Formal Interpreters for Diagram Notations

LUCIANO BARESI
Politecnico di Milano
and
MAURO PEZZÈ
Università degli Studi di Milano Bicocca

The paper proposes an approach for defining extensible and flexible formal interpreters for diagram notations with significant dynamic semantics. More precisely, it addresses semi-formal diagram notations that have precisely-defined syntax, but informally-defined (dynamic) semantics. These notations are often flexible to fit the different needs and expectations of users. Flexibility comes from the incompleteness or informality of the original definition and results in different interpretations.

The approach defines interpreters by means of a mapping onto a semantic domain. Two sets of rules define the correspondences between the elements of the diagram notation and those of the semantic domain, and between events and states of the semantic domain and visual annotations on the elements of the diagram notation.

Flexibility also leads to notation families, i.e., sets of notations that share core concepts, but present slightly different interpretations. Existing approaches usually interpret these notations in isolation; the approach presented in this paper allows the interpretation of a family as a whole. The feasibility of the approach is demonstrated through a prototype generator that allows users to implement special-purpose interpreters by defining relatively small sets of rules.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*Methodologies*; D.2.10 [**Software Engineering**]: Design—*Methodologies*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*

General Terms: Design, Theory, Verification

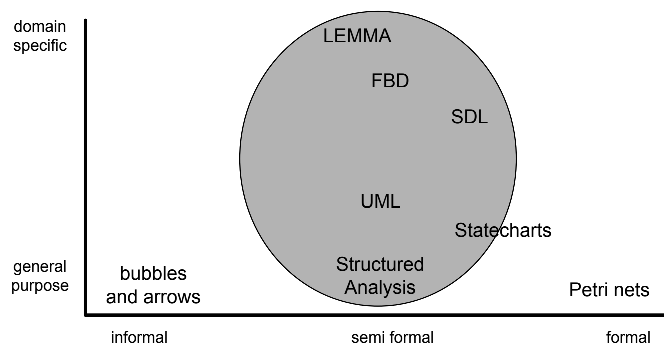
Additional Key Words and Phrases: Semi-formal notations, Semantics, High-level Petri nets, Graph transformation

1. INTRODUCTION

Diagram notations [Burnett and Baker 1993] — like UML, Structured Analysis, Statecharts, Petri nets, and SDL — are often used for specification in several different domains since they allow users to create models as graphs suitably annotated and decorated. These notations are quite different among them, but can be classified in terms of degree of formality and domain specificity (Figure 1). The first dimension identifies: *formal notations*, whose syntax and semantics are both formally defined, *semi-formal notations*, whose syn-

Author's address: L. Baresi, Dipartimento di Elettronica e Informazione, Politecnico di Milano, Piazza L. da Vinci, 32, I20133 Milano (Italy). M. Pezzè, Dipartimento di Informatica, Sistemistica e Comunicazione, Università degli Studi di Milano Bicocca, via Bicocca degli Arcimboldi, 8. I20126 Milano (Italy).
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20XX ACM 1529-3785/20XX/0700-00000 $5.00

tax is formal and semantics leaves room for different interpretations, and *informal notations*, where both syntax and semantics are stated only informally. The second dimension identifies: *general-purpose notations*, which are usually employed to specify different types of problems, and *special-purpose notations*, used only in particular domains. For example, Petri nets [Murata 1989] would fit the class of formal and general-purpose notations, UML [Fowler and Scott 2003] is a semi-formal notation used in several different domains, while IEC FBD (Function Block Diagram) [Lewis 1998] is an example of a domain-specific and semi-formal notation.



UML: OMG Unified Modeling Language, **LEMMA**: Language for Easy Medical Model Analysis,
FBD: IEC 1131-3 Function Block Diagram, **SDL**: Specification and Description Language

Fig. 1. Formality vs. domain specificity

In this paper, we concentrate on *semi-formal* notations with significant dynamic semantics – hereafter *dynamic diagram notations* – that have precisely-defined syntax, but informally-defined dynamic semantics. We do not consider notations that can be animated rather than executed, like UML use cases.

In many cases, these notations are *flexible* to fit the different needs and expectations of users. Flexibility comes from the incompleteness or informality of the original definition and results in different *interpretations*. This is the case, for example, of Structured Analysis [Baresi and Pezzè 1998], Statecharts [von der Beeck 1994], and UML [Evans and Kent 1999] that are available in many different dialects and flavors: None of them comes with formally-defined dynamic semantics. Most of these notations are supported by powerful tools, for example Rational Rose [IBM 2004] for UML or ISaGRAPH [Arcom Control Systems 2002] for FBD, that force users to specify *syntactically correct* models[1], but do not offer analysis capabilities to assess the dynamic semantics. The same applies to automatic generators of graphical editors, for example DIAGEN [Minas and Köth 2000] and GENGED [Bardohl 2000], that only deal with the syntactical correctness of designed models, but do not cover dynamic semantics.

Several researchers have attempted to increase analysis capabilities, as to dynamic semantics, by mapping dynamic diagram notations on formal semantic domains. A lot of

[1] According to the OMG terminology, a *model* is a user specification and a *meta-model* specifies the abstract syntax of a language.

preliminary work focuses on Structured Analysis [France 1992; Liu et al. 1998; Fencott et al. 1994], most recent work concentrates on UML [Evans and Kent 1999; Engels et al. 2001; Evans et al. 1998], less attention has been paid to other domain-specific notations like SDL [Fischer et al. 1995] and IEC 61131-3 [Baresi et al. 2000]. These proposals provide fixed dynamic semantics to the addressed notations, but fail in addressing extensibility and flexibility.

The paper proposes an approach for defining extensible and flexible formal interpreters for diagram notations. The approach adopts a denotational style to define the dynamic semantics by means of a mapping onto a semantic domain. It is based on two sets of rules. *Building rules* define the correspondences between the elements of the diagram notation and those of the semantic domain. In this paper, we use High-Level Timed Petri Nets (HLTPNs [Ghezzi et al. 1991]) as semantic domain, but the approach would also be applicable to other formal methods (e.g., Engels et al. [Engels et al. 2001] propose a similar approach that uses CSP as semantic domain). *Visualization rules* define the correspondences between events and states of the semantic domain and visual annotations on the elements of the diagram notation. The first set is used to create the semantic representation of user models, while the second set renders execution and analysis results on the semantic model in terms of the diagram notation. Users, who are proficient in diagram notations, do not define new rules (interpretations), but use existing rules for their experiments and define the requirements for new ones. Experts transform these requirements into consistent and complete sets of rules. These users must be proficient in HLTPNs, building rules, and visualization rules to be able to ascribe meaningful semantics.

The approach is exemplified by drafting the rules to interpret a simple notation for specifying medical diagnosis processes, called LEMMA (Language for Easy Medical Models Analysis [Baresi et al. 1997]). It is the result of the requirements elicitation process undertaken with a team of doctors. Its definition evolved through different phases that progressively highlighted user needs, evaluated alternatives, and refined the notation. The approach eased the implementation of prototypes to support the elicitation process. LEMMA comes with fixed syntax, but its dynamic semantics depends on the to-be-performed analyses and accuracy of results.

Flexibility also leads to *notation families*, i.e., sets of notations that share core concepts, but present slightly different interpretations. Existing approaches deal with these notations in isolation. In contrast, the approach presented in this paper permits that the interpretation of new family members be obtained by changing some elements of existing ones. It stresses separation of concerns to better scope the domain of each rule, thus facilitating the definition of new interpretations and the adaptation of existing ones. The definition of notation families is exemplified by means of *Structured Analysis*, a well-known representative of notation families in software engineering.

The feasibility of the approach has been demonstrated through a prototype generator that allows users to implement special-purpose interpreters through the definition of relatively small sets of rules. The generator allowed us to apply the approach to both general- and special-purpose notations.

Specifically, the main contributions of this paper are:

—The definition of an approach for ascribing diagram notations with precise dynamic semantics. The approach considers these notations as *families* of related languages instead of tackling them in isolation.

—The proposal of graph grammars as underlying means to specify the chosen behavior. We have identified two grammars that specify the modifications of the abstract syntax representation and the corresponding changes of the high level timed Petri net.

—The definition of a framework to constrain the impact of each graph grammar production and modularize the definition of the dynamic semantics. This helps both reuse elements and highlight similarities among members of the same family.

—The design and implementation of a prototype toolset to support the approach.

The paper is organized as follows. Section 2 describes the core technology, namely building and visualization rules. Section 3 discusses notation families and presents the approach. Section 4 describes the prototype generator used in our experiments, while Section 5 summarizes the main results gained with the experiments. Section 6 surveys relevant related work and Section 7 concludes the paper.

2. FORMAL INTERPRETERS

A *diagram(matic) language* is a language based mainly on graphical elements. The complete definition of a diagram language [Baresi and Heckel 2002], also called *visual formalism*, comprises *syntax* and *semantics*. Syntax specializes in *concrete syntax*, which describes the notational symbols of the language, and *abstract syntax*, which describes the machine's internal representation. Semantics defines the meaning of the notation: *static semantics* deals with the structure of each sentence and *dynamic semantics* defines its dynamic behavior.

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

—*Building rules* map diagram models onto High-Level Timed Petri Net (HLTPN, see Appendix A). The semantics can be adapted, modified, and extended by modifying, deleting or adding these rules. The rules are given as pairs of graph grammar 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.

In this paper, we describe the rules using a simple notation, called LEMMA, developed to help doctors treat their patients.

2.1 LEMMA

LEMMA is a diagram notation for modeling diagnostic processes. Nodes correspond to actions done on patients and edges define the precedences among them. A LEMMA model defines all relevant diagnostic paths. Processes (models) are designed with the elements of Figure 2:

—*Entry points* are the starting points in the diagnostic process. A process can have many entry points. Different patients may enter the process from different entry points according to their symptoms.

—*Clinical tests* model medical investigations. They have either two or three outcomes: *positive* (+), *negative* (-), and if needed *undefined* (?).

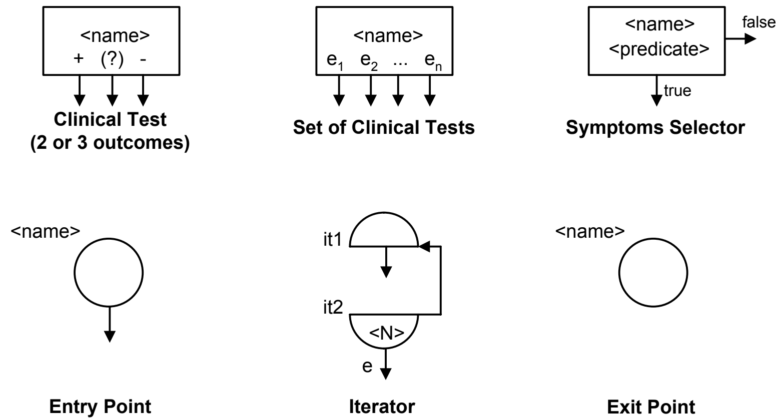


Fig. 2. The lexical elements of LEMMA

—*Sets of clinical tests* model the execution of sets of investigations where the actual order among the tests is not important. The outcomes ($\mathtt{e}_i$) are computed using the results of the single tests.

—*Symptoms selectors* are split points to help decide the actual path. The patient is routed by checking his/her symptoms against the `predicate` of the selector.

—*Iterators* make patients repeat enclosed actions `n` times, where `n` is the integer associated with the iterator.

—*Exit points* define the conditions for exiting the process. This means that either the doctor is able to formulate a diagnosis or the patient needs further investigations.

Figure 3 presents a simple LEMMA model that help explain how the approach works. It detects if a woman is pregnant and assesses the status of the fetus. The patient starts with a `Pregnancy test`. If the outcome is negative, the diagnosis is easy; otherwise the patient is submitted to `Amniocentesis` to probe the fetus.

2.2 Building rules

Building rules are pairs of attributed programmable graph grammar productions [Göttler 1983]. 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 (HLTPNs). Special purpose attributes, associated with semantic elements, specify the correspondences between semantic and syntactic elements.

Figure 4 shows the LEMMA meta-model. 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 standard UML class diagrams to draw meta-models, UML object diagrams to render abstract syntax models, and standard Petri net symbols, augmented with diamonds for markers, to draw HLTPNs.

The approach only applies to special-purpose meta-models designed with the goal of interpreting the notation they describe. For example, the LEMMA meta-model embeds the abstract elements of the notation and their mutual relationships. A *LEMMA Model*

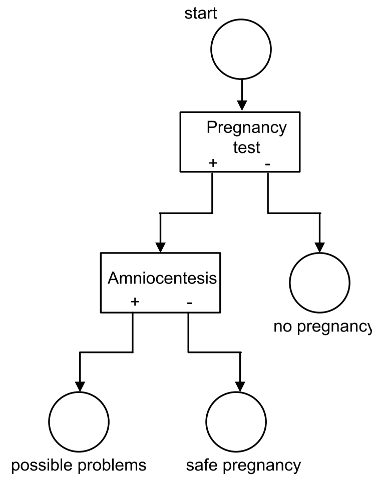


Fig. 3. A simple LEMMA model (User view)

contains one or more *LEMMA Elements*, which in turn can have zero or more *In Ports* and *Out Ports*. *Connectors* join corresponding ports. *LEMMA Element* is an abstract class (denoted by the italics font) and is specialized in *Symptom Selector*, *Clinical Test*, *Entry Point*, *Exit Point*, *Iterator*, and *Set of Clinical Tests*.

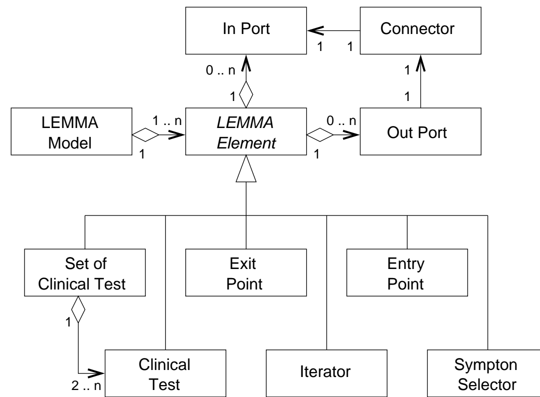


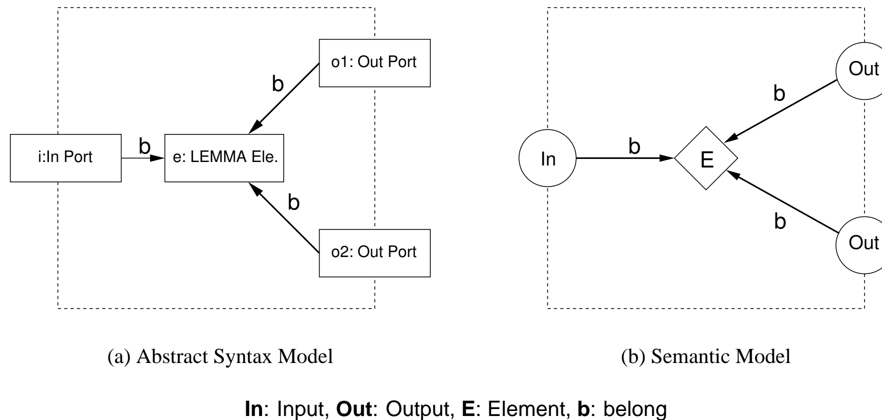
Fig. 4. The LEMMA meta-model

Each element of the meta-model is paired with an equivalent HLTPN that defines the associated semantics. For example, Figure 5 shows the abstract syntax and semantic models of a generic *LEMMA Element* `e` with one *In Port* `i` and two *Out Ports* `o1` and `o2`. *b*-labeled edges indicate a *belong* relationship between the ports (`i`, `o1`, and `o2`) and the element `e`. In this case, the abstract element (i.e., the *Lemma Element*) is not only a means to factorize common properties, but is rendered with a special-purpose marker. This leads to considering the inheritance relationship in Figure 4 as an aggregation between *Lemma Element*

and its sub-classes. This is because we want to transform separately the common part (a generic element with in and out ports) and its specializations. A different approach could only translate the specific elements without ascribing semantics to the abstract component.

The corresponding semantic model comprises a node of type *E*, one node of type *In* and two nodes of type *Out*, connected to the *E* node with *belong* edges. Nodes of types *In* and *Out* are HLTPN places and model *In Ports* and *Out Ports*. The node of type *E* is a marker that identifies the considered element (i.e., the *LEMMA Element*).

After describing models, we need to explain how to create and modify them. A graph grammar production comprises typed nodes and edges. In this context, nodes correspond to objects of the classes of the meta-model. Thus, 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.



(a) Abstract Syntax Model

(b) Semantic Model

In: Input, **Out**: Output, **E**: Element, **b**: belong

Fig. 5. A *LEMMA Element* with one *In Port* and two *Out Ports*

Building rules describe transformations on abstract syntax and semantic models. Figure 6 shows an example. The two productions are represented as graphs. Each production is composed of three parts that correspond to the three graphical regions identified by a Y, as proposed in [Göttler 1992]. 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 itself.

Figure 6 presents the building rule that specializes a *LEMMA Element* in a *Clinical Test*. The ASGG production (Figure 6(a)) applies to a node of type *LEMMA Element*,

which belongs to the left-hand side graph, and preserves 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 application of the ASGG production to the node `e` of Figure 5(a) is shown in Figure 7(a), where grey elements identify what has been added.

The corresponding SGG production is shown in Figure 6(b). The pair of productions 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 attribute `absNode`, which is described later. The production, which is programmed, comprises a main production (the top of Figure 6(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 fixed[2]. 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. Non-programmed productions can add only a fixed number of elements, while the use of sub-productions permits the addition a variable number of elements.

The sub-production shown on the bottom of Figure 6(b) indicates the substitution of a *ca* sp-edge that connects the *CT* place to an *Out* place. The sub-production substitutes the sp-edge with a transition of type *ResProd* (node `3`)[3] and two HLTPN arcs (nodes `4` and `5`). The resolution of the instances of the sp-edges of Figure 6(b) adds as many transitions and pairs of arcs as the number of *Out* places that belong to the *E* marker.

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 =`

[2]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.
[3]Transitions are typed to indicate the class of modeled events. In this case *ResProd* means *Result Production*.

```
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

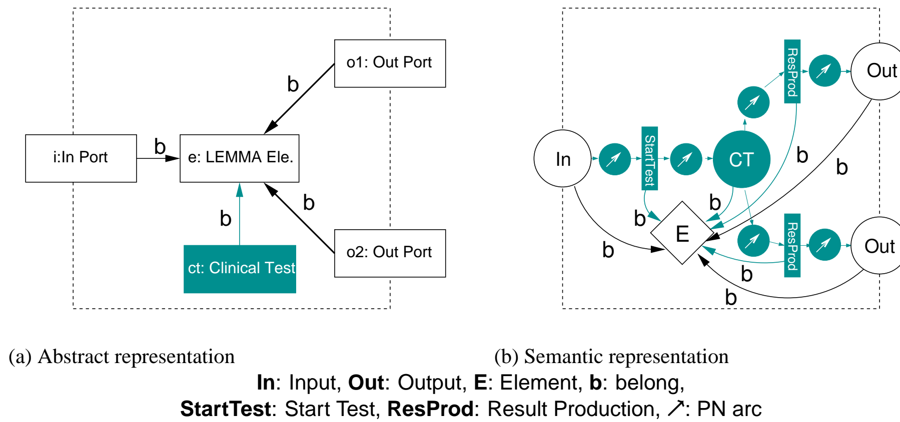
In: Input, **Out**: Output, **E**: Element, **CT**: Clinical Test, **ResProd**: Result Production
↗: PN arc, **b**: belong, **a**: arc, **t**: transition, **p**: place, **ca**: connect arc

Fig. 6. The building rule Add Clinical Test

`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. This information is mainly used by visualization rules – illustrated in the next section – to map back execution and analysis results, but can also be useful to compute the values of other textual attributes.

The application of the SGG production to the *E* node of Figure 5(b) is shown in Figure 7(b). In this case, the production identifies two *Out* places that *belong* to the *E* marker, and thus adds two transitions and the corresponding arcs.



(a) Abstract representation (b) Semantic representation

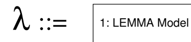
In: Input, **Out**: Output, **E**: Element, **b**: belong,
StartTest: Start Test, **ResProd**: Result Production, ↗: PN arc

Fig. 7. The result of applying production Add Clinical Test to the models of Figure 5

The simple model of Figure 7(b) is just one possible semantics for a clinical test. As soon as the test is enabled, it executes and produces one output. If the user decided to detail the semantics of clinical tests, e.g., by further modeling the start and end of the test and its duration, the model could be easily adapted by modifying the rule of Figure 6 to add a different HLTPN. The ease of changes depends on their scopes. In this example, the change only involves the internals of the HLTPN associated with a *Clinical Test*. Changes that affect the interfaces of the sub-net, e.g., the request for modeling the interaction with an analysis laboratory, may affect other rules. The problem of identifying the scope of changes and thus the number of affected rules is discussed in Section 3. The complete semantics of LEMMA was given with eleven rules, which were changed several times to meet the changing requirements from the medical team. A different meta-model would impact the rules defined so far: The trade-off is between separation of concerns and number of rules. The example of Figure 4 privileges the first aspect to better exemplify the approach, but in many cases also the number of rules becomes an issue.

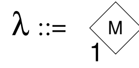
After presenting the approach, we describe the step-by-step construction of the LEMMA model used as example (cf. Figure 3). It uses seven (out of eleven) rules to create the *Entry Point* `start` and the *Clinical Test* `pregnancyTest`, to which we only add the *In Port*. For each step, we present the changes on the abstract syntax model, those on the semantic model, and the textual attributes of newly created elements. The first steps are:

(1) Rule `addModel(1, "example")` creates the new model where `1` is the identifier of the new *LEMMA Model* and `example` is its name. The transformation of the abstract syntax model is:



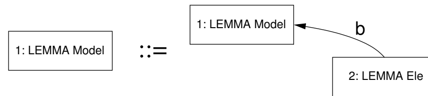

```
1.name = "example";   1.type ="LEMMAModel";
```

The transformation of the HLTPN is:



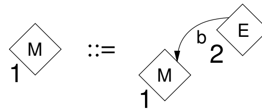

```
1.name = "exampleMM";   1.type = "M";   1.absNode = "example";
```

(2) Rule `addElement(1, 2, "start")` adds a new *LEMMA Element*. Its parameters are the identifier of the model to which the element is to be added (1), the identifier of the new element (2), and its name. The transformation of the abstract syntax model is:



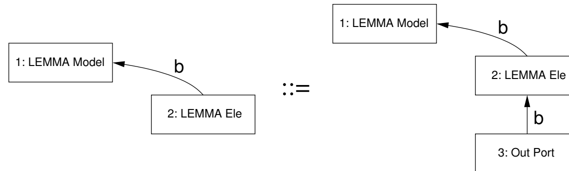

```
2.name = "start";   2.type = "LEMMAElement";
```

The transformation of the HLTPN is:



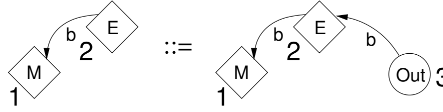

```
2.name = "startEM";   2.type = "E";   2.absNode = "start";
```

(3) Rule `addOutPort(2, 3, "out1", "any")` adds a new out port (3) to the element 2. Besides the name (`out1`), we pass a string (`any`, in this case) to let the port filter incoming data. In this case, the port accepts any data, but if the element had more out ports, these strings would help decide the port that must be enabled. The transformation of the abstract syntax model is:



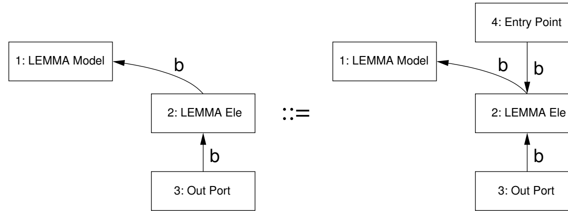

```
3.name = "out1";   3.type = "OutPort";   3.value = "any";
```

The transformation of the HLTPN is:



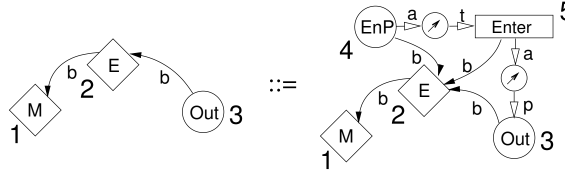

```
3.name = "out1OP";   3.type = "Out";   3.absNode = "out1";
```

(4) Rule `addEntryPoint(2, 4)` specializes element 2 by adding the *Entry Point* 4. The two parameters are the identifiers for the generic element and the new one. The SGG of this rule decides the actual semantics of the component and adds the needed HLTPN elements. The transformation of the abstract syntax model is:



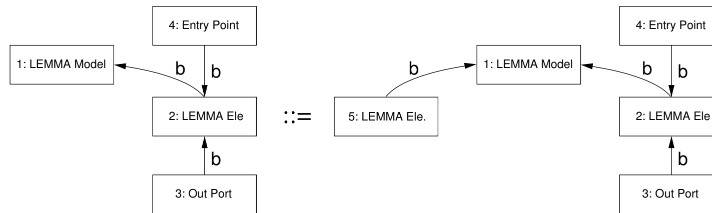

```
4.name = "startEP";   4.type = "Entry Point";
```

The transformation of the HLTPN is:



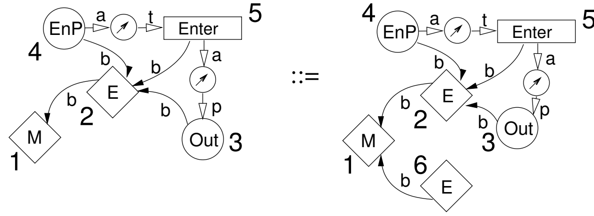

```
4.name = "startP";      4.type = "EnP";      4.absNode = "startEP";
5.name = "startT";      5.type = "Enter";    5.absNode = "startEP";
5.predicate = true;     5.action = ;         5.tMin = enab; 5.tMax = enab;
```

(5) Rule `addElement(1, 5, "pregnancyTest")` adds another *LEMMA Element*. The transformation of the abstract syntax model is:



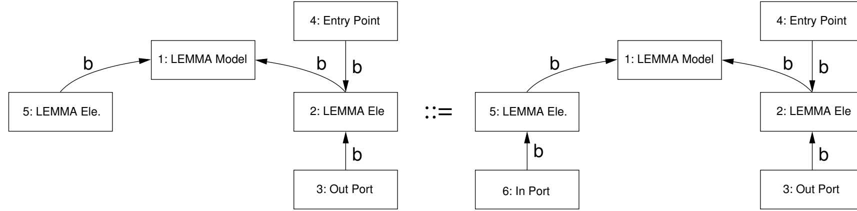

```
5.name = "pregnancyTest";   type = "LEMMAElement";
```

The transformation of the HLTPN is:



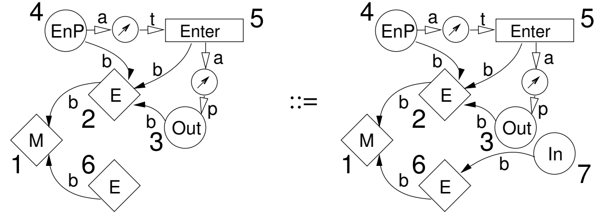
`6.name = "pregnancyTestEM";   6.type = "E";   6.absNode = "pregnancyTest";`

(6) Rule `addInPort(5, 6, "in1", "any")` attaches an in port to the element created by the previous rule, that is, node `5`. Again, parameters are the identifier of the element to which the port is added, the identifier for the port, its name, and the filter. The transformation of the abstract syntax model is:



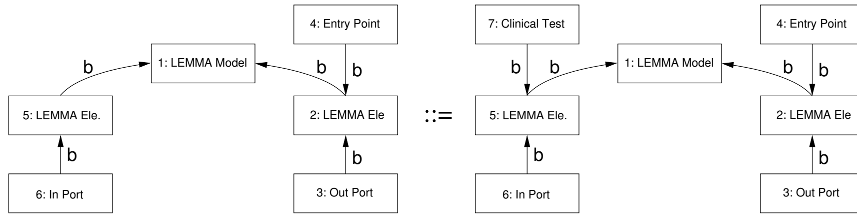
`6.name = "in1";   type = "In Port";   value = "any";`

The transformation of the HLTPN is:



`7.name = "in1IP";   7.type = "In";   7.absNode = "in1";`

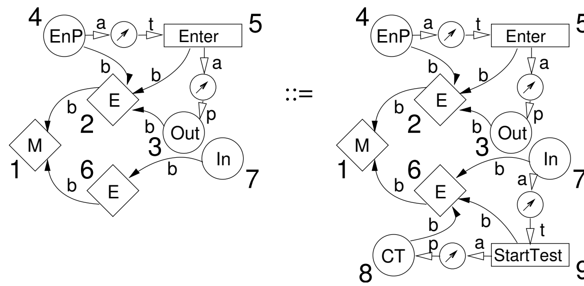
(7) Rule `addClinicalTest(5, 7)` specializes element `5` into a *Clinical Test* (`7`). The transformation of the abstract syntax model is:



`7.name = "pregnancyTestCT";   type = "Clinical Test";`

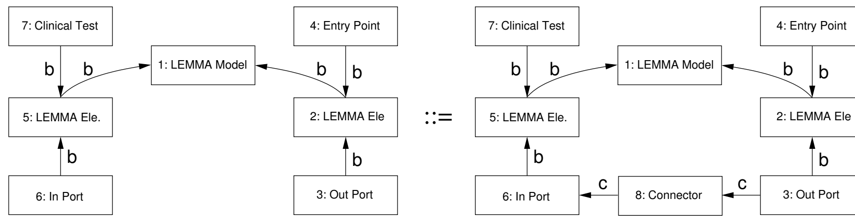
The transformation of the HLTPN is[4]:

[4]Notice that the rule of Figure 6(b) adds no *ResProd* transitions since the *LEMMA Element* has no *Out Ports*. Needless to say, this is because we only present a fragment of the building process.



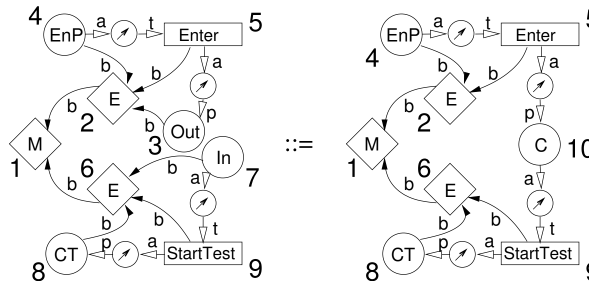

```
8.name = "pregnancyTestP";  8.type = "CT";         8.absNode = "pregnancyTestCT";
9.name = "pregnancyTestT";  9.type = "StartTest";  9.absNode = "pregnancyTestCT";
9.predicate = true;         9.action = ;           9.tMin = enab;
9.tMax = enab;
```

(8) Rule `addConnector(8, 3, 6)` sets a connector (`8`) between the out port (`3`) and the in port (`6`). Notice that the connector does not belong to any element and thus the invocation does not comprise the identifier for it. The transformation of the abstract syntax model is:



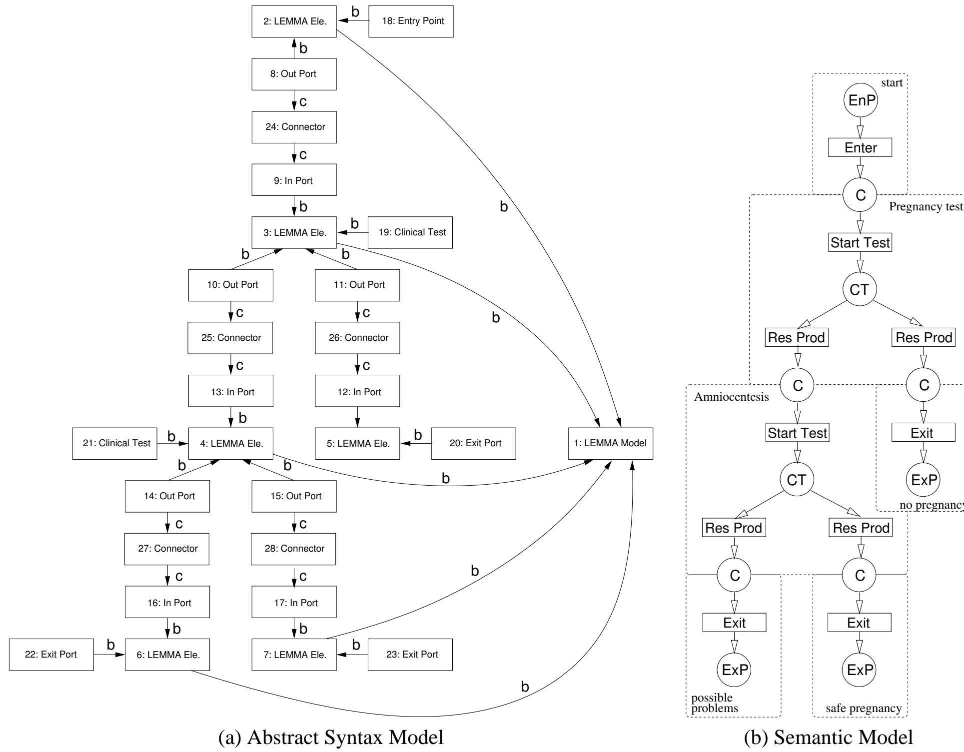

```
8.name = "out1in1";   8.type = "Connector";
```

The transformation of the HLTPN is:




```
10.name = "out1in1P";   10.type = "C";   10.absNode = "out1in1";
```

The complete sequence produces the models of Figure 8. The first model higlights the connections between elements through *ports* and *connectors*. The HLTPN, presented here without markers for the sake of simplicity, uses dashed boxes to show the sub-nets that correspond to the different elements. Notice that the connectors of the abstract syntax model become shared places in the Petri net. and are created by gluing the two places that correspond to the interested ports.



(a) Abstract Syntax Model (b) Semantic Model

Fig. 8. The models of the LEMMA process of Figure 3 (Internal views)

2.3 Visualization rules

HLTPNs allow diagram models to be formally validated through execution, reachability analysis and model checking. *Visualization rules* translate obtained results in terms of suitable visualizations on the abstract syntax elements. The rules define the "observer" policy ([Gamma et al. 1995]) that we want to use 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[5]. The predefined function `getAbsId()` returns the abstract component associated with the Petri net element.

[5]Notice that the same rules can be used to render the results produced by external analysis tools, like counter examples generated by model checkers, in terms of the diagram notation.

The complete animation of the example model of Figure 3 can be obtained with four visualization rules: *enter Process*, to move the patient in the process, *start Clinical Test* and *complete Clinical Test*, to make the patient start and complete a test, and *leave Process*, to move the patient on an exit point. Among these, Figure 9 presents rule *complete clinical test*.

```
// v is the visualization

Visualization v = new Visualization();

// tr: the fired transition
// pl: a place in its postset
// an: an animation
// getAbsId: returns the id of the abstract element
//           associated with the PN element

if (tr.type() == "ResProd") {

// we create an animation for each abstract element associated
// with the place (pl) in the postset of the transition (tr)
// and add it to the visualization

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

// we create an animation for the abstract element associated
// with the transition itself and add it to v

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

Fig. 9. The visualization 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*. This behavior is exemplified in Figure 10 that visualizes the completion of the pregnancy test of Figure 3. If we suppose that the token in the *CT* place allows the right transition to fire, this transition becomes `tr` in the rule. The place in which `tr` produces a token becomes `pl`, i.e., the post-set of `tr`. The animation associated with `pl` identifies element 26 in the abstract syntax model and sets its animation type to `prodOutput`. The animation related to `tr` concerns element 19 and is of type `completeTest`. The two animations, which can be seen as a single visual transaction, define the actual visualization, that is,

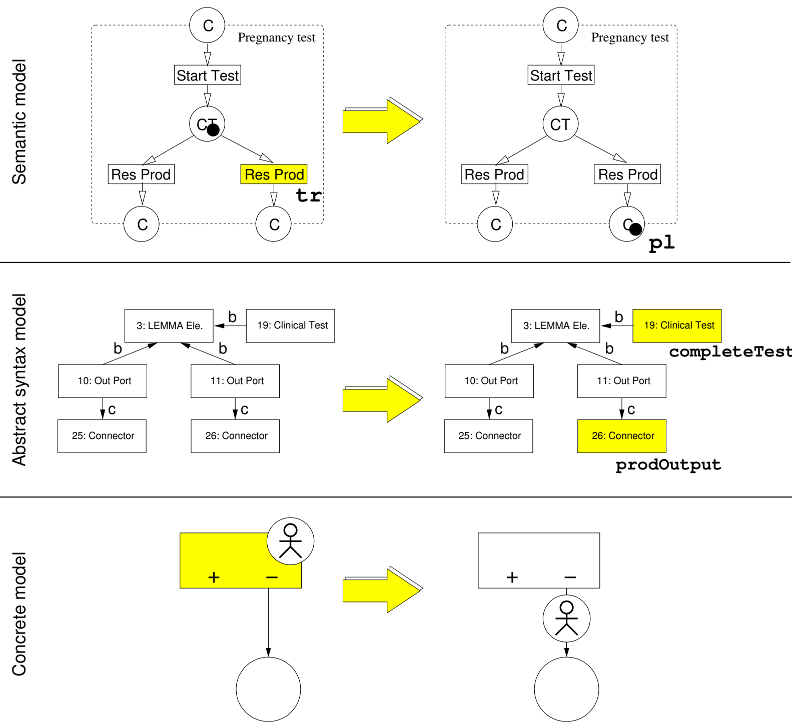


Fig. 10. How visualization rules work

the specific instantiation of the rule of Figure 9. The animations on the concrete model are specified in the translation from abstract to concrete visualizations. In the example of Figure 10, the first animation moves the patient on the outgoing flow, while the second animation changes the color of the *Clinical Test* element and thus the global effect is the one shown in bottom part of the figure.

2.4 Consistency, completeness, and correctness

A set of rules formalizes an informal interpretation of a given notation, thus the *correctness* of the set can only be defined with respect to the informal interpretation and can only be verified informally by inspecting obtained behaviors on well-known benchmarks. In contrast, we can define – and thus verify – the *completeness* and *consistency* of the formal interpretation (i.e., of the set of rules). Tools like AGG (Attributed Graph Grammars [Ermel et al. 1999]) help in many practical cases.

Completeness can be verified by proving that the mapping onto the semantic domain covers all the elements. This can be easily proved by checking the existence of at least a rule for each element of the meta-model. Notice that incomplete sets of rules may 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.

Consistency can be verified by checking the functional behavior (termination and conflu-

ence) of the selected graph transformation system ([Heckel et al. 2002]). The termination and confluence of the system prevent ambiguities in the mapping process.

3. ADAPTABILITY

Diagram notations may change to adapt to the changing context in which they are used. Experimental notations, like LEMMA, require continuous maintenance to meet the changing requests from users. They usually modify their expectations as soon as they increase their familiarity with the notation. Widely used diagram notations, like UML, Structured Analysis, or FBD, present similar changing requirements. The same notation is employed by different users with different meanings: Some symbols are different, some modeling elements are added or removed, and the behavior associated with these elements varies according to the particular domain, user experience, and specific problem.

These considerations lead to the definition of diagram notations as *notation families*, that is, sets of highly related languages that share the core definitions, but allow for different extensions and interpretations [Scholz and Petersohn 1997]. Besides the concept of *notation family*, the paper presents *consistency frameworks* as a means to formalize these families. The framework identifies coherent subsets of rules that can be suitably substituted and modified to capture existing and new interpretations of a diagram notation. It also identifies the set of rules that may be affected when we modify a rule. Thus, it addresses the problem of interpreting notation families and not just single notations. The presented framework is exemplified on Structured Analysis, one of the most complex notation families used in software engineering.

3.1 Notation families

Our definition of *notation family* uses the meta-modeling framework proposed by the Object Management Group (OMG). This *four layer meta-data architecture* [Object Management Group 2002] accommodates the *data* (instances or objects) that we want to describe, the *models* used to describe (instantiate or specify) our data, the *meta-models*, i.e., the languages used to create models, and a *meta-meta model*, that is, the notation used to describe meta-models. In this framework, notations are specified by means of a proper *meta-model*. Oftentimes, we use UML class diagrams to identify the elements of the abstract syntax of the notation.

Borrowing these definitions, we define a *notation family* as a set of notations that share the same meta-model, but that ascribe different (dynamic) semantics to its elements. A first example of notation family is Structured Analysis. If we considered only a single variant, say the one proposed by Hatley and Pirbhai ([Hatley and Pirbhai 1987]), we could reason on its many plausible interpretations ([Baresi and Pezzè 1998]). Differences would mainly deal with the way input, output, and unused values are treated.

Statecharts is another example of a domain-specific notation that can be interpreted as family. Nowadays, we tend to use Statecharts ([Harel 1987]) as if it were a formal notation by borrowing its formal semantics from STATEMATE ([Harel et al. 1990]), but more thoroughly Statecharts can be interpreted in several different ways ([von der Beeck 1994]). They all use the same graphical symbols, but differ for key aspects. For example, some Statecharts dialects consider an implicit priority mechanism along the hierarchy of *OR* states: UML associates higher priorities to inner transitions of an *OR* state, while STATEMATE does not (it does the other way around). In STATEMATE, events generated in one step cannot be used before the next step; other interpretations restrict the set of acceptable

events, or do not consider inter-level transitions and even history states.

The former definition of notation family can be relaxed by requiring that the notations belonging to a family share only the core concepts of the meta-model, while a few elements can be specific to a particular dialect. Again, this is the case of Structured Analysis: All proposed alternatives originate from the first proposal by De Marco ([De Marco 1978]), but each "dialect" reinterprets the notation by slightly changing the concrete syntax, adding new special purpose notation elements, and interpreting the loose concepts in some peculiar ways.

Another notable example is Function Block Diagram (FBD), one of the graphical notations proposed by the IEC standard 61131-3 ([Lewis 1998]) to design programmable logic controllers. The core set of elements is fixed, but designers are free to add as many modeling concepts as they want through ad-hoc libraries. The standard codes only basic elements; designers can extend the notation and add new blocks to model particular control problems.

Similarly, we can relax the definition of notation family to allow designers to extend the concepts they model without affecting the meta-model itself. This is the case of UML where designers can extend the notation with special-purpose mechanisms, like stereotypes and tagged values, to ascribe particular semantics to their modeled elements and thus add new subjects to the UML family.

These cases motivate the interpretation of diagram notations as families and demand for a structured approach for defining their (dynamic) semantics to supply consistent and coherent definitions, clearly identify the scope of each component, and be able to define new interpretations by assembling already available components.

3.2 Consistency framework

The members of a notation family share the same meta-model, that is, the same syntactic elements and their relationships, but interpret them in different ways. Given the meta-model, the dynamic semantics of a family member is defined by specifying a building rule for each element of the meta-model. We formalize the whole family when we associate each element with a set of rules to cover all its alternative behaviors. For example, different interpretations of LEMMA, which constitute a small notation family, associate different rules with each element of the meta-model of Figure 4. This also means that we exploit the meta-model to identify a suitable granularity with which we specify rules. The relationships between the elements in the meta-model suggest how changes in a rule could propagate through the set of rules:

—If changes do not affect the interfaces, and only modify the "internals", they are *local* and do not propagate to any other rule.

—If changes modify the interfaces, i.e., those HLTPN elements used to interconnect the sub-nets, they propagate transitively to the elements that share the modified interfaces and maybe to those related to them.

For example, if we substituted the internals of the building rule for adding a *Clinical Test*, changes would be local only. But if we change its interfaces, this would also impact the rules to add *In* and *Out Ports* since *Clinical Test* is a sub-class of *LEMMA Element*, which has associations with both *In Port* and *Out Port*. On the other hand, changing the rule for adding *In/Out Ports* may affect other rules, since several elements depend on them.

In general, changes on the interfaces of key elements require that several other rules be changed consequently.

Although useful, meta-models can be fairly complex and contain more information than needed for managing and constraining changes among building rules. A neater organization among rules can be obtained if we order elements of the meta-model according to a partial order and distinguish between supplied and used interfaces. The partial order must satisfy the following constraints:

(1) It includes all classes in the meta-model. We do not need to add abstract classes if they only factorize common properties, but do not own proper semantics.

(2) It includes all aggregations in the meta-model; i.e., if node A is part of node B in the meta-model, then A follows B;

(3) It includes a sub-set of the associations in the meta-model, i.e., if node A follows node B, then there is an association between node A and node B in the meta-model.

We refer to such a partial order as *consistency framework*. In a nutshell, a consistency framework can be obtained from a meta-model by considering all classes in the meta-model, suppressing abstract classes if needed, and ignoring some associations to break cycles. Notice that complex meta-models embody several consistency frameworks; the choice of the consistency framework is left up to the user. We have no general rules for breaking cycles, but a simple effective heuristic that follows from the previous observations about the impact of semantic changes on rules. We sort notation elements according to the stability of their semantics within the considered family, i.e., according to the likeliness that elements can change semantics in different interpretations. Sound consistency frameworks place elements with higher stability in higher positions of the hierarchy. Leaf elements are those that change in the different interpretations. This way obtained consistency frameworks reduce the need for changes to support new interpretations.

The consistency framework also 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: A rule can only change the interfaces it supplies, but cannot modify those that it uses. If used interfaces must be modified, we should move up in the hierarchy, modify the rule that supplies the interface, and then modify the rule that wants to use it. We constrain and limit changes, but we obtain an efficient way to trace changes among the set of rules. The more complex the set becomes, the more useful the framework is. In fact, the usefulness of the consistency framework becomes clear when we think of complex notations with several elements in the meta-model and some rules for each element. In these cases, the framework helps specify rules with a homogeneous granularity. If we use a few rules, the granularity has almost no impact, but when the number becomes higher, too-big rules would lead to specifying rules for all particular cases. A finer granularity – constrained by the meta-model – allows particular cases to be tackled by composing rules. The framework, along with the meta-model, also fosters modularity and re-use. In both cases, the explicit definition of the framework in which rules work, gives a precise identification of how rules interact and cooperate. It clearly specifies the constraints to model new rules, but it could also help identify new family members by combining one rule for each element.

Figure 11 shows a consistency framework for LEMMA. It contains all classes: a *LEMMA Element* has a proper semantics since it supplies a container for its sub-classes (elements),

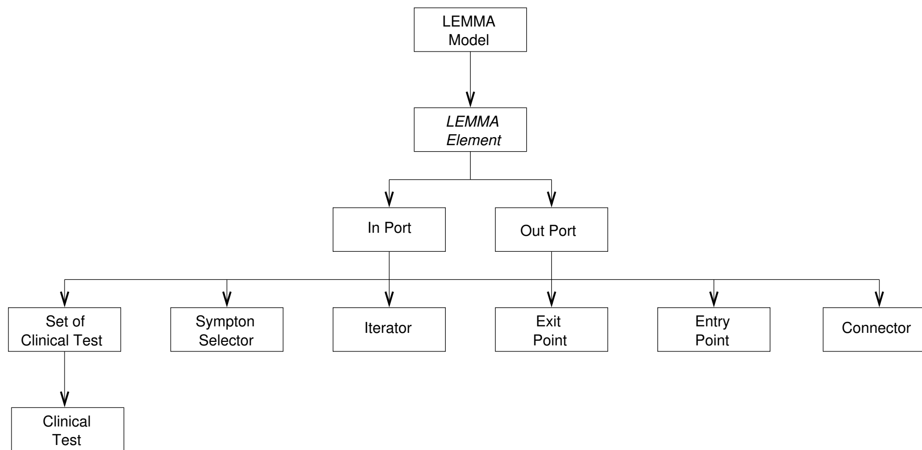


Fig. 11. A consistency framework for LEMMA

thus it is part of the consistency framework. The chosen 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.

3.3 Structured Analysis

The variety of interpretations within a notation family grows with its popularity and the informality of its definition. The popularity of Structured Analysis over the last decades and the flexibility of the informal definitions of the many variants in its plain (SA) and real-time (SA-RT) versions make Structured Analysis probably the most variegate notation family used so far in software engineering.

In this section, we present only the key concepts; [Baresi and Pezzè 2001] provides a detailed description of how to ascribe formal semantics to SA with the approach described in this paper. In particular, we present a consistency framework for SA and we discuss two of the most interesting elements of the notation: *Input Consumption* and *Hierarchy*. The consistency framework shows the difficulties of capturing a complex notation family. *Input Consumption* describes the way data on the input flows are consumed by *Processes* and is one of the SA elements with the largest number of interpretations (see [Baresi and Pezzè 1998] for a detailed discussion of the many interpretations of SA and the input consumption mechanism, in particular). The set of rules for *Input Consumption* and their simple substitutability illustrate the power of the approach in terms of modeling a highly populated family. *Hierarchy* is one of the farthest element of SA with respect to the chosen semantic domain. In fact, SA models can be decomposed in a hierarchy of diagrams, which can be controlled by *Control Transformations* in SA-RT, but HLTPNs do not provide features for modeling hierarchy. The building rules for rendering the hierarchy illustrates the capability of the approach to deal with this type of problems.

For example, Figure 12 presents an excerpt of an SA model for a vending machine for hot drinks. It can sell coffee, milk, or capuccino. The diagram uses dashed circles to summarize the hierarchical organization of the model. Processes `Vend drinks` and `Validate money` group lower level processes and data stores. The flattened model re-

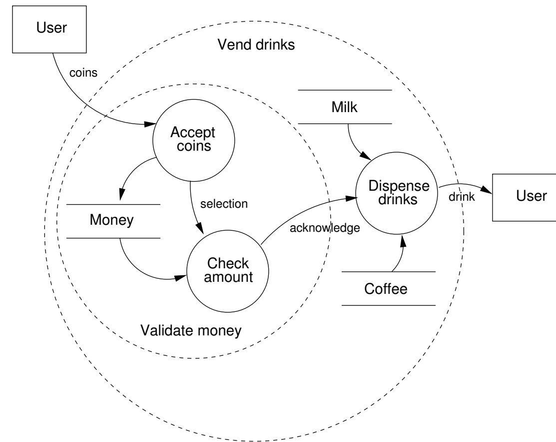


Fig. 12. A simple vending machine for hot drinks

quires that the `User` enter `coins`. Process `Accept coins` validates them and stores them in `Money`. Process `Check amount` controls that inserted coins are enough to buy the selected drink (`selection`) and "enables" process `Dispense drinks`. This process uses `Milk` and `Coffee` to provide users with their drinks.

If we concentrate on the first problem, process `Dispense drinks` helps reason about different dynamic semantics for SA processes. The obvious intuition here is that the process is enabled when there is a datum on flow `acknowledge` and can use the two data stores (`Milk` and `Coffee`) freely. This means that SA processes can use: (1) all their inputs, (2) exactly one input, (3) any particular subset of inputs. All these options are meaningful, but affect the actual drinks supplied by the dispenser. For example, if process `Dispense drinks` only used the second option, it could not dispense capuccino.

The hierarchical organization, i.e., the second problem, does not affect the behavior of the model[6], but we need to mimic it to suitably render execution and analysis results.

Both aspects, which are discussed here as significant samples of the problems related to formalizing SA, must be suitably rendered in terms of HLTPNs with enough freedom to let users select the interpretations they prefer. This is why we need the consistency framework.

Consistency Framework

Figure 13 presents a subset of a meta-model for Structured Analysis: It includes the main elements of SA, but omits the details for *Control Transformations* that characterize SA-RT.

An *SA Model* contains one or more *Diagrams*, according to the chosen hierarchical organization. A *Diagram* contains one or more *Active Data Element* and zero or more *Diagram Control*. A *Diagram Control* can control many different *Diagrams* and vice versa, since a *Diagram Control* impacts both the *Diagram* it belongs to and its heir diagrams. The relationship between *Diagram Control* and *Process Execution* indicates that a *Diagram Control* can control the execution of many processes and that the execution of a process

[6]It would impact the propagation of hierarchical control in SA-RT models, but this aspect is not addressed in this paper. Interested readers can refer to [Baresi and Pezzè 2001] for a complete discussion of possible alternatives.

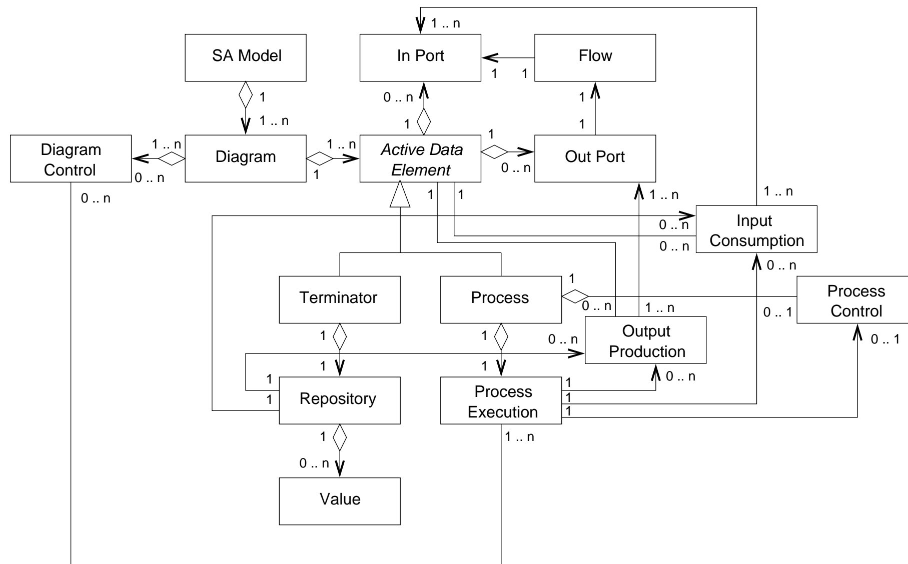


Fig. 13. A meta-model for Structured Analysis

can depend on many *Diagram Controls*. Additional control aspects of SA models, i.e., the automata that specify the control, are not captured in the subset of the SA meta-model shown in Figure 13. An *Active Data Element* can have *In* and *Out ports*, connected by *Flows*, and can contain *Input Consumptions* and *Output Productions* to model the consumption and production of data. An *Active Data Element*, which in this case is only a place holder for a neater model, can be specialized in *Process* or *Terminator*. A *Terminator* contains a *Repository* which can contain zero or more *Values*. A *Process* contains a *Process Execution* and may contain a *Process Control* to model the way the process is executed and controlled, respectively.

Figure 14 shows a possible consistency framework for the meta-model of Figure 13. The nodes of the consistency framework correspond to the classes of the meta-model (*Active Data Element* is not part of the framework since it is only a placeholder). The aggregations in the meta-model correspond to direct or indirect precedence relationships in the consistency framework and all precedences in the consistency framework correspond to relationships in the meta-model.

Building Rules

After defining the consistency framework for constraining the relationships among building rules, we explain how to solve the problems described so far: input consumption and hierarchy.

Input Consumption. Processes can be given several interpretations that differ in the number of consumed inputs, number of produced outputs, and duration of execution ([Baresi and Pezzè 1998]). If we limit our attention to how processes consume inputs, Figure 15 shows the abstract syntax and the semantic models of four possible interpretations for consuming the input values of a process with two input and one output ports. The abstract syntax models differ for the number of nodes of type *Input Consumption* and their con-

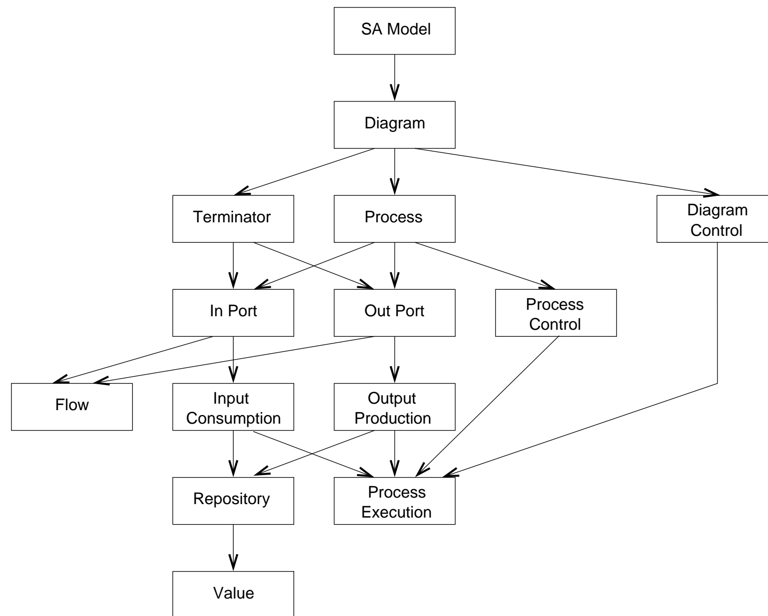
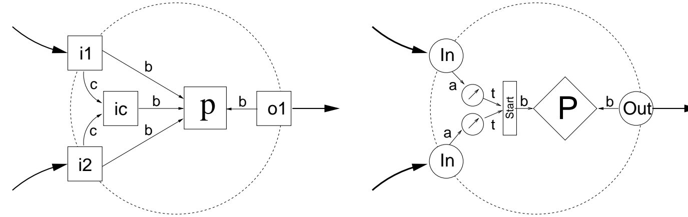


Fig. 14. A consistency framework for Structured Analysis

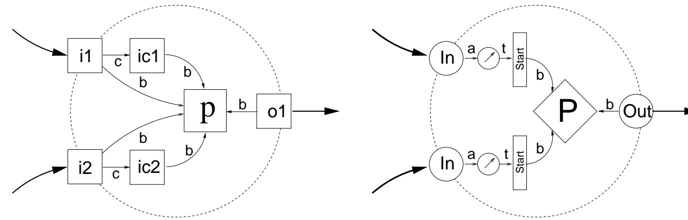
nections to the two nodes of type *In Port*. The semantic models differ for the number of transitions and arcs connected to the input places. The different interpretations can be modeled with different rules, but it is important to notice that the four nets of Figure 15 show the same interfaces. Figures 16 and 17 show the rules for cases (a), always consume from all inputs, and (c), consume from any subset of inputs. The rules for cases (b), consume from exactly one input, and (d), consume from user-defined subsets, are similar to those illustrated in this section.

The ASGG production of Figure 16 adds an *Input Consumption* node and connects it to all *In Ports* with *c* (connects) edges. The `?` in the textual annotation refers to a value that is supplied when invoking the rule. The SGG production adds a *Start* (process) transition and connects it to all *In* places through suitable arcs. The sub-production is needed because we do not statically know the number of input ports of a process and thus we need to add a variable number of arcs. The main production adds a *aa* (add arc) sp-edge for each *In* place. The *aa* sp-edge invokes the sub-production that substitutes each *aa* edge with a HLTPN arc. Textual attributes set all properties of the *Start* transition. The single transition connected to all input places fires only when all input places are marked and consumes all input values. It thus models the "consume all inputs" semantics. The firing interval does not constrain the firing since the transition is enabled from time `enab` (see Appendix A for further details) to infinity (`enab + ∞`).

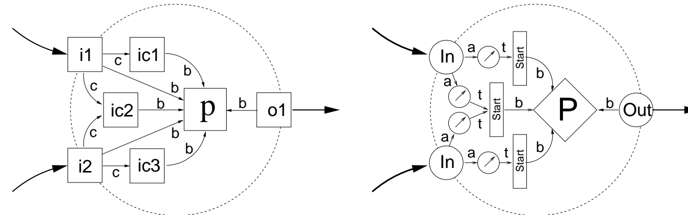
The rule of Figure 17 is one of the most complex rules for giving semantics to SA elements. The ASGG production first adds an sp-edge of type *ai* (all inputs) between the *Process* marker and all its *In Ports*. The first sub-production substitutes all *ai* sp-edges with an *Input Consumption* node and adds an sp-edge of type *oic* (other input consumption) between the considered *In Port* and all the other *Input Consumption* nodes that belong



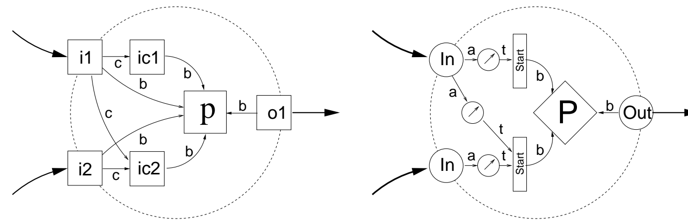
(a) Always from all input flows



(b) From one input flow at a time



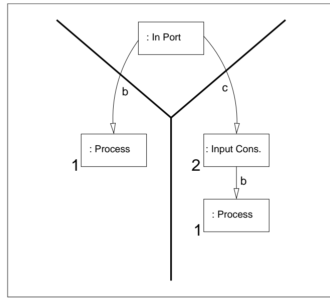
(c) From one of all possible subsets



(d) From one of user-defined subsets

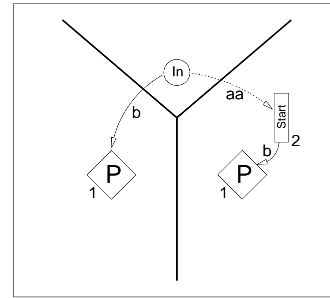
b: belong, **c**: connect, **In**: Input, **Out**: Output
P: Process, **Start**: Start Process, **a**: arc, **t**: transition, **p**: place

Fig. 15. Four interpretations of input consumption exemplified on a process with two inputs and one output (To keep diagrams simple, objects are identified with their names only: `i` identifies *In Ports*, `o` identifies *Out Ports*, `ic` identifies *Input Consumptions*, and `p` identifies *Processes*)

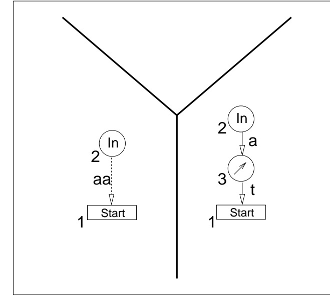



```
2.name = 1.name + "IC";
2.type = "Input Consumption";
2.action = ?;
```

(a) ASGG production




```
2.name = @2.name@ + "T";
2.type = "Start";
2.predicate = "TRUE";
2.action = @2.action@;
2.tMin = "enab"; 2.tMax = "enab + ∞";
2.absNode = @2.name@;
```



(b) SGG production

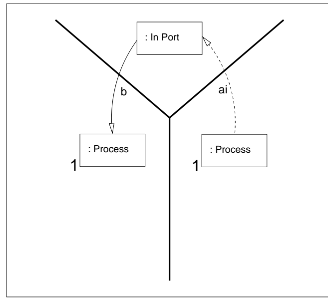
Input Cons.: Input Consuption, **P**: Process, **In**: Input, **Start**: Start Process
↗: PN arc, **b**: belong, **aa**: add arc, **t**: transition, **a**: arc

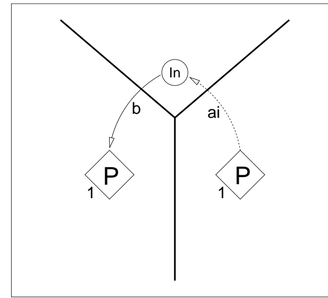
Fig. 16. Rule Add Input Consumption: The process consumes values from all input flows

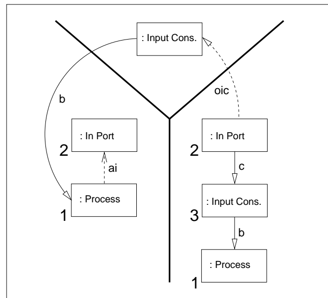
to the *Process*. The last sub-production substitutes the *oic* sp-edges with a new *Input Consumption* node connected to the selected *In port*. The combination of the two sub-productions adds a number of *Input Consumption* nodes equal to the powerset of the *In Ports*, as illustrated in Figure 18, which demonstrates how to apply the rule of Figure 17.

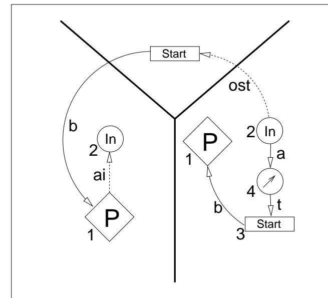
Similarly to the ASGG production, the SGG production of Figure 17 uses sub-productions to add a number of transitions equal to the powerset of the input places. It first adds an sp-edge of type *ai* (all inputs) between the *Process* marker and all its *In* places. Then, it substitutes each *ai* sp-edge with a transition *Start*, an arc between the new transition and the selected *In* place, and an sp-edge of type *ost* (other start transition). The application of the main rule and the first sub-production adds one transition for each *In* place and one *osp* edge between each transition and each input place. The second sub-production substitutes the *ost* sp-edge with yet another transition, an arc, and a *aa* (add arc) sp-edge that is then substituted with an arc by a simple sub-production not shown in the figure.

The consistency framework of Figure 14 indicates that *Input Consumption* only precedes



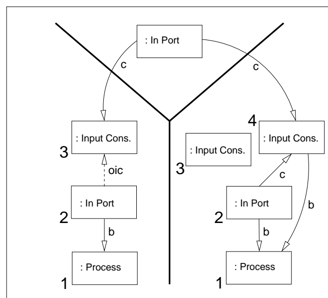


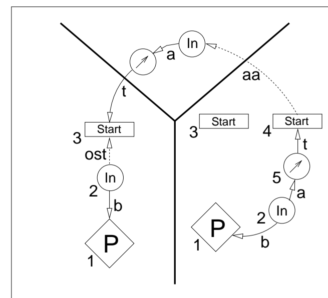





```
3.name = 1.name + 2.name + 1.cnt;
3.type = "Input Consumption";
3.action = ?;
1.cnt += 1;
```

```
3.name = @3.name@ + "T";
3.type = "Start";
3.predicate = "TRUE";
3.action = @3.action@;
3.tMin = "enab"; 3.tMax = "enab + ∞";
3.absNode = @3.name@;
```






```
4.name = 1.name + 2.name + 1.cnt;
4.type = "Input Consumption";
4.action = ?;
1.cnt += 1;
```

```
4.name = @4.name@ + "T";
4.type = "Start";
4.predicate = "TRUE";
4.action = @4.action@;
4.tMin = "enab"; 4.tMax = "enab + ∞";
4.absNode = @4.name@;
```

(a) ASGG production (b) SGG production

Input Cons.: Input Consumption, **b**: belong, **c**: connect, **ai**: all inputs,
oic: other input consumption, **P**: Process, **In**: Input, **Start**: Start Process,
ost: other start transition, **a**: arc, **t**: transition, **aa**: add arc, ↗: PN arc

Fig. 17. Rule Add Input Consumption: The process can consume values from any non-empty subset of input flows

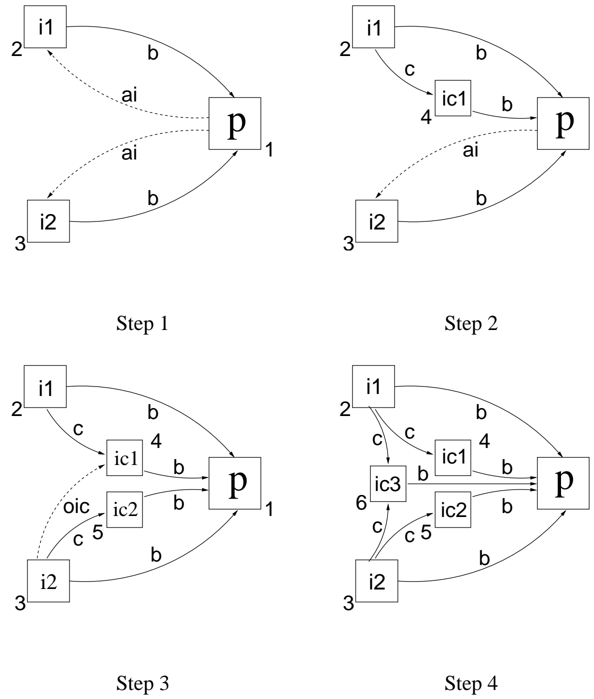


Fig. 18. Sample application of rule Add Input Consumption of Figure 17 to a process with two inputs (To keep diagrams simple, objects are identified with their names only: `i` identifies *In Ports*, `o` identifies *Out Ports*, `ic` identifies *Input Consumptions*, and `p` identifies *Processes*)

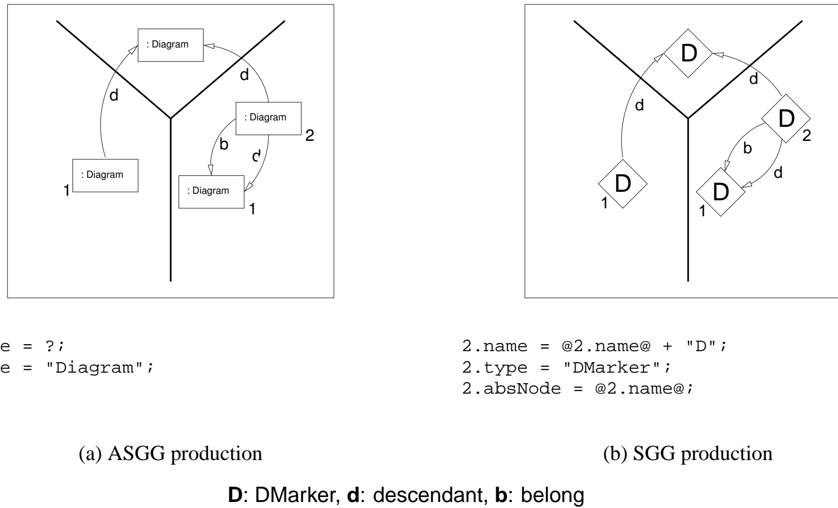
Process Execution (*Repository* and *Value* concern *Terminators* and not *Processes*). Thus substituting one rule with another to give a different interpretation impact at most the rule that describes *Process Execution*, while all other rules are not affected. This means, for example, that we can produce several rules for giving different interpretations to *Output Production* and combine them freely with any of the rules given above.

Hierarchy. SA models are usually decomposed in sets of hierarchically organized diagrams. The root is the *context diagram* that identifies the whole system and its interfaces; leaf diagrams model primitive functional transformations. The hierarchical organization helps users decompose the model and define the scope of designed components. Besides this, SA-RT extensions use hierarchy to properly propagate control flows. Each diagram can both have a local controller, that is, a *Diagram Control* that selectively regulates process activation, and be fully controlled by parent diagrams.

Hierarchical control is one of the most problematic aspects to be modeled with HLTPNs, along with the "powerset" interpretation of input consumption illustrated above. In both cases, the informal semantics of the diagram language (SA-RT) does not find corresponding constructs in the semantic domain (HLTPNs). We have illustrated above how it is possible to design building rules for the "powerset" interpretation taking advantage of the power of programmable graph grammars. The modeling of hierarchical control presents

additional challenges: Non-local elements that affect a set of elements whose number depends on the depth of the decomposition hierarchy and thus is not bound a-priori.

We solve the problem by representing hierarchy with a set of nodes of type *Diagram*. Each *Diagram* node is connected through a *b* edge to the *Diagram* node it belongs to and through a *d* (descendant) edge to all diagrams it is a descendant of. *Diagram* nodes are given semantics using markers of type *DMarker* that mirror the abstract syntax model. The marker that corresponds to the context diagram is introduced by the *Axiom* rule, which is invocated to create an empty model. Markers that correspond to other diagrams are introduced by rule Add Diagram of Figure 19.




```
2.name = ?;
2.type = "Diagram";
```

```
2.name = @2.name@ + "D";
2.type = "DMarker";
2.absNode = @2.name@;
```

(a) ASGG production (b) SGG production

D: DMarker, **d**: descendant, **b**: belong

Fig. 19. Rule Add Diagram

The two productions add a new *Diagram* marker (node *2*) as descendant of an already existing marker (node *1*). The newly added element belongs (*b* edge) to the old one, and is registered as descendant (*d* edge) of all *Diagram* markers of which node *1* was already a descendant. Edges of type *d* are not strictly necessary: All descendants of a marker could be identified by following the hierarchy defined by *b* edges, but the direct knowledge of the descendants simplifies the rules to hierarchically propagate the control. The rules that give semantics to *Control Transformations* connect a transition representing a control action to the *Process Execution* subnet of the current process, as well as of all the descendant processes indirectly affected by the control. This simple trick allows us to easily model the hierarchical propagation of control. This is an example of the power of the proposed technique that can bridge significant gaps between the diagram notation and the formal model: a hierarchical construct is rendered onto a flat model.

4. INTERPRETER GENERATOR

The approach proposed in this paper is the basis of METAENV, our prototype interpreter generator. Similar to generators for lexical and syntactical analyzers, like `Lex`, `Bison`, and `JavaCC`, METAENV is a tool for deriving formal interpreters for different diagram notations. Building and visualization rules, introduced in Section 2, are the basis for tailoring

METAENV for a particular diagram notation and a given interpretation. The *consistency framework*, defined in Section 3, is the basis for efficiently managing complex notation families.

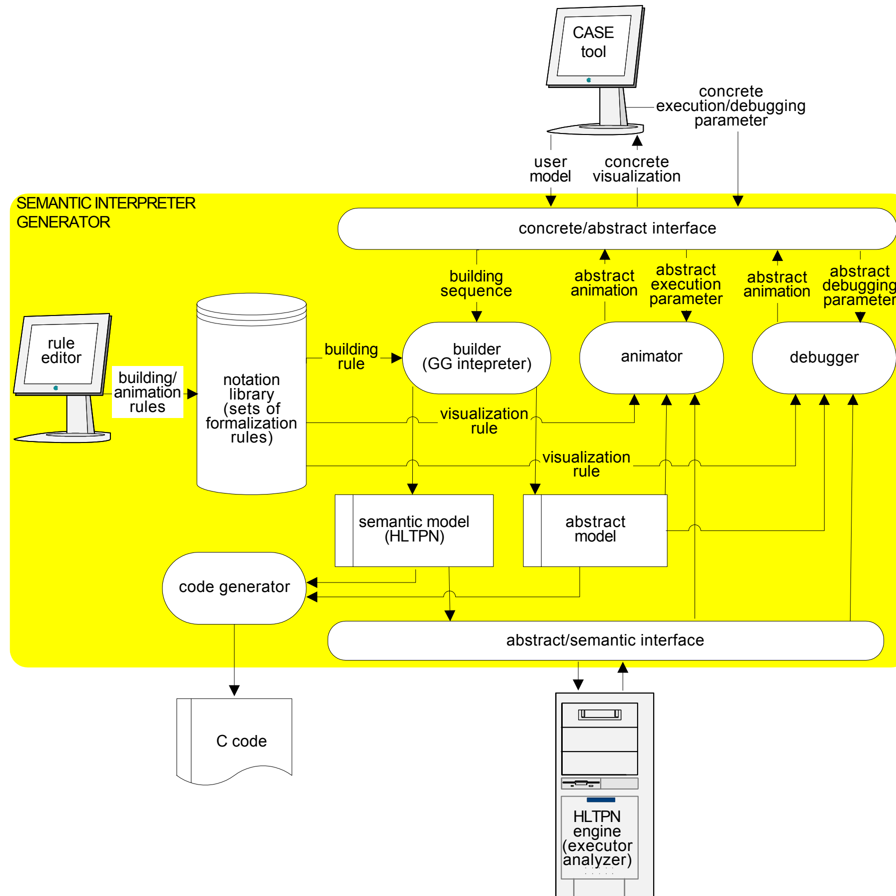


Fig. 20. Architecture of METAENV

METAENV is built around the architecture illustrated in Figure 20. The *concrete/abstract interface* plugs METAENV in an external *CASE tool*. The interface defines a two-way communication channel: It transforms user models into suitable sequences of building rules and abstract animations into concrete visualizations for the employed CASE tool. The *CASE Tool* must be a service-based graphical editor [Reiss 1990], i.e., it must supply an API that can be used to properly store, retrieve, and animate models.

The experiments described in Section 5 required extremely different efforts to deploy and integrate METAENV. For example, the integration with StP costed a lot of effort for two main reasons: The notation, SA-RT, is complex and imposes that several details

be taken into account. In addition, the proprietary language supplied to program the interaction with the tool was complex and not user-friendly. Things become easier if we only consider the identification of the right sequence of building rules and the creation of HLTPNs. Most of the complexity relayed in the transformation of visualizations into concrete animations. In contrast, this task was quite simple in all the cases where we developed the editor and thus we paid attention to the problem during the implementation of the front-end. Besides the intrinsic complexity, the effort for transforming visualizations into concrete animations also depends on the "observer" policy adopted. Needless to say, the higher the number of rules is, the more complex the task becomes.

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. A detailed analysis of the two approaches can be found in [Baresi 1997]. Roughly, in the first case, we should define a rule for each user action supported by the tool. This also means that we should have rules to delete elements since we can make mistakes or change our mind. In contrast, if we adopt an off-line approach, the number of rules would be less than in the previous case. We could decide how to scan the representation of diagrams supplied by the tool and thus the rules that we need. 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. Differently from other proposals (for example, [Arcom Control Systems 2002]), which supply the code along with the abstract machine to execute it, our code generator produces raw code to be compiled and linked using standard C compilers.

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* [Pezzè and Silva 1994] 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.

The attempt to make the two classes of users become closer would require a simplified way to specify rules. We have no general purpose approach so far, but we conducted some experiments on particular notations. For example, the peculiarities of FBD allowed us to supply users with a simplified HLTPN interface to let them define how blocks transform inputs into outputs. In this case [Baresi et al. 2000], rules can be created by adding automatically the infrastructure that completes the HLTPN and makes it become a building rule. This was possible because of the specialties of the notation, but in general we believe that domain experts need some knowledge of the internals to fully exploit the approach.

5. EXPERIMENTAL VALIDATION

The approach has been validated by plugging METAENV in different commercial and special-purpose CASE tools for experimenting with various diagram notations. All considered CASE tools provide a service-based graphical editor pluggable in METAENV.

The flexibility of the approach has been validated by experimenting with diagram languages with different characteristics. The prototype allowed for both adapting the interpretation to changes of the considered notations and experimenting the rules by formalizing and analyzing selected case studies. The cost of generating an interpreter for a given notation using METAENV is measured with the number of rules required to formalize the notation. The design of rules depends on both the difficulty of understanding the to-be-formalized notation, along with the desired interpretation, and the familiarity with the approach. All considered notations include some difficult and several straightforward rules. The formalization of notations speeded up while acquiring experience with the approach.

The main experiments conducted to validate the approach are summarized in Table 1, which lists both notations and used CASE tools. More specifically, we experimented with:

Structured Analysis. Structured Analysis ([De Marco 1978]) has been chosen as one of the richest notation families. The formalization of Structured Analysis comprises about 50 sets of rules[7]. 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

[7] The consistency framework of Figure 14 is only an excerpt of the complete framework that comprises some 50 elements.

Table I. Summary of our experiments

Notation	CASE tool	Rules
Structured Analysis [De Marco 1978]	StP [Interactive Development Environments 1994]	50
FBD [Lewis 1998]	MATLAB (*) [The MathWorks Inc. 2000]	40
UML [Fowler and Scott 2003]	Rose [IBM 2004]	20
Control Nets [Caloini et al. 1998]	tcl-tk (*) [Ousterhout 1993]	30
LEMMA [Baresi et al. 1997]	tcl-tk (*), Java (*)	11

(*) indicates special-purpose CASE tools implemented with the technology indicated in the cell. Column *Rules* indicates the approximate number of building rules required for formalizing the diagram notation.

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.

An interpretation of the SA-RT dialect proposed by Hatley and Pirbhai [Baresi 1997] was used for modeling and analyzing the hard real-time component of a radar control system by Alenia (The experiment was conducted within the ESPRIT IDERS Project – EP8593). Details on the formalization of the SA family can be found in [Bove et al. 1996].

Control Nets. 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. The CASE tool obtained by integrating METAENV with a special-purpose interface implemented in TCL-TK was successfully used to model and analyze the control of a robot arm developed by Comau. Details can be found in [Orso 1997].

Function Block Diagram. IEC Function Block Diagram (FBD) is one of the graphical languages proposed by the IEC standard 61131-3 [Lewis 1998] 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. This way, new blocks can be added by users who are not familiar with graph grammars and this facilitated the construction of libraries.

METAENV was interfaced with a special-purpose editor, PLCEDITOR, developed within MATLAB/SIMULINK. PLCEDITOR and METAENV are integrated through CORBA.

Formally interpreted FBD was used to model and analyze controls of electrical motors developed by Ansaldo (The experiment was conducted within the ESPRIT INFORMA project – EP23163). Details on the formalization of FBD and the customized *rule editor* can be found in [Carmeli et al. 2000].

LEMMA. 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.

We implemented two versions of the LEMMA toolbox by plugging METAENV in graphical interfaces generated with TCL-TK and Java. The toolbox was used at the 4th Institute of General Surgery in Rome to model and analyze the diagnosis process of colon-rectal cancer ([Baresi et al. 1997]).

UML. We also applied the approach to the Unified Modeling Language (UML) [Baresi and Pezzè 2001]. 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 Statecharts 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. They have been defined to empirically study both the applicability of our approach to the elements that characterize object-oriented notations, and the suitability of the approach to deal with multiple languages that define different aspects of the same system. Details about the results can be found in [Baresi and Pezzè 2001].

6. RELATED WORK

The problem of "interpreting" diagram notations through formal methods has been studied by several researchers who proposed different approaches for integrating formal and informal notations [Broy et al. 1998].

As already pointed out, most of the preliminary work was on Structured Analysis. For example, Semmens and Allen [Semmens and Allen 1990] complement De Marco-like SA with Z, while Wing and Zaremski [Wing and Zaremski 1991] use Larch. De Marco-like SA is supplemented also with object-oriented methodologies and VDM by Liu et al. in [Liu et al. 1998]. Several interpretations are proposed also for SA-RT. For example, the extension by Ward-Mellor (SA-WM) is formalized by France [France 1992], Fencott et al. [Fencott et al. 1994], and Petersohn et al. [Petersohn et al. 1994]; the extension by Hatley-Pirbhai (SA-HP) is studied by France and Wu [France and Wu 1995]; ESML is analyzed by Shi and Nixon [Shi and Nixon 1996] and Richter and Maffeo [Richter and

Maffeo 1993].

A wider approach is presented by Paige [Paige 1997a; 1997b], where several diagram notations are addressed, . The proposed 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.

All these proposals exploit the formal model as a means to supply a single fixed formal semantics to the considered diagram notation. None of them addresses the problem of the backward mapping of analysis and simulation results.

More recently, some proposals concentrated on SDL [Fischer et al. 1995; Sherratt 2003] and many on UML. In some cases, like the pUML approach ([Evans and Kent 1999]) 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 Statecharts diagrams and not the whole language. For example, Engels et al. [Engels et al. 2001] use CSP (Communicating Sequential Processes [Hoare 1978]) to formally define the dynamic semantics, Kuske [Kuske 2001] uses graph transformation as a means to define ad-hoc interpreters, Traorè [Traoré 2000] exploits PVS, and Paltor and Lilius [Lilius and Paltor 1999] utilize state term graphs. Only the last proposal describes a round-trip approach where the results of model checking are rendered visually on UML models. Quite different is the proposal by Engels et al. [Engels et al. 1999] where UML interaction diagrams are transformed directly into Java code: The formalization remains implicit, but is mandatory to define automatic translation mechanisms.

A notation-independent approach is supplied by GENGED [Bardohl 2000], where a diagram notation can be specified through three different graph grammars. The *syntax grammar* defines the actual syntax of the language, enough to implement a syntax-directed editor. For free-hand editing, GENGED also requires a *parse grammar*, but if users want to simulate their models, they must provide a *simulation grammar* to specify how their models behave when fed correctly.

Other proposals (for example, UPGRADE [Böhlen et al. 2002], DIAGEN [Minas and Köth 2000], and DOME [Honeywell 2000]) are mainly meta-CASE tools, which only support the capabilities of defining diagram languages, but do not support dynamic semantics and thus related analysis capabilities.

A number of researchers propose parametric semantic models as means to customize the dynamic semantics of diagram models. For example, Day and Joice use higher-order logic [Day and Joyce 1999], Pezzè and Young propose hypergraph rules [Pezzé and Young 1997], Dillon and Stirewalt combine process algebras and temporal logic [Dillon and Stirewalt 2003], Niu et al. introduce hierarchical state-transition machines [Niu et al. 2003]. These approaches focus directly on the formal model and make it become parametric; our approach moves a step beyond and supplies rules to create the formal models. We do not limit the choices on the formal model to some parameters, but we offer a more flexible solution to address a wider set of solutions.

7. CONCLUSIONS

This paper proposes an approach and a supporting toolset for defining formal interpreters for diagram notations. The approach is based on *building rules* to create HLTPNs that

are equivalent to diagram models, and *visualization rules* to map analysis and simulation results from HLTPNs to proper visualizations of elements in diagram models.

The paper also extends the approach to interpret diagram notations as *notation families* and supplies the concept of *consistency framework* to help ascribe formal dynamic semantics to notation families and better scope changes in proposed interpretations.

The approach was validated through METAENV that supplies a general-purpose interpreter that can be tailored to a particular notation with two proper sets of 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

Arcom Control Systems 2002. *ISaGRAPH Programming Suite*. Arcom Control Systems.

BARDOHL, R. 2000. GenGED - Visual Definition of Visual Languages based on Algebraic Graph Transformation. Ph.D. thesis, Technische Universität Berlin.

BARESI, L. 1997. Formal Customization of Graphical Notations. Ph.D. thesis, Dipartimento di Elettronica e Informazione – Politecnico di Milano. In Italian.

BARESI, L., CONSORTI, F., PAOLA, M. D., GARGIULO, A., AND PEZZÈ, M. 1997. LEMMA: A Language for an Easy Medical Models Analysis. *Journal of Medical Systems – Plenum Publishing Co. 21,* 6 (December), 369–388.

BARESI, L. AND HECKEL, R. 2002. Tutorial Introduction to Graph Transformation: A Software Engineering Perspective. In *Proceedings of the First International Conference on Graph Transformation (ICGT 2002)*. Lecture Notes in Computer Science, vol. 2505. Springer-Verlag, 402–429.

BARESI, L., MAURI, M., MONTI, A., AND PEZZÈ, M. 2000. PLCTools: Design, Formal Validation, and Code Generation for Programmable Controllers. In *Proceedings of the IEEE Conference on System, Man, and Cybernetics*. Vol. 4. IEEE-CS, 2437–2442.

BARESI, L. AND PEZZÈ, M. 1998. Towards Formalizing Structured Analysis. *ACM Transactions on Software Engineering and Methodology 7,* 1 (January), 80–107.

BARESI, L. AND PEZZÈ, M. 2001. Formal Semantics for Notation Families: The Case of Structured Analysis. Tech. rep., Dipartimento di Elettronica e Infomazione – Politecnico di Milano.

BARESI, L. AND PEZZÈ, M. 2001. On Formalizing UML with High-Level Petri Nets. In *Concurrent Object-Oriented Programming and Petri Nets*. Springer-Verlag, 276–304.

BÖHLEN, B., JÄGER, D., SCHLEICHER, A., AND WESTFECHTEL, B. 2002. UPGRADE: A Framework for Building Graph-Based Interactive Tools. In *Proceedings of the International Workshop on Graph-Based Tools*. Electronic Notes in Computer Science, vol. 72. Elsevier, 149–159.

BOVE, R., DIPOPPA, G., AND PERROTTA, A. 1996. The RPCM System Core Specified and Animated Using IDERS. Tech. rep., Alenia. Apr. IDERS Doc.id : IDERS-ALENIA-42-V2.2.

BROY, M., COLEMAN, D., MAIBAUM, T., AND RUMPE, B., Eds. 1998. *Workshop on Precise Semantics for Software Modeling Techniques*. In conjunction with ICSE 1998.

BURNETT, M. AND BAKER, M. 1993. A Classification System for Visual Programming Languages. Tech. Rep. 93-60-14, Department of Computer Science, Oregon State University.

CALOINI, A., MAGNANI, G., AND PEZZÈ, M. 1998. A Technique for Designing Robotic Control Systems based on Petri Nets. *IEEE Transactions on Control Systems Technology 5,* 1 (January), 72–87.

CARMELI, S., COSATTO, E., AND PENNO, C. 2000. Ansaldo Demonstration: Design of Application Components. Tech. Rep. INFORMA-AI-17, Ansaldo Sistemi Industriali.

DAY, N. A. AND JOYCE, J. J. 1999. Symbolic Functional Evaluation. In *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*. Springer-Verlag, 341–358.

DE MARCO, T. 1978. *Structured Analysis and System Specification*. Prentice-Hall.

DILLON, L. AND STIREWALT, R. 2003. Inference Graphs: A Computational Structure Supporting Generation of Customizable and Correct Analysis Components. *IEEE Transactions on Software Engineering 29,* 2 (February), 133–150.

ENGELS, G., HECKEL, R., AND KÜSTER, J. 2001. Rule-based Specification of Behavioral Consistency based on the UML Meta Model. In *Proceedings of UML 2001*. Lecture Notes in Computer Science, vol. 2185. Springer-Verlag, 272–287.

ENGELS, G., HÜCKING, R., SAUER, S., AND WAGNER, A. 1999. UML Collaboration Diagrams and their Transformation to Java. In *Proceedings of UML'99*. Lecture Notes in Computer Science, vol. 1723. Springer-Verlag, 473–488.

ERMEL, C., RUDOLF, M., AND TAENTZER, G. 1999. The AGG Approach: Language and Tool Environment. In *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages, and Tools*, G. Engels, H.-J. Kreowski, and G. Rozenberg, Eds. World Scientific, 551 – 601.

EVANS, A., FRANCE, R., LANO, K., AND RUMPE, B. 1998. The UML as a Formal Modelling Notation. In *Proceedings of UML'98*. Lecture Notes in Computer Science, vol. 1618. Springer-Verlag, 336–348.

EVANS, A. AND KENT, S. 1999. Core Meta-Modelling Semantics of UML: The pUML Approach. In *Proceedings of UML'99*. Lecture Notes in Computer Science, vol. 1723. Springer-Verlag, 140–155.

FENCOTT, P. C., GALLOWAY, A. J., LOCKYER, M. A., O'BRIEN, S. J., AND PEARSON, S. 1994. Formalising the Semantics of Ward/Mellor SA/RT Essential Models using a Process Algebra. In *Proceedings of FME94: Industrial Benefit of Formal Methods*. Lecture Notes in Computer Science, vol. 873. Springer-Verlag, 681–674.

FISCHER, J., DIMITROV, E., AND TAUBERT, U. 1995. Analysis and Formal Verification of SDL92 Specifications using Extended Petri Nets. Tech. Rep. 43, Department of Computer Science, Humbolt University Berlin.

FOWLER, M. AND SCOTT, K. 2003. *UML Distilled: A Brief Guide to the Standard Object Modeling Language (3nd Edition)*. Addison-Wesley.

FRANCE, R. 1992. Semantically Extended Data Flow Diagrams: A Formal Specification Tool. *IEEE Transactions on Software Engineering 18,* 4, 329–346.

FRANCE, R. AND WU, J. 1995. Rigorous Dynamic Analysis of SA/RT Requirements Models. Tech. rep., Department of Computer Science & Engineeering - Florida Atlantic University.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns*. Addison-Wesley.

GHEZZI, C., MANDRIOLI, D., MORASCA, S., AND PEZZÈ, M. 1991. A Unified High-Level Petri Net Model For Time-Critical Systems. *IEEE Transactions on Software Engineering 17,* 2 (Feb.), 160–172.

GÖTTLER, H. 1983. Attribute Graph Grammars for Graphics. In *Graph Grammars and Their Application to Computer Science*. Lecture Notes in Computer Science, vol. 153. Springer-Verlag, 130–142.

GÖTTLER, H. 1992. Diagram Editors = Graphs + Attributes + Graph Grammars. *International Journal Man-Machine Studies 4,* 37, 481–502.

HAREL, D. 1987. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming 8,* 3, 231–274.

HAREL, D., LACHOVER, H., NAAMAD, A., PNUELI, A., POLITI, M., SHERMAN, R., SHTULL-TRAURING, A., AND TRAKHTENBROT, M. 1990. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering 16,* 4 (Apr.), 403–414.

HATLEY, D. J. AND PIRBHAI, I. A. 1987. *Strategies for Real-Time System Specification*. Dorset House, New York.

HECKEL, R., KÜSTER, J., AND TAENTZER, G. 2002. Confluence of Typed Attributed Graph Transformation Systems. In *Proceedings of Graph Transformation, 1st Int. Conference, ICGT 2002*. Lecture Notes in Computer Science, vol. 2505. Springer-Verlag, 161–176.

HOARE, C. 1978. Communicating sequential processes. *Communicat. Associat. Comput. Mach. 21,* 8, 666–677.

HONEYWELL. 2000. What is DOME? Tech. rep., Honeywell. `www.htc.honeywell.com/dome/description.htm`.

IBM 2004. *Rational Rose XDE Modeler: User's Manuals* . IBM.

Interactive Development Environments 1994. *Structure Environment: Using the StP/SE Editors*. Interactive Development Environments. Release 5.

KUSKE, S. 2001. A Formal Semantics of UML State Machines Based on Structured Graph Transformation. In *Proceedings of UML'01*. LNCS, vol. 2185. Springer-Verlag, 241–256.

LEWIS, R. 1998. *Programming Industrial Control Systems Using IEC 1131-3*. IEE Publishing.

LILIUS, J. AND PALTOR, I. P. 1999. vUML: A Tool for Verifying UML Models. In *14th IEEE International Conference on Automated Software Engineering*. IEEE CS, 255–258.

LIU, S., OFFUTT, A., HO-STUART, C., SUN, Y., AND OHBA, M. 1998. SOFL: A Formal Engineering Methodology for Industrial Applications. *IEEE Transactions on Software Engineering 24,* 1 (Jan.), 24–45.

MINAS, M. AND KÖTH, O. 2000. Generating Diagram Editors with DIAGEN. In *Proceedings of AGTIVE'99*. Lecture Notes in Computer Science, vol. 1779. Springer-Verlag, 433–440.

MURATA, T. 1989. Petri Nets: Properties, Analysis, and Applications. *Proceedings of the IEEE 77,* 4 (Apr.), 541–580.

NIU, J., ATLEE, J. M., AND DAY, N. A. 2003. Template Semantics for Model-based Notations. *IEEE Transactions on Software Engineering 29,* 10 (October), 866–882.

OBJECT MANAGEMENT GROUP. 2002. Meta Object Facility (MOF) Specification - v.1.4. Tech. rep., OMG. Mar.

ORSO, A. 1997. An Environment for Designing Real-Time Control Systems. Tech. Rep. 97-56, Dipartimento di Elettronica e Informazione - Politecnico di Milano.

OUSTERHOUT, J. 1993. *TCL and the TK Toolkit*. Professional Computing Series. Addison Wesley.

PAIGE, R. 1997a. Case Studies in Using a Meta-Method for Formal Method Integration. In *Proceedings of the Internation Conference on Algebraic Methodology and Software Technology*. Lecture Notes in Computer Science, vol. 1349. Springer-Verlag, 395–403.

PAIGE, R. 1997b. A Meta-method for Formal Method Integration. Lecture Notes in Computer Science, vol. 1313. Springer-Verlag, 473–485.

PETERSOHN, C., DE ROEVER, W., HUIZING, C., AND PELESKA, J. 1994. Formal Semantics for Ward & Mellor's Transformation Schemas. In *Proceedings of the Sixth Refinement Workshop of the BCS FACS*. Springer-Verlag.

PEZZÈ, M. AND SILVA, S. 1994. Cabernet User Manual. Tech. Rep. 47-94, Politecnico di Milano. May.

PEZZÉ, M. AND YOUNG, M. 1997. Constructing Multi-formalism State-space Analysis Tools: Using Rules to Specify Dynamic Semantics of Models. In *Proceedings of the 19th International Conference on Software Engineering*. ACM Press, 239–250.

REISS, S. 1990. Connecting Tools using Message Passing in the FIELD Program Development Environment. *IEEE Software 7,* 4 (July), 57–66.

RICHTER, G. AND MAFFEO, B. 1993. Toward a Rigorous Interpretation of ESML - Extended Systems Modeling Language. *IEEE Transactions on Software Engineering 19,* 2, 165–180.

SCHOLZ, D. AND PETERSOHN, C. 1997. Towards a Formal Semantics for an Integrated SA/RT & Z Specification Language. In *Proceedings of 1st International Conference on Formal Engineering Methods (ICFEM'97)*. IEEE CS, 28–37.

SEMMENS, L. AND ALLEN, P. 1990. Using Yourdon and Z: An Approach to Formal Specification. In *Proceedings of the 5th Z User Workshop*, J. Nicholls, Ed. Springer-Verlag.

SHERRATT, E. 2003. *Telecommunications and beyond: The BroaderApplicability of SDL and MSC: Third International Workshop*. Lecture Notes in Computer Science, vol. 2599. Springer-Verlag.

SHI, L. AND NIXON, P. 1996. An Improved Translation of SA/RT Specification Model to High-Level Timed Petri Nets. In *Proceedings of Formal Methods Europe 96*. Lecture Notes in Computer Science, vol. 1051. Spriger-Verlag, 518–537.

The MathWorks Inc. 2000. *MATLAB 5.3*. The MathWorks Inc.

TRAORÉ, I. 2000. An Outline of PVS Semantics for UML Statecharts. *Journal of Universal Computer Science 6,* 11 (Nov.), 1088–1108.

VON DER BEECK, M. 1994. A Comparison of Statecharts Variants. In *Proceedings of Formal Tecniques in Real Time and Fault Tolerant Systems*. Lecture Notes in Computer Science, vol. 863. Springer, 128–148.

WING, J. AND ZAREMSKI, A. 1991. Unintrusive ways to integrate formal specifications in practice. In *Proceedings of Formal Software Development Methods (VDM '91)*. Lecture Notes in Computer Science, vol. 551. Springer, 545–570.

A. HIGH-LEVEL TIMED PETRI NETS

Petri nets have been augmented with data and time in several ways. In our work we used High-Level Timed Petri Nets (HLTPNs), which were introduced by Ghezzi et al.

in [Ghezzi et al. 1991].

HLTPNs are Petri nets, i.e., bipartite graphs, augmented with data and time. Tokens are associated with both data values and timestamps. Values allow to model additional characteristics of the application domain; the timestamp records the time of the token, which is defined as the firing time of the transition that creates the token. Transitions are associated with a predicate, an action, and a time-function. The predicate is evaluated on both the values and timestamps of the input tokens, and indicates the condition that must be verified to enable the transition. The action defines the values associated with the tokens produced by the firings as a function of the values and timestamps of the tokens removed by the firing. The time-function indicates the firing interval of the transition, i.e., its minimum and maximum firing times expressed as functions over the values and timestamps of the tokens removed by the firing.

METAENV associates a type with places and transitions to allow for statically analyzing the compatibility of predicates, actions, and time-functions with the pre- and post-sets of transitions, and thus avoiding annoying run-time problems. The type of places indicates the type of the tokens that can mark the place. The type of transitions indicates the signature of the associated predicates, actions, and time-functions. METAENV also sub-classes types associated with places and transitions to better characterize them and ease the definition of suitable visualization rules.

Figure 21 presents a simple HLTPN excerpted from the HLTPN of Figure 8(b). This net describes the semantics of the first steps of the example clinical process: A patient with symptoms that suggest pregnancy enters the process and undergoes a pregnancy test.

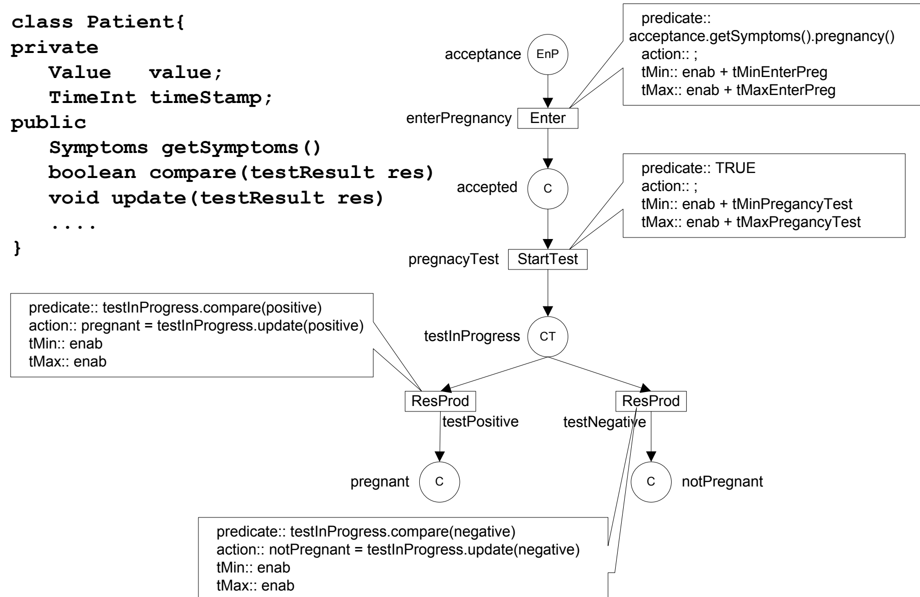


Fig. 21. A simple HLTPN

All five places have `Patient` as super-type that contains the data to identify patients.

Type `Patient` is partially given in the bottom part of Figure 21. The inscriptions in places and transitions indicate the actual sub-type used by visualization rules. Transitions are labelled for easy referencing. Arcs have weight 1, thus transitions can access at most one token per place at a time. Consequently, predicates, actions and time-functions can use the name of the place to refer to the token in that place. For example, the predicate of transition `enterPregnancy` refers to method `getSympthoms` on the token in the input place with name `acceptance`.

The predicate of transitions is a boolean predicate. The special value `TRUE` denotes the constant predicate that does not restrict the set of enabling tokens. The predicate of transition `enterPregnancy` selects the patient with pregnancy symptoms (`acceptance.getSymptoms().pregnancy()`)[8]. The predicates of transitions `testPositive` and `testNegative` enable the transitions if `pregnancyTest` produces positive or negative results, respectively.

Actions describe the modifications caused by the firing of transitions on input tokens. The full specifications of produced tokens are only needed when transitions remove or produce tokens with different types. The empty predicate ';' indicates that the values associated with created tokens are not modified by the firing, except for the timestamp. The actions of transitions `testPositive` and `testNegative` modify field `pregnant` of the created token with value `positive` or `negative`, respectively.

Time functions are given as pairs of functions $\langle tMin, tMax\rangle$ that compute the minimum and maximum firing times of the transitions. The special value `enab` indicates the enabling time, i.e., the maximum among the times associated with the tokens considered for firing the transition. Constants `tMinEnterPreg`, `tMinPregnancyTest`, `tMaxEnterPreg`, and `tMaxPregnancyTest` indicate the timings of the different activities.

According to the strong time semantics used in this paper, a transition is enabled if it is functionally enabled, i.e., the associated predicate evaluates to *true*, and if the firing time interval is non empty:

(1) The minimum firing time is given by the maximum between the *tMin* associated with the transition and the time of the last firing (this is to ensure monotonicity of time);

(2) The maximum firing time is given by the minimum of the *tMax* of all transitions that are functionally enabled in the current marking (this is to ensure strong time semantics, i.e., enabled transitions cannot be disabled by the firing of unrelated transitions).

For example, let us assume that after the last firing at time 2, we have three functionally enabled transitions, *T1*, *T2*, and *T3*, with the following *tMin* and *tMax*: $\langle 1, 6\rangle$, $\langle 3, 4\rangle$, and $\langle 5, 8\rangle$. Transition *T1* is enabled between 2 and 4, in fact it cannot fire before the last firing time (2), otherwise time would regress, and later than the maximum firing time for *T2* (4), otherwise it would disable transition *T2* advancing time beyond the maximum firing time for *T2*. This would contradict strong time semantics that forces transitions to fire within their firing time interval. Transition *T2* is enabled between 3 and 4. Transition *T3* is not enabled in the current marking, since it cannot fire before time 5, thus its firing would violate the strong time semantics by disabling transition *T3* (and possibly also transition *T1*, as side effect of its firing).

[8]Methods `getSymptoms` and `pregnancy` are not explicitly given in the figure.

The firing of a transition removes selected tokens from the pre-set and adds a token to each place of the post-set with the values specified by the action and a timestamp chosen among the possible firing times. In the example, `tMinEnterPreg` and `tMinPregnancyTest` indicate the minimum and maximum times for accepting the patient, while `tMaxEnterPreg` and `tMaxPregnancyTest` indicate the minimum and maximum times needed for a pregnancy test. Transitions `testPositive` and `testNegative` fire immediately, i.e., they present choices and not the termination of activities with a non-null execution time.