among them. Tokens at this level are related to the semantic interpretation, that is, we think of the example

of Fig. 6 in terms of states (initial, AND-decomposed, and OR-decomposed) transitions, events, and so on. Figure 7 shows a simplified abstract syntax graph

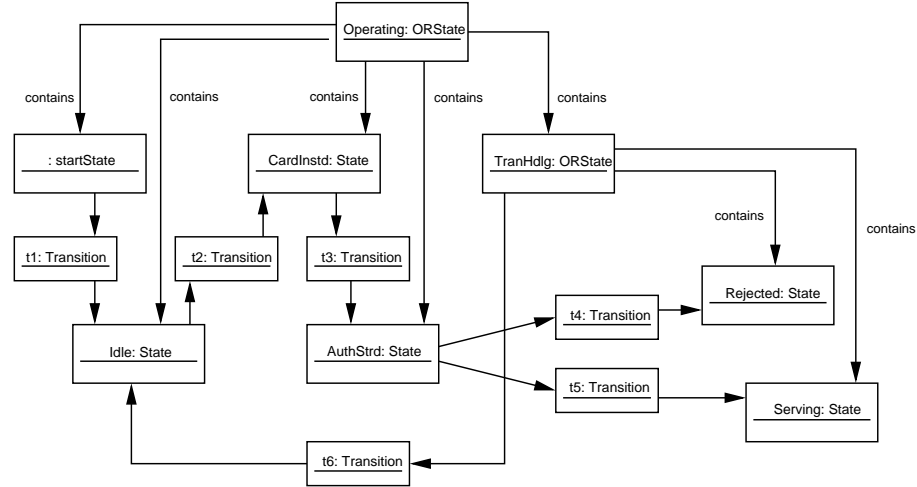


Fig. 7. Abstract syntax graph for the Statechart diagram of Fig. 3 (The figure is taken from [5]).

for the Statechart diagram of Fig. 6. Nodes are instances of a simple type graph 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.

If we wanted to map abstract to concrete syntaxes, this implies the definition of 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 algorithms for defining the layout of user models can be implemented using GT rules and textual attributes to compute the coordinates of each graphical symbol.

GT systems can be used to specify the rules that govern the elements at a given level, but also to define the mappings and transformations between levels. The choice is between two parallel systems, with paired rules, or a single complex system. In the first case each pair would comprise a rule that specifies a step in the first domain and the corresponding step in the other domain. The selection and application of the first rule would also trigger the application of the second rule. In the second case, we could foresee complex rules that start from symbols of the first domain and rewrite them into their equivalent elements of the second domain.

Moving to the semantics of the model of Fig. 6, we can specify it both *operationally* and *denotationally*.

An operational semantics can be given directly on the abstract syntax of the language through yet another GT system that specifies an interpreter for the language ([8]). Each model can also be “compiled” into a set of dedicated rules ([18]) to specify the behavior of each single model separately. For example, transitions for *syntactically correct* Statechart diagrams can be generated by applying the rules of the transformation unit $term(S)$ presented in [18]. Fig. 8 shows the result for some of the transitions of Fig. 6. Rule $t1$ moves the current state from the start state to *Idle*; Rule $t5$ moves the current state from *Authentication Started* to the hierarchy *Transaction Handling / Serving*. Rule $t6$ moves the current state from the hierarchy *Transaction Handling / any contained state* back to *Idle*.

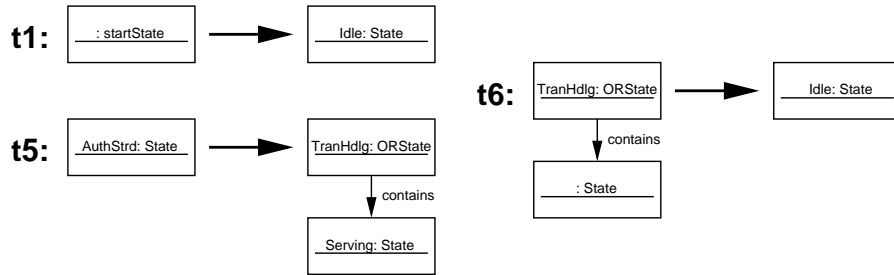
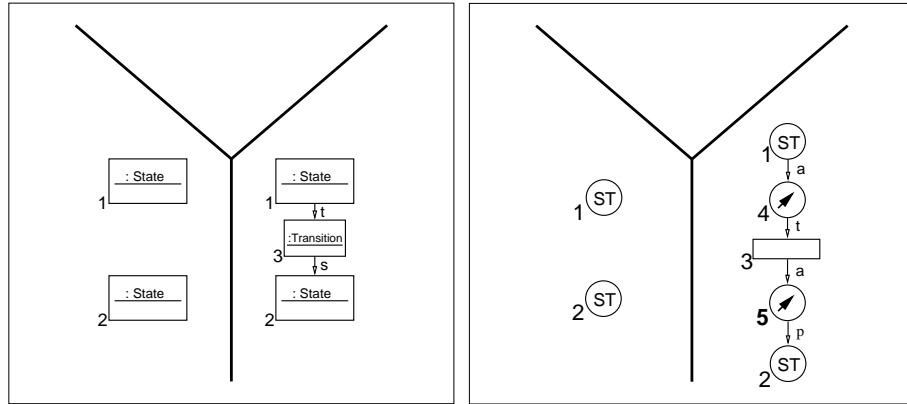


Fig. 8. Some transitions of Fig. 3 as GT rules (The figure is taken from [5])

Denotational semantics is given by mapping the abstract syntax to a *semantic domain*. The role played by GT depends on the chosen semantic domain. 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 automatically triggers the application of the paired textual production [27].

For example, Engels et al. define the dynamic semantics of Statecharts through CSP (Communicating Sequential Processes [13]). The left-hand side of each rule defines how UML-like metamodel instances can be built for Statecharts (GT rules); the right-hand side codes how the corresponding CSP specification must be modified accordingly (textual grammar productions). Baresi uses high-level timed Petri nets as semantic domain. The mapping is defined with pairs of GT production: the first production defines the evolution of the abstract syntax representation, while the second production states the corresponding changes on the semantic Petri nets [2, 4]. Figure 9 taken from [2] shows a simple transformation rule that formalizes the connection of two Statechart states with a Statechart

transition. A transformation rule comprises two productions. The left-hand side production applies to pairs of Statechart states and connects them through a Statechart transition, as indicated by the right-hand side of the rule. The right-hand side production applies to the Petri net places corresponding to the selected Statechart states and adds a Petri net transition and two arcs to connect them. The figure omits the textual annotations that can be found in [2].



(a) Abstract Syntax Model

(b) Semantic Model

Fig. 9. A simple transformation rule taken from [2]

More sophisticated mappings could be implemented using triple graph grammars [28], that is, besides the two grammars that define the abstract syntax and the corresponding modifications of the semantic domain, a third grammar would state the mapping between the two paired productions explicitly.

The different models defined so far pave the ground to additional analysis techniques, e.g., modern refactoring approaches. For example, Mens et al. start from the abstract syntax representation of UML class diagrams to reason on them and improve the quality of designed models [21]. Transformation rules work at this level to modify and improve user-defined class diagrams.

Similarly, with the help of pair grammars, transformation rules can be used to maintain, enforce or access the consistency of different views on the same problem. The fact that a modification of a given model can trigger modifications in other models, that is, the application of a rule on a model triggers the application of other rules on other models, is a way to enforce the consistency among views in complex models. Rules can be used both as a generative means, i.e., consistency is enforced by construction, or they can be a validation means and thus access the consistency of views defined independently. For example Haus-

mann et al. show how to use GT to enforce and access the consistency between requirements and high-level design of software systems [11].

4 Software engineering for graph transformation

After discussing how GT can be proficiently used in software engineering, we briefly discuss how graph transformation can benefit from software engineering principles and techniques.

Software engineers can help graph transformation experts improve tools. Currently, the most popular tools for GT are excellent supports for early experimentation and validation of new ideas, but do not always meet the needs of software developers. They work well with research-size models, but become slow or crash when the size of models increases. They support well the direct use of graph transformation, but provide less help to software experts that would like to exploit GT without being proficient in the formalism. They need open tools that interoperate with sophisticated development environments [22].

Software engineers look for “problem-oriented”, rather than “solution-oriented” tools. They are not ready to use graph transformation systems per-se, but they want means to solve development problems. This means that many details and technicalities should be hidden, while the adoption of standard notations and formats should be emphasized. This approach was adopted by Fujaba with its use of UML, as standard modeling notation (class diagrams to define concepts, object diagrams to define rules, and activity diagrams to compose them), and its hiding of many details related to graph transformation. The GT community is using standard notations to renders type graphs and rules. UML is the de-facto standard also in this community and the adoption of this notation is a first step towards the readability and usability of GT.

Software engineers need techniques that interface well with other CASE tools. Software engineering can contribute to GT with well-known standards for information exchange like XMI or MOF [23]. Besides special-purpose XML-based standards to encode graphs and graph transformation (GXL and GTXL, namely), GT tools should also support common “generic” standards to import and export artifacts and improve their interfaceability with other tools in the development process.

When designing real(istic) artifacts, modularization becomes an important issue. Nowadays, many graph transformation systems adopt a flat organization and the number of rules is constrained. Some proposals were presented to add modules to graph transformation; none of them has been widely accepted, but modules are a key element to handle real problems through graph transformation. They help understand the specification, by allowing the designer to reason at different levels, and foster the idea of reusing organized and coherent sets of rules. Something like a *package* in UML, to group the rules, and a *component* to show its interfaces would clearly improve the organization and management of graph transformation systems.

Software engineering can also help validate designed graph transformation systems. Flow-based analysis techniques, but also unit and integration testing techniques may help understand how a set of rules (i.e., a system) works. Other more formal approaches could also exploit process calculi and model checking techniques to reason on the dynamic behavior of such systems and prove interesting properties.

The last aspect concerns the methodologies and heuristics usually adopted by software engineers to model the different aspects of a software system, reason on them, and solve possible problems. This help is nothing concrete, but the right mix between capabilities in abstracting and modeling and the knowledge of graph transformation systems should pave the ground to better graph-based models of addressed aspects.

5 Conclusions

This overview was motivated by the idea that graphs provide a direct and intuitive way to represent, visualize and analyze a variety of structures that are typical to software engineering, and graph transformation provides general means for both describing and analyzing changes on graphs and specifying their governing rules and constraints. Sections 2 and 3 sketch some approaches, which are intended as example applications and do not aim at providing a complete survey of existing experiments with graph transformation and software engineering.

Moving to future directions, modern software development paradigms and applications, such as mobility, pervasive computing and component-based development raise new critical challenges. They come from the impossibility of predicting all possible uses of the software at development time. Traditional quality assurance techniques are mostly based on pre-deployment test and analysis, and provide limited support to these new challenges. Unpredicted interactions with new components or agents may lead to unexpected behaviors not previously verified. Current research investigates techniques such as in-field testing, run-time monitoring and self-healing mechanisms. They make use of run-time information to capture and analyze the in-field behavior of complex systems ([20, 25, 26, 9, 16]).

A main challenge of “post-deployment verification” is modeling the expected and experienced run-time behavior of the system, to identify and record potentially harmful runs. An excellent example of application of post-deployment modeling is the STAT intrusion detection framework proposed by Kemmerer and Vigna [15]. STAT models suspect security threats with finite state machines. The run-time behavior of the monitored systems is compared to the STAT model that detects and signals possible security threats. Finite state machines are effective in this context because the behaviors of interest are fairly simple and can be captured with a simple model. In general, the behavior of interest may be very complicated and may not be easily captured with a simple finite state machine. Graph transformation systems provide an excellent support for modeling complex evolutions, as illustrated in the previous sections, and thus are an obvious

candidate to extend post-deployment modeling for monitoring complex system evolutions.

We think that these examples are only the first fruits of a cooperation that has all the chances to become wider. The marriage will be complete as soon as the knowledge also addresses the other way around. Software engineers could contribute to improve the development of graph transformation tools and standard exchange formats for graphs and graph transformation. They can lend their experience on modularity and components, to add these features to graph transformation specifications, and on analysis and testing to validate produced GT systems.

References

1. 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.
2. L. Baresi. *Formal customization of graphical notations*. PhD thesis, Dipartimento di Elettronica e Informazione – Politecnico di Milano, 1997. In Italian.
3. L. Baresi, R. Heckel, S. Thone, and D. Varro. Modeling and validation of service-oriented architectures: application vs. style. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 68–77. ACM Press, 2003.
4. L. Baresi, A. Orso, and M. Pezzè. Introducing formal methods in industrial practice. In *Proceedings of the 20th International Conference on Software Engineering*, pages 56–66. ACM Press, 1997.
5. L. Baresi and R. Heckel. Tutorial introduction to graph transformation: A software engineering perspective. In *Proceedings of the First International Conference on Graph Transformation (ICGT 2002)*, volume 2505 of *Lecture Notes in Computer Science*, pages 402–429. Springer-Verlag, 2002.
6. E. Clarke, J. Wing, R. Alur, R. Cleaveland, D. Dill, A. Emerson, S. Garland, S. German, J. Guttag, A. Hall, T. Henzinger, G. Holzmann, C. Jones, R. Kurshan, N. Leveson, K. McMillan, J. Moore, D. Peled, A. Pnueli, J. Rushby, N. Shankar, J. Sifakis, P. Sistla, B. Steffen, P. Wolper, J. Woodcock, and P. Zave. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.
7. A. Corradini, F. Dotti, and L. Ribeiro. A graph transformation view on the specification of applications using mobile code. In *Proceedings of the International Symposium on Graph Transformation and Visual Modeling Techniques (GT-VMT)*, volume 50 (3). Electronic Notes in Computer Science, 2001.
8. G. Engels, J.H. Hausmann, R. Heckel, and S. 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 *Lecture Notes in Computer Science*, pages 323–337. Springer-Verlag, 2000.
9. D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, pages 27–32. ACM Press, 2002.
10. D. Garlan. Software architecture: a roadmap. In *The Future of Software Engineering*, pages 91–101. ACM Press, 2000.

11. J. 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 *Proceedings of the International Conference on Software Engineering (ICSE'2002)*, pages 105–155. ACM Press, May 2002.
12. D. Hirsch, P. Inverardi, and U. Montanari. Graph grammars and constraint solving for software architecture styles. In *ISAW '98: Proceedings of the Third International Workshop on Software Architecture*, pages 69–72, 1998.
13. C. Hoare. Communicating sequential processes. *Communicat. Associat. Comput. Mach.*, 21(8):666–677, 1978.
14. 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.
15. R.A. Kemmerer and G. Vigna. Intrusion Detection. *IEEE Computer*, 2002. Special publication on Security and Privacy.
16. J.O. Kephart and D.M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, January 2003.
17. M. Koch, L. V. Mancini, and F. Parisi-Presicce. A graph based formalism for RBAC. *ACM Transactions on Information and System Security (TISSEC)*, 5(3):332–365, August 2002.
18. S. Kuske. A formal semantics of UML state machines based on structured graph transformation. In M. Gogolla and C. Kobryn, editors, *Proc. UML 2001*, volume 2185 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.
19. D. Le Métayer. Software architecture styles as graph grammars. In *Sigsoft*, pages 15–23. ACM Pres, 1996.
20. S. McCamant and M. Ernst. Predicting problems caused by component upgrades. In *Proceedings of ESEC/FSE 2003*, pages 287–296. ACM Press, 2003.
21. T. Mens, N. Van Eetvelde, D. Janssen, and S. Demeyer. Formalising refactorings with graph transformations. *Journal of Software Maintenance and Evolution*, pages 1001–1025, 2004.
22. T. Mens, A. Scürr, and G. Taenzer. *Proceedings of the Workshop on Graph-Based Tools*. ENTCS, 2002.
23. OMG. Meta object facility (MOF) specification, September 1999.
24. OMG. *Unified Modeling Language (UML), version 1.5*. OMG Standard, 2003.
25. A. Orso, D. Liang, M. Harrold, and R. Lipton. Gamma system: continuous evolution of software after deployment. In *Proceedings of the international symposium on Software testing and analysis*, pages 65–69, Roma, Italy, 2002. ACM Press.
26. C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *International Conference on Software Engineering*, pages 277–284, 1999.
27. T. W. Pratt. Pair grammars, graph languages and string-to-graph translations. *Journal of Computer and System Sciences*, 5:560–595, 1971.
28. A. Schürr. Specification of graph translators with triple graph grammars. In *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 904 of *LNCS*, pages 228–253. Springer Verlag, 1994.
29. D. Varró. Towards symbolic analysis of visual modelling languages. In Paolo Bottoni and Mark Minas, editors, *Proc. GT-VMT 2002: International Workshop on Graph Transformation and Visual Modelling Techniques*, volume 72 of *ENTCS*, pages 57–70, Barcelona, Spain, October 11-12 2002. Elsevier.
30. W3C. *SVG: Scalable Vector Graphics (SVG) version 1.2*. W3C, May 2004. <http://www.w3.org/TR/2004/WD-SVG12-20040510/>.