

Customizable Notations for Kernel Formalisms *

Luciano Baresi, Alessandro Orso, Mauro Pezzè
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Piazza Leonardo da Vinci 32
Milano, Italy, I-20133
[baresilorso|pezze]@elet.polimi.it

Abstract

Rigorous formal methods and intuitive graphical notations can greatly enhance the development of complex computer systems. Formal methods guarantee non-ambiguity and support powerful analysis techniques. Intuitive graphical notations facilitate the communications between engineers preventing errors due to misunderstandings.

Unfortunately, tools and techniques based on formal methods do not usually support adequate graphical notations; while tools and methods based on powerful graphical notations often lack formal foundations.

This paper proposes a technique that allows kernel formalisms to be accessed through powerful graphical notations. The proposed technique allows graphical notations to be tailored to the needs of the specific application domain. This paper focuses on the tool support.

1 Introduction

The development of large and complex computer systems can be improved by using intuitive graphical notations and unambiguous formal methods. Intuitive graphical notations facilitate the communications between software engineers by capturing the complex relations among components, separating concerns, and providing powerful browsing facilities. Formal methods add rigor to the produced models, avoiding problems of interpretation and improving readability by providing a unique interpretation.

Unfortunately, the graphical notations supported by formal methods and tools are often simple and primitive and retain only few of the advantages of graphical approaches. On the other hand, most powerful graphical notations are ambiguous and provide analysis capabilities limited to syntactic aspects [6]. In the past, the benefits of formal methods and rich graphical notations have been integrated by either enriching the essential graphics of formal methods or by giving formal semantics to rich graphical notations through a mapping to a kernel formalism. Examples of the former approach can be found in [7, 1]; examples of the latter can be found in [9, 4]. The aforementioned approaches solve the problem for specific

graphical notations and formal methods, but do not provide a general solution. Each solution imposes a specific graphical notation and kernel formalism, that cannot be easily adapted to fit different classes of users and problem needs. Moreover, these approaches require substantial reengineering of existing tools or the development of a new generation of CASE tools.

This paper presents a general solution that allows new graphical notations to be mapped to the required kernel formalism. This approach imposes neither a specific notation nor a specific formalism: users can go on using their familiar formalism and the interface of their favorite CASE tool. The approach of this paper offers the opportunity to animate and analyze graphical specifications through execution and analysis of a kernel operational formalism. Moreover, the proposed approach do not require the development of new CASE tools, but can be based on the existing mature CASE technology.

The proposed technique relies on two components:

- A customization editor, that allows the definition of a mapping between the chosen graphical notation and a kernel operational formalism.
- A Run-Time semantic Translator, hereafter RTT, that translates input models into the kernel formalism referring to the mapping introduced through the customization editor; the produced representation can be executed and analyzed; all the significant events are translated into events of the end-user notation and presented in a suitable fashion.

2 The Approach

The goal of the proposed approach is to provide a tool that can be customized for different graphical notations and kernel formalisms to better fit the specific needs of the application domain.

The tool itself provides neither specific graphical notations nor kernel formalisms, but relies on existing CASE tools supporting the required graphical notation and kernel formalism. Given a graphical notation and/or a kernel formalism no changes to the tool itself are required, but only the definition of a set of rules that customize the tool to work with the chosen formalisms.

*This work has been partially sponsored by the ESPRIT Project EP8593 IDERS.

The customization approach offers the following advantages:

- it supports evolution of CASE platforms. Changes in the end-user notation do not require re-implementation of the environment from scratch, but simple changes to the set of customization rules.
- As the effort to customize an environment is much less than the one to develop a new environment, customization can support applications that would benefit from formal methods but do not own a market wide enough to justify the development of specific tools.
- It allows the experimentation of new notations. On the contrary, existing tools tend to restrict the range of the used formalism to the ones that obtain a positive answer from the market, thus limiting the creativity and the growth of new formal methods.

Figure 1 gives an high-level view of the environment. The main components are:

- **Customization Editor:** it supports the construction of the rules that define a graphical notation with respect to a specific kernel formalism. The rules produced during the customization activity are stored in the **Repository**, that represents the interface between the off-line definition of a graphical notation and the **RTT**.

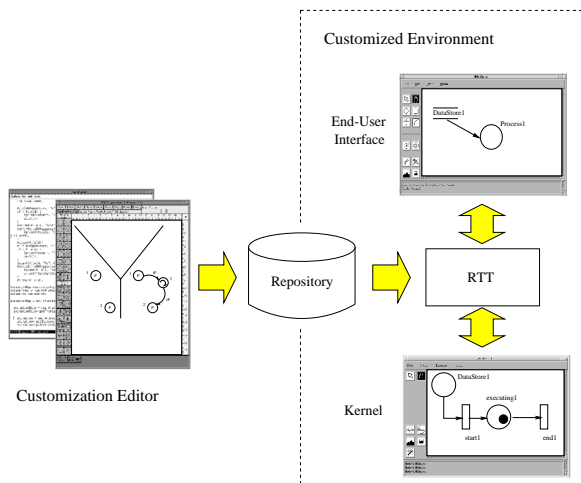


Figure 1: A high-level view of the environment

- **Customized Environment:** it constitutes the run-time support of the customization process. It comprises:
 - **End-User Interface:** it is the CASE tool chosen by end-users to interact with the environment and to design their own models, using the formalism they prefer.

- **RTT:** it is the run-time support to interact with the kernel formalism. During model construction, it transforms end-user models into kernel models. During kernel execution, it maps firings and states of the kernel model to the end-user interface according to the notation in use. The rules to perform the translations are retrieved from the **Repository**.
- **Kernel:** it is the formal engine. It creates the kernel formal models of the specifications, executes and analyzes them. For simplicity, in this paper we use high-level timed Petri nets (HLTPNs [2]) as kernel formalism, although in principle the approach described can refer to a generic operational formalism.

3 The Customization Process

The customization process produces all the information needed to tailor the RTT for the chosen graphical notation. This activity produces a formal definition of the syntax and the semantics of the selected notation in terms of the kernel formalism.

The formal definition of the syntax of a well known graphical notation is a matter of assessing the interpretations of common practice. The experimentations carried on so far reveal a tedious but straightforward process.

The definition of the semantics often requires careful examination of all the details of the graphical notation and can result in a complex activity.

Another aspect is the definition of the appearance of notation symbols, i.e., the concrete syntax. This customization depends on the end-user interface. Since we add formality and executability to notations already supported by existing CASE tools, a concrete syntax already exists.

The customization activity consists in defining the following aspects:

- **Abstract Actions-Rules Mapping:** it states the correspondences between end-user actions and graph-grammar productions. The abstract actions-rules mapping associates a rule, i.e., a pair of graph grammar productions, with each legal end-user action. The absence of an entry for non legal end-user actions is signaled as an error.
- **Abstract Syntax Graph Grammar:** ASGG defines the syntax of the chosen notation. It specifies the actions of a syntax directed editor that creates the abstract model according to the chosen notation. A detailed explanation of graph-grammars and their use to define the syntax of graphical languages can be found in [3, 5, 10].
- **Semantic Graph Grammar:** SGG defines the semantics of the notation, i.e., how the kernel model is modified with respect to the changes in the end-user model made by the application of the related ASGG productions.

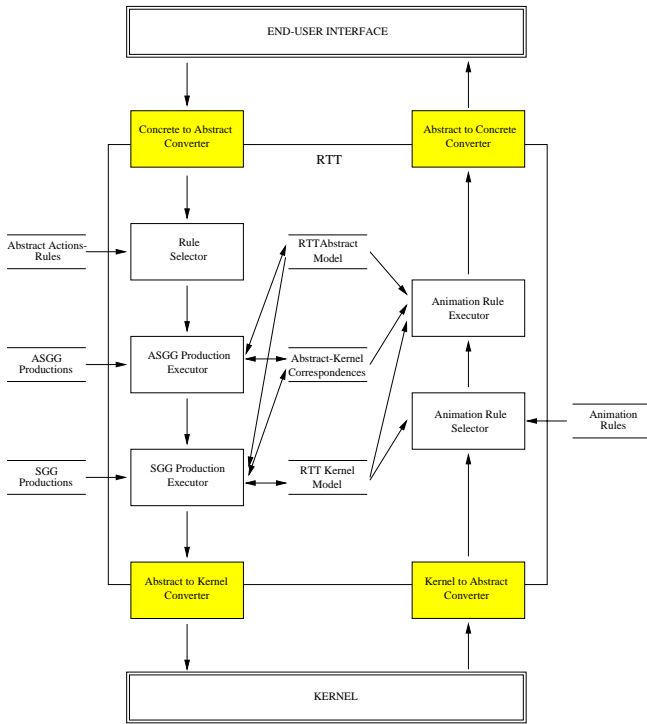


Figure 2: RTT architecture

- **Animation Rules:** they translate events of the kernel model into suitable animations of the end-user notation. Animation rules are associated with transition “types”. When a transition is added to the kernel model, it is assigned a “type”. Animation rules relate transition firings to changes of the corresponding syntax symbols.

The translation of abstract animations into the concrete end-user interface format is up to the *Abstract to Concrete Converter*, described in the next section.

4 The Run-Time semantic Translator

Figure 2 shows the architecture of the Run-Time semantic Translator. Boxes represent functional components: doubly bordered boxes indicate external components, gray boxes represent elements specific to the particular external tools, and white boxes indicate components that do not depend on specific tools. Components indicated with white boxes are customized for the selected notation through suitable rules in the repository. Gray boxes are the only domain-specific components, that must be re-designed to interact with a different CASE tool. Arcs indicate data flows and open boxes stand for data repositories, either provided by the *Customization Editor* or internal to the *RTT*.

The *End-User Interface* is not constrained by *RTT*. Any CASE tool can be used, provided that it allows end-user actions to be exported. The only intrinsic limitation on the *End-User Interface* is related

to the definition of a mapping of the supported end-user notation into the kernel formalism. Although there exist no theoretical limitations, the mapping can be complex, especially if the end-user notation and the kernel formalism do not present any homogeneity. The experiments recalled in Section 5 provide enough evidence of the feasibility of the approach, at least for CASE tools supporting operational end-user notations.

The *Kernel* must support the required kernel operational formalism; it must react to external requests and return execution and analysis results.

The *Concrete to Abstract Converter* translates legal end-user actions into corresponding abstract actions. The implementation of this component strictly depends on the communication mechanism adopted to connect the *End-User Interface* and the *RTT*.

The *Rule Selector* selects from the *Abstract Actions-Rules* repository the rule corresponding to the abstract action identified by the *Concrete to Abstract Converter*, i.e., identifies a couple of ASGG and SGG productions, and return their identifiers.

The *ASGG Production Executor* retrieves the production, identified by the *Rule Selector*, from the *ASGG Productions* repository and executes it. The execution of ASGG productions modifies the two repositories referred to as *RTT Abstract Model* and *Abstract-Kernel Correspondences*, that are the *RTT* internal representation of the current end-user model and a mapping between end-user symbols and kernel elements respectively.

The *SGG Production Executor* executes the semantic production, extracted from the *SGG Productions* repository. Since the computation of the attributes of an SGG production can refer to elements of the corresponding ASGG production, the *SGG Production Executor* needs to access the *RTT Abstract Model*. The execution of a semantic production modifies the *RTT Kernel Model*, adds the identifiers of the new objects to the *Abstract-Kernel Correspondences*, completing the mapping between *RTT Abstract Model* and *RTT Kernel Model* elements, and produces a set of directives for the *Kernel* indicating the modifications.

The *Abstract to Kernel Converter* is part of the *RTT* interfaces: it translates messages from the *RTT* format to the *Kernel* format.

Once end-users terminate the construction of a model, they can choose to validate the specification by visualizing the execution of the kernel representation.

The *Kernel to Abstract Converter* translates the data produced by the *Kernel* into the format required by the *RTT*.

The *Animation Rule Selector* identifies the animation rule, that is then executed by the *Animation Rule Executor*

The *Abstract to Concrete Converter* translates the animation events produced by the *Animation Rule Executor* into the syntax used by the CASE tool, that displays the results.

5 Customization Experiences

To validate the approach, we built a prototype of the *Customization Environment*, that interacts with a formal kernel based on high-level Petri nets (Cabernet [8]) and with several end-user interfaces that include commercial CASE tools, e.g., Software through Picture (StP), and in-house tools based on Motif and Tcl-Tk.

The prototype has been incrementally validated with several specific customizations. The first case study has been investigated for a preliminary version of the prototype that included only semantic translation, but not animation. Its purpose was to experiment the usage of graph-grammars in defining both the abstract syntax and the semantics of a notation. In conducting these first trials we used StateCharts. The transformation of StateCharts constructs into Petri nets is not trivial, and provided important insides of the approach.

The first notation for which we provided full customization, i.e., semantic translation and animation, has been FIFONets. We customized both StP and the in-house editors to be used with FIFONets.

A third case study refers to a design notation based on Petri nets. In this case Petri nets were enriched with POSIX compliant constructs, i.e., messages, mailboxes, semaphores, shared memories and tasks. This experiment provided important results, being the first complex case study where end-user notation and kernel formalism are not completely disjoint.

We are currently working on an evolution of De Marco's structured analysis proposed by Hatley and Pirbhai [6] and supported by Software through Picture.

6 Conclusions

The customization approach presented in this paper allows end-users to benefit from formal techniques without changing their preferred notations and CASE tools. The experiments carried on so far indicates that the technique applies to a wide range of end-user notations and the customization toolset can be used with commercial as well as in-house CASE tools.

The experimentations conducted with the RTT presented so far revealed the usefulness and the potentialities of the approach based on customization.

We plan to define an end-user interface general framework and a way to customize it for any specific notation. Basically the plan is to extend the customizable toolset up to the concrete interface.

As user notations often own a textual part in addition to the graphic one, it would be useful to offer customization services for textual aspects too. Such aspects are currently investigated within the IDERS Project.

Finally, new notations will be examined and all the customization rules produced. These case studies will help both to tune the approach and to test and validate the prototypes. Planned applications are formalisms to design electricity networks, to specify the control of robot arms and to model diagnostic processes.

Acknowledgments

The authors wish to thank Carlo Ghezzi, Fabio Lameri, Michele Sfondrini, Paola Cherubini, Piero Zanchi, Dario Galbiati, and the "Cab Crew".

References

- [1] E. Brinksma. A tutorial on LOTOS. In *Protocol Specification, Testing, and Verification, V*, pages 171–194. M. Diaz editor. Elsevier Publishing Company 1986.
- [2] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezzè. A Unified High-Level Petri Net Model For Time-Critical Systems. *IEEE Transactions on Software Engineering*, 17(2):160–172, February 1991.
- [3] H. Ehrig. Introduction to the Algebraic Theory of Graph Grammars. In *Proceedings of the International Workshop on Graph Grammars and their Application to Computer Science and Biology*, pages 1–69. V. Claus, H. Ehrig, and G. Rozenberg editors. Lecture Notes in Computer Science, Volume 73. Springer Verlag 1979.
- [4] M. Fraser, K. Kumar, and V. Vaishnavi. Informal and Formal Requirements Specification Languages: Bridging the Gap. *IEEE Transactions on Software Engineering*, 17(5), May 1991.
- [5] C. Ghezzi and M. Pezzè. Towards Extensible Graphical Formalisms. In *Proceedings of the 7th International Workshop on Software Specification and Design*. IEEE-Computer Society Press, December 1993.
- [6] D. Hatley and I. Pirbhai. *Strategies for Real-Time System Specification*. Dorset House, New York, 1987.
- [7] P. Huber, K. Jensen, and R. Shapiro. Hierarchies in Coloured Petri Nets. In *Proceedings of the 10th International Conference on Application and Theory of Petri Nets*, Bonn (Germany). Springer Verlag, June 1989.
- [8] M. Pezzè and C. Ghezzi. Cabernet: An Environment for the Specification and Verification Analysis of Real-Time Systems. In *Proceedings of 1992 DECUS Europe Symposium*, Cannes (France), September 1992.
- [9] R. Elmstrøm, R. Lintulampi, and M. Pezzè. Automatic Translation of SA/RT to ER Nets. *The International Journal of Time-Critical Computing Systems*, 5(2/3), Kluwer Academic Publisher 1993.
- [10] J. Rekers. On the Use of Graph Grammars for Defining the Syntax of Graphical Languages. In *Proceedings of Colloquium on Graph Transformation and its application in Computer Science*, March 1994.
- [11] P. Ward and S. Mellor. *Structured Development for Real-Time Systems*, volume 1-3. Yourdon Press, New York, 1985-1986.