

A Toolbox for Automating Visual Software Engineering

Luciano Baresi¹ and Mauro Pezzè²

¹ Dipartimento di Elettronica e Informazione - Politecnico di Milano
Piazza L. da Vinci 32, I-20133 Milano, Italy
bares@elet.polimi.it

² Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano - Bicocca
Via Bicocca degli Arcimboldi 8, I-20126 - Milano, Italy
pezz@disco.unimib.it

Abstract Visual diagrammatic (VD) notations have always been widely used in software engineering. Such notations have been used to syntactically represent the structure of software systems, but they usually lack dynamic semantics, and thus provide limited support to software engineers. In contrast, formal models would provide rigorous semantics, but the scarce adaptability to different application domains precluded their large industrial application. Most attempts tried to formalize widely used VD notations by proposing a mapping to a formal model, but they all failed in addressing flexibility, that is, the key factor of the success of VD notations.

This paper presents *MetaEnv*, a toolbox for automating visual software engineering. *MetaEnv* augments VD notations with customizable dynamic semantics. Traditional meta-CASE tools support flexibility at syntactic level; *MetaEnv* augments them with semantic flexibility. *MetaEnv* refers to a framework based on graph grammars and has been experimented as add-on to several commercial and proprietary tools that support syntactic manipulation of VD notations.

1 Introduction

Visual diagrammatic (VD) notations [10] have always been widely used in software engineering. Examples of these notations, which are daily exploited in industrial practice, are Structured Analysis [13], UML [9], SDL [14], IEC Function Block Diagram [21]. Graphical elements make these languages appealing and easy to understand. Users can glance the meaning of their models without concentrating on complex textual descriptions. The graphical syntax of these models is well defined, but the dynamic semantics is not clearly (formally) stated. The same graphical model can be interpreted in different ways, relying on human reasoning as the main means to analyze and validate produced specifications.

Formal models have been widely studied in academia and research laboratories [35]. Formal models would provide powerful formal engines that support automatic reasoning about important properties, but the supplied VD interfaces lack domain specificity [17,27]. The many proposals, which tried to combine graphical notations with

formal models, fail in addressing flexibility, i.e., the real problem: They freeze a fixed semantics and over-constrain the VD notation ([16,24]). Consequently, these proposals remained limited to very specific projects and never got any general acceptance.

This paper proposes *MetaEnv*, a toolbox that complements domain-specific VD notations with formal dynamic semantics without missing either flexibility, which is the key factor for the success of VD notations, or formality, which is the base of the benefits of formal methods. *MetaEnv* works with VD notations that can be ascribed with an intuitive operational dynamic semantics [18]. It improves VD notations by adding analysis capabilities, and fostering customizability of the associated semantics. Meta-CASE technology provides flexibility at syntactic level [29], and thus supply the concrete syntax and visualization capabilities; *MetaEnv* provides a formal engine to execute and analyze designed models according to the chosen semantics. Users can both exploit already defined semantics and define their own interpretations. In both cases, they gain all benefits of a formal engine without being proficient in it.

MetaEnv is based on the approach for defining customizable semantics for VD notations described in [6,4]. It supports the definition of formal dynamic semantics through rules that define both a High-Level Timed Petri Net (HLTPN), which captures the semantics associated with the model, and proper visualizations of executions and analysis results. Rules can easily be adapted to new needs of software engineers who may want to change or extend the semantics of the adopted VD notation. This paper presents the conceptual framework of *MetaEnv*, its software architecture, and the results of a set of experiments performed with prototypes. Experiments have been carried out on integrating our prototype with both commercial tools (Software through Pictures [2], ObjectMaker [28], and Rose [32] and in-house user interfaces implemented in tcl-tk, Java, and MATLAB/SIMULINK.

2 Technical Background

In *MetaEnv*, a VD notation consists of two sets of rules: *Building rules* and *Visualization rules*. Building rules define both the abstract syntax, i.e., the syntactic elements and their connectivity, and the operational semantics of each element, i.e., the corresponding HLTPN. Visualization rules translate execution and analysis from HLTPNs to the VD notation.

Building rules are pairs of attributed programmed graph grammar productions [30]. These rules work on the abstract syntax, that is, models are described as directed graphs with typed nodes and arcs, and represent admissible operations for designing models with the chosen VD notation. For each rule, the first production belongs to the *Abstract Syntax Graph Grammars* (ASGG) and describes the evolution of the abstract syntax graph; the second production is part of the *Semantic Graph Grammar* (SGG) and defines the operational semantics as modifications to the associated HLTPN. For example, Figure 1 presents a simple building rule taken from a formalization of Structured Analysis [8]. The rule defines how to connect a Process (P) to an input port (I) (connected to a data flow) through an input consumption (IC) element. The two productions are represented graphically using a Y: the left-hand side indicates the elements on which the production applies; the right-hand side indicates the elements introduced by applying

the production; the upper part indicates how new elements are connected to the graph. The same identifiers associated with nodes in the left-hand and right-hand sides indicate that the two objects are kept while applying the production. The Abstract Syntax production of Figure 1(a) adds a node of type `InCons` (IC) and an edge of type `belong` (b). It adds also an edge of type `connect` (c) between the added IC node and each I node that is connected to the P node through a b edge. The textual attributes indicate that the newly created node (node 2) has three attributes: *name*, *type*, and *action*. The value of the attribute *name* is the concatenation of the name of node 1 (the node of type P) and the string `IC`, the value of attribute *type* is the string `InCons`, and the value of attribute *action* is provided externally (typically by the user).

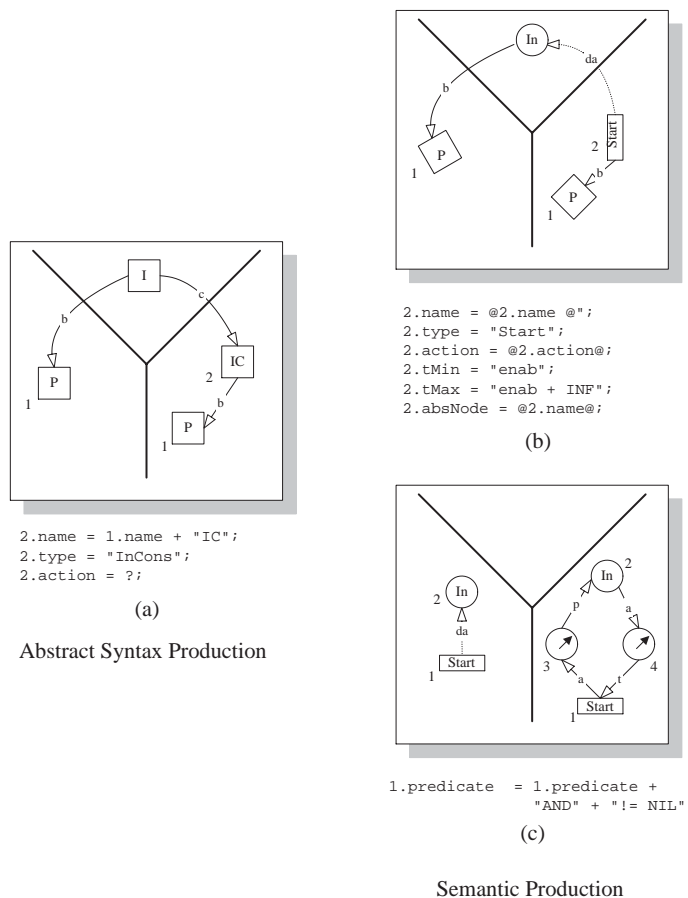


Figure 1. A simple *Building Rule*

The corresponding Semantic production of Figure 1(b) adds a transition of type `Start` and two nodes of type `arc` (graphically identified with an arrow in a circle) be-

tween the newly created `Start` transition and all `In` places that belong to the semantic representation of the process. Special edges, called `sp`-edges, represent sub-production invocations and are drawn using dashed lines. The `sp`-edge of Figure 1(b) triggers the sub-production illustrated in Figure 1(c) that substitutes each `da` `sp`-edge with a pair of arcs. Notice that nodes of type `arc` represent HLTPN arcs and edges define the connectivity among nodes.

Semantic HLTPNs allow VD models to be formally validated through execution, reachability analysis and model checking. Visualization rules translate obtained results in terms of suitable visualization actions on the abstract elements.

```

VisRule va = new VisAction();

if (tr.type() == "Start") {
    foreach pl in tr.preset() {
        Animation an = new Animation();
        an.setEntityId(pl.getAbsId());
        an.setAnimType("readFlow");
        va.addAnimation(an);
    }
    Animation an = new Animation();
    an.setEntityId(tr.getAbsId());
    an.setAnimType("start");
    an.setAnimPars("executing");
    an.addAnimation(an);
}

```

Figure 2. A simple Visualization rule

For example, the rule of Figure 2 describes how the firing of a transition of type `Start` is visualized in terms of Structured Analysis. `Start` transitions (see the building rule of Figure 1) correspond to starting process executions. When transition `tr` is fired, the visualization action (`va`) defines how to animate the data flows, which correspond to the places (`pl`) in the preset of `tr`, and the process corresponding to the transition itself. In particular, action `va` associates animation `readFlow` to all selected data flows and animation `start` to the process.

3 *MetaEnv*

MetaEnv consists of a set of tools that integrate CASE technology with an HLTPN (High Level Timed Petri Nets) engine to augment current CASE tools with analysis and validation capabilities.

MetaEnv is built around a layered architecture (Figure 3). Its components can easily be organized in *concrete*, *abstract*, and *semantic* components according to the level at which they work on the VD notation.

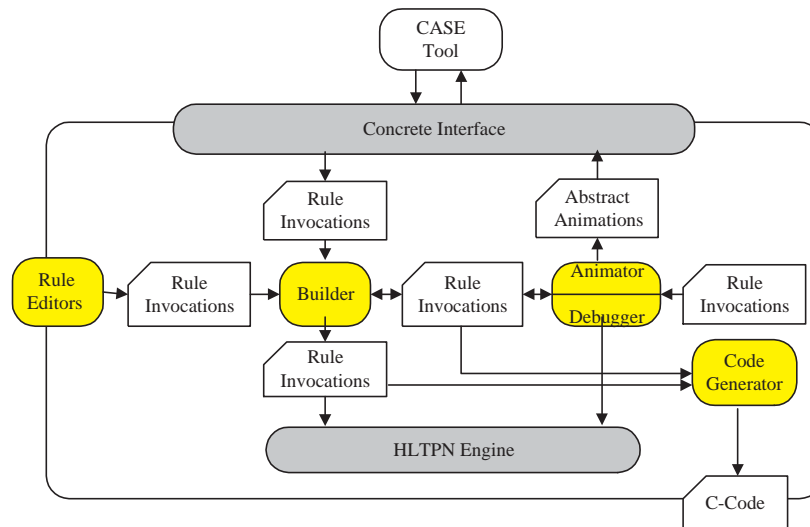


Figure 3. Architecture of *MetaEnv*

Concrete components are the *CASE Tool* and the *Concrete Interface*. The interface allows *MetaEnv* to be plugged in the chosen *CASE Tool*. The only constraint *MetaEnv* imposes is that the *CASE Tool* must be a service-based graphical editor [33]: It must supply a set of services that *MetaEnv* can use to properly store and animate user models. These services are used by the *Concrete Interface* that connects the *CASE tool* to the inner components of *MetaEnv*. This interface defines a two-way communication channel: It transforms user models in suitable invocation sequences of building rules and animation actions in concrete visualizations.

To transform user models in invocation sequences of building rules, the concrete interface can use different policies. The simplest solution is that user actions are immediately translated into rule invocations. Alternatively, the interface can adopt a generative approach that reads complete user models and then defines the sequence according to a predefined partial order among building rules. A detailed analysis can be found in [4].

The data flow in the opposite direction transforms the abstract animations produced by visualization rules in concrete animations. Abstract animations describe visualizations of notation elements in a plain and tool-independent way. The interface adds all details that depend on the specific tool. Differently from all other *MetaEnv* components, the concrete interface varies according to the *CASE tool* that "imposes" the communication means, supplied services and actual implementation. *MetaEnv* requires only that building and visualization rules be defined in a special purpose format.

Abstract components are a *Builder*, an *Animator/Debugger*, and a *Rule Editor*. The *Builder* is a graph grammar interpreter that applies building rules according to the application sequence defined by the concrete interface and it builds both the abstract model (i.e., the abstract representation of the user model) and the HLTPN that correspond to the current model. The *Animator/Debugger* applies visualization rules to place mark-

ings and transition firings from the HLTPN engine. For example, it transforms the execution of the HLTPN, that is, a sequence of transition firings, into a sequence of abstract visualizations. The debugger allows users to control the execution of their models as if they were using a "standard" debugger: They can set breakpoints and watch-points, chose step-by-step execution, and trace the execution. All these services, which are set through the CASE tool, are transformed in terms of constraints on the sequence of abstract animations. A step-by-step execution becomes an execution that stops after each abstract animation; a break point on a specific element of the model means that the execution is suspended at the first abstract animation that involves the selected element.

Both the *Builder* and the *Animator/Debugger* read their rules from external repositories that store all required rules. Rule editors are the personalization-oriented user interfaces of *MetaEnv*. They let users define new rules, but they are not part of the runtime environment. Building rules, i.e., graph grammar productions, are defined with a graphical editor and then processed to move from the graphical representation to the required textual format. Visualization rules can be designed with any textual editor. The use of these editors requires proficiency not only in the particular VD notation, but also in HLTPNs to be able to ascribe meaningful semantics. Most users will never cope with designing new rules; they will do their experiments with already existing formalizations or they will ask experts for special-purpose formalizations.

The *Code Generator* is an abstract/semantic component because it uses inputs from both layers. The code generator automatically produces ANSI C code from VD models. Again, it does not use the concrete representation, but it starts from the HLTPN to get the code semantics and the abstract representation to get a reasonable way to partition the code. Topology and computations are "read" from the HLTPN, while modules and functions are defined according to the structure of the abstract model. Differently from other proposals (for example, [12]), which supplies the code together with the abstract machine to execute it, this component produces raw code to be compiled and linked using standard C compilers.

The Semantic component is the *HLTPN Engine*. It defines, executes and analyzes the HLTPNs obtained through the builder. *MetaEnv* reuses a light version of Cabernet [31] as Petri net tool. Other Petri net tools or even other formal engine would be usable as well. The layered and component-based architecture of *MetaEnv* allows for experiments with different formal engines without rebuilding the system from scratch. The interfaces and the services provided by/required from the engine are clearly stated and thus plugging in a new engine is quite easy.

4 Experimental Validation

The approach described in this paper has been validated by plugging prototype implementations of *MetaEnv* in different front-ends. The integration of the prototype with a front-end requires the selection of the CASE tool and the selection of a set of rules to formally define the addressed VD notation. The experiments described in this section aim at showing the flexibility of *MetaEnv* both in interfacing with different CASE tools and in formalizing different VD notations. The flexibility in interfacing with different CASE tools has been shown by both choosing commercial CASE tools and imple-

menting special-purpose concrete interfaces developed using different languages. The commercial CASE tools used in the experiments are Software through Pictures [2], ObjectMaker [28], and Rational Rose [32]. The special-purpose interfaces described in this section have been developed using tcl-tk, MATLAB-SIMULIK, and Java. The capability of adapting to different VD notations has been shown by both formalizing de-facto standard notations and developing ad-hoc VD notations to address specific needs. The standard VD notations addressed in the experiments include Structured Analysis [13], the Unified Modeling Language (UML) [9], and Function Block Diagram (FBD) [21]. The special-purpose VD notations described in this section are Control Nets, a variant to timed Petri nets for the design of control systems, and LEMMA, a VD notation for modeling diagnostic and therapeutic medical processes. All experiments are summarized in Table 1 that lists the different VD notations, the CASE tools (ad-hoc interfaces) that we used (defined), and the number of building rules required to ascribe the formal semantics.

Structured Analysis	StP, ObjectMaker	50
Function Block Diagram	Ad-hoc interface (MATLAB)	40
UML	Rose	20
Control Nets	Ad-hoc interface (tcl-tk)	30
LEMMA	Ad-hoc interface (tcl-tk, Java)	10

Table 1. Summary of experiments

In all cases, the cost of plugging *MetaEnv* in existing CASE tools, and providing rules for the chosen VD notation, is negligible with respect to the cost of developing a new specific graphical editor. The flexibility of the obtained environments permitted also the experimentation with different semantics without significant extra costs, while ad-hoc "semantic" editors would have hardly be adapted to new semantics.

The following sections briefly overview the main results.

4.1 Structured Analysis

The first VD notation considered in the experiments was Structured Analysis (SA) [13]. SA was chosen because of the variety of uses and related interpretations as well as supporting CASE tools. *MetaEnv* was plugged in two different CASE tools: Software through Pictures (StP) [2] and ObjectMaker [28]. In both cases *MetaEnv* was integrated by filtering the information stored in the repository of the two tools. StP was also connected using the open interface provided by the tool. The integration with the repository works off-line, i.e., the HLTPN is built after completing the models. The integration with the open interface allows the construction of the HLTPN incrementally. We provided a family of rules for addressing the main variants of SA, as described in [8].

The effect of combining different interpretations for the same notation is illustrated in Figure 4 that presents a sample of alternative semantics for a process, i.e., a functional transformation. The figure presents the semantics for a process with two input flows

and one output flow. Semantics (a) and (b) define an atomic instantaneous functional transformation. Semantics (c) defines a functional transformation with duration that can be interrupted. Semantics (a) requires exactly one input to execute the transformation, while semantics (b) and (c) require both inputs. All different semantics can be given by simply selecting suitable building rules, as discussed in [8].

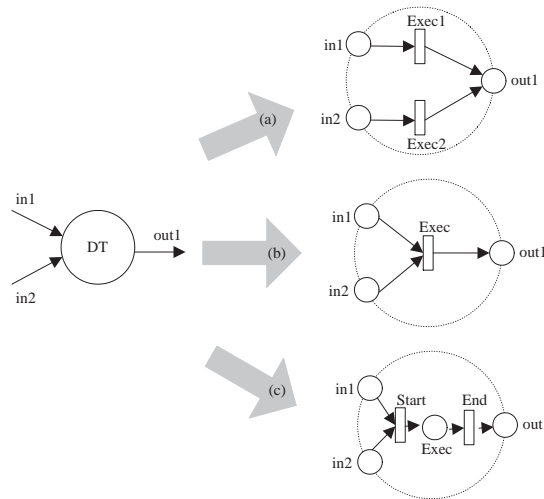


Figure 4. Different semantics for an SA process

An interpretation of the SA-RT dialect proposed by Hatley and Pirbhai [19] was used for modeling and analyzing the hard real-time component of a radar control system by Alenia¹.

4.2 Function Block Diagrams

The second standard VD notation considered in the experiments was IEC Function Block Diagram (FBD), one of the graphical languages proposed by the IEC standard 1131-3 [21] for designing programmable controllers. FBD was chosen because it presents different challenges. In particular FBD is used at a much lower abstraction level than Structured Analysis, and the IEC standard is mostly limited to the syntax, while the semantics of the components is highly programmable to adapt the notation to different platforms and applications. Another interesting option of FBD is the possibility of adding new syntactic elements (blocks) provided with proper semantics according to user needs. We formalized FBD by providing rules for defining different libraries of components and a library manager that allows new libraries to be added and existing libraries to be modified by producing suitable rules. The possibility of adding rules to

¹ The experiment was conducted within the ESPRIT IDERS Project (EP8593).

MetaEnv to modify or create a formalization greatly facilitated the construction of libraries and the library manager. The formalization supported by *MetaEnv* requires the description of a block usually given as Structured Text (ST)² to be formalized by means of a HLTPN. Figure 5 shows two alternative semantics (HLTPNs) for a simple block. In the first case, the textual description is mapped in the action of a single transition. In the second case, the action is semantically described with two alternative transitions.

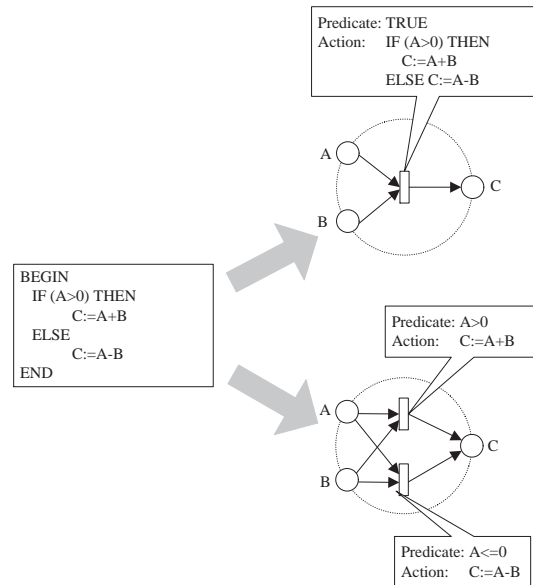


Figure 5. Alternative ways for expressing the semantics of an FBD block

MetaEnv was interfaced with an ad-hoc editor, *PLCTools*, developed with MATLAB/SIMULINK. *PLCTools* and *MetaEnv* are integrated through CORBA in a way that allows only generative construction of HLTPNs. Formal FBD was used to model and analyze controllers of electrical motors developed by Ansaldo³.

4.3 UML

We are currently applying the approach to the Unified Modeling Language (UML) [9]. UML has been chosen because the semantics derived from the object-oriented nature of the notation includes aspects that radically differ from the hierarchical approach of both SA and FBD. Moreover, the different diagrammatic notations provided within UML allow alternative descriptions of the same elements, thus raising consistency and

² ST is another standard language of IEC1131-3.

³ The experiment was conducted within the ESPRIT INFORMA project (EP23163).

completeness issues. This led us to consider the UML meta-model as integration means, choice that significantly impacted the definition of the abstract representation.

The work aims at analyzing mainly the dynamic behavior of UML models. HLTPNs are used to animate and validate the dynamics of object interactions (mainly class diagrams and collaboration diagrams). Static aspects (e.g., the consistency among classes) are not covered by these experiments.

In this case, *MetaEnv* has been plugged in Rational Rose by interfacing with the data repository, as done for StP and Object-Maker. The rules defined so far (only a subset of all rules needed to formalize UML) formalize class and Statecharts diagrams partially. We are currently extending the set of rules to cope with other UML diagrams and we are applying the formalization to significant examples. Details can be found in [7] where we used the "formalized" UML to model the well-known problem of dining philosophers. Figure 6 sketches the *Forks* that philosophers should use and the corresponding HLTPN, where methods are rendered using pairs of places (one for the "request of execution" and one for the results) and the internal semantics mimics a state diagram.

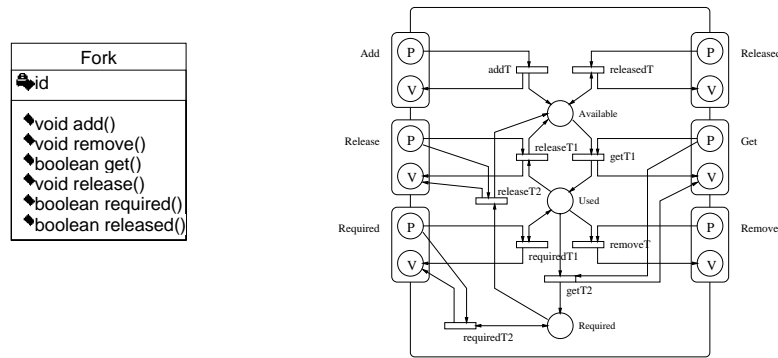


Figure 6. Class *Fork* and its HLTPN semantics

4.4 Control Nets

Control Nets have been defined for designing embedded control systems. Control nets enrich Petri nets with a higher-level VD notation to identify subnets to be reused in further developments. The notation is open, i.e., new elements can be added to the set of reusable components by simply defining their syntax, their external ports and the corresponding HLTPN. In this case, *MetaEnv* has been integrated with a special-purpose interface implemented in tcl-tk. The obtained CASE tool has been successfully used to model and analyze the controller of a robot arm developed by Comau; details can be found in [11].

4.5 LEMMA

The last experiment reported in this paper is the use of *MetaEnv* for giving operational semantics to LEMMA (a Language for Easy Medical Model Analyses) [5]. LEMMA is a notation for specifying, executing, and analyzing clinical diagnostic and therapeutic processes, which has been developed jointly with the 4th Institute of General Surgery in Rome (Italy). Doctors usually describe diagnostic and therapeutic processes informally. Formal notations represent a barrier for doctors who are not able to take advantage from formal analysis. LEMMA conjugates the high expressiveness of VD notations with the rigor of formal methods necessary to simulate and analyze defined models. The lack of experience of doctors with formal methods made it very hard to freeze the specification notation. *MetaEnv* made it possible to incrementally adapt the toolset to the evolving requirements of doctors, who modified or added new elements to the notation while modeling new processes and learning the expressiveness and usefulness of LEMMA. Figure 7 shows the evolution of the semantic model of medical laboratories involved in the analysis required during diagnostic and therapeutic processes. Model (a) simply describes the two possible outcomes of a medical analysis. Model (b) describes the limited resources and capacity of the laboratory and thus allows more accurate evaluation of the timing aspects of the processes.

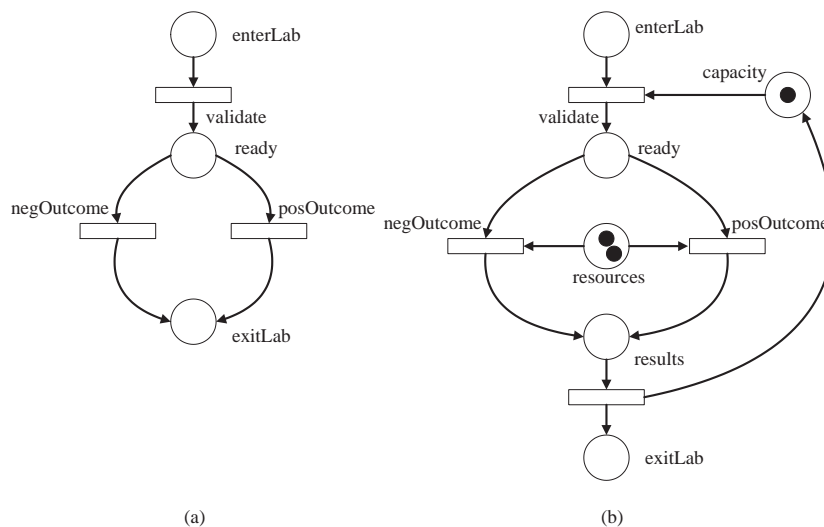


Figure 7. Two different models of medical laboratories

The toolbox has been implemented by plugging *MetaEnv* in graphical interfaces generated with tcl-tk and Java. The tool-box has been used at the 4th Institute of General Surgery in Rome to model and analyze the diagnostic process of colon-rectal cancer.

5 Related Work

Meta-CASE technology, according to the taxonomy presented in [22], introduces customizable environments, that is, tools that offer a set of core functionalities to define, tailor and extend VD notations. To the best of authors' knowledge all these tools are repository-based. All information is stored in a repository in predefined and fixed format; customization deals only with the way this information is presented to users. For example, *MetaEdit+* [23] is based on an object repository that contains all the information about available - both built-in and user defined - notations. Users can exploit all supported notations at the same time; it is the repository that enforces and ensures consistency among them. The repository supplies also data for model reporting and code generation. *ObjectMaker* [28] supplies two different tools for customizing the information stored in the repository. The tool developer kit customizes the notation syntax, the syntactic and static semantic checks, and the way models are stored in the repository. The method maker guides users while defining new notations at a higher abstraction level: Notations are specified by filling built-in forms, customizing general-purpose diagrams, and answering specific questions.

ToolBuilder [25] is organized around the object repository LINCOLN. It supplies both a general-purpose CASE tool, which allows users to manipulate the data stored in the object repository, and a tool for designing new notations and generating the rules required by the CASE tool. A new notation consists of a data model and a frame model. The data model describes how data are organized; the frame model defines the view hierarchy.

Software through Pictures [2] offers a family of products: StP/SE for structured analysis, StP/UML for object modeling, and StP/IM for information modeling). They all are built around a centralized repository and share the same engine called StPCore. StPCore supplies the access to the centralized repository, an API to the graphical interfaces, a predefined framework to associate notations with static semantics, and the control of client-server interaction through messages and notifications. All information is stored according to a simple meta-model called Persistent Data Model (PDM). PDM cannot be extended, but users can define new "views" on the data stored according to this format. The graphical interfaces can be customized by means of textual rules that specify notation elements, syntactic checks, and system answers to user actions.

Metaview [15] is organized around a layered architecture. The meta-model is defined using an extension to entity-relationship data model. At this level users specify notation elements and the constraints among them. Graphical symbols are defined at the environment level. Conceptual tables define the notation symbols and conceptual constraints ensure the consistency among them. The user level defines the concrete interfaces.

All tools described so far privilege the syntax and static semantics of VD notations. They supply appealing features to define or modify the syntax of VD notations, but analysis capabilities are limited to the syntactic correctness and static constraints among data. In contrast, we do not concentrate on these aspects, but we aim at supplying customizable formal semantics to existing notations and tools and thus we consider that all these tools could easily be complemented by *MetaEnv*, but they do not provide the same functionality.

Examples of research prototypes are *GENGED* [3] and *DIAGEN* [3] that allow users to define specialized graphical editors. *GENGED* is based on algebraic specifications and algebraic graph grammars; *DIAGEN* is based on hyper-edge graph grammars. The first approach "formalizes" the definition of symbols and links used within the language; the second approach addresses also animation of designed models, but again they both do not address dynamic semantics.

A slightly different approach is taken by *Dome* [20], which does not limit itself to syntactical aspects, but let users define the semantics of their blocks using Scheme, a Lisp-like language. In this case, the main difference with respect to our approach is the difficulty associated with the customization of semantic aspect and the need of using a predefined interface to interact with the tool. Again, even if *Dome* moves in the same direction as we did, the use of Petri nets for designing the semantics and the freedom of using existing CASE tools, instead of just one tool, are the key differences between our approach and *Dome*.

Other "semantic" customizable tools (for example, [26]) offer large-grained components to program computer aided generation, analysis, verification, and testing of complex systems. They do not aim at letting users customize graphical notations, but they offer large-grained components to program complex systems in a graphical way. In this context, meta-modeling capabilities allow users to define new building blocks by combining existing ones. The user notation is always the same; no customization of the graphical interfaces is permitted.

Finally, we can mention [1,34]: two theoretical frameworks for defining graphical notations from concrete aspects to semantics. They supply a hierarchy of graph grammars to concentrate on concrete and abstract syntax aspects, but the semantics is relegated to textual annotations in natural language.

5.1 Conclusions and Future Work

This paper presented *MetaEnv* and its supporting methodology to add formal semantics to VD notations. *MetaEnv* does not significantly reduce the flexibility and adaptability of VD notations and thus it can be seen as a means to smoothly introduce formal methods in industrial practice. The experiments briefly outlined in the paper demonstrate the suitability of the approach in different application domains. Differently from other approaches, *MetaEnv* neither significantly alters the formalized VD notation nor limits its flexibility. Moreover, *MetaEnv* can be adapted to different environments with a small effort and thus it can be used in situations that could not be efficiently addressed with expensive solutions due to the limited scope of the notation or of the chosen interpretation in the specific situation.

References

1. M. Andries, G. Engels, and J. Rekers. *How to Represent a Visual Program?* In Proceedings of International Workshop on Theory of Visual Languages (1996).
2. Aonix. Structure Environment: Using the StP/SE Editors (1998).

3. R. Bardohl, M. Minas, A. Schuerr, and G. Taentzer. *Application of graph transformation to visual languages*. Handbook of Graph Grammars and Computing by Graph Transformation, volume II: Applications, Languages and Tools, pp. 105-180 (1999).
4. L. Baresi. *Formal Customization of Graphical Notations*. PhD thesis, Dipartimento di Elettronica e Informazione – Politecnico di Milano, 1997. in Italian.
5. L. Baresi, F. Consorti, M. Di Paola, A. Gargiulo, and M. Pezzè. *LEMMA: A Language for an Easy Medical Models Analysis*. Journal of Medical Systems - Plenum Publishing Co., 21(6):369-388 (1997).
6. L. Baresi, A. Orso, and M. Pezzè. *Introducing Formal Methods in Industrial Practice*. In Proceedings of the 20th International Conference on Software Engineering, pp. 56-66. ACM Press (1997).
7. L. Baresi and M. Pezzè. *On Formalizing UML with High-Level Petri Nets*. In G. Agha and F. De Cindio (eds.) *Concurrent Object-Oriented Programming and Petri Nets* (a special volume in the Advances in Petri Nets series); 2001, pages 271-300. Volume 2001 of Lecture Notes in Computer Science.
8. L. Baresi and M. Pezzè. *A formal Definition of Structured Analysis with Programmable Graph Grammars*. In *Proceedings of AGTIVE99*; volume 1779 of *Lecture notes in Computer Science*, pages 193–208. Springer/Verlag, 1999.
9. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. The Addison-Wesley Object Technology Series (1998).
10. M. Burnett and Marla J. Baker. *A Classification System for Visual Programming Languages*. Journal of Visual Languages and Computing, pp. 287-300 (1994).
11. A. Caloini, G. Magnani, and M. Pezzè. *A Technique for Designing Robotic Control Software Based on Petri Nets*. IEEE Transactions on Control Systems (1997).
12. CJ International. *IsAGRAPH 3.3 Documentation* (1999).
13. T. De Marco. *Structured Analysis and System Specification*. Prentice-Hall (1978).
14. O. Fargemand and A. Olsen. *Introduction to SDL-92*. Computer Networks and ISDN Systems, 26:1143-1167 (1994).
15. P. Findeisen. *The Metaview System*. Technical report, University of Alberta (Canada) (1994).
16. R. B. France and M. M. Larrondo-Petrie. *From Structured Analysis to Formal Specifications: State of the Theory*. In Proceedings of Computer Science Conference, pp. 249-256. ACM Press (1994).
17. J.A. Goguen and Luqi. *Formal Methods and Social Context in Software Development*. In 6th International Conference on Theory and Practice of Software Development (TAPSOFT'95), number 915 in Lecture Notes in Computer Science, pp. 62-81. Springer-Verlag, (invited talk), Aarhus (Denmark) (1995).
18. D. Harel and B. Rumpe. *Modeling Languages: Syntax, Semantics and All That Stuff - Part I: The Basic Stuff*. Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, Israel, MCS00-16, September 2000.
19. D. J. Hatley and I. A. Pirbhai. *Strategies for Real-Time System Specification*. Dorset House, New York (1987).
20. Honeywell. *Dome Extensions Manual (ver. 5.2.2)* (1999).
21. IEC. *Part 3: Programming Languages, IEC 1131-3*. Technical report, International Electrotechnical Commission - Geneva (1993).
22. A. S. Karrer and W. Scacchi. *Meta-Environments for Software Production*. Technical report, University of Southern California, Atrium Laboratory (1994).
23. S. Kelly, K. Lyytinen, and M. Rossi. *MetaEdit+ A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment*. In Proceedings of CAiSE'96, volume 1080, pp. 1-21. Springer-Verlag, Lecture Notes in Computer Science (1996).

24. C. Kronlof, editor. *Method Integration - Concepts and Case Studies*. John Wiley & Sons (1993).
25. Lincoln Software Limited. *What is a Meta-Tool?* (white paper). see <http://www.ipsys.com/mc-wp.htm>.
26. Luqi. Real-time constraints in a rapid prototyping language. *Computer Languages*, 18(2):77-103 (1993).
27. T.S.E. Maibaum and B. Rumpe. *Automated Software Engineering: Special Issue on Precise Semantics for Software Modeling Techniques* Volume 7, Issue 1, Kluwer Academic Publishers, March 2000.
28. Mark V Systems. *ObjectMaker User's Guide* (1997).
29. METACase Consulting. *ABC To Metacase Technology*. Technical report (1999).
30. M. Nagl. *Set theoretic approaches to graph grammars*. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Graph Grammars and Their Application to Computer Science*, volume 291 of *Lecture Notes in Computer Science*, pp. 41-54. Springer-Verlag (1987).
31. M. Pezzè. *Cabernet: A Customizable Environment for the Specification and Analysis of Real-Time Systems*. Technical report, Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy (1994).
32. Rational Software Corporation. *Rational Rose 2001: User's Manuals* (2001).
33. S. Reiss. *Connecting Tools using Message Passing in the FIELD Program Development Environment*. *IEEE Software*, pp. 57-67 (1990).
34. J. Rekers and A. Schuerr. *A Graph Based Framework for the Implementation of Visual Environments*. In *Proceedings of VL'96 12th International IEEE Symposium on Visual Languages*. IEEE-CS Press (1996).
35. H. Saiedian. *An Invitation to Formal Methods*. *IEEE Computer*, pp. 16-30 (1996).