Journal of Zhejiang University SCIENCE A
ISSN 1673-565X (Print); ISSN 1862-1775 (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

Towards automated software model checking using graph transformation systems and Bogor

Vahid RAFE†, Adel T. RAHMANI

(*Department of Computer Engineering, Iran University of Science and Technology, Tehran, Iran*)

†E-mail: rafe@iust.ac.ir

Received June 2, 2008; Revision accepted June 9, 2009; Crosschecked Apr. 28, 2009

Abstract: Graph transformation systems have become a general formal modeling language to describe many models in software development process. Behavioral modeling of dynamic systems and model-to-model transformations are only a few examples in which graphs have been used to software development. But even the perfect graph transformation system must be equipped with automated analysis capabilities to let users understand whether such a formal specification fulfills their requirements. In this paper, we present a new solution to verify graph transformation systems using the Bogor model checker. The attributed graph grammars (AGG)-like graph transformation systems are translated to Bandera intermediate representation (BIR), the input language of Bogor, and Bogor verifies the model against some interesting properties defined by combining linear temporal logic (LTL) and special-purpose graph rules. Experimental results are encouraging, showing that in most cases our solution improves existing approaches in terms of both performance and expressiveness.

Key words: Graph transformation, Verification, Bogor, Attributed graph grammars (AGG), Software model checking
doi:10.1631/jzus.A0820415 **Document code:** A **CLC number:** TP31

INTRODUCTION

Today, software development is a complex task, because most of the modern software systems are large in size and involve different and complex artifacts (e.g., distributed, real time and embedded systems). Hence, to overcome these complexities it is important to undertake software system modeling and design before implementation. However, to have accurate models, using a proper language for modeling is mandatory and formal methods have proven to be a crucial solution for automated software engineering.

Graphs and diagrams are a very useful means to describe complex structures and systems and to model concepts and ideas in a direct and intuitive way. For example, the structure of an object-oriented system or the execution flow of a program can be considered as a graph. Regardless of the actual process for modeling, a designer always will end up with some diagrams or in fact, annotated boxes and lines. These annotated boxes and lines can easily be conceived as annotated directed/undirected graphs. Graph transformation (Ehrig *et al*., 1999; Baresi and Heckel, 2002) is a popular formalism as a well-known and expressive specification language (e.g., to formally capture software requirements). Therefore, using graphs and graph transformation systems as a formal background for software modeling is a natural choice. Software architectures, component diagrams, and state charts are only a few well-known examples in which graphs have been used to software development process (Baresi *et al*., 2008). These models and many others can easily be described by means of suitable graph transformation systems to formalize their syntax and define the formal semantics of used notations (Kuske, 2001; Baresi *et al*., 2003).

Rule-based features of graph transformation systems can play an important role in modeling of complex and large systems. Modeling is often not enough because designers want to be able to 'discover' whether stated requirements (such as the

absence of deadlocks, safety and liveness properties) are fulfilled in the system model. This is why even the perfect graph transformation system must be complemented with automated analysis capabilities to let users reason on it and understand whether such a formal specification fulfills their requirements, and model checking has proven to be a viable solution for this purpose.

Having completed our previous preliminary research (Baresi *et al*., 2008), we present in this paper a novel solution using Bogor (Robby *et al*., 2003) to model check attributed graph grammars (AGG)-like (www.tfs.cs.tu-berlin.de/agg/) graph transformation systems. The key characteristics that make our solution highlighted from the existing proposals are: (1) supporting models of complex types for verification, and thus using attributed graphs is mandatory, (2) using the available data structures in Bogor to support dynamic systems, that is, systems with dynamic node creation/deletion, (3) supporting layered graph transformation systems for verification, and (4) using graph transformation systems including type graphs that support meta modeling techniques.

In our approach, graph transformation systems are translated to Bandera intermediate language (BIR), the input language of the model checker called Bogor (Corbett *et al*., 2000), while properties are defined by combining linear temporal logic (LTL) and special-purpose graph rules. Then Bogor generates the transition system and performs the verification (via temporal logics interpreted on the transition system). If the result of the verification is negative, Bogor will generate a counter example to show it to the designers.

RELATED WORKS

There are different approaches and tools for software model checking. Model checkers [like Murφ (http://verify.stanford.edu/dill/murphi.html), SAL (Bensalem *et al*., 2000), SPIN (Holzmann, 1997)] are used to verify finite state systems automatically. As it is hard to use the low-level input language of model checker tools for modeling systems directly, many transformation techniques have been developed to translate high-level modeling languages like unified modeling language (UML) based models into the input languages of model checker tools (e.g., Latella *et al*., 1999; Paltor and Lilius, 1999; Compton *et al*., 2000). But the problem is that UML is not formal, hence automatic and precise translation of UML diagrams to the input languages of the model checkers is not straightforward. Therefore, in our proposal we are dealing with graph transformation systems instead of UML or other informal modeling languages.

The theoretical foundations for the verification of graph transformation systems through model checking have been studied by Heckel (1998). The author suggested that graphs should be interpreted as states. Then transformation rules can be considered as transitions between states. This idea is used by both GROOVE (Rensink, 2004) and CheckVML (Schmidt and Varró, 2003). Also, we have exploited this idea in our solution.

GROOVE applies adapted model checking algorithms on graph transformation systems by considering graphs as states and applications of transformation rules as transition between them. Properties are defined by means of a combination of graph rules and computational tree logic (CTL) expressions containing rule names as atoms. Since GROOVE cannot support type graphs (in contrast to our proposal), the model checking of real models becomes complex or infeasible. Later, Kastenberg (2005) proposed a solution to extend GROOVE with attributed graphs; however, it supports attributed graphs partially and in a non-native way. Kastenberg suggested in his proposal that attributes and values are kept separate, which makes it difficult for users to work with the graph transformation systems, and performance decreases as soon as the size of graphs increases. Furthermore, GROOVE cannot support layered graph transformation systems, while our approach not only supports them, but also supports attributed and type graph transformation systems.

CheckVML (Schmidt and Varró, 2003) exploits SPIN (Holzmann, 1997) to model check graph transformation systems. A graph transformation system including a type graph, rules and a host graph is fed to CheckVML as input. Then it produces an equivalent model in Promela, which is SPIN's input language. In CheckVML, properties are defined by means of a combination of graph rules and LTL. CheckVML can support only safety and reachability properties (Schmidt, 2004), while our approach can support a wide range of properties like safety, liveness,

reachability and deadlock freeness. In the case of dynamic systems, CheckVML has insufficient performance (Rensink *et al*., 2004) (in contrast to our approach) owing to its use of a fixed two-dimensional 0-1 array as a data structure to store graphs in Promela. Moreover, Gyapay *et al*.(2004) proposed an approach to solving the optimization problems in graph transformation systems with time using CheckVML, but the dynamic creation and deletion of nodes and edges is bounded a priori, which is a major restriction in comparison with our work. This work also cannot support layered graph transformation systems.

Grunske *et al*.(2008) investigated how graph transformation systems can improve the specification of the abstract syntax of a visual modeling language. They defined the abstract syntax of behavior trees (BTs). BT is a graphical modeling language for specifying functional requirements. Then, the authors defined a translation schema to translate BTs to the input language of the SAL model checker. In contrast to our approach, this work describes a special kind of graph transformation systems restricted to BTs, while we investigate on the general graph transformation systems and layered graphs.

Baldan and König (2002) introduced a different theoretical proposal aiming at verifying a special class of hypergraph transformation systems (and not typical graph transformation systems) by means of a special class of Petri nets which is a static analysis technique. Later, Baldan *et al*.(2004a) extended this proposal by providing a precise (McMillan style) unfolding strategy. In comparison with our work, they concentrate on hypergraphs, without considering typical graph transformation systems or layered graphs.

Dotti *et al*.(2003) suggested object-based graph grammars to model object-oriented systems. They also described an approach to translating object-based graphs into Promela. The authors restricted the structure of graph transformation rules to only model the message exchange mechanism in the object-oriented systems. Although the used representation in terms of Promela constructs can support only a restricted system, the equivalent system in Promela might increase the runtime performance. Ferreira *et al*.(2007) extended the work presented in this approach to verify concurrent object-oriented systems. They used a special class of object-oriented graph grammars and defined a translation from such specifications to Promela. According to their claim, they were not dealing with object creation or deletion. Hence, it is not suitable for dynamic systems. In contrast to these approaches, our proposal has no such restriction for the shape of the graphs and can support dynamic systems efficiently. In addition, these proposals cannot support layered systems.

Baresi and Spoletini (2006) described a proposal to verify graph transformation systems by means of Alloy (Jackson, 2006) based on first order logic. They defined an approach to translating AGG transformation systems in Alloy. Due to the nature of Alloy, the presented proposal can support only bounded and a priori limited domains. Finally, Liu *et al*.(2007) presented a constructive method to check if a transformation satisfies a set of correctness constraints and proposed an algorithm based on critical pair analysis to automatically prove whether a transformation rule satisfies a transformation construct or not.

BACKGROUND

In this section we briefly introduce the required background, i.e., graph transformation systems and Bogor.

Graph transformation system

The mathematical foundation of graph transformation systems returns to 40 years ago in response to shortcomings in the expressiveness of classical approaches to rewriting (e.g., Chomsky grammars) to deal with nonlinear grammars (Ehrig *et al*., 1999). In this subsection, we describe graph transformation briefly, as a modeling means. For more information about the theoretical background and semantics of graph transformation, interested readers can refer to (Ehrig *et al*., 1999; Baresi and Heckel, 2002).

Definition 1 (Attributed type graph transformation) An attributed type graph transformation system is a triple: $AGT=(TG, HG, R)$, where TG is the type graph, HG is the host graph and R is the set of rules.

Definition 2 (Type graph) Let TG_N be a set of node types and TG_E be a set of edge types. A type graph TG is a tuple: $TG=(TG_N, TG_E, src, trg)$, with two functions src: $TG_E \rightarrow TG_N$ and trg: $TG_E \rightarrow TG_N$. These functions assign to each edge a source and a target node. Each node type NT in TG_N is a triple: $NT=(Mult,$

Attr, *Out*). *Mult*, being a pair, is the multiplicity of the node: *Mult*=(*min*, *max*), which specifies lower and upper bounds for the number of nodes of type *NT* in the host graphs. *Attr*, being a tuple, is the set of its attributes: *Attr*=(*SN*, *V*, *X*, *sort*), where *SN* is a set of sort names (or type names), *V* is a set of attribute values including a subset $X \subseteq V$ of variable names, and a function *sort*: $V \to SN$ associating every value and variable with a sort (type). *Out* is the set of its outgoing edges (or associations) with corresponding multiplicity and destination nodes. More precisely, each outgoing edge *OE* in *Out* is a pair: *OE*=(*Card*, *Dest*), where *Card* is formally defined by two pairs of functions *minSrc*, *minTrg*: $TG_E \to \mathbb{N}_+ \cup \{0\}$ and *maxSrc*, *maxTrg*: $TG_E \to \mathbb{N}_+ \cup \{*\}$ with $minSrc(OE) \le maxSrc(OE)$ and $minTrg(OE) \le maxTrg(OE)$, and *Dest* is the destination node of the edge.

Definition 3 (Host graph) A host graph *HG*, also called 'instance graph' over *TG*, is a graph equipped with a graph morphism $type_G: HG \to TG$ that assigns a type to every node and edge in *HG*.

Definition 4 (Graph rules) In this study, we follow the algebraic double pushout (DPO) approach to graph transformation as first introduced by Ehrig *et al*.(1973) for untyped graphs. A graph transformation rule *P* over an attributed type graph *TG* is given by $P = (L \xleftarrow{l} K \xrightarrow{r} R, type, NAC)$, where $L \xleftarrow{l} K \xrightarrow{r} R$ is a rule span with injective graph morphisms *l*, *r* and graphs *L* (left hand side or LHS), *K* (gluing graph) and *R* (right hand side or RHS) typed over *TG*, $type=(type_L: L \to TG, type_K: K \to TG, type_R: R \to TG)$ is a triple of morphisms, and *NAC* is a set of triples $nac=(N, n, type_N)$ with *N* being a graph, $n: L \to N$ a graph morphism, and $type_N: N \to TG$ a morphism.

The application of a rule to a host graph *H* replaces a matching of the LHS in *H* by an image of the RHS. This is performed by (1) finding a matching of the LHS in *H*, (2) checking the *NAC* (which prevents the existence of certain nodes and edges), (3) deleting a part of the host graph (that can be mapped to LHS but not to RHS) producing the context model, and (4) connecting the context model with a matching of the RHS by adding new nodes and edges (that can be mapped to the RHS but not to the LHS) and resulting in a new model *H'*.

By recursively applying all enabled graph transformation rules to the host graph, a transition system can be generated. Transition systems are frequently used to represent the behavior semantics of software systems. In the case of graph transition systems, one considers graphs as representations of system states. If the resulting state space of the graph transition system is finite, we can easily check different properties (e.g., reachability, safety, and liveness), even for unrestricted forms of graph transformation systems, by searching the state space.

Bogor

In this paper we use Bogor (Robby *et al*., 2003) to verify graph transformation systems. Bogor is an extensible software model checking tool developed at Kansas State University. Bogor has novel capabilities for checking different properties on a variety of modern software artifacts, and its internal, modular architecture lets domain experts extend it to provide a domain-specific model checker.

Bogor's input language, Bandera intermediate representation (BIR), provides the different instructions that are supported by the modeling languages of verification tools (e.g., SPIN). These instructions include primitive and non-primitive data types, like arrays and records. Bogor also supports advanced features, like function pointers, dynamic creation of threads and objects, automatic memory management (garbage collector), and generic data types. Control-flow and actions in BIR are expressed in a guarded command format: 'guard expressions' are devoted to check expressions, while 'actions' (commands) change the value of variables in the system.

For example, the BIR model of Fig.1 comprises one thread (i.e., MAIN) and a global integer variable *x* whose initial value is set to 100. Thread MAIN defines a simple loop. For example, the first guard checks whether *x* is even and, if it is the case, computes its new value (*x*/2). Notice that both guards are evaluated simultaneously.

When there is more than one true guard, Bogor chooses non-deterministically one of them; when all the guards are false, Bogor detects a deadlock. In the BIR example of Fig.1, instruction 'goto loc0' at the end of the code shows that loc0 must be reached again, and thus it causes a loop.

```
system example {
   int x:=100;
   main thread MAIN() {
      loc loc0;
        when x%2==0 do  {x:=x/2;}
          goto loc0;
          when x%2!=0 do  {x:=3*x+1;}
      goto loc0;
   }
}
```

Fig.1 Example BIR model

While the BIR code is executed, Bogor creates an automaton. In the generated automaton, states show a configuration of the system based on the values of the variables in the code. Hence, in the example, when the value of variable x is changed, Bogor adds a new state to the automaton and continues until no new state is found; in this case the execution is terminated and the automaton represents all reachable program states.

Bogor also has a module for checking different properties expressed in LTL (Bogor extensions for LTL checking, www.projects.cis.ksu.edu/projects/gudangbogor/). Each property must be stated in BIR as a function *fun*. Then Bogor checks the function. Fig.2 shows two examples (Baresi *et al*., 2008). The first LTL formulae, $G((x>0)\rightarrow F(x<0))$, is true if in each execution of the automaton there is a state where $x>0$, and eventually there must be a state in the postfix of that path where $x<0$. Based on the BIR model of Fig.1, this property function is not satisfied.

```
fun fail() returns boolean=
 LTL.temporalProperty(
   Property.createObservableDictionary(
    Property.createObservableKey("p", x>0),
    Property.createObservableKey("q", x<0)
   ),
 LTL.always(LTL.implication(LTL.prop("p"),
  LTL.eventually(LTL.prop("q"))))
 );

fun hold() returns boolean=
 LTL.temporalProperty(
   Property.createObservableDictionary(
    Property.createObservableKey("p", x>0),
    Property.createObservableKey("q", x<=100)
   ),
 LTL.always(LTL.conjunction(LTL.prop("p"),
  LTL.prop("q")))
 );
```

Fig.2 Example property functions (Baresi *et al*., 2008)

In Fig.2, "p" and "q" correspond to the proposition "$x>0$" and "$x<0$", respectively. The second LTL formula, $G(x>0 \wedge x\leq 100)$, will be true if in each execution of the automaton and in all states, $0<x\leq 100$. The example model of Fig.1 satisfies this property. In this example, "LTL.always" and "LTL.conjunction" are equivalents for operators G and $\wedge$, respectively.

ENCODING GRAPH TRANSFORMATION TO BIR

We can summarize the main steps of the proposed solution as follows: (1) we define the required data structures in BIR to translate graph transformation systems by using the type graph; (2) we initialize the data structures in BIR using the host graph in the graph transformation system; (3) we translate the rules to BIR: the LHS of each rule is encoded as one or more guarded commands in BIR, while the RHS as actions (the body of the guards). These steps are explained through the well-known Dining Philosopher example, which has been modeled in (Schmidt, 2004) as a graph transformation system. We choose this example because it is a simple model, able to illustrate our approach without any complexity. In addition, it is also a usual benchmark for evaluating the performance of model checkers. In the 'VALIDATION' section, we will show some experimental results for more realistic models.

Fig.3a shows the type graph of the example. This type graph includes two types of nodes, each having its own attributes, multiplicities, and associations. Rules of Fig.4 describe all the actions that a philosopher can carry out on the model. For example, rule "GetHungry" defines an event in which a philosopher with the status "thinking" (LHS of the rule) gets hungry (RHS of the rule). Fig.3b shows the example host graph. It shows the starting configuration of the system. In the example host graph, three philosophers are thinking.

At first, the type graph must be encoded. In this step the data structure in BIR is defined. Each node in the type graph is encoded to a record in BIR. This record contains all the attributes and associations of the node. Then a further record must be defined to store the whole type graph. These steps can be listed formally as follows: $\forall n\in NT$, a record is defined in

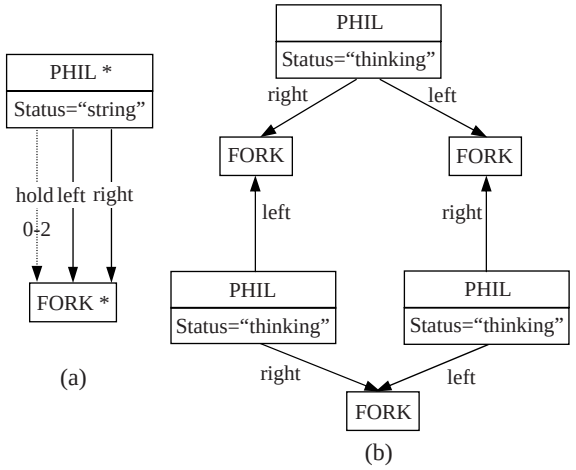

Fig.3 Example type graph (a) and host graph (b)

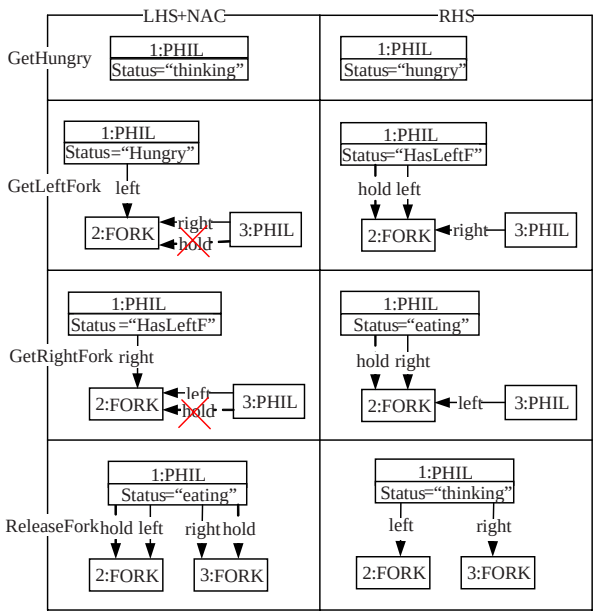

Fig.4 Example transformation rules

BIR with the following fields:

$\forall a \in n.Attr$, a field is added to the record with the same type as a;

$\forall o \in n.Out$, if $o.Card.maxTrg(o)=1$, an element of type $o.Dest$ is added as field of the record; if $o.Card.maxTrg(o)>1$, an array of elements of type $o.Dest$ is added as field of the record.

In general, each node in the type graph is a record and each attribute in the node type is encoded to an equivalent field of the record. Associations in the type graph, along with their multiplicities, are encoded to a field of the type of the destination node with a multiplicity equal to 1, or an array otherwise.

As an example, Fig.5 shows the PHIL node of Fig.3 encoded to an equivalent record in BIR. Its first field, "status", corresponds to the attribute of the node, while the other fields represent the associations.

```
record PHIL {
    string status;
    FORK left;
    FORK right;
    FORK[] hold;
  }
```

Fig.5 The BIR record equivalent to type PHIL

In Fig.5, "left" and "right" are single elements of type "FORK", while "hold" is an array of "FORK", since a philosopher can hold more than one FORK (at most two in this case).

A further record must be defined in BIR to store the whole type graph. This further record represents all the nodes that a type graph has. For example, the record of Fig.6 shows the type graph of Fig.3. In this example, all the fields have been encoded to arrays because the multiplicity of these nodes is *.

```
record graph {
    PHIL[] PHILs;
    FORK[] FORKs;
  }
```

Fig.6 Graph data structure in BIR

After defining the data structures, we define a main thread in BIR. This thread contains all the behavior of the graph transformation. The main thread consists of different locations (loc). In the first location (loc0), the type graph is instantiated and the host graph is implemented. At first, a variable of type "graph" must be defined. Then, based on the existing nodes in the host graph the size of contained arrays must be determined.

Fig.7 presents a portion of generated BIR code for loc0 in the main thread, where "instance" is a variable of type "graph". As an example, instruction instance.PHILs:=new PHIL[3] allocates the necessary space for the philosophers in the host graph (three in this example), and instance.PHILs[0].status:="thinking" sets the value of field status for the first philosopher in the model to "thinking".

```
loc loc0:
    do {
        instance:=new graph;
        instance.PHILs:=new PHIL[3];
        instance.FORKs:=new FORKs[3];
        instance.PHILs[0]:=new PHIL;
        instance.FORKs[0]:=new FORK;
        instance.PHILs[0].status:="thinking";
        instance.PHILs[0].right:=instance.FORKs[0];
    }
```

Fig.7 A portion of loc0 in the main thread

Up to now, we have described our approach to translating the type and host graph to BIR. After the first location was generated in BIR, we should transform the rules. The encoding of rules can be divided into two different sub-problems: "matching" and "acting", i.e., the LHS (and the NAC, if it exists) and the RHS, respectively. The matching routine must be implemented in the second location for non-layered graph transformation systems or in consecutive locations for layered systems (one location per each layer) as guarded commands, while each action is implemented as bodies of the guards.

In the previous work (Baresi *et al*., 2008), we used a different way to implement "matching" and "acting". Therein we found the main components in the LHS (the nodes of the LHS through which all the other nodes in that LHS are reachable), and generated a thread for each RHS. But using threads decreases the performance, because it generates many additional states. In this study, we use another way to translate rules (both LHS and RHS), which leads to a better performance as will be discussed in the next section.

The matching procedure computes all possible combinations for matching between nodes in the LHS of each rule and the nodes in the host graph. Notice that the NAC is handled as the LHS; the only difference is that we treat it as a negative condition. The acting procedure is based on the RHS of the rule.

As an example, consider rule GetHungry of Fig.4. There are three possible matches for the LHS of this rule on the host graph. Fig.8 shows the guarded commands and actions to detect this rule.

These instructions are generated automatically, located in the second location (loc1) in BIR. Bogor evaluates the guards and executes the action associated with a selected true guard.

```
when instance.PHILs[0].status=="thinking" do {
    instance.PHILs[0].status:="hungry";
    }
goto loc1;
when instance.PHILs[1].status=="thinking" do {
    instance.PHILs[1].status:="hungry";
    }
goto loc1;
when instance.PHILs[2].status=="thinking" do {
    instance.PHILs[2].status:="hungry";
    }
goto loc1;
```

Fig.8 Guarded commands to detect rule GetHungry

As an additional example, consider rule GetLeftFork of Fig.4. In the LHS of this rule, there are two nodes of type PHIL and one node of type FORK. Hence, there are six possible combinations for matching of philosophers in the host graph and three different combinations for matching of forks in the host graph. Also, as each philosopher can hold two forks, we should check in which cells of the "hold" array this fork can be stored. Consequently, there are 18×2 (or 36) different matchings for this rule, so 36 different guards and actions must be generated in BIR to support this rule. Fig.9 shows some of the guards and actions. Notice that some of these guards might never be true (because we only use nodes to detect matchings); but to implement the matching procedure as generally as possible, we need to consider all situations. Furthermore, in the preprocessing step where we generate these guards and actions, it is impossible to foresee future changes in the model (host graph).

These instructions check the status of the first philosopher to be "hungry". If its left hand side fork was not held by another philosopher, then he should hold the fork and change the value of his status.

When detecting a matching for the LHS of a rule, we generate the suitable guarded commands (based on the LHS and NACs), and then generate the actions associated with those guards (based on the RHS):

(1) If there are some nodes in the RHS but not in the LHS, these nodes must be set to "active", and instantiated as stated by the rule [The maximum number of dynamic nodes is not always known a priori, and the host graph only gives a lower bound. Hence, our approach gets the maximum numbers of dynamic

```
when instance.PHILs[0].status==“hungry” &&
    instance.PHILs[0].left==instance.FORKs[0] &&
    instance.PHILs[1].right==instance.FORKs[0] &&
    instance.PHILs[1].hold[0]==null do {
          instance.PHILs[0].hold[0]:=instance.FORKs[0];
          instance.PHILs[0].status:=“HasLeftF”;
       }
goto loc1;
when  instance.PHILs[0].status==“hungry” &&
       instance.PHILs[0].left==instance.FORKs[1] &&
       instance.PHILs[1].right==instance.FORKs[1] &&
       instance.PHILs[1].hold[0]==null do {
          instance.PHILs[0].hold[0]:=instance.FORKs[1];
          instance.PHILs[0].status:=“HasLeftF”;
       }
goto loc1;
when  instance.PHILs[0].status==“hungry” &&
       instance.PHILs[0].left==instance.FORKs[0] &&
       instance.PHILs[2].right==instance.FORKs[0] &&
       instance.PHILs[2].hold[0]==null do {
          instance.PHILs[0].hold[0]:=instance.FORKs[0];
          instance.PHILs[0].status:=“HasLeftF”;
       }
goto loc1;
```

Fig.9 A portion of generated guards and actions for rule GetLeftFork

nodes as parameters and considers an attribute named “isactive” to manage them as in (Schmidt, 2004).].

(2) If there are some edges in the RHS but not in the LHS, as to the edge that is a unary association, the variable corresponding to the destination node is assigned to the corresponding variable in the record of the source node and, as to the edge that is stored in an array, an inactive cell in the array is set as in the case of unary associations.

(3) If the RHS does not add nodes or edges to the graph and it only modifies the attributes of a node, the fields corresponding to the attributes in the record of the corresponding variable are changed accordingly.

(4) If there are nodes in the LHS but not in the RHS, the corresponding variables are de-allocated, set as inactive, and the associations that have these nodes as sources or destinations are cleared.

(5) If there are edges in the LHS but not in the RHS, the corresponding variables, i.e., the fields corresponding to the associations in the source nodes, are deleted.

For example, the RHS of the GetLeftFork in Fig.9 sets the “hold” association of the philosopher to the “fork” which is on the left hand side of this philosopher. As this attribute is an array, only one of its cells is used. Additionally, the status of the philosopher is changed to “HasLeftF”.

VALIDATION

In this section, we show the experimental results and we compare them with those of CheckVML and GROOVE on well-known examples to demonstrate the validity of our proposal.

To use the Bogor for checking the properties, we first need a way to define properties as an LTL formula, i.e., the notion of temporal logic that Bogor understands. Before translating the properties into the language that Bogor understands (i.e., property BIR functions), we need to define the LTL formula that will be used to express the properties:

Definition 5 (LTL formula) Let $TS=(S, \rightarrow, h_0)$ be a transition system generated by recursively applying all enabled rules to host graph (h_0). Let $Path(h_0)$ be the set of all linear paths in the transition system starting with state h_0, and let p be some atomic propositions. Then

TS|=$G(p)$ (or $\Box(p)$):

$\Leftrightarrow \forall h_0h_1h_2\ldots \in Path(h_0)$, $\forall k\in\mathbb{N}$, p holds in h_k;

TS|=$F(p)$ (or $\Diamond(p)$):

$\Leftrightarrow \forall h_0h_1h_2\ldots \in Path(h_0)$, $\exists k\in\mathbb{N}$, p holds in h_k.

Note that this is only a subset of LTL. In case of finite paths, k has to be restricted to the length of the path.

Now, we should find a way to define propositions. For example, consider this safety property on the Dining Philosopher example: “a fork may never be held by two different philosophers”, or formally:

$\forall p_1, p_2$: PHIL, $\forall f$:

FORK: $\sim(hold(p_1, f)\wedge hold(p_2, f))$ holds in all states.

“*hold*” in this formula refers to the “*hold*” relation between PHIL and FORK node types of Fig.3. To avoid Bogor-specific knowledge for stating properties, users can use special-purpose graph rules for graphical representation of the properties. We follow both CheckVML and GROOVE to state the properties for checking as the combination of graph transformation rules and LTL operators. Properties are defined by

special transformation rules, where NACs show negative conditions and LHSs positive conditions. We define a property rule to visually describe the characteristics that nodes must (or must not) have. So the RHS is identical to the LHS because it does not change the host graph.

As an example, consider a property rule as shown in Fig.10. We use this property rule to state the mentioned safety property. Its LTL expression is: $G(\neg$HoldTwoForks). "HoldTwoForks" is the name of the rule.

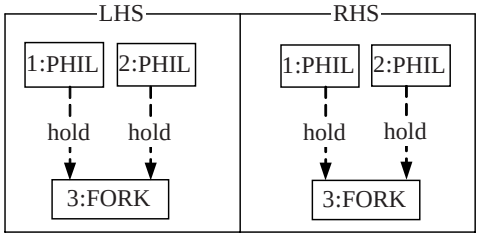

Fig.10 Example property as a rule

To encode properties rules to BIR, we carry out the following steps (Baresi *et al*., 2008):

$\forall n \in$LHS where n is a node type, we consider all the possible variables $v_1, v_2, \ldots, v_k$ of the type corresponding to n in the host graph: $\forall v_i \in \{v_1, v_2, \ldots, v_k\}$ and $\forall attr_j \in v_i$ attributes in LHS$\cup$NAC, we create a proposition $p_i^{attr_j}$ that checks the attribute for v_i.

$\forall a \in$LHS, where a is an association, we consider all the possible variables $v_1, v_2, \ldots, v_h$ of the type corresponding to the source of a: $\forall v_i \in \{v_1, v_2, \ldots, v_h\}$ and $\forall a_j = a \in v_i$ associations in LHS$\cup$NAC, we create a proposition $p_i^{a_j}$ that checks whether the required association exists in v_i.

We create proposition p formed by the disjunction of all p_i generated in the previous steps. In the end, using the LTL operator defined by the user, we make the final LTL expression.

Fig.11 shows how the property is translated to BIR.

Since there are 36 different possible matchings for the nodes in the LHS of this property rule and due to the lack of space, we only consider PHILs[0], PHILs[1] and FORKs[0] to translate a portion of this property in Fig.11 (only two cases). The result of checking this property on the model will be valid.

```
fun Property() returns boolean=
 LTL.temporalProperty(
  Property.createObservableDictionary(
   Proerty.createObservableKey("p1", (instance.PHILs[0].
    hold[0]!=null && instance.PHILs[1].hold[0]!=null &&
    instance.PHILs[0].hold[0]==instance.FORKs[0] &&
    instance.PHILs[1].hold[0]==instance.FORKs[0]) ||
    (instance.PHILs[0].hold[0]!=null &&
    instance.PHILs[1].hold[1]!=null &&
    instance.PHILs[0].hold[0]==instance.FORKs[0] &&
    instance.PHILs[1].hold[1]==instance.FORKs[0]))
  ),
 LTL.always(LTL.negation(LTL.prop("p1")))
);
```

Fig.11 Property in Fig.10 rendered in BIR

As another property example, which causes a deadlock, consider the following liveness property: "if the philosopher p gets hungry, then the status of p must be 'eating' once in the future". This property can be stated formally as the following formula:

$$\forall p\text{: PHIL}, \forall h_0h_1h_2\ldots \in Path(h_0), \exists i \in \mathbb{N}, Hungry(p) \text{ holds in } h_i \rightarrow \exists j \in \mathbb{N}, j>i\text{: } eating(p) \text{ holds in } h_j.$$

Fig.12 describes this property as two different rules. The following LTL expression states this liveness property: G(HungryPhil$\rightarrow F$(EatingPhil)).

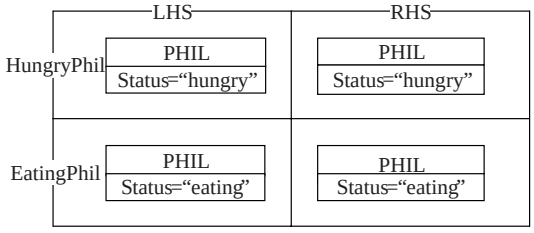

Fig.12 Two example property rules

The LTL expression states that, in every execution if there is a state where a philosopher gets hungry in a path, eventually there must be a state in the following of that path in which the status of the philosopher is "eating". Naturally, the result of the verification is not valid, as a deadlock easily appears in the model by holding just the left side forks by each philosopher.

Another interesting point about Bogor is that it detects deadlocks automatically; i.e., without using any LTL property, it can detect deadlocks on the

model. In a deadlock state no transitions are enabled. Hence, without using the above liveness property, Bogor will detect that there is a deadlock in the model, thus generating a counter example to show it. Checking the counter example, we find out that there is exactly one state in the transition system where all philosophers hold their left side forks and therefore, no transitions (rules) are enabled in that state.

Case studies

To show the performance of our approach and to compare our approach with existing ones, we implemented five different case studies:

(1) The Dining Philosopher problem, introduced in this article.

(2) The Concurrent Append example presented in (Rensink *et al*., 2004). In this example, two, three or four nodes are added concurrently at the end of a linked list. For example, 3:7 in Table 1 shows that three nodes are added to a list with seven nodes in parallel.

(3) The Shopping Cart example presented in (Hausmann *et al*., 2002). In this example the process of purchasing goods by customers in a market has been modeled as a graph transformation system.

(4) The Airport case study introduced in (Baldan *et al*., 2004b). This example models a system representing planes landing and taking off from airports and transports passengers as layered graph transformation systems.

(5) The SmartCar example specified as a service-oriented architecture in (Baresi *et al*., 2006). This example is a highly dynamic and big model. This example contains nearly 30 graph rules, a host graph initially with 40 nodes and nearly 20 dynamic nodes which are added to the host graph while the system evolves. Baresi *et al*.(2006) presented an approach to modeling service-oriented architectures as a graph transformation system. Then to verify the designed model, they could not use model checking but did some experiments by simulation. Because this model is a big and dynamic model, existing tools for verification graph transformations are not capable of verifying it. By simulation, they only checked that a specific state was reachable. But by our approach, we can check different properties on this model.

Our experimental results were obtained on a 3-GHz Pentium IV processor with 1 GB memory; for CheckVML, we used the results in (Rensink *et al*., 2004) where a 3-GHz Pentium IV processor was used with 1 GB memory; for GROOVE, we used the results provided by the GROOVE group (on a 3.2-GHz processor with 500 MB memory).

Table 1 presents the results of our experiments (some of these models and their corresponding generated BIR codes along with the translator are available at http://webpages.iust.ac.ir/rafe/files/expriments.rar) and compares them with those of CheckVML and GROOVE. Considering Table 1, we understand that for the dining philosophers, our approach is similar to CheckVML but weaker than GROOVE. The biggest model for dining philosophers that our approach can verify (on the same machine) is a model with 11 philosophers. In the case of Concurrent Append examples, our approach is better than CheckVML with respect to both memory usage and the time taken to run. Compared with the GROOVE in this example, our approach is weaker based on the memory used and the states and transitions generated. The use of more memory can be explained with the need to manage typed graphs. But as our approach is designed to handle attributed type graphs (i.e., Shopping, SmartCar examples and Airport case studies) and layered graph transformation systems (i.e., the Airport case study), it proves its efficiency in the next three case studies (the last three rows of Table 1).

As compared with CheckVML, our approach is more efficient in all the cases but the dining philosophers (where the result is similar). Unfortunately, CheckVML is not publicly available now; therefore we could not test the Shopping example, but as mentioned in (Rensink *et al*., 2004), CheckVML has some drawbacks for dynamic cases and this case study is also a dynamic model. Furthermore, CheckVML cannot handle layered graphs like the Airport case study.

About the range of properties we can check, we implemented some examples like (Hausmann, 2005; Engels *et al*., 2007). In these works, the authors proposed an approach to formally defining semantics for dynamic meta modeling using graph transformation systems. They used an activity diagram as a case study and defined a formal semantics for the activity diagram based on token flow semantics to model and analyze workflows. To implement the semantics, they used GROOVE and finally checked a liveness property on activity diagrams. As they mentioned, their

Table 1 Comparison of GROOVE, CheckVML and our approach in terms of memory used, states and transitions generated, and processing time

Example	Number of states			Number of transitions			Memory (MB)			Time (s)		
	GR	CV	OA	GR	CV	OA	GR	CV	OA	GR	CV	OA
Dining Philosopher												
3	17	57	46	35	125	112	0.1	2.6	0.1	0.1	0.2	0.01
4	45	181	162	124	554	533	0.1	2.6	0.2	0.1	0.2	0.06
5	117	603	574	403	2397	2366	0.2	2.6	0.4	0.2	0.2	0.5
8	3261	25 961	25 890	17 984	171 058	170 985	0.6	8.8	10.6	2.2	0.6	32
12	347 337	OM	OM	2 873 308	OM	OM	72.6	OM	OM	367.6	OM	OM
Concurrent Append												
2:3	57	22	40	94	169	59	0.2	2.6	0.01	0.2	0.5	0.01
2:5	145	86	116	290	395	199	0.4	2.6	0.12	0.3	1.1	0.03
3:5	1125	3311	2124	3161	5764	5428	0.6	37	1.3	1.2	40	1.3
3:7	2617	OM	3386	7766	OM	15 574	1.0	OM	5.4	2.2	OM	4.8
4:8	31 104	OM	40 669	116 642	OM	1 116 697	18.3	OM	467.5	30.8	OM	561
Shopping example	8584	NA	3816	23 196	NA	141 987	5.9	NA	8.9	6.1	NA	7.5
Airport case study	CNS	CNS	145	CNS	CNS	412	CNS	CNS	0.15	CNS	CNS	0.1
SmartCar example	CNS	CNS	538 936	CNS	CNS	3 338 856	CNS	CNS	598	CNS	CNS	1186

GR: GROOVE, CV: CheckVML, OA: our approach. In the first column, 3, 4, 5, 8, 12 are the number of philosophers, and 2:3, 2:5, 3:5, 3:7, 4:8 are the number of appends vs the number of cells. OM: out of memory, NA: not available, CNS: cannot support

proposal cannot check the liveness property of a single activity node in the diagram. To do a comparison, we implemented this example with our approach (Rafe and Rahmani, 2008) and found that it can check not only different properties on activity diagrams but also the liveness for all the nodes in the model. We believe that it can check a wide range of properties (in contrast to CheckVML that can check only such properties as safety and reachability).

It remains to discuss the time that our translator will take to generate the BIR code. In general, it can be quite expensive: in a graph transformation system with $|R|$ rules, where each rule R_i (i=1, 2, …, $|R|$) has $|R_i|$ nodes in its LHS, and each node n_j (j=1, 2, …, $|R_i|$) can appear $|n_j|$ times in the host graph, the number of possible matchings is $\sum_{i=1}^{|R|}\prod_{j=1}^{|R_i|}|n_j|$. But this time is considerably less than that the Bogor takes to run for the same model. To show this, we measured the running time of our translator for various input model sizes (Table 2). Comparison of the time in Table 2 with that in Table 1 reveals that in all cases the verification time is more than the translation time. For each example, the number of generated BIR code lines is also shown in the table.

Table 2 Running time of translator for different case studies

Example	BIR code lines	Time (s)
Dining Philosopher #12	343	1
Concurrent Append 4:8	1007	4
Shopping example	1117	5
Airport case study	352	1
SmartCar example	3120	475

#12 in the Dining Philosopher shows the number of philosophers in the model; 4:8 in the Concurrent Append denotes the number of nodes that must be added to the list that initially has 8 nodes

Architecture of the translator

The main components of the translator are shown in Fig.13. The translator is written in Java to ensure platform independence. As is shown, the graph transformation system along with the properties to be checked and the corresponding LTL expression, are designed in AGG by designers. The translator gets the graph transformation and the LTL expression as its inputs. From these inputs our translator derives a semantically equivalent model in BIR. Then Bogor runs the generated BIR model and performs the verification.

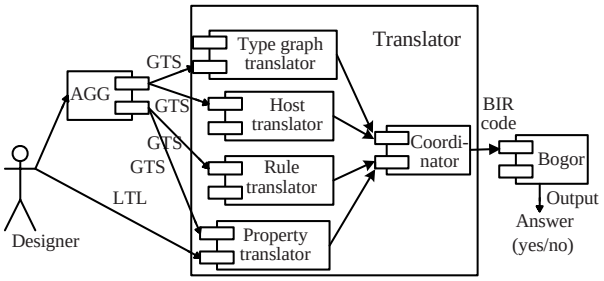

Fig.13 The proposed architecture of the translator

CONCLUSION AND FUTURE WORK

In this study, we completed our previous preliminary research (Baresi *et al*., 2008), for model checking graph transformation systems. To do so, we exploited Bogor and implemented a translator to encode graph transformation systems to the input language of Bogor (i.e., BIR) using an efficient algorithm. In addition, we used the approach presented by both GROOVE and CheckVML to state properties by a combination of LTL and special-purpose graph transformation rules. Then Bogor verifies the model against the properties by generating the transition system and sends the results of the verification back to the designers. Supporting dynamic metamodeling, layered graphs and attributed type graphs is the key characteristic that makes our solution highlighted from the existing proposals.

In the future, we intend to complete the implementation of a prototype analysis framework for back-annotating analysis results so that they could be simulated in AGG. In addition, we plan to exploit the extensibility of Bogor to reduce the state space. To do so, Bogor must consider only the states that contain different graphs. Obviously, the benefit is a better performance and less memory use.

ACKNOWLEDGEMENTS

This research was partially done while the first author was in University of Politecnico di Milano, Italy as a visiting researcher and we would like to thank Prof. Luciano Baresi and Dr. Paola Spoletini for providing supports.

References

Baldan, P., König, B., 2002. Approximating the behavior of graph transformation systems. *LNCS*, **2505**:14-29. [doi:10.1007/3-540-45832-8]

Baldan, P., Corradini, A., König, B., 2004a. Verifying finite-state graph grammars: an unfolding-based approach. *LNCS*, **3170**:83-98.

Baldan, P., Corradini, A., Gadducci, F., 2004b. Specifying and verifying UML activity diagrams via graph transformation. *LNCS*, **3267**:18-33. [doi:10.1007/b103251]

Baresi, L., Heckel, R., 2002. Tutorial introduction to graph transformation: a software engineering perspective. *LNCS*, **2505**:402-429. [doi:10.1007/b100934]

Baresi, L., Spoletini, P., 2006. On the use of Alloy to analyze graph transformation systems. *LNCS*, **4178**:306-320. [doi:10.1007/11841883_22]

Baresi, L., Heckel, R., Thöne, S., Varró, D., 2003. Modeling and Validation of Service Oriented Architectures: Application vs. Style. European Software Engineering Conf. and ACM SIGSOFT Symp. on the Foundations of Software Engineering, p.68-77.

Baresi, L., Heckel, R., Thöne, S., Varró, D., 2006. Style-based modeling and refinement of service-oriented architectures: a graph transformation-based approach. *Software Syst. Model*., **5**(2):187-207. [doi:10.1007/s10270-006-0001-4]

Baresi, L., Rafe, V., Rahmani, A.T., Spoletini, P., 2008. An Efficient Solution for Model Checking Graph Transformation Systems. 3rd Workshop on Graph Transformation for Verification and Concurrency, ENTCS, **213**:3-21. [doi:10.1016/ j.entcs.2008.04.071]

Bensalem, S., Ganesh, V., Lakhnech, Y., Munoz, C., Owre, S., Rueß, H., Rushby, J., Rusu, V., Saïdi, H., Shankar, N., *et al*., 2000. An Overview of SAL. Fifth NASA Langley Formal Methods Workshop, p.187-196.

Compton, K., Gurevich, Y., Huggins, J., Shen, W., 2000. An Automatic Verification Tool for UML. Technical Report, CSE-TR-423-00, Department of Electrical Engineering and Computer Sciences, University of Michigan, USA.

Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Pasareanu, C.S., Robby, Zheng, H., 2000. Bandera: Extracting Finite-state Models from Java Source Code. 22nd Int. Conf. on Software Engineering, p.439-448. [doi:10.1145/337180.337234]

Dotti, F.L., Foss, L., Ribeiro, L., Santos, O.M., 2003. Verification of object-based distributed systems. *LNCS*, **2884**:261-275. [doi:10.1007/b94120]

Ehrig, H., Pfender, M., Schneider, H.J., 1973. Graph Grammars: An Algebraic Approach. 14th Annual Symp. on Switching and Automata Theory, p.167-180. [doi:10.1109/SWAT.1973.11]

Ehrig, H., Engels, G., Kreowski, H., Rozenberg, G. (Eds.), 1999. Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools. World Scientific, USA.

Engels, G., Soltenborn, C., Wehrheim, H., 2007. Analysis of UML activities using dynamic meta modeling. *LNCS*, **4468**:76-90. [doi:10.1007/978-3-540-72952-5]

Ferreira, A.P.L., Foss, L., Ribeiro, L., 2007. Formal Verification of Object-oriented Graph Grammars Specifications. Proc. Third Workshop on Structural Operational Semantics, ENTCS, **175**:101-114.

Grunske, L., Winter, K., Yatapanage, N., 2008. Defining the abstract syntax of visual languages with advanced graph grammars—a case study based on behavior trees. *J. Vis. Lang. Comput.*, **19**(3):343-379. [doi:10.1016/j.jvlc.2007.11.003]

Gyapay, S., Schmidt, Á., Varró, D., 2004. Joint Optimization and Reachability Analysis in Graph Transformation Systems with Time. Int. Workshop on Graph Transformation and Visual Modeling Techniques, **109**:137-147.

Hausmann, J.H., 2005. Dynamic Meta Modeling: A Semantics Description Technique for Visual Modeling Languages. PhD Thesis, University of Paderborn, Germany.

Hausmann, J.H., Heckel, R., Taentzer, G., 2002. Detection of Conflicting Functional Requirements in a Use Case-driven Approach: A Static Analysis Technique Based on Graph Transformation. Proc. 24th Int. Conf. on Software Engineering, p.105-115.

Heckel, R., 1998. Compositional verification of reactive systems specified by graph transformation. *LNCS*, **1382**:138-153. [doi:10.1007/BFb0053578]

Holzmann, G.J., 1997. The model checker SPIN. *IEEE Trans. Software Eng.*, **23**(5):279-295. [doi:10.1109/32.588521]

Jackson, D., 2006. Software Abstractions: Logic, Language, and Analysis. The MIT Press, USA.

Kastenberg, H., 2005. Towards Attributed Graphs in GROOVE. First Workshop on Graph Transformation for Verification and Concurrency, ENTCS, **154**:47-54. [doi:10.1016/j.entcs.2005.03.030]

Kuske, S., 2001. A formal semantics of UML state machines based on structured graph transformation. *LNCS*, **2185**:241-256. [doi:10.1007/3-540-45441-1_19]

Latella, D., Majzik, I., Massink, M., 1999. Automatic verification of UML statechart diagrams using the SPIN model checker. *Formal Aspects Comput.*, **11**(6):637-664. [doi:10.1007/s001659970003]

Liu, H., Ma, Z.Y., Shao, W.Z., 2007. Description and proof of property preservation of model transformations. *J. Software*, **18**(10):2369-2379. [doi:10.1360/jos182369]

Paltor, I., Lilius, J., 1999. vUML: A Tool for Verifying UML Models. 14th IEEE Int. Conf. on Automated Software Engineering, p.255-258.

Rafe, V., Rahmani, A.T., 2008. Formal analysis of workflows using UML 2.0 activities and graph transformation systems. *LNCS*, **5160**:305-318.

Rensink, A., 2004. The GROOVE simulator: a tool for state space generation. *LNCS*, **3062**:479-485.

Rensink, A., Schmidt, Á., Varró, D., 2004. Model checking graph transformations: a comparison of two approaches. *LNCS*, **3256**:226-241. [doi:10.1007/b100934]

Robby, Dwyer, M.B., Hatcliff, J., 2003. Bogor: an extensible and highly-modular software model checking framework. *ACM SIGSOFT Software Eng. Notes*, **28**(5):267-276. [doi:10.1145/949952.940107]

Schmidt, Á., 2004. Model Checking of Visual Modeling Languages. MS Thesis, Budapest University of Technology, Hungary.

Schmidt, Á., Varró, D., 2003. CheckVML: a tool for model checking visual modeling languages. *LNCS*, **2863**:92-95. [doi:10.1007/b14063]