Customizable Notations for Kernel Formalisms *

Luciano Baresi, Alessandro Orso, Mauro Pezzè
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Piazza Leonardo da Vinci 32
20133 Milano, Italy
tel: 39-2-2399-3400
[baresi|orso|pezze].elet.polimi.it

April 27, 1995

Abstract

Both 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 and describes the experiences gained so far in accessing different formal kernels through commercial and ad-hoc CASE tools.

Keywords: formal methods, graphical notations, CASE tools, customization.

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, supporting different views of the system, and providing powerful browsing facilities [26]. Formal methods add rigor to the produced models, avoiding problems of interpretation and improving readability by providing

*This work has been partially sponsored by the ESPRIT project EP 8593 IDERS and by Progetto Finalizzato Sistemi Informativi e Calcolo Parallelo (CNR).

a unique non-ambiguous interpretation. Moreover, formal methods support powerful semantic analysis capabilities that include animation, reachability analysis, model checking and theorem proving [19, 5].

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 [2, 1]. On the other hand, most powerful graphical notations are ambiguous and provide analysis capabilities limited to syntactic aspects [31, 14, 8]. In the past, the benefits of formal methods and rich graphical notations have been joined 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. Example of the former approach are [17]; examples of the latter are [23]. 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 problems. 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 does not impose a specific notation nor a specific formalism. Users can go using their familiar formalism and, in some cases, even 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 techniques 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.

This paper is organized as follows. Section 2 illustrates the general approach. Section 3 presents the customization process. Section 4 gives an insight of the RTT architecture. Section 5 describes the current status of the prototype used for experimenting the approach. Section 6 presents the experiences with different notations used as case studies to test and validate the methodology. Section 7 concludes discussing future plans.

2 The Approach

The ultimate 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 either the required graphical notation or 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 for the chosen notation and kernel formalism. The technique for integrating the tool with existing CASE tools and the few related requirements are discussed in Section 4.

The customization approach offers the following advantages:

- it supports evolution of the CASE platform. 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, while 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.

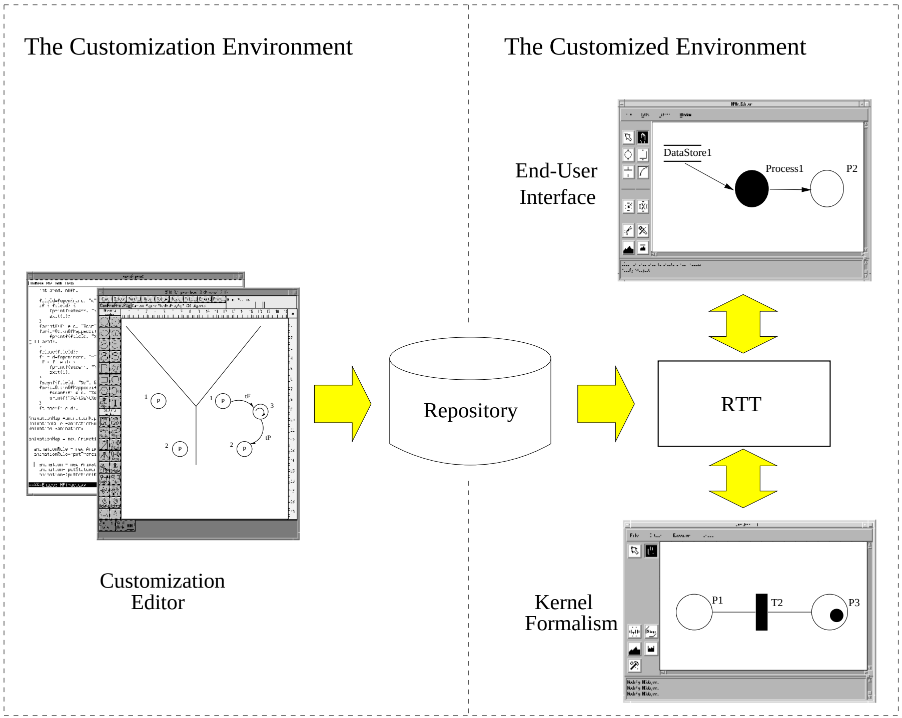

Figure 1: A High-Level View of the Environment

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

Customization Editor, that supports the construction of the rules that define a graphical notation with respect to a specific kernel formalism. Details on the data to be provided are given in Section 3. Graphical editing can improve the customization editor, however customization data can be given textual form. A general-purpose text editor (vi, emacs) provides all the functionalities of a customization editor. 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 run-time semantic translator.

Customized Environment, that constitutes the run-time support of the customization process. It comprises:

End-User Interface, which is the CASE tool chosen by the end-users to interact with the environment and to design their own models, using the formalism they prefer.

RTT, which is the run-time support for executing and analyzing users' models. Roughly speaking, it can be seen as a two-way translator. During model construction, it receives data from the modeling tool and transforms them to build the kernel representation. 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**. Although the mapping from the kernel model to the graphical end-user interface can involve both execution and analysis results in terms of firings and states of the kernel model, in this paper we concentrate only on animation based on firings to simplify the presentation.

Kernel, which is the actual formal engine. It creates the kernel formalism representation of the current model, executes it and analyzes it. Although in principle the approach described in this paper can refer to a generic operational formalism, so far we experimented only with high-level timed Petri nets (HLTPNs [4]) to take advantage from the available tools within the project. For simplicity, hereafter we assume HLTPNs as the kernel formalism.

The two main components outlined so far lead to two different kind of users:

Super-users, that are the experts of the customization. They provide the rules that describe end-user notations.

End-users, that are the experts of the applications and of the notation the environment is customized for. They use the environment through the end-user interface.

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 syntax and 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. The semantics can be defined either before starting the production of the rules, or it can be obtained as a side effect of the customization process. From the authors' point of view the first solution is to be preferred, since it avoids rules reworking each time a change in the definitions happens. However in absence of a clear and unique view of the semantics an incremental definition which flows in parallel with the customization can help the super-user in evaluating different solutions. The choice does not depend on the RTT, but is up to super-users. 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 mainly in defining two appropriate graph-grammars, as illustrated in [6], however the complete set of rules needed for a complete mapping are:

Abstract Actions-Rules Mapping, that 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, for 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) Productions, that define the syntax of the chosen notation by specifying the actions of a syntax directed editor that creates the abstract model according to the chosen notation. A detailed explanation on the use of graph-grammars can be found in [10]; a similar approach is in [25].

Semantic Graph-Grammar (SGG) Productions, that define 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 production.

Animation Rules, that 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", depending on the role played with respect to the semantics of the user symbol it translates. For instance the kernel representation of a *Data Transformation*[1] could comprise two transitions, that model the start and termination of the transformation. Animation Rules relate transition firings to changes of the corresponding syntax symbols. For example the firing of transitions "start" of *Data Transformations* can be related to the change of the state of the transformation from idle to executing[2].

The animations are currently related to transition "types" only, while, in future, they will consider also the values of the tokens consumed and produced by the firing.

We refer to abstract animations since we do not consider a specific format that depends on the concrete user-interface. The translation of abstract animations into the concrete end-user interface format is up to the *Abstract to Concrete Converter* (see Figure 2).

[1]Represented by a bubbly in usual Data Flow Diagrams.
[2]It is often modeled by the bubbly changing color

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; white boxes indicate components that do not depend on specific tools, customized for the selected notation. 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 chosen architecture well supports a distributed environment, in which the *End-User Interface*, RTT and the *Kernel* are three distinct processes, that run on different machines.

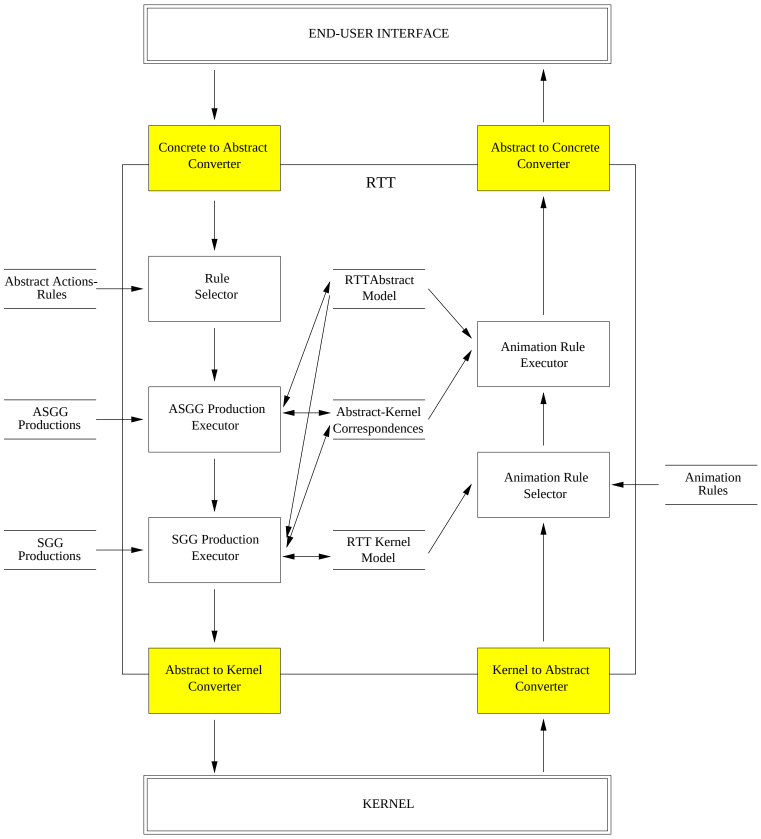

Figure 2: RTT Architecture

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 to 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. Moreover, the experiments recalled in Section 6 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, must react to external requests and must return execution and analysis results. Basically they both must be service-based tools (see [24] for a complete explanation). Details on the CASE tools used so far as *End-User Interface* and *Kernel* are given in Section 6.

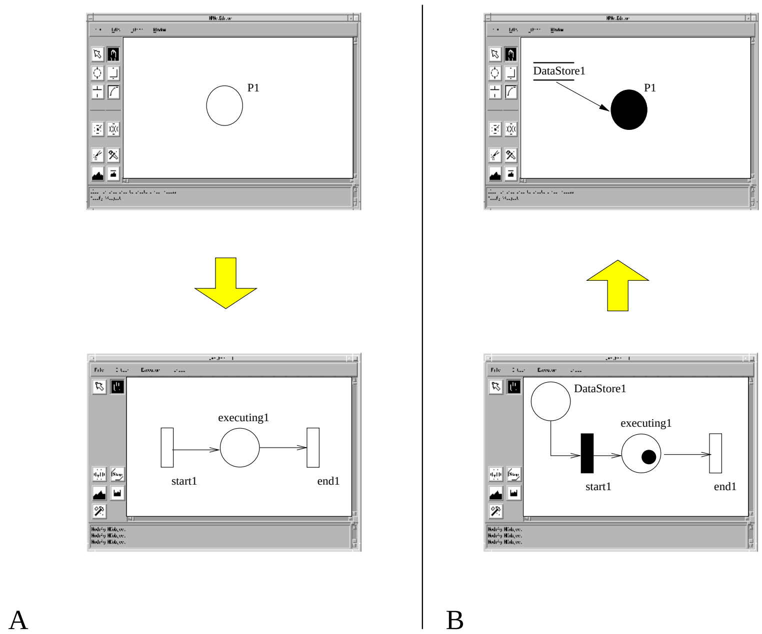

Figure 3: A Usage Example

The *Concrete to Abstract Converter* translates legal end-user actions into corresponding abstract actions. Legal actions to be triggered and converted concern either model construction, e.g., adding or deleting new elements, or "general services", e.g., loading, saving models or quitting the application. The implementation of this component strictly depends on the communication mechanism adopted to connect the *End-User Interface* and the RTT. The experience gained so far is presented in Section 6.

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* element(s), 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. The RTT produces these messages according to a general abstract format, that can be different from the interface of the underlying *Kernel*. The experience of integrating with different kernels is described in Section 6.

The components described so far are responsible for building the kernel model corresponding to the end-user model. An example of usage can described referring to Figure 3.A, that assumes a general data flow notation as end-user graphical notation and Petri nets as the internal kernel. If an end-user draw a Data Transformation in the canvas, the *End-User Interface* traps this action and propagates it to the *Concrete to Abstract Converter*. The *Concrete to Abstract Converter* translates the addiction of a Data Transformation into a suitable RTT internal format. The *Rule Selector* retrieves the rule, say `addDT`, associated with the incoming abstract action. The *ASGG Production Executor* executes the ASGG production `addDT`, that adds a Data Transformation to the *RTT Abstract Model* and initializes an entry in the *Abstract-Kernel Correspondences* with the Data Transformation identifier. The *SGG Production Executor* selects the semantics production `addDT` and executes it. The execution of the semantic production `addDT` creates two transitions, a place and two arcs in the *RTT Kernel Model*. The identifiers of the created elements complete the entry in the *Abstract-Kernel Correspondences*. Finally the *Abstract to Kernel Converter* sends suitable messages to the *Kernel* to add the required transitions, places and arcs to the kernel model.

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

The *Kernel to Abstract Converter* translates the data produced by the *Kernel* into the format required by the RTT. In the current prototype the mapping is limited to the identifier of the fired transition, a more complete mapping would take into account also the enabling tuple and the tokens produced by the execution.

The *Animation Rule Selector* identifies the "type" of the fired transition accessing the *RTT Kernel Model*, and retrieves the corresponding animation rule from the *Animation Rules* repository.

The *Animation Rule Executor* executes the animation rule identified by the *Animation Rule Selector*. The abstract object related to the fired transition is retrieved from the *Abstract-Kernel Correspondences* repository, while the connected objects are retrieved from the *RTT Abstract Model*.

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

As an example, Figure 3.B illustrates the animation of the firing of transition `start1` corresponding to the starting of execution of Data Transformation `P1`. The *Kernel* fires transition `start1` of "type" `StartDataTransformation`. The event is notified to the *Animation Rule Selector* after being filtered by the *Kernel to Abstract Converter*. The *Animation Rule Selector* retrieves the animation rule associated with transition type

`StartDataTransformation`.

The *Animation Rule Executor* identifies the syntax elements involved in the animation action. For example, the animation rule shown in Figure 4 identifies the Data Transformation `P1` (`OBJECT`) and all its input flows (`ALLINNODES`), and associates the retrieved objects with the corresponding animation events (`Blacken P1` and `Blink` its input flows).

The *Abstract to Concrete Converter* converts animations for the *End-User interface*, that blinks all the flows entering the Data Transformation and blacken the Data Transformation itself.

```
OBJECT      ''                        EventType ''Blacken''
ALLINNODES  Type=DataTransformation   EventType ''Blink''
```

Figure 4: A Trial Animation Rule

5 A Prototype

A prototype of the *Customization Environment* has been implemented as part of the ESPRIT Project IDERS. The prototype has been demonstrated in April 95 at the tools fair of the International Workshop on Industrial-Strength Formal Techniques (WIFT95). The first prototype focuses mostly on RTT and its interfaces with *End-User Interfaces*, *Kernels*, and the customization editor, i.e., the *Repository*.

The support to super-users, i.e., the customization editor is provided by conventional text editors (like vi or emacs). Abstract syntax and semantic graph-grammars productions as well as abstract action-rules mapping and animation rules are supplied to the *Repository* in a textual format. The need of manually deriving this textual representation is a consistent drawback. We are currently designing and developing an ad-hoc graph-grammar editor. We decided to implement a new editor instead of using existing software, e.g., [11, 16, 28], because none of the quoted editors easily supports our graph-grammar notation. The first step, still to be completed, is the design of a filter from a widely available graphical editor, xfig, to the textual format used to provide the RTT with graph-grammar productions. The lessons learned during this phase will be the base of the development of our ad-hoc editor.

The currently available RTT prototype implements all the functionalities outlined in Figure 2. It is implemented in C++ and runs on UNIX platforms. The currently available prototype is interfaced with several end-user interfaces and kernels to validate the proposed mechanism against different case studies.

Currently available end-user interfaces include commercially available CASE tools, such as Software through Picture (StP) [30], and in-house tools based on Motif [15] and `tcl-tk` [20]. The communication mechanism adopted in the three cases are different and are worth some comments.

Being StP a commercial product we do not have access to the source code; thus all the mechanisms have been implemented using the customization facilities provided with the environment. The identified solution is based on `QRL`, the StP Query and Reporting Language. An appropriate `QRL` script has been written for each relevant end-user action. Each time an action is performed, the related script writes the relevant information on the repository (a UNIX file, in this case). Alternatively, the relevant information could be extracted from the StP repository. This solution is currently under investigation; related problems are outlined in Section 7.

On the contrary, we had complete visibility of in-house tools. In these cases we augmented the interfaces of in-house tools with a communication layer providing ad-hoc message-exchange mechanisms. The interface and the RTT can communicate either through the repository or through UNIX sockets.

The currently available kernels are all based on Petri nets. They include Cabernet, a tool developed at Politecnico di Milano (Italy) presently licensed to more than forty sites world wide [22] and IPTES, a CASE tool based on Petri nets developed as part of an ESPRIT Project[3]. In both cases, we have enough access to the source code of the tools to provide socket-based interfaces. Accessing the kernels do not imply specific conceptual problems, being in both cases a simple invocation of methods of the object kernel.

6 Customization Experiences

The current 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 [12]. The transformation of StateCharts constructs into Petri nets is not trivial, and provides important insides of the approach.

The first notation for which we provided full customization, i.e., semantic translation and animation, has been FIFONets [27]: Petri nets with places in which tokens are queued according to their production. In this case our aim was to study how to provide users with the animation of their models; thus we chose a notation in which the mapping between executions of the kernel model and front-end animations were straightforward, to focus only on the problem of animation. We customized both StP and the in-house editors to be used with FIFONets.

A third case study [9] refers to a design notation based on Petri nets. In this case Petri nets were enriched with POSIX compliant constructs [18], i.e., messages, mailboxes, semaphores, starved 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.

The formalism we are currently working on is an evolution of De Marco's structured analysis proposed by Hatley and Pirbhai [13]. This notation will be used within the ESPRIT Project IDERS for modeling the requirements of a radar control system [21]. Presently, we defined most of the semantic translation rules and the converters for StP and the in-house editors as end-user interfaces, and for Cabernet and IPTES as kernels. The complete set of semantic transformations and animation rules will be available for the specification and analysis of the radar control system by the end of July 95.

A positive side effect of this last case study is to provide formal semantics to the Hatley and Pirbhai notation, often object of different interpretations. Such semantics

[3] The ESPRIT Project EP5570 IPTES: Incremental Prototyping Technology for Embedded Real-Time Systems.

is the most effective way of providing animation and analysis capabilities not supplied within StP yet.

7 Conclusions

The experimentation conducted with the RTT presented so far revealed the usefulness and the potentialities of the approach based on customization. Our experience in ESPRIT suggests that customization of end-user graphic notations can be a winning strategy to better support formal methods in industrial practice, allowing step by step migrations instead of imposing drastic changes.

The experiments described in the previous sections highlights some limitations of the current prototype and suggests future evolutions.

We plan to enhance the support to super-users by developing graphical editors specific to the customization activity, to support easily design of graph-grammar productions.

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.

Further studies on graph-grammars and graph rewriting theory are needed to find mechanisms allowing post-mortem translation of end-user models instead of imposing on-line transformations, as it is currently done.

As user notations often own a textual part in addiction to the graphic one, it would be useful offering customization services for this aspect too. It would vastly increase the number of formalisms the RTT could be fully customized for. Finally, new notations will be examined and all the customization rules produced. These case studies will help both in testing and validating the prototypes and in raising new requirements to the RTT. Planned applications are formalism to design an electricity network ([7]), to specify the control of a robot arm ([3]) and to model a diagnostic process ([29]).

Acknowledgments

The authors wish to thank Carlo Ghezzi for his precious comments and suggestions, Fabio Lameri, Michele Sfondrini, Paola Cherubini, Piero Zanchi and Dario Galbiati for their valid aid in different stages of experiments with prototypes and notations. A thought goes to everyone that has been part of the "Cab Crew".

References

[1] *IEEE Software*, volume 7(5). IEEE, September 1990.

[2] *IEEE Transactions on Software Engineering*, volume 21(2), chapter Best Papers of Formal Methods Europe '93. IEEE, February 1995.

[3] A. Caloini, G. Magnani, and M. Pezzè. Designing Sensor Based Control Systems Using Petri Nets. Technical report, Politecnico di Milano, Dipartimento di Elettronica ed Informazione, 1995.

[4] S. M. Carlo Ghezzi, Dino Mandrioli and M. Pezzé. A Unified High-Level Petri Net Model For Time-Critical Systems. *IEEE Transactions on Software Engineering*, 17(2):160–172, February 1991.

[5] E. W. Dijkstra. *Formal Development of Programs and Proofs.* Addison-Wesley, 1990.

[6] H. Ehrig. Introduction to the algebraic theory of graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Graph Gramars and their Application to Computer Science and Biology*, pages 1–69. Springer, 1979. Lecture Notes in Computer Science, Volume 73.

[7] S. Faltracco and P. Gentina. Analisi di sistemi in tempo reale. Master's thesis, Politecnico di Milano, Dipartimento di Eletronnica e Informazione, 1994.

[8] A. Fuggetta. A Classification of CASE Technology. *Computer*, December 1993.

[9] R. Gargioli. Una metodologia per la specifica di reti di task per mezzo di reti di petri. Master's thesis, Politecnico di Milano, Dipartimento di Elettronica e Informazione, 1994.

[10] C. Ghezzi and M. Pezzè. Towards extensible graphical formalisms. In *Proceedings of the 7th International Workshop on Software Specification and Design.* IEEE-CS, December 1993.

[11] H. Göttler. Diagram editors = graphs + attributes + graph grammars. *International Journal Man-Machine Studies*, (37):481–502, 1992.

[12] D. Harel. StateCharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, (8), 1987.

[13] D. Hatley and I. Pirbhai. *Strategies for Real-Time System Specification.* Dorset House, New York, 1987.

[14] D. J. Hatley and I. A. Pirbhai. *Strategies for Real-Time System Specification.* Dorset House, New York, 1987.

[15] D. Heller. *Motif Programming Manual*, volume 6 of *The Definitive Guides to the X Window System.* September 1991.

[16] M. Himsolt. Graph^{Ed}: The design and implementation of a graph editor. Technical report, University of Passau, Germany, 1994.

[17] P. Huber, K. Jensen, and R. Shapiro. Hierarchies in coloured Petri nets. In *10th International Conference on Application and Theory of Petri Nets*, Bonn (Germany), June 1989. Springer Verlag.

[18] IEEE. *Real-Time extensions to POSIX. IEEE Standard P1003.1b.* IEEE, March 1993.

[19] IEEE. *Workshop on Industrial-Strength Formal Specification Techniques*, April 1995.

[20] J. K. Ousterhout. *TCL and the TK Toolkit.* 1993.

[21] A. Perrotta, G. Massara, G. Dipoppa, and R. Bove. A Logical Model of the RPCM System. Technical report, Alenia, January 1995.

[22] M. Pezzé. Cabernet: A Customizable Environment for the Specification and Analysis of Real-Time Systems. *Submitted for publication*, 1994.

[23] R. L. R. Elmstrø m and M. Pezzé. Automatic translation of SA/RT to ER nets. Technical report, IFAD, VTT and PDM, February 1992.
IPTES Doc.id.: IPTES-PDM-17-V1.3.

[24] S. Reiss. Connecting Tools using Message Passing in the FIELD Program Development Environment. *IEEE Software*, pages 57–67, July 1990.

[25] 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.

[26] G.-C. Roman and K. C. Cox. Program visualization: The art of mapping program to pictures. In *Proceedings of the 14th ICSE*, pages 412–420, Melbourne, Australia, 1992.

[27] G. Roucariol. *Advanced in Petri Nets, Part I*, chapter FIFO-Nets. Lecture Notes in Computer Science N. 254. Springer Verlag, Berlin, 1987.

[28] A. Schürr. PROGRESS, A Visual Language and Environment for PROgramming with Graph REwriting Systems. Technical Report AIB 94-11, Aachener Informatik-Berichte, 1994.

[29] T. Catarci and M. Di Paola and A. Gargiulo and M. Pezzè. A Formal Model for Quality Assurance in Coloproctologic Surgery. In *Proceedings of Medinfo95*, Toronto, July 1995.

[30] IDE. *Software through Pictures: Using the StP/SE Editors*. Interactive Development Environments, February 1994. Release 5.

[31] P. Ward and S. Mellor. *Structured Development for Real-Time Systems*, volume 1-3. Yourdon Press, New York, 1985-1986.