Survey of Graph Rewriting applied to Model Transformations

Francisco de la Parra and Thomas Dean
School of Computing, Queen's University, Kingston, Ontario, Canada

Keywords: Graph Grammars, Graph Rewriting, Model Transformations, Domain Analysis, Model-Driven Engineering.

Abstract: Model-based software development has become a mainstream approach for efficiently producing designs, test suites and program code. In this context, model-to-model transformations have become first-class entities and their classification, formalization and implementation are the subject of ongoing research. This work surveys the characteristics and properties of graph rewriting systems and their application to formalize and implement transformations of language-based software models. A model's structure or behaviour can be abstracted into the definition of a given graph type. Its structural and behavioural changes can be represented by rule-based transformation of graphs.

1 INTRODUCTION

Model Driven Engineering (MDE) has become a sound and efficient alternative for the design and development of complex software of the kind found in industrial and networked applications. Efficient application of MDE can produce powerful and reusable designs, as well as readily verifiable program code (Neema and Karsai, 2006).

In MDE, model-to-model transformations are first-class entities and their classification, formalization and research has focused on: 1) analyzing their intrinsic properties; 2) defining formal syntactic and semantic aspects of their realization; and 3) developing approaches for their implementation (Bézivin, 2004; Czarnecki and Helsen, 2006; Mens and Gorp, 2006).

Many modeling languages utilize graphs in the core of their representation. A model's structure can be abstracted as a given graph type. Its structural and behavioural changes can be represented by rule-based transformations of graphs (Boyd and McBrien, 2005; Karsai, 2010). This work surveys the characteristics of graph rewriting systems, known as: 1) *graph grammars* if the emphasis is on studying the types of graphs they generate (i.e., *graph languages*), 2) graph transformation systems if the primary interest is in describing the graph rewriting mechanisms they use; and their application to formalize, specify and implement transformations on models. The presentation focuses on providing a concise and system-oriented view of the more versatile graph rewriting techniques which could aid MDE researchers in devising rule-based model transformations.

1.1 Background

Graph grammars and graph transformations have been used in software engineering as formalisms for representing design and computational aspects of systems at a very high level of abstraction.

Application Domains and Modeling. Application domain analysis (Prieto-Díaz, 1990) provides precise descriptions and formalizations of the objects and activities in a given structured environment. Graphs and graph transformations are a capable meta-language to abstract from real systems or other models (Kent, 2002; Kühne, 2006). In the language-oriented context of MDE, they represent a promisory alternative to provide more precise syntactic and semantic definitions of diagrammatic modeling languages.

Graph Grammars. From its intial introduction in the late 1960's, graph grammar theory has developed into a number of approaches: algebraic techniques, set theoretic approaches, node-label controlled (NLC) approaches, etc. (Ehrig et al., 1991; Nagl, 1987; Rozenberg, 1997; Schürr, 1995). From a software engineering perspective, the initial focus on graph grammars has shifted from understanding the properties of the graph languages they generate to an interest in describing and exploiting the graph transformation mechanisms they use, how they integrate into graph

de la Parra F. and Dean T..
Survey of Graph Rewriting applied to Model Transformations.
DOI: 10.5220/0004731504310441
In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2014), pages 431-441
ISBN: 978-989-758-007-9

rewriting systems, and how they can be applied as high level specifications in modeling systems and processes.

Graph Rewriting Tools. Graph^{Ed} (Himsolt, 1991) and Algebraic Graph Grammar (AGG) system (Taentzer, 2004) represent early efforts to prototype graph grammars in a standalone fashion. PROGRES was one of the first integrated environments for interactively prototyping graph grammars and transformations with the goal of producing software artifacts (Nagl and Schürr, 1991; Schürr et al., 1995). The environment consisted of: 1) a visual/operational specification language for graphs and graph rewriting rules, 2) a set of tools for editing, analyzing and executing specifications, and 3) a methodology to use of the language and tools, denominated graph grammar engineering. PROGRES can be considered a precursor for a number of more specialized prototyping tools dealing with models and model transformations: AToM^3 (de Lara et al., 2004b), the Graph Rewriting and Transformation (GReAT) system (Balasubramanian et al., 2006; Karsai et al., 2003), FUJABA (Nickel et al., 2000), MOFLON (Weisemöller et al., 2011), VIATRA2 (Balogh and Varró, 2006), and MATE (Legros et al., 2011).

2 GRAPH GRAMMAR APPROACHES

2.1 General Concepts

The following is an abbreviated description of key components.

2.1.1 Graphs and Hypergraphs

A *simple labeled directed graph G* over a pair of label alphabets (Σ_V, Σ_E) (where $\Sigma_V = \{vl_1, ..., vl_p\}$: node label alphabet, and $\Sigma_E = \{el_1, ..., el_q\}$: edge label alphabet) consists of a sextuple $(V, E, s_V, t_V, l_V, l_E)$ where $V = \{v_1, ..., v_n\}$ (set of nodes; or vertices), $E = \{e_1, ..., e_m\}$ (set of directed edges), $s_V : E \rightarrow V$ assigns an edge's source node, $t_V : E \rightarrow V$ assigns an edge's target node, $l_V : V \rightarrow \Sigma_V$ assigns node labels, $l_E : E \rightarrow \Sigma_E$ assigns edge labels (fig. 1-(a)).

From the previous definition of a simple labeled graph, one can derive the more specific classes of graphs: 1) *undirected graphs*, if for all edges the source and target are not identified; 2) *multigraphs* multiple edges can be defined between a pair of source and target nodes; 3) *unlabeled graph*, if the label alphabets Σ_V, Σ_E are empty sets; 4) *typed graph*, if the sets of node and edge labels are partitioned into classes called *types*; 5) *attributed graph*, if nodes are assigned attributes of a given data type; and 6) *Star graphs* consisting of nonterminal and terminal nodes, where the node labeled as nonterminal is at the center of a star configuration (fig. 1-(b)).

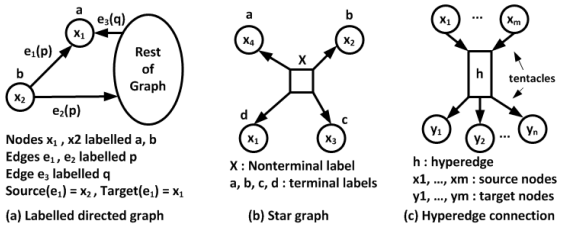

Figure 1: Graphs and Hypergraphs.

Hypergraphs represent an extension of the structural concept of a graph to contain *hyperedges*, instead of edges, connecting to multiple source and target nodes (fig. 1-(c)) (Habel, 1992).

Triples of Graphs extend the idea of a single graph as the unit of analysis to a triple (LG, CG, RG) consisting of three graphs: LG (left graph), RG (right graph), and CG (connection graph), of the same class, where graph CG represents associations between the elements of graphs LG and RG. *Triple graph grammars* are the structures associated with generating and parsing these graph triples (Schürr, 1995).

Hierarchical Graphs constitute a most general structure with *higher-order nodes* that are abstractions of graphs and *higher-order edges* that represent relations between graphs which are represented as graphs again. With these structures, it is possible to abstract an entire sub-graph to a node or to a single edge between two abstracted nodes (Dean and Cordy, 1995).

2.1.2 Graph Rewriting Productions

The following is a general definition of a production that is applicable to most graph transformation (i.e., graph grammar) approaches found in the literature (Andries et al., 1999).

A graph rewriting production is a sextuple $p = (LG, K, RG, ac, gl, em)$[1] that allows the transformation of a *host graph G* into a *direct derivation graph H*, whose components are defined as follows:

LG is the production's left graph. Applying a production to a host graph *G* involves searching for one or more isomorphic occurrences of *LG* in *G* which will eventually be replaced by instantiations of graph *RG*.

[1] A production either uses embedding or gluing but not both simultaneously

K is the production's interface graph which is a sub-graph (i.e., no dangling edges: all its edges have a target and a source node). *LG* and *RG* contain an isomorphic occurrence of it.

RG is the production's right graph which will be attached to an intermediate graph $(G - LG) \cup K$, in the more common case, using graph gluing or embedding, or both operations, to produce a direct derivation graph H as the overall result of applying the production.

ac are the application conditions under which a production will be applied. They could represent the existence or non-existence of certain nodes, edges, or subgraphs in the host graph G, as well as embedding restrictions of *LG* in G or of *RG* in H.

gl is the gluing operation which consists of three steps: 1) Build a context graph D which contains the graph G, minus the nodes and edges of the isomorphic occurrence of *LG* in G that are not preserved in *RG*; 2) Glue graphs D and *LG* through K to generate the host graph G. This operation is the disjoint union of graphs D and *LG*; and 3) Glue graphs D and *RG* through K to generate the direct derivation graph H. This operation is the disjoint union of graphs D and *RG*.

em is the embedding (or connecting) operation, which creates edges between designated nodes in the context graph D and nodes in the graph *RG*, to produce the final direct derivation graph H.

2.1.3 Graph Grammars

A graph grammar is a quadruple $GG = (T, S, P, A)$ where:

T is the type or class of graph (e.g., labeled, hypergraph), that can be generated, modified and recognized by applying a set P of productions.

S is the start graph to which a set P of productions will be initially applied.

P is a set of productions $P = \{p_1, ..., p_n\}$.

A is a set of additional specifications extending the scope and nature of applying the set of productions P to graphs of type T (e.g., attributes, programmed rules, global application conditions, structural conditions).

The language $L(GG)$ generated by graph grammar *GG* is the set of terminal graphs that can be generated from the *start graph S* by repeated application of the graph rewriting productions.

Applying a set of productions $P = \{p_1, ..., p_n\}$ to one or more graphs involves scheduling the application of the individual productions in a non-deterministic or deterministic manner. Deterministic approaches can utilize ad-hoc programmed rules, prioritization schemes, application conditions and hybrid schemes to schedule productions. It is well known that these extensions reduce the generative power of a grammar, making the parsing and recognition of the graphs in the generated language a more manageable task (Nagl, 1987). In general, they facilitate using graph grammars in applications such as subject modeling, entity description, process specification and pattern recognition, by increasing their descriptive power (Drewes et al., 2008).

2.2 Context-free Graph Grammars

Context-free grammars replace a single node or edge by a complete graph when applying a production. They are important in that graph derivations can be modelled by derivation trees which produce the same set of graphs for any different ordering used to apply the productions (Courcelle, 1987). They are useful in describing and generating graphs with recursive properties.

2.2.1 Algorithmic Approaches

These approaches to graph grammars implement a graph rewriting mechanism which replaces a node v_l, labeled with the symbol l from an alphabet Σ, in a host graph G, by reconnecting to the context graph $D = G - v_l - E_d$ (E_d: set of dangling edges in G after removing v_l) an isomorphic copy of a grammar production's *RG* graph through edges determined by path expressions (Nagl, 1987), connection relations (Rozenberg, 1997), or some other algorithm.

The following three sections introduce the grammar types: NLC, NCE and edNCE, which have completely local graph rewriting mechanisms, where a production's *RG* graph is reconnected only to the neighbour nodes of the node being replaced.

NLC Grammars. A *node-label-controlled* (NLC) graph grammar is a quadruple (NU, P, C, S) where:

NU = (V, E, Σ, L_v) is a *node-labeled undirected graph type* where: 1) V is a finite set of nodes; 2) E is a set of undirected edges; 3) $\Sigma = \Sigma_N \cup \Sigma_T$ is a finite alphabet of symbols, with Σ_N: non-terminals, Σ_T: terminals; and 4) $l_v : V \to \Sigma$ is the labelling function which assigns a terminal or non-terminal symbol to each node in V.

P = $\{p_1, ..., p_n\}$ is a finite set of productions, with each $p_i = (L_i, RG_i, C_i)$, L_i: non-terminal symbol in Σ_N (left side of production), RG_i: graph of type *NU* (right side of production), C_i: connection relation of p_i.

C = $\{(\rho, \sigma) \mid \rho, \sigma \in \Sigma\} = \bigcup_{1 \le i \le n} C_i$ ($C \subseteq \Sigma \times \Sigma$) is the connection relation. A pair (ρ, σ) in this relation indicates that an edge should be created between each

node labeled σ in graph *RG* and each node labeled ρ in the context graph *D*.

S is the start graph, usually a single node labeled with a non-terminal symbol.

The graph replacement and embedding mechanisms in NLC grammars are strictly local to the vicinity of a single labeled node.

NCE Grammars. A *neighbourhood controlled embedding* (NCE) grammar represents a refinement of a NLC grammar to provide enhanced control over the local graph embedding mechanism. It is defined as a quadruple (NU, P, C, S) where: 1) ***NU*** as in a NLC grammar; 2) ***P*** is a set of productions as specified for NLC grammars, with the addition that the right-hand-sides of the pairs in the connection relation C_i of each production now refer to specific nodes in graph *RG*; 3) $\boldsymbol{C} = \{(\rho, v) \mid \rho \in \Sigma, v \in V\} = \bigcup_{1 \le i \le n} C_i$ $(C \subseteq \Sigma \times V)$ is the connection relation. A pair (ρ, v) in this relation indicates that an edge should be created between a specific node labeled v in one of the productions and any node labeled ρ in the *context graph*; and 4) ***S*** is the start graph as defined in a NLC grammar. These grammars still generate undirected node-labeled graphs with unlabeled edges.

edNCE Grammars. *Edge-labeled directed-graph neighbourhood controlled embedding* (edNCE) grammars represent an improvement of NLC grammars with an enhanced graph embedding mechanism and enhanced structural rules to generate and recognize more general classes of graphs. An edNCE grammar is a quadruple (ND, P, C, S) where: 1) ***ND*** is defined similarly to *NU* with the enhancement of labeled directed edges; 2) ***P*** is a finite set of productions whose left-hand-sides are defined as in the case of NLC grammars, whose right-hand-sides are now graphs of type *ND*, and the right-hand-sides of the pairs in the connection relation C_i refer now to specific nodes in graph *RG*; 3) $\boldsymbol{C} = \{(\rho, v, d) \mid \rho \in \Sigma, v \in V, d \in \{in, out\}\} = \bigcup_{1 \le i \le n} C_i$ $(C \subseteq \Sigma \times V)$ is the connection relation, a triple (ρ, v, d) in this relation indicates that an incoming $(d = in)$ or outgoing $(d = out)$ directed edge should be created between a specific node labeled v in one of the productions and any node labeled ρ in the context graph; and 4) ***S*** is the start graph as defined for a NLC grammar.

It is well known that some other possible extensions to the graph embedding mechanism, for example, allowing flipping directions on edges, or allowing multiple edges in the tuples of the connection relation, do not increase the generation power of an edNCE grammar (Rozenberg, 1997).

Grammars with Arbitrary Embedding. In general, the embedding mechanism of a production's *RG* graph can be extended to an arbitrary degree of complexity, beyond local schemes, allowing reconnecting it to not only neighbour nodes of the node being replaced but also to any node in the context graph *D*. This would allow a further increase in the generation power of a grammar using the node reconnecting approach. However, the trade-off is increased complexity of the generated graphs, which could make parsing them a very difficult or even intractable task. Furthermore, applying these grammars to modeling could become cumbersome.

The algorithmic approach to grammars can be used to describe the syntax of simple diagrams such as object models in UML.

2.2.2 Hypergraph Grammars

The central element of a hypergraph is the *hyperedge*, which is an abstraction consisting of a single edge h with N links ("tentacles") that are connected to a set $S(h)$ of m source nodes and a set $T(h)$ of n target nodes. Similar to ordinary graphs, one can define different types of hypergraphs: unlabeled, directed labeled, etc. A special type is a *directed labeled multi-pointed hypergraph*, denominated an (m,n)-*hypergraph*, which contains two sets of labeled sequential nodes: *begin* and *end*, with cardinalities m and n, representing overall source and target nodes of the hypergraph, respectively. This structure provides the elements to implement hyperedge replacement grammar productions using a node gluing approach, where a single hyperedge is replaced by a (m,n)-hypergraph. The gluing operation occurs between the hyperedge's $S(h)$ and $T(h)$ nodes and sets *begin* and *end* of a (m,n)-hypergraph, respectively.

Hyperedge-Replacement Grammars. A *hyperedge replacement* (HR) grammar is a quadruple (H_M, P, C, S) where: 1) H_M is a (m,n)-hypergraph specification; 2) P is a set of productions p_i, each $p_i = (L_i, RH_i, C_i)$, L_i is the label of a single hyperedge h (production's left side), RH_i: hypergraph of type H_M (production's right side), C_i: re-connection mapping for RH_i, production p_i applied to a host hypergraph *GH* replaces hyperedge L_i by hypergraph RH_i reconnecting it according to mapping C_i to produce derived hypergraph *HH*; 3) $C = \{(\delta_b, \delta_e) \mid \delta_b : \pi(begin) \rightarrow S(h), \delta_e : \pi(end) \rightarrow T(h); \delta_b, \delta_e$ surjective functions$\}$ ($\pi(begin), \pi(end)$: permutations of *begin* and *end*, respectively. $C = \bigcup_{1 \le i \le n} C_i$) is the node-gluing mapping; 4) S is the start hypergraph, usually a single

handle[2] labeled with a non-terminal symbol.

HR-grammars have convenient context-free properties, mostly due to the fact that their hypergraph replacement mechanism is completely local to a single hyperedge. They are quite flexible structures with a wide spectrum of generation power, considering that they can generate graphs and hypergraphs of varying complexity just by changing the *type* (i.e., number of tentacles) on hyperedges. The hyperedge concept is a fundamental structure in many different types of models (Habel, 1992), hence it is applicable to consider HR-grammars as having a wide range of descriptive power. For example, transitions in Petri nets are more easily represented as hypergraphs than as complex subgraphs.

Hypergraph Replacement Grammars. In these structures, denominated HGR-grammars, productions can replace arbitrary sub-hypergraphs in a given host hypergraph. They use the double pushout graph rewriting technique defined in the algebraic approach to graph grammars (Section 2.3) to generate derived hypergraphs. HGR-grammars can provide more flexible and less granular productions than HR-grammars, and can be reduced to the latter grammars when the sub-hypergraph replaced is a single hyperedge.

The generative power of HR- and HGR-grammars and the classes of languages they generate have been extensively studied (Habel, 1992; Rozenberg, 1997), however, the attention has shifted to studying specific hypergraphs such as *jungles* [3] (Ehrig et al., 1999) and their rewriting mechanisms. The nodes and edges of a jungle can represent the terms and operations of functional or algebraic expressions. This approach allows a more efficient evaluation of expressions by avoiding the multiple representation, hence the re-calculation, of repeated terms.

2.3 Algebraic Approach to Graph Grammars

The algebraic approach to graph grammars focuses on the rewriting mechanism of direct derivations of graphs, rather than on the study of general constructs to obtain graph languages (Ehrig, 1979; Ehrig et al., 2006; Rozenberg, 1997). Its basic operation is graph gluing, where common nodes and edges in a production's *LG* (left) and *RG* (right) graphs play a key role in maintaining graph consistency when replacing *LG*

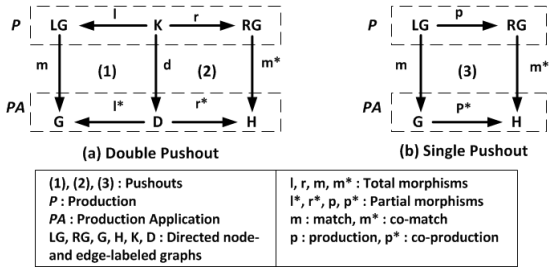

Figure 2: Single and Double Pushout Production Application (adapted from (Ehrig et al., 2006)).

by *RG* in a host graph *G*. In this approach, given a node- and edge-labeled simple graph *G*, the application of a production *p* yields a graph *H* if the following conditions are met:

1) The definition of production *p* is based on partial graph morphisms between its *LG* (left graph) and its *RG* (right graph) in the case of the single pushout approach (SPO) (fig. 2-(b)), or is based on total graph morphisms between *LG*, *RG* and an auxiliary interface graph *K*, containing the nodes and edges required for gluing *RG* to a context graph *D* (fig. 2-(a)), in the case of the double pushout approach (DPO).

2) The application of production *p*, represented by pushout constructions (1) and (2), or (3) in the DPO and SPO approaches, respectively, removes the nodes and edges of *G* that can be mapped to *LG* but not to *RG*, and adds to *G* the new nodes and edges of *RG* to produce *H*.

Similar to all approaches, a graph grammar GG consists of a start graph G_0 and a finite set of productions $P = \{p_1, ...p_n\}$, each production as defined above. The language *L*(GG) generated by the grammar is the set $\{G_n \mid G_0 \overset{*}{\Rightarrow} G_n; n = 1, 2, ...\}$.

The grammars obtained using this approach, generally context-sensitive, can be highly tuned to specific graph modeling situations with arbitrary start graphs and application-specific graph derivation steps.

The focus on direct graph derivations, typical of the algebraic approach, facilitates detailed analyses of comparative properties, structural transformations, and semantics associated with specific subsets of productions in a grammar and on specific graphs of its generated language, which can be used to find optimized ways to transform graphs of a given class.

If two productions p_1 and p_2 must work cooperatively on graphs to model, for example, a concept of synchronized execution of tasks or updates to systems states, then they must synchronize their application to common items accessed by both on those graphs. One solution to this problem is to amalgamate the shared

[2] A handle is a single (m,n)-hyperedge connected to sets of nodes *begin* and *end*.

[3] *jungles* are forests of coalesced trees with shared structures that can be represented by acyclic hypergraphs

effects of both productions into a common subproduction p_s, embedded in p_1 and p_2.

If a derivation frequently occurs in some modeling situation and the interest concentrates on the initial and final graphs, it seems natural to search for an optimization of the productions that would bypass the derivation of the intermediate graphs. This problem translates to finding a derived production $p^* : G_0 \rightsquigarrow G_n$. This production exists if, given a derivation $\alpha \equiv G_0 \overset{p_1}{\Rightarrow} ... \overset{p_n}{\Rightarrow} G_n$ and an injective morphism $G_0 \overset{e_0}{\rightarrow} H_0$, e_0 induces an embedding $e : \alpha \rightarrow \beta$ (where $\beta \equiv H_0 \overset{p_1}{\Rightarrow} ... \overset{p_n}{\Rightarrow} H_n$) and $H_0 \overset{p^*}{\Rightarrow} H_n$. Derived production p^* for derivation sequence α is given by $p^* = p_1^*;...;p_n^*$ (i.e., sequential application of coproductions p_i^* (fig. 2-(b))).

2.4 Triple Graph Grammars

The formalism of triple graph grammars (TGG) was devised (Schürr, 1995) to generate and transform pairs of related graphs (usually called source and target graphs) under a well-formed synchronization mechanism (i.e., a connection graph) which would preserve the relations in the pair after applying a transformation. The initial motivation of the formalism was to provide a high-level graph-based modeling and specification tool for problems involving related diagrams (i.e., syntax trees, control flow diagrams) and information structures (i.e., requirements, design and traceability documents) requiring simultaneous update for consistency purposes. More recently, key concepts found in TGGs have been used in the OMG's QVT model transformation language standard (Object Management Group, 2011) to define the "mappings" of a transformation. Also, applications of TGGs to the specification of bidirectional and incremental model transformations are becoming more common (Czarnecki et al., 2009; Stevens, 2010; Hermann et al., 2013).

The categorical framework (Ehrig et al., 2006) is the standard mathematical tool used to describe TGGs: their elements and operations. TGG's elementary unit of transformation is the *graph triple*, denoted as $GT \equiv (LG \overset{gl}{\leftarrow} CG \overset{gr}{\rightarrow} RG)$, where LG: left graph, CG: connection graph, RG: right graph (usually node- and edge-labeled directed graphs, however, other types of graphs can be used), and gl, gr are graph morphisms (fig. 3). The transformation operator is the *triple production*, denoted as $p \equiv (lp \overset{lr}{\leftarrow} cp \overset{rr}{\rightarrow} rp)$, and consisting of three monotonic single productions [4]: $lp \equiv (LL, LR)$, $cp \equiv (CL, CR)$ and

[4]Production $p \equiv (L, R)$ is monotonic if $L \subseteq R$.

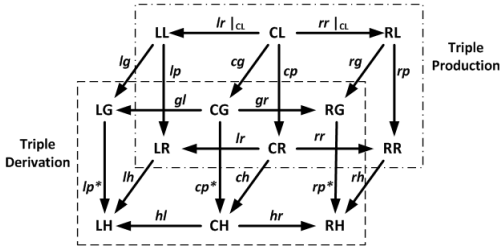

Figure 3: TGG Production Application (adapted from (Schürr, 1995)).

$rp \equiv (RL, RR)$ (with graph morphisms $lp : LL \rightarrow LR$, $cp : CL \rightarrow CR$ and $rp : RL \rightarrow RR$, respectively, and the induced graph morphisms $lr|_{CL} : CL \rightarrow LL$, $rr|_{CL} : CL \rightarrow RL$[5], $lr : CR \rightarrow LR$ and $rr : CR \rightarrow RR$), which are applied simultaneously (fig. 3).

Applying a triple production p to a graph triple GT produces a directly derived triple $HT \equiv (LH \overset{hl}{\leftarrow} CH \overset{hr}{\rightarrow} RH)$ (fig. 3), derivation denoted as $GT \overset{p}{\rightsquigarrow} HT$, where the application of each of its component productions lp, cp and rp are modeled by the single pushout categorical construct described in fig. 2-(b). That is, productions lp, cp and rp, match LL, CL and RL in graphs LG, CG and RG through total morphisms lg, cg and rg, respectively. Then, they transform the latter triple into graphs LH, CH and RH, which are related to the applied production and original graphs by the morphism pairs (lh, lp*), (ch, cp*), and (rh, rp*), respectively (fig. 3).

The basic definition of TGGs with monotonic productions, although partially restrictive, makes it more feasible to specify translation tools based on inter-graph relationships by simplifying the manipulations required to obtain a triple's target graph that an arbitrary source graph, as shown below (Schürr, 1995).

A triple production $p \equiv ((LL, LR) \overset{lr}{\leftarrow} (CL, CR) \overset{rr}{\rightarrow} (RL, RR))$ can be split into a pair of equivalent productions: a left-local production p_L and a left-to-right production p_{LR}, with:

- $p_L \equiv ((LL, LR) \overset{\varepsilon}{\leftarrow} (\phi, \phi) \overset{\varepsilon}{\rightarrow} (\phi, \phi))$ [6]
- $p_{LR} \equiv ((LR, LR) \overset{lr}{\leftarrow} (CL, CR) \overset{rr}{\rightarrow} (RL, RR))$

When p is applied to a triple GT, of which, only graph LG is known, denoted by $GT \overset{p(lh)}{\rightsquigarrow} HT$ (HT: derived triple), if its component productions are monotonic, the following equivalence holds:

[5]$lr|_{CL}$ and $rr|_{CL}$ indicate that lr and rr are total morphisms over graph CL

[6]ϕ: empty graph; ε: inclusion of empty graph in any graph.

$$GT \overset{p(lh)}{\rightsquigarrow} HT \Longleftrightarrow \exists\, XT \mid GT \overset{p_L(lh)}{\rightsquigarrow} XT \wedge XT \overset{p_{LR}(lh)}{\rightsquigarrow} HT$$

This result extends to the application of a sequence of triple productions $p_1, ..., p_n$ in the form of the following equation:

$$p_1(lh_1) \circ ... \circ p_n(lh_n) = (p_{1L}(lh_1) \circ ... \circ p_{nL}(h_n)) \circ (p_{1LR}(lh_1) \circ ... \circ p_{nLR}(lh_n))$$

which translates to say that the application of a *sequence of triple productions* is equivalent to the application of a sequence of left-local productions followed by the application of a sequence of left-to-right productions. The previous analysis is also valid for right-local and right-to-left productions.

3 GRAPH TRANSFORMATION SYSTEMS

A significant body of concepts is common to all the graph grammar approaches found in the literature. Graph types and production types are common examples of useful abstractions. When extending these ideas to applications involving large numbers of rules (i.e., productions) and complex systems, resorting to proven software engineering principles (i.e., classification, encapsulation, reuse and module-based semantics) to further structure the graph transformation process is essential if the requirement is to produce manageable and scalable graph-based systems.

Graph transformation systems (Ehrig et al., 1999; Kreowski et al., 2010) is a formalism that focus on defining encapsulation abstractions such as transformation units and modules, using precise graph-based semantics, which can be reused and aggregated to build larger systems. The basic structuring generalization is a *graph transformation approach* $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \Rightarrow, \mathcal{E}, \mathcal{C})$, where:

$\mathcal{G}$ is a class of graphs of a specified type (i.e., labelled graphs, hypergraphs, etc.).

$\mathcal{R}$ is a class of rules with the same structure (i.e., single pushout, double pushout, etc.).

$\Rightarrow$ is a rule application operator that for each rule $r \in \mathcal{R}$ produces a relation $\overset{r}{\Rightarrow} \equiv \{(g_1, g_2) \in \mathcal{G} \times \mathcal{G} \mid g_1 \overset{r}{\Rightarrow} g_2\}$.

$\mathcal{E}$ is a class of *graph class expressions*, where each one of them defines: a finite enumeration of graphs, or a set theoretic definition of graphs, or a graph theoretic property, or the exhaustive output of a given transformation unit, or some graph meta-definition scheme, or the result of boolean operations on graph classes.

$\mathcal{C}$ is a class of *elementary control conditions* over a set ID of identifiers, which are expressed as rule execution schedules, priority schemes, boolean expressions and regular languages. For a given environment E, defined by the mapping $E : ID \rightarrow 2^{\mathcal{G} \times \mathcal{G}}$ [7], each $c \in \mathcal{C}$ specifies a relation $SEM_E(c)$ consisting of pairs $(G, G') \in \mathcal{G} \times \mathcal{G}$, where G: initial graph, G': derived graph.

This scheme allows to refer to the multiple grammar types in an approach-independent manner that facilitates the analysis of similarities and differences in properties, structures and rules.

Given a graph transformation approach, a *transformation system* consists of a set of transformation units with the capability to import each other forming acyclic or cyclic import paths. Each transformation unit produces an output terminal graph from an initial graph by interleaving rule applications with calls to other transformation units, where the latter are executed in an atomic manner.

3.1 Transformation Units

A transformation unit (TU) consists of a set of graph transformation rules conforming to a graph transformation approach $\mathcal{A}$ and a set of atomic-execution calls to imported transformation units, where elements of both sets execute interleaving each other ("interleaving semantics") to transform an initial graph of a given class into a specific terminal graph of the same or another class. The calls to other import transformation units can be cyclic or acyclic, depending on whether a "called unit" calls one of its "calling units" in a given calling path or not.[8] The input and output graph classes are defined by class expressions.

A TU can also include a *control condition* (*cc*) whose purpose is to regulate, restrict, and possibly eliminate the inherent non-determinism appearing in graph transformations due to the fact that: 1) multiple rules can simultaneously be applicable to an existing graph, 2) there can exist multiple places in a graph where a given rule could be applied. A *cc* can have properties of minimality (i.e., the graphs transformed by a TU are only the ones allowed by the *cc*), invertibility (i.e., for a given *cc* C, C^{-1} exists) and continuity (i.e., given a TU with a *cc* C over a sequence of environments $\{E_1, ..., E_n\}$, then $SEM_{E_1 \cup ... \cup E_n}(C) = SEM_{E_1}(C) \cup ... \cup SEM_{E_n}(C)$). The following are the most common types of control conditions:

1) *Control conditions of language type* which only allow interleaving sequences of rules and imported

[7] Given a set A, 2^A denotes its power set

[8] This analysis assumes acyclic calling paths.

TUs that can be represented by strings belonging to a specific control language L.

2) *Priorities* assigned to the rules of a TU can represent control conditions that allow only specific pairs $(G,G') \in SEM_E(C)$.

3) *Reduced graphs* with respect to a control condition (denoted $C!$) is a type of control condition where the only pairs $(G,G') \in \mathcal{G} \times \mathcal{G}$ allowed, have the property that there is no graph G'' such that $(G',G'') \in SEM_E(C!)$.

4) *Rules applied as-long-as-possible* (denoted $R!$) for a given rule set R constitutes a special type of reduced graphs control condition.

5) A TU can be a control condition as it defines a binary relation on graphs.

As a generative structure, a TU over a graph transformation approach $\mathcal{A}$ is a 5-tuple $TU = (IG,L,R,CC,TG)$, where: 1) IG and TG are the initial and terminal graph class expressions, respectively; 2) L is a finite set of identifiers to refer to imported transformation units; 3) R is a finite set of labelled rules ($R \subseteq \mathcal{R}$), whose identifiers are not in L; and 4) CC is a control condition.

A TU has a graph-based interleaving operational semantics, denoted as $SEM(TU)$, implemented by: 1) a function $SEM : L \rightarrow 2^{\mathcal{G} \times \mathcal{G}}$ capable of assigning a set of graphs to each identifier associated with a TU, either a graph transformation rule or an imported transformation unit, 2) a set of graphs resulting from applying the rules from R. The overall semantics $SEM(TU)$ of a TU is also a binary relation on graphs having pairs (G,G') where G and G' are compliant with the initial and terminal graph class expressions, respectively, and the control condition CC.

3.2 Transformation Modules

A transformation module (Ehrig et al., 1999) is useful in the specification of very large systems to encapsulate a set of transformation units, making some hidden from the outside and some visible through an export interface. Formally, it is defined over a graph transformation approach $\mathcal{A}$[9] as a triple $MOD = (IMPORT,BODY,EXPORT)$ where: 1) $IMPORT$ is a set of *imported names* referring to external TUs; 2) $BODY$ and is a finite set of local named TUs, each of which can use TUs from sets $BODY$ and $IMPORT$ only, and named rules over some set L of names; and 3) $EXPORT$ is a set of exported names referring to TUs in $BODY$ and/or $IMPORT$. The following restrictions apply: 1) Imported names cannot be reused for a local rule or TU (i.e., $L \cap IMPORT = \phi$); and 2)

[9]The set of all transformation modules over a graph transformation approach $\mathcal{A}$ is denoted $\tau_{\mathcal{A}}$

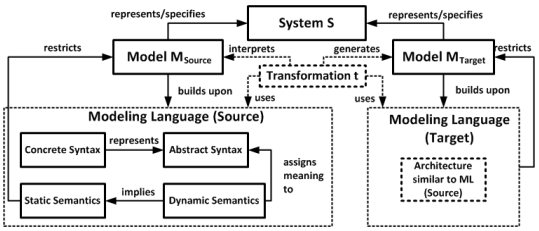

Figure 4: Modeling and Model Transformation Architecture.

Local TUs use only names that are imported or represent local TUs to refer to imported TUs.

The semantics of a module over a graph transformation approach $\mathcal{A}$, denoted as $SEM_{E_{mod}}(MOD)$, defines an environment E_{mod} that associates binary relations on graphs to each imported name, each local rule, each locally defined TU, and each exported name in the $EXPORT$ set. The export semantics of a module is obtained as the restriction of E_{mod} to names in $EXPORT$ only.

4 GRAPH-BASED MODEL TRANSFORMATION

The suitable use of graphs and graph transformations for representing models and model transformations could potentially provide the tools to define domain modeling and transformations languages with very precise syntax and semantics.

4.1 Model Transformations

Reliable transformations on language-based models have to correctly interpret the syntax and semantics of the modeling language(s) defining the source models, and at the same time, correctly reproduce the syntax and semantics of the modeling language(s). During software development, the views of the system, represented by models, can change (i.e., model-to-model transformations), and models are refined to lower abstraction levels to generate system implementations. The following model, adapted from (Metzger, 2005), provides a unifying and systemic understanding of these multiple options occurring in MDE development, which would include graph languages: 1) A modeling language (ML) is a formalism which precisely defines the notation and meaning of a model (fig. 4); 2) The ML's notation consists of concrete and abstract syntax; 3) The ML's meaning consists of the static and dynamic semantics.

Using the conceptual model of fig. 4, one can define a model transformation as a mapping:

$$f : M_{Source}(S)|_{ML_{Source}} \rightarrow M_{Target}(S)|_{ML_{Target}}$$

where ML_{Source} and ML_{Target} are the modeling languages defining the source $ML_{Source}(S)$ and target $M_{Target}(S)$ models of the system S, respectively.

Visualizing and interpreting some of the existing relationships and elements in fig.4, one can provide a brief taxonomy of model transformations, based on the more relevant properties, as follows:

1) *Directionality:* a unidirectional transformation only interprets its source and reproduces its target models. A bidirectional transformation can also switch interpretation to target and generation to source models.

2) *Degree of Evolution:* a horizontal transformation basically generates an equivalent view of the system at the same abstraction level, often using the same source and target modeling languages. A vertical transformation generally uses different modeling languages for source and target models, for example in code generation, and reduces the abstraction level.

3) *Implementation Type:* a transformation can be implemented using a declarative approach based on rules and pre/post-conditions, or an operational approach which explicitly specifies the sequence of actions to transform a model.

4) *Atomicity:* a transformation can be executed as a single non-decomposable step or as multiple steps allowing for partial rollbacks.

5) *Degree of Automation:* the configuration and execution of a transformation can be completely automated by a software tool, or it can be partially automated, requiring user intervention to complete some manual steps.

Other classifications of transformations exist (Czarnecki and Helsen, 2006; Metzger, 2005; Mens and Gorp, 2006), but the properties considered tend to be refinements of the five above.

4.2 Model Transformations using Graphs

Models in MDE routinely express complex structural and dynamic aspects of a system by using fairly intuitive visual languages. If one formalizes syntactic and semantic aspects of these languages using graphs, then one can make use of graph grammars and graph transformations to specify precisely how these models should be built and how they should be transformed (Grunske et al., 2005).

Graph transformations, with their defined graph types and rule sets, provide a solid foundation for reasoning about structural, semantic and operational aspects of model transformations in the following roles:

- as a *semantic domain* that directly provides a generic specification language and semantic model for very high level definitions of application domains, their abstractions and the transformations of those abstractions to support software development activities such as the specification of functional requirements and description of architectural changes (Buchmann et al., 2008; Grunske et al., 2005).
- as a *meta-language* to be used in the formal specification of the syntax, semantic and manipulation rules of the DSLs and model transformation languages (MTLs) defining source/target models and model transformations, respectively (Karsai et al., 2003; de Lara et al., 2004a).

Graph-based Metamodeling. Metamodeling is a widely accepted technique used to define the rules of visual/diagrammatic languages, which provides a flexible configuration environment, especially useful when working with DSLs, and allows the checking of whether a model produced in a specific language is valid or not (Amelunxen et al., 2008; Levendovzky et al., 2009). The construction of a metamodel requires two basic elements: 1) a meta-language capable of describing the metamodel's structure and relations between modeling items (e.g., class diagrams, graphs) and 2) an instantiation relationship indicating how model instances will be generated as a result of using the metamodel.

In the context of graphs one can formally think of metamodels as sets of typed graphs and of models as sets of instance graphs obtained from enacting type-instance mappings from the former sets. Furthermore, given a model of a certain type (e.g., diagrams, object structure, architectures), one can specify its transformations (e.g., visual/notational representation changes, functional requirements, architectural changes) as graph transformation rules over the typed graphs of the associated metamodel. These graph-based transformations are sufficiently generic to: 1) specify syntactic changes when using different source and target modeling languages and 2) define semantic mappings when the target language is used to define a semantic domain for the source language.

5 OPEN PROBLEMS AND CONCLUSIONS

Research on the application of graph grammars and graph transformations to model transformations is currently fairly active, showing two main trends: 1)

mostly theoretical studies characterized by increasingly abstract grammar-based formalisms employed in analyzing and specifying high level properties of model transformations, and 2) software engineering perspectives interested in automating the implementation of model transformations occurring in specific domains using graph-based specifications. There is a noticeable gap between both trends that gives way to an interesting set of open problems, among them: 1) Specialized graph structures that perform well in the specification and implementation of graph- and metamodel-based model transformations would make the declarative approach suitable for building complex model transformations; 2) Most research prototyping tools are often cumbersome to use, offer solutions for very narrow paradigms and incomplete support for systems integration and management. Making graph transformations a solid tool in the context of MDE requires further research into the requirements that industrial strength graph-based tools should have, given the foreseeable nature of applications in engineering and business.

In general, this work has found that there is a large gap between a theoretical body of numerous graph grammar and graph transformation approaches and its application to practical problems in software engineering. Their use in formalizing syntactic and semantic aspects of software artifacts still remains far removed from proven software engineering methodologies and tools that have produced reliable, efficient, user-friendly and maintainable applications. The graph transformation research community has recently become aware of these deficiencies and has responded with studies placing the graph transformation in a more systemic context of transformation units and systems, although with a marked theoretical flavour.

REFERENCES

Amelunxen, C., Legros, E., Schürr, A., and Stürmer, I. (2008). Checking and enforcement of modeling guidelines with graph transformations. In *AGTIVE'07, 3rd International Workshop on Application of Graph Transformations with Industrial Relevance*, volume 5088 of *LNCS*, pages 241–255. Springer.

Andries, M., Engels, G., Habel, A., Hoffmann, B., Kreowski, H.-J., Kuske, S., Plump, D., Schürr, A., and Taentzer, G. (1999). Graph transformation for specification and programming. *Science of Computer Programming*, 34:1–54.

Balasubramanian, D., Narayanan, A., van Buskirk, C., and Karsai, G. (2006). The graph rewriting and transformation language: GReAT. In *GraBats 2006, 3rd International Workshop on Graph Based Tools*, volume 1 of *Electronic Communications of the EASST*.

Balogh, A. and Varró, D. (2006). Advanced model transformation language constructs in the VIATRA2 framework. In *SAC'06, 2006 ACM Symposium on Applied Computing*, New York, USA.

Bézivin, J. (2004). In search of a basic principle for model driven engineering. *UPGRADE -The European Journal for the Informatics Professional*, 2:21–24.

Boyd, M. and McBrien, P. (2005). Comparing and transforming between data models via an intermediate hypergraph data model. *Data Semantics*, 3730:69–109.

Buchmann, T., Dotor, A., Uhrig, S., and Westfechtel, B. (2008). Model-driven software development with graph transformations: A comparative case study. In *AGTIVE '07, 3rd International Workshop on Application of Graph Transformations with Industrial Relevance*, volume 5088 of *LNCS*, pages 273–280. Springer.

Courcelle, B. (1987). An axiomatic definition of context-free rewriting and its application to NLC graph grammars. *Theoretical Computer Science*, 55(2-3):141–181.

Czarnecki, K., Foster, J., Hu, Z., Lämmel, R., Schürr, A., and Terwilliger, J. (2009). Bidirectional transformations: A cross-discipline perspective. In *ICMT2009, Theory and Practice of Model Transformations: Second International Conference*, volume 32 of *LNCS*, pages 260–283. Springer.

Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645.

de Lara, J., Guerra, E., and Vangheluwe, H. (2004a). Metamodeling, graph transformation and model checking for the analysis of hybrid systems. In *AGTIVE '03, 2nd International Workshop on Application of Graph Transformations with Industrial Relevance*, volume 3062 of *LNCS*, pages 292–298. Springer.

de Lara, J., Vangheluwe, H., and Fonseca, M. (2004b). Meta-modeling and graph grammars for multi-paradigm modelling in AToM^3. *Journal of Software and Systems Modeling*, 3(3):194–209.

Dean, T. and Cordy, J. (1995). A syntactic theory of software architecture. *IEEE Transactions on Software Engineering*, 21(4):302–313.

Drewes, F., Hofmann, B., and Minas, M. (2008). Adaptive star grammars for graph models. In *ICGT 2008, 4th International Conference on Graph Transformations*, volume 5214 of *LNCS*, pages 442–457. Springer.

Ehrig, H. (1979). Introduction to the algebraic theory of graph grammars (a survey). In *International Workshop on Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *LNCS*, pages 1–69. Springer.

Ehrig, H., Ehrig, K., Prange, U., , and Taentzer, G. (2006). *Fundamentals of Algebraic Graph Transformation*. EATCS. Springer.

Ehrig, H., Engels, G., Kreowski, H., and G.Rozenberg, editors (1999). *Handbook of Graph Grammars and Com-*

puting by Graph Transformation, volume 2. World Scientific Publishing.

Ehrig, H., Habel, A., Kreowski, H.-J., and Parisi-Presicce, F. (1991). From graph grammars to high level replacement systems. In *4th International Workshop on Graph Grammars and their Application to Computer Science*, volume 532 of *LNCS*, pages 269–291. Springer.

Grunske, L., Geiger, L., Zündorf, A., van Eetvelde, N., van Gorp, P., and Varró, D. (2005). Using graph transformation for practical model-driven software engineering. In *Model-Driven Software Development*, pages 91–117. Springer.

Habel, A. (1992). *Hyperedge Replacement: Grammars and Languages*, volume 643 of *LNCS*. Springer.

Hermann, F., Ehrig, H., Orejas, F., Czarnecki, K., Diskin, Z., Xiong, Y., Gottmann, S., and Engel, T. (2013). Model synchronization based on triple graph grammars: correctness, completeness and invertibility. *Software and Systems Modeling*.

Himsolt, M. (1991). Graph^{Ed}: An interactive tool for building graph grammars. In *4th International Workshop on Graph Grammars and their Application to Computer Science*, volume 532 of *LNCS*, pages 61–65. Springer.

Karsai, G. (2010). Lessons learned from building a graph transformation system. In *Graph Transformations and Model-Driven Engineering*, volume 5765 of *LNCS*, pages 202–223. Springer.

Karsai, G., Agrawal, A., Shi, F., and Sprinkle, J. (2003). On the use of graph transformation in the formal specification of model interpreters. *Journal of Universal Computer Science*, 9(11):1296–1321.

Kent, S. (2002). Model driven engineering. In *3rd International Conference on Integrated Formal Methods*, volume 2335 of *LNCS*, pages 286–298. Springer.

Kreowski, H.-J., Kuske, S., and von Totth, C. (2010). Stepping from graph transformations units to model transformation units. In *GraMot 2010, International Colloquium on Graph and Model Transformation*, volume 30 of *Electronic Communications of the EASST*.

Kühne, T. (2006). Matters of (meta-) modeling. *Journal of Software and Systems Modeling*, 5(4):369–385.

Legros, E., Schäfer, W., Schürr, A., and Stürmer, I. (2011). MATE - a model analysis and transformation environment for MATLAB Simulink. In *International Dagstuhl Workshop on Model-Based Engineering of Embedded Real-Time Systems*, volume 6100 of *LNCS*, pages 323–328. Springer.

Levendovzky, T., Lengyel, L., and Mézáros, T. (2009). Supporting domain-specific model patterns with metamodeling. *Journal of Software and Systems Modeling*, 8(4):501–520.

Mens, T. and Gorp, P. V. (2006). A taxonomy of model transformation. In *GraMoT 2005, International Workshop on Graph and Model Transformation*, volume 152 of *Electronic Notes in Theoretical Computer Science*, pages 125–142. Elsevier.

Metzger, A. (2005). A systematic look at model transformations. In *Model-Driven Software Development*, pages 19–33. Springer.

Nagl, M. (1987). Set theoretic approaches to graph grammars. In *3rd International Workshop on Graph Grammars and their Application to Computer Science*, volume 291 of *LNCS*, pages 41–54. Springer.

Nagl, M. and Schürr, A. (1991). A specification environment for graph grammars. In *4th International Workshop on Graph Grammars and their Application to Computer Science*, volume 532 of *LNCS*, pages 599–609. Springer.

Neema, S. and Karsai, G. (2006). Software for automotive systems: Model-integrated computing. In *Automotive Software - Connected Services in Mobile Networks*, volume 4147 of *LNCS*, pages 116–136. Springer.

Nickel, U., Niere, J., and Zündorf, A. (2000). The FUJABA environment. In *ICSE'00, 22nd International Conference on Software Engineering*, pages 742–745, New York, NY, USA. ACM.

Object Management Group (2011). Meta Object Facility (MOF) 2.0 Query / View / Transformation Specification, version 1.1, January 2011. version 1.1.

Prieto-Díaz, R. (1990). Domain analysis: An introduction. *ACM SIGSOFT Software Engineering Notes*, 15(2):47–54.

Rozenberg, G., editor (1997). *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. World Scientific Publishing.

Schürr, A. (1995). Specification of graph translators with triple graph grammars. In *20th International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 903 of *LNCS*, pages 151–163. Springer.

Schürr, A., Winter, A. J., and Zündorf, A. (1995). Graph grammar engineering with PROGRES. In *ESEC '95, 5th European Software Engineering Conference*, volume 989 of *LNCS*, pages 219–234. Springer.

Stevens, P. (2010). Bidirectional model transformations in QVT: Semantic issues and open questions", JOURNAL = "Journal of Software and Systems Modeling. 9(1):7–20.

Taentzer, G. (2004). AGG: A graph transformation environment for modeling and validation of software. In *AGTIVE 2003, 2nd International Symposium on Applications of Graph Transformations with Industrial Relevance*, volume 3062 of *LNCS*, pages 446–453. Springer.

Weisemöller, I., Klar, F., and Schürr, A. (2011). Development of tool extensions with MOFLON. In *International Dagstuhl Workshop on Model-Based Engineering of Embedded Real-Time Systems*, volume 6100 of *LNCS*, pages 337–343. Springer.