



ELSEVIER

Available online at www.sciencedirect.com

SCIENCE @ DIRECT®

**Electronic Notes in
Theoretical Computer
Science**

Electronic Notes in Theoretical Computer Science 127 (2005) 29–44

www.elsevier.com/locate/entcs

Petri Nets as Semantic Domain for Diagram Notations

Luciano Baresi¹*Dipartimento di Elettronica e Informazione
Politecnico di Milano
Milano, Italy*Mauro Pezzè²*Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano Bicocca
Milano, Italy*

Abstract

This paper summarizes the work carried out by the authors during the last years. It proposes an approach for defining extensible and flexible formal interpreters for diagram notations based on high-level timed Petri nets.

The approach defines interpreters by means of two sets of rules. The first set specifies the correspondences between the elements of the diagram notation and those of the semantic domain (Petri nets); the second set transforms events and states of the semantic domain into visual annotations on the elements of the diagram notation. The feasibility of the approach is demonstrated through *MetaEnv*, a prototype tool that allows users to implement special-purpose interpreters.

Keywords: Graph transformation systems, Petri nets, Formal denotational semantics, MetaEnv

1 Introduction

This paper summarizes the work carried out by the authors during the last years. The work started with the goal of finding a suitable solution to the problem of adding precision and rigor to the many informal diagram notations

¹ Email: baresil@elet.polimi.it

² Email: pezzem@disco.unimib.it

used by software engineers. We wanted to improve current practice; we did not want to change it and force designers adopt new formal methodologies. In fact, many attempts already proved that formal methodologies per se only pay off in very specific domains where special purpose requirements and needs justify their use.

Our goal was the definition of suitable dynamic semantics for common notations without forcing the designer to learn new notational elements. The flexibility of used notations, like Structured Analysis —when we started the work— and UML nowadays, does not allow the definition of a single formal semantics. This is why we opted for a rule-based approach that allows experts to associate the chosen notation with as many dynamic semantics as needed. We adopted a denotational style to specify the semantics as a suitable translation into high-level Petri nets. The use of Petri nets came from the background of the authors and the availability of a suitable validation framework.

This approach posed two problems: (1) the definition of a means to specify the translations of designed models into functionally equivalent Petri nets and (2) the translation of execution and analysis results of obtained nets into representations suitable for the user. Petri nets remain hidden, the user does not “touch” them, but they supply the execution/validation engine and their outcome (e.g., the token game or deadlock situations) is rendered in such a way that the user understands them.

We addressed the first problem by means of a pair of graph transformation systems. The first graph transformation system specifies all feasible steps for constructing user models. These transformation rules predicate on the meta-model of such models, that is, they consider their abstract syntax representations. The second transformation system, whose rules are paired with those of the first system, specifies equivalent transformations on the corresponding Petri net. In other words, while generating the abstract representation (meta model) of a user model, we also create the corresponding Petri net.

We also defined a consistency framework to allow experts define a consistent set of semantics for the same notation by framing rules and fostering their reuse. The consistency framework helps the user assess the completeness and consistency of the proposed semantics.

We addressed the second problem (i.e., the backward translation of execution and analysis results) by defining textual grammars that transform transition firings and place mappings into suitable annotations of the elements of the meta model. These annotations are then rendered into actual animations, graphical effects, and textual annotations according to the used tool.

The integration with standard CASE technology and tools introduces the last aspect touched by this paper. The two-way translation approach is im-

plemented in a tool called *MetaEnv* [4]. It supplies the core support to graph transformation systems and was conceived to be easily integrated with well-known CASE tools. We did experiments with StP (Software through Pictures), but also with IBM Rational Rose and with special-purpose prototypes developed for this purpose. *MetaEnv* and all studied CASE tools allowed us to do experiments with very different notations: for example, Structured Analysis [1], our first case study, LEMMA [2], a special-purpose notation for medical diagnostic processes, and a subset of UML [3]. All these experiments gave encouraging results and demonstrated the feasibility of the approach.

The rest of the paper is organized as follows. Section 2 surveys related approaches. Section 3 presents the approach and its characterizing features. Section 4 describes *MetaEnv*, the prototype toolset developed to support the approach. Section 5 summarizes the main experiments conducted with the approach and Section 6 concludes the paper.

2 Related work

The blending of diagram notations and formal methods has been addressed by several proposals in the last decades. Initially, the approaches concentrated on Structured Analysis (SA). For example, Semmens and Allen [19] complement De Marco-like SA with Z, while Wing and Zaremski [20] use Larch. De Marco-like SA is supplemented also with object-oriented methodologies and VDM by Liu et al. in [15].

A wider approach is taken by Paige [16,17]. His approach, called *meta-method*, integrates specification notations by using a *heterogeneous basis*, which contains a set of formal and informal notations along with all relationships among them. The meta-model is given by defining fixed correspondences among formal models and by providing particular interpretations of informal notations.

More recently, many proposals concentrated on UML. In some cases, like the pUML approach ([8]) for example, the goal is the static semantics of UML and dynamic aspects are often neglected. Other approaches, in contrast, concentrate on the dynamic semantics, but they all address only state diagrams and not the whole language. For example, Engels et al. [6] use CSP (Communicating Sequential Processes [13]) to formally define the dynamic semantics. Engels et al. [7] also propose a completely different approach where UML interaction diagrams are transformed directly into Java code and the formalization is given implicitly.

3 The MetaEnv approach

The approach presented in this paper defines the dynamic semantics of a notation by means of a *mapping* from the abstract syntax to a *semantic domain*. The mapping is based on two sets of rules:

- *Building rules* map diagram models onto high-level timed Petri nets (HLTPN, [9]). The semantics can be adapted, modified, and extended by modifying, deleting or adding these rules. The rules are given as pairs of graph transformation productions.
- *Visualization rules* map semantic actions, i.e., HLTPN firings and markings, back to the diagram model, thus allowing animation and visualization of analysis results.

3.1 Building rules

Building rules are pairs of attributed programmable graph grammar productions [10]. The *Abstract Syntax Graph Grammar (ASGG)* productions identify the transformations on abstract syntax models. The corresponding *Semantic Graph Grammar (SGG)* productions define the semantics by suitably transforming the corresponding Petri nets. Special purpose attributes, associated with semantic elements, specify the correspondences between semantic and syntactic elements.

In this paper, we use the term *meta-model* to indicate the abstract syntax of a notation, and the term *abstract syntax model* to refer to an instance of the meta-model. We also use UML object diagrams to render abstract syntax models, and standard Petri net symbols, augmented with diamonds for markers, to draw HLTPNs.

Each element of the meta-model is paired with an equivalent HLTPN that defines the associated semantics. Their transformations are stated by means of suitable graph grammar productions. In this context, nodes of ASGG productions correspond to the elements that define abstract syntax models, while nodes of SGG productions correspond to HLTPN places, transitions, arcs, and markers (i.e., placeholders that relate elements). Edges represent relationships between elements and instantiate the associations in the meta-model. Edges in ASGG productions correspond to links between notation elements, while edges of SGG productions link arcs with places and transitions. They also connect HLTPN elements with the markers to which they are related.

In this paper, we use LEMMA as example notation [2]. LEMMA is a simple language to allow doctors to precisely specify their diagnostic processes. Figure 1 shows its meta-model. Nodes correspond to actions done on pa-

tients and edges define the precedences among them. LEMMA models are designed with the following elements: *entry points* are starting points, *clinical*

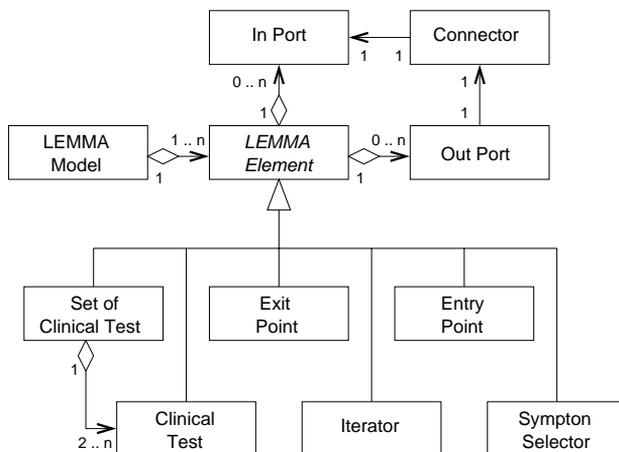


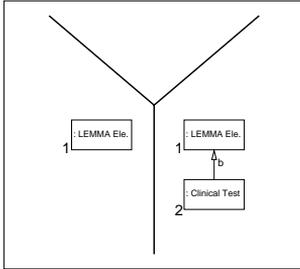
Fig. 1. LEMMA meta-model

tests model medical investigations, *sets of clinical tests* model the execution of sets of investigations where the actual order among the tests is not important, *symptoms selectors* are split points to help decide the actual path, *iterators* make patients repeat enclosed actions n times, and *exit points* define the conditions for exiting the process.

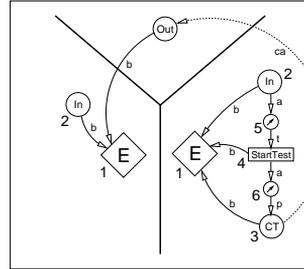
The meta-model acts as *type graph* for the ASGG productions of the building rules. The meta-model and building rules only define necessary constraints on LEMMA models, but they are not sufficient to identify the class of correct LEMMA models. It is the user interface that imposes further constraints and thus allow the designer to specify only correct models.

Figure 2 shows an example building rule. The two productions are represented as graphs. Each production is composed of three parts that correspond to the three graphical regions identified by a Υ , as proposed in [11]. The left-hand side graph indicates the sub-graph to be substituted by applying the production. The right hand-side graph indicates the graph to be added. The edges between left- and right-hand side graphs, through the top graph, indicate the connectivity of the added sub-graph with respect to the host graph (i.e., the graph on which the production is applied). Each node is associated with a unique identifier; nodes with the same identifier in both the left- and right-hand side of the production are preserved while applying the production.

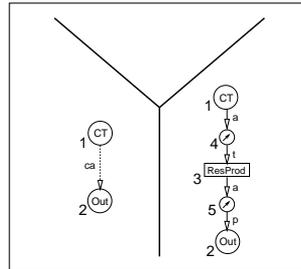
Figure 2 presents the building rule that specializes a *LEMMA Element* in a *Clinical Test*. The ASGG production (Figure 2(a)) applies to a node of type *LEMMA Element*, which belongs to the left-hand side graph, and preserves



```
2.name = 1.name + "CT";
2.type = "Clinical Test";
```



```
3.name = @1.name@ + "P";
3.type = "CT";
3.absNode = @2.name@;
4.name = @1.name@ + "T";
4.type = "StartTest";
4.predicate = "TRUE";
4.action = ;
4.tMin = "enab";
4.tMax = "enab";
4.absNode = @2.name@;
```



```
3.name = 2.name + "T";
3.type = "ResProd";
3.predicate = "compare(1.value,
                    2.absNode.value)";
3.action = ;
3.tMin = "enab";
3.tMax = "enab";
3.absNode = @2.name@;
```

(a) ASGG production

(b) SGG production

Fig. 2. Building rule *AddClinicalTest*

it, since the node with the same identifier (1) belongs to both the left- and right-hand side graphs. The production adds a node of type *Clinical Test* and an edge of type b to connect the newly added node to the *LEMMA Element*. The textual annotations of the rule indicate that the **name** of the new element (node 2) is the same as the name of the *LEMMA Element* 1 plus suffix **CT** and its **type** is **Clinical Test**.

The pair of productions of Figure 2(b) identifies the changes on the HLTPN that correspond to the modifications on the abstract syntax model. The actual correspondences between ASGG and SGG elements are established by means of the textual attribute **absNode**. The programmed production comprises a main production (the top of Figure 2(b)) and one sub-production. The production adds a *CT* place to model the status of the clinical test. It also connects the *In* place to this place by means of a *Start Test* transition and two arcs, depicted using an arrow in a circle. Notice that HLTPN arcs are modeled as nodes; edges model the usual input/output relationships between arcs and places/transitions. The top of the production indicates the embedding, i.e., the context that must be considered when applying the rule. The b edge from the top *Out* place to element 1 selects all the *Out* places that belong to the E marker, that is, the *LEMMA Element*. Notice that the number of nodes identified by the embedding can vary, while the number of nodes selected by the left-hand side of a production is fixed³. The dotted edge that connects the *Out* place with the *CT* place (node 3) indicates a new special-purpose edge of type ca (*connect arc*), which is added between each place of type *Out* selected by the embedding and the new *CT* place.

Dotted edges indicate *sp-edges* that trigger sub-productions. A production with sp-edges is a programmed production, whose application requires that the main production and all instantiations of the sub-productions be applied. Since sp-edges connect nodes in the embedding, whose instantiation happens only dynamically, the number of times the sub-productions must be applied varies consequently.

The sub-production shown on the bottom of Figure 2(b) indicates the substitution of a ca sp-edge, which connects the *CT* place to an *Out* place, with a transition of type *ResProd* (node 3)⁴ and two HLTPN arcs (nodes 4 and 5). The resolution of the instances of the sp-edges of Figure 2(b) adds as many transitions and pairs of arcs as the number of *Out* places that belong to the E marker.

³ Since a clinical test has exactly one input, but two or three outputs, we can use the left-hand side graph to reason about the *In* place, but we need the embedding to deal with *Out* places.

⁴ Transitions are typed to indicate the class of modeled events. In this case *ResProd* means *Result Production*.

The textual annotations of the main SGG production set the properties of the newly created *CT* place and *Start Test* transition. We use pairs of @ to make the textual annotations of SGG productions refer to the values of the attributes of ASGG elements. The name of place 3 is the same as the name of the ASGG node 1 plus P and its type is CT. The name of transition 4 is the same as the name of the ASGG element augmented with suffix T. Its predicate is `true`, that is, the transition is enabled as soon as there is at least a token in each place of its pre-set. The empty action simply moves the token from the place of the pre-set to that of the post-set. The enabling interval is `tMin = enab` and `tMax = enab` to indicate that the transition must fire as soon as enabled. The textual annotations of the sub-production define the attributes of the added transition(s): The `name` is the name of the output port (i.e., the *Out* place) augmented with suffix T. The `type` is `ResProd`, the `predicate` calls the external function `compare` that enables/disables the transition by comparing the token in place 1 (`1.value`) and the value associated with the output port, that is, the `value` associated with the `absNode` of node 2. The enabling interval indicates that the transition must fire as soon as enabled.

For all these SGG elements, attribute `absnode` indicates the abstract syntax element that corresponds to the semantic node.

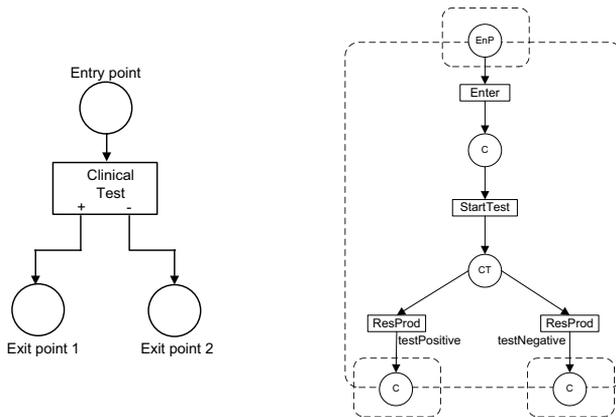


Fig. 3. Example LEMMA model and its corresponding Petri net

Figure 3 shows a simple LEMMA model and its corresponding Petri net. Space limitations do not allow us to describe the step by step construction of the model. Given the LEMMA representation, we start applying the *Axiom* to create a new *LEMMA Model*, then we need rule *AddLEMMAElement* four times to create the four elements that belong to the model. These elements are then converted into an entry point, a clinical test and two exit points by means of dedicated rules: the rule of Figure 2 is the one that transforms the

LEMMA Element into the *Clinical Test*.

3.2 Visualization rules

Visualization rules translate obtained results in terms of suitable visualizations on the abstract syntax elements. The rules define the policy to propagate the events generated by analysis and simulation back to the abstract level. HLTPN events are firings of transitions and markings of places; once translated, they become *abstract animations*. The mapping from abstract to concrete is straightforward and deals with substituting an abstract animation with a concrete action, which depends on the used CASE tool.

Since propagation of events depends on both the diagram notation and user needs, visualization rules are externally provided in the form of C-like code and are interpreted to produce abstract animations. Each rule produces a `visualization` that usually comprises some `animations`, that is, the visualization actions associated with the elements of the abstract syntax model. The triggering event is the firing of a transition; the `visualization` defines how to animate the element associated with the transition itself and those related to the places of its pre- and post-sets. The predefined function `getAbsId()` returns the abstract component associated with the Petri net element. Figure 4

```
Visualization v = new Visualization();

if (tr.type() == "ResProd") {
    foreach pl in tr.postSet() {
        Animation an = new Animation();
        an.setEntityId(pl.getAbsId());
        an.setAnimType("prodOutput");
        v.addAnimation(an);
    }

    Animation an = new Animation();
    an.setEntityId(tr.getAbsId());
    an.setAnimType("completeTest");
    v.addAnimation(an);
}
```

Fig. 4. The visualization rule *complete clinical test*

presents rule *complete clinical test*. It describes how the firing of transitions of type `ResProd` is visualized in terms of LEMMA elements. Since these transitions identify the completion of clinical tests, the places in their post-sets correspond to the flows that leave the *Clinical Test*. The rule associates animation `prodOutput` to all selected *Connector* (in this case, we always select one place and thus one connector) and animation `completeTest` to the *Clinical test*.

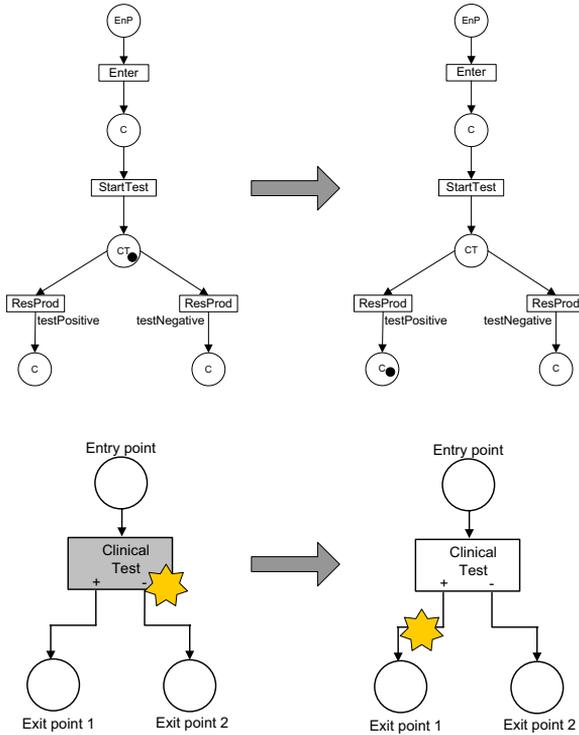


Fig. 5. How visualization rules work

The application of the visualization rule of Figure 4 to the example model of Figure 3 is exemplified in Figure 5. The firing of the *ResProd* transition makes the patient move from the *Clinical Test* to the outgoing flow. This is because the rule states that we must animate both the element associated with the fired transition and its outgoing flows.

3.3 Consistency framework

Given the meta-model of a diagram notation, a formal dynamic semantics (i.e., an interpretation) is defined by specifying a building rule for each element of the meta-model. The definition of a *consistency framework* help define consistent interpretations.

For example, Figure 6 shows a consistency framework for LEMMA, that is, a partial order among the elements of the meta-model. This organization forces all notation elements to use the interfaces supplied by *In* and *Out Port*. This means that all changes in *Symptom Selector*, for example, can only be local, while changes to *LEMMA Element* could impact both *In Port* and *Out*

Port and all other notation elements.

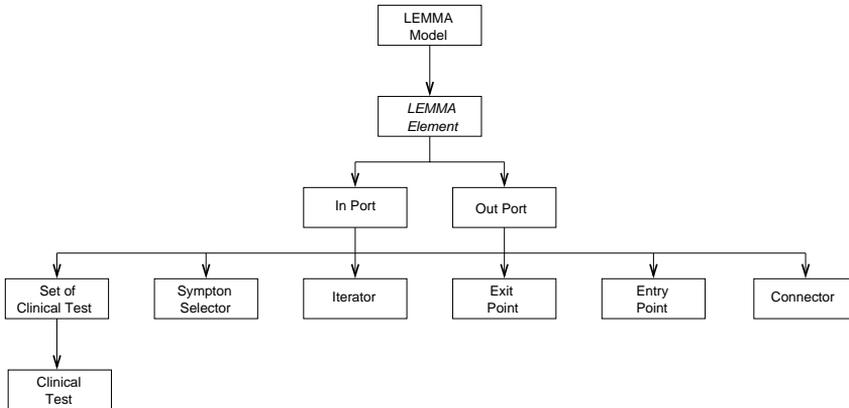


Fig. 6. A consistency framework for LEMMA

The consistency framework imposes a dependency among shared interfaces. The rule that is higher in the hierarchy is in charge of defining (supplying) the interface, while the lower-level rule can only use it. This way, we clearly constrain the changes in building rules and foster consistency of defined interpretations.

The consistency framework allows us to assess the *correctness* of a given set of rules by inspecting informally obtained behaviors on well-known benchmarks. This is because in many cases the starting point is the informal interpretation ascribed with the considered notation. The *completeness* can be verified by proving that the mapping onto the semantic domain covers all the elements of the meta model. Incomplete sets of rules might reflect incomplete informal interpretations. The possibility of proving the completeness of the formalization also allows us to identify lacks in the informal interpretations. Some partiality can be by-passed by providing default transformations to complement the original specification. The *consistency* can be verified by checking the functional behavior (termination and confluence) of the designed graph transformation system ([12]).

4 Tool support

The approach proposed in this paper is supported by *MetaEnv*, our prototype interpreter generator. Building and visualization rules are the basis for tailoring *MetaEnv* for a particular diagram notation and a given interpretation. The *consistency framework* is the basis for efficiently managing complex notation families.

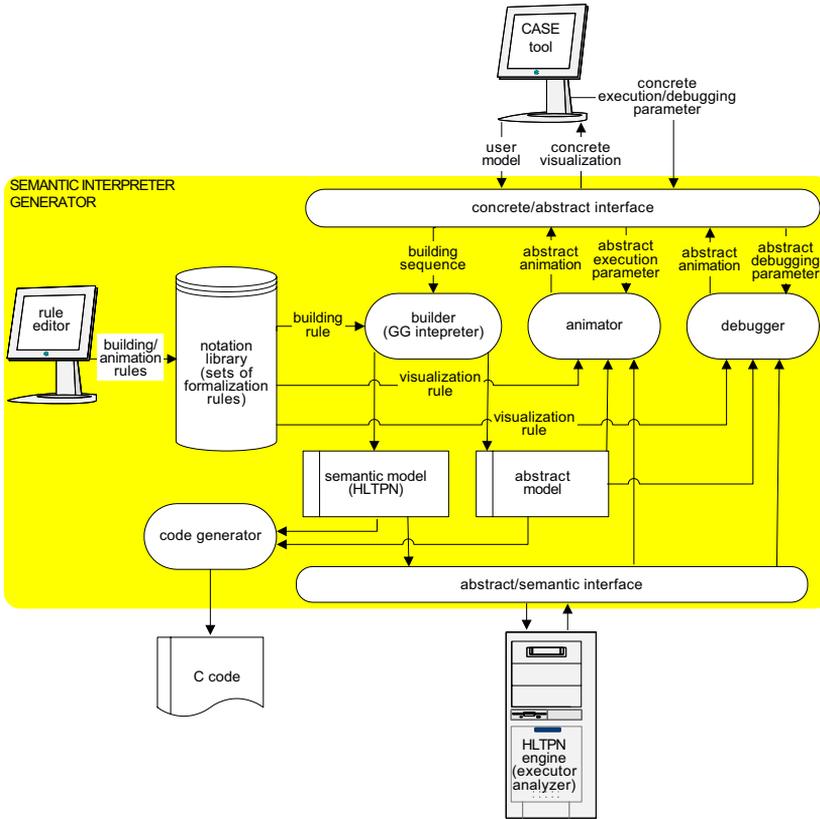


Fig. 7. Architecture of METAENV

Figure 7 shows the architecture of *MetaEnv*. The *concrete/abstract interface* plugs *MetaEnv* in an external *CASE tool*. The interface defines a two-way communication channel to transform user models into suitable sequences of building rules and abstract animations into concrete visualizations for the employed *CASE Tool*. The *CASE Tool* must supply an API that can be used to properly store, retrieve, and animate models.

The *concrete/abstract interface* can use different policies to transform models in sequences of invocations of building rules. The simplest solution is an on-line translation that maps user actions into invocations of rules. Alternatively, the interface can adopt an off-line approach that reads complete models and defines the sequence according to the predefined partial order among building rules identified by the consistency framework. Rules that delete elements are necessary only to allow for incremental transformations, while would be useless if we think that we always scan models from scratch.

The data flow in the opposite direction transforms abstract animations produced by animation rules into concrete visualizations. Abstract animations describe visualizations of notation elements in a tool-independent way. The interface adds all details that depend on the particular tool. Differently from all the other components of *MetaEnv*, the *concrete/abstract interface* varies according to the employed CASE tool.

The *builder* is a graph grammar interpreter that applies building rules, according to the sequence supplied by the concrete interface, and builds both the abstract syntax model and the semantic model, i.e., the HLTPN. The *animator* and *debugger* apply visualization rules to firings and markings produced by the HLTPN engine. For example, the *animator* transforms the execution of the HLTPN, that is, a sequence of firings, into a sequence of abstract animations. The *debugger* allows users to control the execution of their models by setting break-points and watch-points, choosing step-by-step execution, and tracing the simulation. The *debugger* transforms debugging parameters in terms of constraints on the sequence of abstract animations. A step-by-step execution is an execution that stops after each abstract animation; a break point on a particular element of the model suspends the execution at the first abstract animation that involves the selected element.

The *builder*, *animator*, and *debugger* read their rules from the *notation library* that stores all rules. The *rule editor* lets users define new rules through a graphical editor and processes them to move from the graphical representation to the required textual format.

The *code generator* automatically produces ANSI C code from diagram models, using both the semantic and abstract models. The semantic model provides the details to generate the C code; the abstract model provides the structure to split the code in meaningful modules. The automatic derivation is based on special-purpose hard-coded rules that parse the HLTPN to find particular patterns that are associated with the main constructs of the C language.

The *abstract/semantic interface* plugs in a *HLTPN engine* that executes and analyzes the HLTPNs obtained through the builder. The *abstract/semantic interface* adapts the interface of *MetaEnv* to the chosen HLTPN engine. All experiments conducted so far used *Cabernet* [18] as HLTPN engine, but other engines could be plugged as well.

MetaEnv requires two different classes of users. Domain experts are proficient in the diagram notation and interact with the tool-set through the *CASE tool* to design their models. They do not define new rules (interpretations), but do their experiments with existing sets or define the requirements for new ones. *MetaEnv* experts transform these requirements into consistent and

complete sets of rules. These users interact with the tool-set through the *rule editor* and must be proficient in HLTPNs, building rules, and visualization rules to be able to ascribe meaningful semantics.

5 Experiments

The approach has been validated by plugging *MetaEnv* in different commercial and special-purpose CASE tools for experiments with various diagram notations:

- **Structured Analysis** ([5]) has been chosen as one of the richest notation families. The formalization of Structured Analysis comprises about 50 sets of rules. Each set of rules comprises from one to five rules that provide different interpretations of the same construct. A specific interpretation can be obtained by selecting one rule from each of the 50 sets. Some interpretations do not require a rule from each set. For example, about one third of the rules concerns the control model, which belongs to the real-time extensions of Structured Analysis (SA-RT), thus such rules are not used for formalizing “classical” Structured Analysis dialects. The consistency framework indicates coherent subsets of rules. For example, all rules that deal with control aspects are rooted in a single sub-hierarchy and can thus be ignored without affecting the other rules.
- **Control Nets** have been defined for designing embedded control systems. Control Nets enrich Petri nets with graphical elements that 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 defining their syntax, their external ports and the corresponding HLTPNs. The core elements of Control Nets were formalized with 30 rules.
- **IEC Function Block Diagram** (FBD) is one of the graphical languages proposed by the IEC standard 61131-3 [14] for designing programmable controllers. FBD was chosen because it presents new challenges. In particular FBD is used at a lower abstraction level than Structured Analysis, and the IEC standard is mostly limited to the syntax, while the semantics of components is highly programmable to adapt the notation to different platforms and applications. Another interesting option of FBD is the possibility of extending the notation by adding new elements (blocks).

We formalized the core FBD notation and the main libraries with about 40 rules. We used a customized version of the *rule editor* for adding new libraries and modifying existing ones. The customized version of the *rule editor* allows users to define new building rules by simply indicating the interfaces of the new block and giving a HLTPN that models the semantics.

- **LEMMA**, a Language for Easy Medical Model Analyses, was developed jointly with the 4th Institute of General Surgery in Rome (Italy). Diagnosis processes are usually described informally, thus they are often misinterpreted and cannot be fruitfully analyzed. Formal notations represent a barrier for doctors who are not able to take advantage from formal analysis. LEMMA conjugates the high expressiveness of diagram notations with the rigor of formal methods necessary to simulate and analyze defined models.
- **UML** was 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 diagram notations provided within UML allow alternative descriptions of the same elements, thus raising consistency and completeness issues. This led us to consider the UML meta-model as integration means, choice that significantly impacted the representation of abstract syntax models.

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

In this case, METAENV was plugged in Rational Rose. The 20 rules we defined refer to class, state and interaction diagrams only, and represent a subset of all rules needed to formalize UML.

6 Concluding remarks

This paper proposes an approach and a supporting toolset for defining formal interpreters for diagram notations. The paper also demonstrates the suitability of HLTPNs as semantic domain to ascribe diagrammatic notations with formal semantics. The approach is based on *building rules*, to create HLTPNs that are equivalent to user models, and *visualization rules*, to map analysis and execution results of HLPTNs onto user models.

The approach has been validated by means *MetaEnv*, a prototype toolset that supplies a general-purpose interpreter for building and visualization rules. *MetaEnv* was properly customized for special-purpose and well-known notations and produced interpreters were employed to design several example models, from simple exercises to models of real industrial applications.

References

- [1] L. Baresi. *Formal Customization of Graphical Notations*. PhD thesis, Dipartimento di Elettronica e Informazione – Politecnico di Milano, 1997. In Italian.

- [2] 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, December 1997.
- [3] L. Baresi and M. Pezzè. On Formalizing UML with High-Level Petri Nets. In *Concurrent Object-Oriented Programming and Petri Nets*, pages 276–304. Springer-Verlag, 2001.
- [4] L. Baresi and M. Pezzè. A Toolbox for Automating Visual Software Engineering. In *Proceedings of FASE'02*, volume 2306 of *Lecture Notes in Computer Science*, pages 189–198, 2002.
- [5] T. De Marco. *Structured Analysis and System Specification*. Prentice-Hall, 1978.
- [6] G. Engels, R. Heckel, and J.M. Küster. Rule-based Specification of Behavioral Consistency based on the UML Meta Model. In *Proceedings of UML 2001*, volume 2185 of *Lecture Notes in Computer Science*, pages 272–287. Springer-Verlag, 2001.
- [7] G. Engels, R. Hüicking, S. Sauer, and A. Wagner. UML Collaboration Diagrams and their Transformation to Java. In *Proceedings of UML'99*, volume 1723 of *Lecture Notes in Computer Science*, pages 473–488. Springer-Verlag, 1999.
- [8] A. Evans and S. Kent. Core Meta-Modelling Semantics of UML: The pUML Approach. In *Proceedings of UML'99*, volume 1723 of *Lecture Notes in Computer Science*, pages 140–155. Springer-Verlag, 1999.
- [9] 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.
- [10] H. Göttler. Attribute Graph Grammars for Graphics. In *Graph Grammars and Their Application to Computer Science*, volume 153 of *Lecture Notes in Computer Science*, pages 130–142. Springer-Verlag, 1983.
- [11] H. Göttler. Diagram Editors = Graphs + Attributes + Graph Grammars. *International Journal Man-Machine Studies*, 4(37):481–502, 1992.
- [12] R. Heckel, J Küster, and G. Taentzer. Confluence of Typed Attributed Graph Transformation Systems. In *Proceedings of Graph Transformation, 1st Int. Conference, ICGT 2002*, volume 2505 of *Lecture Notes in Computer Science*, pages 161–176. Springer-Verlag, 2002.
- [13] C. Hoare. Communicating Sequential Processes. *Communicat. Associat. Comput. Mach.*, 21(8):666–677, 1978.
- [14] R.W. Lewis. *Programming Industrial Control Systems Using IEC 1131-3*. IEE Publishing, 1998.
- [15] S. Liu, A.J. Offutt, C. Ho-Stuart, Y. Sun, and M. Ohba. SOFL: A Formal Engineering Methodology for Industrial Applications. *IEEE Transactions on Software Engineering*, 24(1):24–45, January 1998.
- [16] R.F. Paige. Case Studies in Using a Meta-Method for Formal Method Integration. In *Proceedings of the International Conference on Algebraic Methodology and Software Technology*, volume 1349 of *Lecture Notes in Computer Science*, pages 395–403. Springer-Verlag, 1997.
- [17] R.F. Paige. A Meta-Method for Formal Method Integration. *Lecture Notes in Computer Science*, 1313:473–485, 1997.
- [18] M. Pezzè and S. Silva. Cabernet User Manual. Technical Report 47-94, Politecnico di Milano, May 1994.
- [19] L. Semmens and P. Allen. Using Yourdon and Z: An Approach to Formal Specification. In J. Nicholls, editor, *Proceedings of the 5th Z User Workshop*. Springer-Verlag, December 1990.
- [20] J. Wing and A. Zaremski. Unintrusive Ways to Integrate Formal Specifications in Practice. In *Proceedings of Formal Software Development Methods (VDM '91)*, volume 551 of *Lecture Notes in Computer Science*, pages 545–570. Springer, 1991.