Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 213 (2008) 3–21
www.elsevier.com/locate/entcs

# An Efficient Solution for Model Checking Graph Transformation Systems

Luciano Baresi[1],

*Dipartimento di Elettronica e Informazione*
*Politecnico di Milano – Milan, Italy*

Vahid Rafe, Adel T. Rahmani[2,3],

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

Paola Spoletini[4]

*Dipartimento di Scienze della Cultura, Politiche e dell'Informazione*
*Università degli Studi dell'Insubria – Como, Italy*

**Abstract**

This paper presents an efficient solution for modeling checking graph transformation systems. The approach transforms AGG specifications into Bogor models and supports both attributed typed graphs and layered transformations. Resulting models are amenable to check interesting properties expressed as combinations of LTL (Linear Temporal Logic) and graph transformation rules. The first experimental results are encouraging and show that in most cases our proposal improves existing approaches, both in terms of performance and expressiveness.

*Keywords:* Attributed Type Graphs, Layered Graph Transformations, Model Checking, Bogor, AGG.

[1] Email: baresi@elet.polimi.it
[2] Email: paola.spoletini@uninsubria.it
[3] Email: rafe@iust.ac.ir
[4] Email: rahmani@iust.ac.ir

1571-0661 © 2008 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2008.04.071

# 1 Introduction

Many of the artifacts software engineers are used to deal with are nothing but suitable annotated graphs. Software architectures, class diagrams, and version histories are only a few well-known examples in which graphs have proven their usefulness in everyday software engineering. These models, and many others, can easily be described by means of suitable graph transformation systems [5,13] to formalize both their syntax and formal semantics [6,23].

Most of the research so far concentrated on graph transformation systems as a modeling means, without considering the need for suitable analysis tools. Oftentimes, however, modeling must be complemented with analysis capabilities to let the user understand how designed transformations behave and whether stated requirements are fulfilled.

If we reason on single rules, which describe local changes, we can only understand how the underlying graph evolves locally, but we loose its global behavior. Powerful analysis solutions must bypass this limitation and allow the user to reason on how the different rules impact the behavior of the graph as a whole. CheckVML [28] and GROOVE [14] exploit model checking techniques to provide such analysis capabilities, but their usefulness is limited for different reasons. They both do not support layered graph transformation systems directly. CheckVML does not efficiently support dynamic systems, that is, systems whose nodes are added/deleted by transformation rules while the system evolves, and GROOVE support attributed graphs partially and in a non native way.

To overcome these limitations, the paper presents an innovative approach based on Bogor [26] for model checking AGG-like [1] graph transformation systems. As already said, the problem per se is nothing new, but our solution has some key characteristics that improve currently available proposals since: (1) we foster models that are rich enough to render *complex* types, and thus *attributed graphs*, (2) we rely on Bogor to tackle dynamic systems, that is, systems with inherent node creation, and (3) we natively support layered graph transformation systems, to allow designers to assign different priorities to their transformation rules [9].

Graphs are translated into the input language of the model checker, called BIR (Bandera Intermediate Language, [10]), while properties are rendered by means of LTL (Linear Temporal Logic) and special-purpose rules. The result is used to feed Bogor, which performs the verification. The approach handles dynamic systems, but obviously their maximum size is limited to a user-defined threshold.

The paper is organized as follows. Section 2 surveys the state of the art.

Section 3 briefly introduces Bogor and motivates the choice of this model checker with respect to other options. Section 4 describes our approach and shows how we encode a graph transformation system in BIR through an illustrative example. Section 5 presents some experimental results and compares them with existing approaches. Section 6 concludes the paper.

## 2 Related Work

Heckel et al. [19] provide the theoretical foundations for the verification of graph transformation systems through model checking by proposing to interpret graphs as states and transformation rules as transitions. This idea is exploited by GROOVE [24], CheckVML [28], and also by our approach.

GROOVE applies graph-specific model checking algorithms by rendering graphs as states and transitions as applications of graph transformation rules. Properties are specified as transformation rules and CTL expressions containing rule names as atoms. Since GROOVE does not support typed graphs, the verification of real models becomes complex (or unfeasible). There is a proposal to extend GROOVE with attributed graphs [22], but it does not support all the "common" types (e.g., strings) and is difficult to understand since attributes and values are kept separate. Moreover, performance easily deteriorates as soon as the size of graphs increases.

CheckVML [28] exploits SPIN [20] to verify graph transformation systems. It takes a type graph, a parameterized graph transformation system, and an initial graph to produce an equivalent Promela model, which is the input of the SPIN model checker. Property graphs are translated into LTL. The approach uses a fixed 0-1 array to encode the creation/deletion of nodes and this choice results in insufficient performance [25]. [15] attempts to tackle optimization problems in graph transformation systems with time by using CheckVML, but the dynamic creation and deletion of system objects is bounded [16].

As for other proposals, Baldan and König [3] describe a different theoretical framework that aims at analyzing a special class of hypergraph rewriting systems by means of a static analysis technique based on approximate foldings and unfoldings of a special class of Petri nets. The authors [2], and Corradini, also extend this work by providing a precise (McMillan style) unfolding strategy. Dotti et al. [11] use object-based graph grammars for modeling object-oriented systems and define a translation into Promela. The authors consider a restricted structure for graph transformation rules that is tailored to modeling the message exchange mechanism typical of object-oriented systems. Even if the chosen representation in terms of Promela constructs only supports a restricted problem, the structure of generated code is interesting

since it might result in good runtime performance. Finally, Baresi and Spoletini [7] describe a methodology to analyze graph transformation systems by means of Alloy [21]. The authors present an encoding of AGG systems into Alloy, but the Alloy analyzer only deals with a-priori limited domains.

## 3 Bogor

The approach presented in this paper is based on Bogor [26], which is an extensible software model checking framework developed at Kansas State University. Its novel capabilities are appealing for checking the properties of a variety of modern software artifacts, while its internal, modular architecture lets domain experts extend the model checker to provide domain-specific analysis capabilities [4].

```
system example {
   int x := 100;

   thread A() {
      loc loc0:
         when x % 2 != 0 do {x := 3 * x + 1;}
         goto loc0;
   }

   thread B() {
      loc loc0:
         when x % 2 == 0 do {x := x / 2;}
         goto loc0;
   }

   main thread MAIN() {
      loc loc0:
         do {start A(); start B();}
         return;
   }
}
```

Fig. 1. An example Bogor model.

The input language, called BIR (Bandera Intermediate Representation), provides the basic constructs commonly supplied by the modeling languages of verification tools (e.g., Promela). For example, it supports primitive and non-primitive data types, function pointers, dynamic creation of threads and objects, automatic memory management (garbage collector), and generic data types. Control-flow and actions in BIR are stated in a guarded command format: *guard expressions* evaluate expressions, while *actions* (commands) modify the state of the system. It is also possible to create different locations by labeling parts of the code and modify the control-flow by explicitly jumping among them.

Concurrent processes are modeled by threads. For example, the BIR model

of Figure 1 comprises three threads (`A`, `B` and `MAIN`) and a global integer variable `x` initialized to 100. Threads `A` and `B` define two simple loops. For example, thread `A` checks whether `x` is odd and, if it is the case, computes its new value (`3*x+1`). Then statement `goto loc0` simply indicates that `loc0` is the next location to reach and thus closes the loop. Notice that when a guard is false, the thread is stopped in that location till the condition becomes true or no other operations are executable in the model, and Bogor detects a deadlock.

The execution of a BIR model always starts from thread `MAIN`. For example, in the model of Figure 1, the thread launches `A` and `B` and initializes `x` to 100. Then, depending on the actual value of `x`, `A` and `B` can change its value. During verification, Bogor creates an automaton whose states represent a configuration of the execution of the whole program (a Bogor state also includes the value of the variables in the code), hence, each time the value of `x` is changed, it generates a new state and it keeps doing so as long as it does not detect any new state. Command `return` in the main thread means that, when the execution of `A` and `B` finishes, the program terminates, and the automaton that represents the whole program goes to the end state.

Bogor also provides a module for checking properties expressed in LTL [8]. The desired property must be defined as a BIR function (`fun`) that is then checked by the model checker. Figure 2 shows two examples. The first property, whose LTL equivalent is $\Box((x>0)\Rightarrow\Diamond(x<0))$, is true if, in every possible execution, every time there is a state in a path in which `x` is greater than zero, then eventually there is a state in the postfix of that path, in which `x` is less than zero. This property cannot be satisfied by the example of Figure 1. The second property, whose LTL equivalent is $\Box(x>0 \wedge x\leq 100)$, is true if for every possible execution, in each state, the value of `x` is greater than 0 and less than or equal to 100. This property is satisfied by the example model.

```
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("p", x<=100)
      ),
      LTL.always(LTL.conjunction(LTL.prop("p"), LTL.prop("q")))
   );
```

Fig. 2. Two example property functions

# 4 Proposed Encoding

The main features of the proposed approach can be summarized in the following steps: (a) we build the BIR data structures needed to encode the type graph of the transformation system, (b) we initialize them through the host graph, and (c) we encode the rules: the LHS of a rule is encoded as a global matching procedure, while the RHS as a thread that applies the modifications stated by the rule. These steps are exemplified in this paper through the shopping cart example, introduced in [18], that describes the process of purchasing goods at a market.

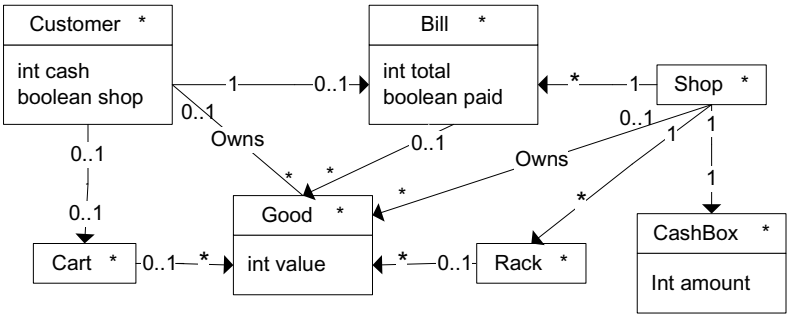


Fig. 3. Example type graph

Figure 3 shows the type graph, which comprises seven types of nodes, along with attributes, multiplicity, and associations. Its elements are manipulated by means of the rules of Figure 4 that describe all the possible actions that a customer can perform. Then, Figure 5 shows the host graph used in the paper as starting point for the analysis.

The first step of our encoding requires that the type graph be translated into BIR to define the data structures needed later. The type graph is presented as a pair: $G = (N, A)$, where $N$ is the set of nodes and $A$ the set of edges, more often called associations. Each node $n$ in $N$ is a triple $\{mult, Attr, O\}$, where $mult$ is the multiplicity of the node, $Attr$ is the set of its attributes, and $O$ is the set of its outgoing associations (with the corresponding multiplicity $mult$ and the destination node $dest$ [5]). For each node in the type graph, we build a record that contains all the information in the node and then we add a further record to represent the graph itself. More precisely, the main steps of the encoding are the following:

[5] We do not care about the multiplicity of the sources since we use AGG to check whether the host graph and defined rules comply with it. This is why we only consider the multiplicity of the destination side to build the data structures.

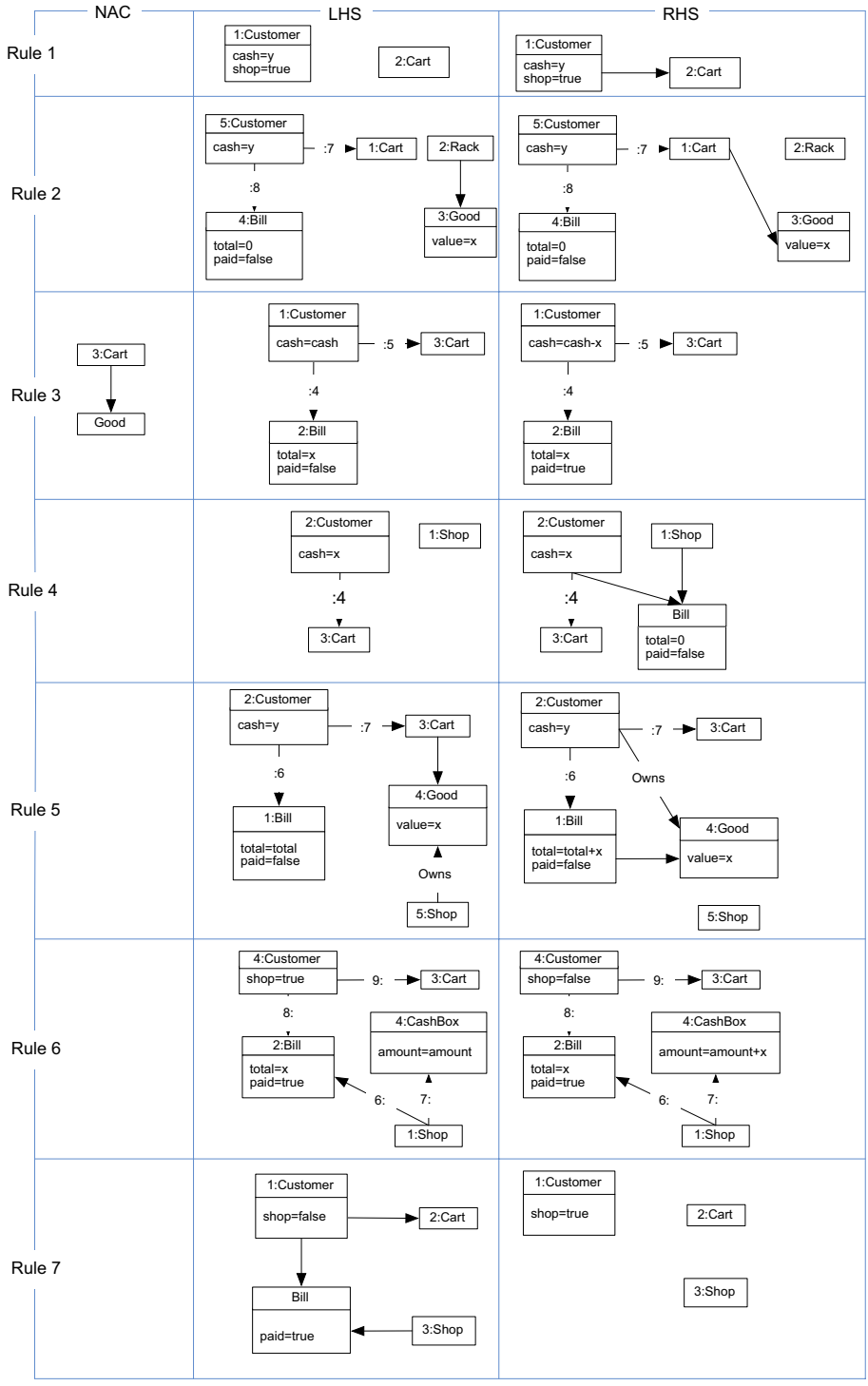


Fig. 4. Example transformation rules

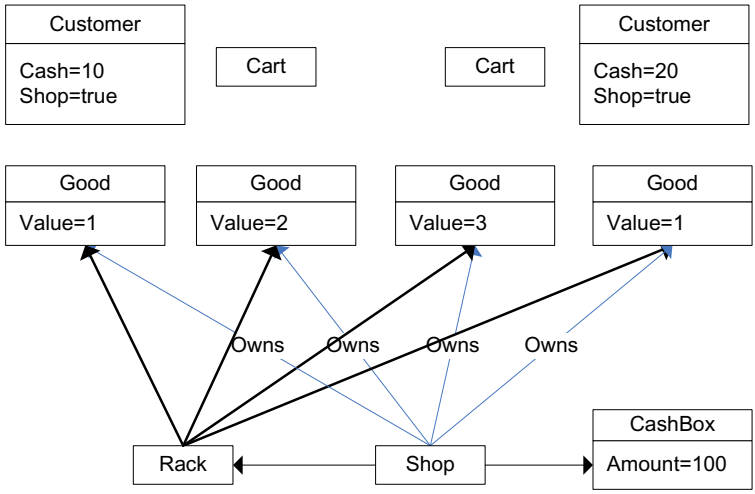


Fig. 5. Example host graph

- $\forall n \in N$, *we create a record with the following fields:*
  - $\forall a \in n.Attr$, *we add a field to the record with the same type as* $a$.
  - $\forall o \in O$,
    *if* $o.mult = 1$, *we add an element of type* $o.dest$ *as field of the record,*
    *if* $o.mult = *$, *we add an array of elements of type* $o.dest$ *as field of the record.*

Roughly, each node is a record and each attribute defined in the node type is mapped onto an equivalent field of the record. For associations, along with their multiplicity, in the type graph, we use a field of the type of the destination node if multiplicity is equal to 1, or an array otherwise.

Our approach also supports inheritance by copying all the attributes and associations of the super type as fields in the record representing the sub type.

```
record customer{
      int cash;
      boolean shop;
      bill bill_outgo;
      good[] owns;
      cart cart_outgo;
}
```

Fig. 6. A BIR record equivalent to type `Customer`

As example, Figure 6 represents the BIR record equivalent to node `Customer` of Figure 3. The first two fields of the record, `cash` and `shop`, correspond to the attributes of the node; the other three fields represent the associations. Notice that while `cart_outgo` and `bill_outgo` are single elements of type `Cart` and `Bill`, respectively, while `owns` is an array of `Goods`

since a `Customer` can buy different `Goods`.

We also define a record to represent the whole type graph:

- $\forall n \in N$,
  - *if* $n.mult = 1$, *we add a field of type* $n$ *to the record*
  - *if* $n.mult = *$, *we add an array of elements of type* $n$ *to the record.*

```
record graph{
   shop[] shops;
   cashbox[] cashboxes;
   rack[] racks;
   good[] goods;
   bill[] bills;
   customer[] customers;
   cart[] carts;
}
```

Fig. 7. Graph data structure in BIR

This record contains all the nodes of a type graph, and thus the record of Figure 7 represents the type graph of Figure 3: all the fields in the example correspond to arrays since the multiplicity of all nodes is $*$.

After defining the data structures, we create a thread `main` to drive the behavior of the whole system. This thread is divided into locations and the first one (`loc0`) is used to instantiate the type graph and implement the host graph. We first create a variable of type `graph`, and we suitably dimension the arrays it contains. More precisely, we dimension the arrays by distinguishing between static and dynamic nodes. The former can neither be added nor deleted in the RHS of a rule, while the latter can. The dimension of the arrays for static nodes is the number of these nodes in the host graph, while the maximum number of dynamic nodes is not always known a-priori, and the host graph only defines a lower bound.

Bogor would allow us to use dynamic lists to represent these nodes (hence, it would be possible not to predefine any upper bound), but since model checking requires that models have a finite number of states, each domain in the model must be finite and thus the maximum size of our arrays is given by the user. Notice that, since dynamic nodes can be added and deleted, some slots of the arrays could sometime remain unused, and thus each slot is associated with a boolean field, `isactive`, as in [27], to distinguish between nodes (slots) that are really instantiated and nodes that exist but are not currently used. We also store the inactive nodes in a table —as a pool for currently passive nodes— that will be used by the rules. Created elements are then initialized with the information in the host graph.

The BIR fragment of Figure 8 shows a portion of `loc0` in the main thread. Notice that `limits` is a constant global variable that con-

tains different constants, one for each array. For example, instruction `instance.customers:=new customer[limits.num_customer]` allocates the space for the `Customer`s in the host graph (two in this case), and `instance.customer[0].cash:=10` sets the value of field `cash` for the first customer in the model to 10. In our example, the only dynamic node is `Bill`, which is not present in the initial host graph (the lower bound is 0). In this particular case, the maximum value is 2, since each `Bill` is associated with at most one `Customer`, and we only have two of these (static) instances in the host graph.

```
loc loc0:
   do{
      instance:=new graph;
      instance.customers:=new customer[limits.num_customer];
      instance.carts:=new cart[limits.num_cart];
      instance.bills:=new bill[limits.num_bill];
      instance.shops:=new shop[limits.shop];
      instance.goods:=new good[limits.num_good];
      instance.racks:=new rack[limits.num_rack];
      instance.cashboxes:=new cashbox[limits.num_cashbox];
      instance.customers[0]:=new customer;
      instance.customers[0].owns:=new good[limits.num_good];
      instance.customers[0].shop:=true;
      instance.customers[0].cash:=10;
```

Fig. 8. A portion of `loc0` in the main thread

### *4.1 Transformation Rules*

The problem of encoding rules is split into two different sub-problems: matching and acting, i.e., the LHS (and the NAC, if present) and the RHS, respectively. The matching procedure is located in the main thread, while each action is defined as a particular thread. The matching is performed in the second location of the main thread (`loc1`) and uses a vector for each rule, which contains the main components of the LHS, that is, the minimum set of node types in the LHS that allows one to access all the other nodes in the LHS. More precisely, the vector only contains the nodes that are not reachable from others. If the LHS graph contains a cycle, it is cut nondeterministically to identify the node(s) to be stored in the vector.

To detect the main components for each LHS, we use a simple algorithm based on DFS search. This means that, since for each rule its main components are stored in the vector, and the host graph instantiates the different node types, the problem becomes checking whether the type instances of the host graph match some of the vectors. In the case of dynamic nodes, since the elements in the vectors may also match inactive elements (corresponding to inactive nodes), we add an additional condition (besides the matching one) to check wether a node is active and, only in this case, we then check the

```
when instance.customers[0].cart_outgo!=null &&
     instance.customers[0].bill_outgo!=null &&
     instance.customers[0].bill_outgo.isactive==true &&
     instance.customers[0].bill_outgo.paid==false &&
     instance.customers[0].cart_outgo.good_outgo[0]==null &&
     instance.customers[0].cart_outgo.good_outgo[1] &&
     instance.customers[0].cart_outgo.good_outgo[2]==null &&
     instance.customers[0].cart_outgo.good_outgo[3]==null
do {
     start rule3(instance.customers[0]);
}

when instance.customers[1].cart_outgo!=null &&
     instance.customers[1].bill_outgo!=null &&
     instance.customers[1].bill_outgo.isactive==true &&
     instance.customers[1].bill_outgo.paid==false &&
     instance.customers[1].cart_outgo.good_outgo[0]==null &&
     instance.customers[1].cart_outgo.good_outgo[1] &&
     instance.customers[1].cart_outgo.good_outgo[2]==null &&
     instance.customers[1].cart_outgo.good_outgo[3]==null
do {
     start rule3(instance.customers[1]);
}
```

Fig. 9. Guarded commands to detect Rule 3

matching condition. Notice that we treat the NAC as the LHS, with the only difference that it is a negative condition. If more than one matching is satisfied, Bogor chooses non deterministically among them.

The matching can be quite expensive and depends on the number of rules, the number of main components in the rules, the number of nodes reachable from the main components, and the cardinalities of the different associations. If the tree, that is, the hierarchy of nodes reachable from a main one, is of depth zero or all the associations have cardinality at most equal to one, given a system with $|R|$ rules, where each rule $R_i$ ($i = 1, \ldots, |R|$) has $|R_i|$ nodes as main components and each main component $n_j$ ($j = 1, \ldots, |R_i|$) can appear $|n_j|$ times in the graph, then the number of comparisons to identify the to-be-applied rule is $\sum_{i=1}^{|R|}(\sum_{j=1}^{|R_i|}(n_j))$, but the number of possible matching is $\sum_{i=1}^{|R|}(\prod_{j=1}^{|R_i|}(n_j))$. If the depth of the tree is greater than zero and the associations have multiplicity greater than one, the number of comparisons and the number of matchings also depend on the number of nodes that satisfy the considered association.

These considerations try to identify an upper bound to the number of possible matchings to give an intuition of the complexity of the most expensive part of the proposed approach. To mitigate it, our algorithm for the calculation of possible matchings is optimized and discards some redundant paths while creating the Bogor code for the matching.

As an example of the matching phase, we consider Rule 3 of Figure 4. The main component of this rule is a node of type `Customer`, and through this node we can access the other nodes of the LHS. The vector corresponding to this

rule contains a `Customer` associated with a `Bill` with attribute `paid` set to false and with an empty `Cart` (no `Good`s in it). Since in this case we only have one main component, and is a static node, we check for all the nodes of type `Customer` in the graph to see whether they have the same associations as the `Customer` represented in the vector of Rule 3. Figure 9 shows the two *guarded commands* to detect this rule, and if we consider the graph of Figure 5, we have two nodes of type `Customer` that do not match the rule.

As additional example, we can consider Rule 5, which has a `Customer` and a `Shop` as main components. Hence the vector that corresponds to this rule contains a `Customer` node associated with a `Bill`, whose attribute `paid` is set to false, and with a `Cart` with a `Good`, and a `Shop` node associated with the same `Good` as the `Cart` above.

When the guard associated with a rule evaluates to true, we execute the action that corresponds to the RHS, that is, the thread associated with the rule. After executing it, Bogor keeps checking and choosing non deterministically among true guards till the end of the state space.

The matching for layered transformation systems is performed as described above, but instead of using one location for all the matchings, we use one location for each layer. Since a location represents the matching for a particular layer, the matching procedure starts from the first location and looks for matchings. If it succeeds, we execute the corresponding action, otherwise the matching is performed on the next location.

As stated above, the RHS of a rule is translated into an atomic thread, whose parameters are the actual main components of the LHS, which are the nodes detected by the matching. The actions performed by the thread are:

- If the RHS adds nodes, these nodes (taken from the pool of inactive nodes) are passed as parameters, set to active, and instantiated as stated by the rule.
- If RHS adds edges, if the edge 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, if the edge is stored in an array, an inactive cell in the array is set as in the case of unary associations.
- If the RHS just modifies attributes, the fields corresponding to the attributes in the record of the corresponding variable are changed accordingly.
- If the RHS deletes nodes, the corresponding variables are deallocated, set as inactive, added to the pool of inactive dynamic nodes, and the associations that have these nodes as source or destination are deleted as a consequence.
- If the RHS deletes edges, the corresponding variables, i.e., the fields corresponding to the associations in the source nodes, are cleared.

Figure 10 shows the thread (BIR code) that corresponds to the RHS of Rule 3 of Figure 4. The thread has a `Customer` parameter, which is the main component of the LHS. Since the rule only modifies the attributes of `Customer` and `Bill`, which has a unary association with `Customer`, we need no further parameters. The body of the thread simply modifies the attributes according to the rule.

```
thread rule3(customer cus) {
   loc loc0:
      do {
         cus.cash:=cus.cash-cus.bill_outgo.total;
         cus.bill_outgo.paid:=true;
      }
      return;
}
```

Fig. 10. Rule 3 as a thread in BIR

The BIR code of Figure 11 represents the RHS of Rule 5 and provides a more complex example. It modifies attributes and adds/deletes associations. According to our encoding, the parameters are the two main components, namely `Customer` and `Shop`, and the four indexes correspond to the two associations to be deleted and to the two free slots to write the associations we add.

```
thread rule5(customer cus,shop sh,int cart_good_outgo_no,int
             shop_owns_no, int customer_owns_no,int bill_good_outgo_no) {
   loc loc0:
      do {
         cus.owns[customer_owns_no]:=sh.owns[shop_owns_no];
         cus.bill_outgo[bill_good_outgo_no]:=sh.owns[shop_owns_no];
         cus.bill_outgo.total:=cus.bill_outgo.total+sh.owns[shop_owns_no].value;
         sh.owns[shop_owns_no]:=null;
         cus.cart_outgo[cart_good_outgo_no]:=null;
      }
      return;
}
```

Fig. 11. Rule 5 as a thread in BIR

In the body, the first two statements set the new associations (from `Customer` to `Good` and from `Bill` to `Good`), the third modifies the values of attribute `total` of `Bill` and the last two statements delete the associations from `Cart` to `Good` and from `Shop` to `Good`, respectively.

## 5 Validation

This section proposes some experiments, and comparisons with CheckVML and GROOVE on well-known examples, aimed at demonstrating the soundness of our approach. First of all, we need to describe the properties we can

check, and we mimic CheckVML and GROOVE to let our users state the properties they want to check as graph transformation rules or as special-purpose graphs. LTL is only used in a fixed way, as a means to verify in Bogor the properties expressed as rules.

Properties are stated as transformation rules, where NACs represent negative conditions and LHSs positive ones. The purpose of a property rule is to describe the characteristics that nodes must/must not have, and hence the RHS, which does not perform any action, is identical to the LHS.

To translate safety properties, we perform the following steps:

- $\forall n(nodetype) \in N \cap LHS$, *we consider all the possible (active) variables* $v_1, \ldots, v_k$ *of the type corresponding to* $n$
  - $\forall v_i \in \{v_1, \ldots, v_k\}$ *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* $v_i$
- $\forall a(associationtype) \in A \cap LHS$, *we consider all the possible (active) variables* $v_1, \ldots, v_h$ *of the type corresponding to the source of* $a$
  - $\forall v_i \in \{v_1, \ldots, v_h\}$ *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* $v_i$
- $\forall n(nodetype) \in N \cap LHS$
  - *if* $n$ *is active, we form a conjunctive proposition* $p_i$ *with a proposition per type involving* $n$
  - *if a proposition involves another node* $n'$, *we add at the conjunction a proposition per type involving* $n'$
- *We create proposition* $p$ *formed by the disjunction of all* $p_i$ *with* $i \in \{1, \ldots, |N \cap LHS|\}$. *In the end, the LTL expression is* $\Box(\neg p)$

As example, we consider the rule of Figure 12 that has no NAC and the LHS and RHS are identical. The property we want to verify simply states that two customers can never share a cart. Figure 13 shows how the property is translated.

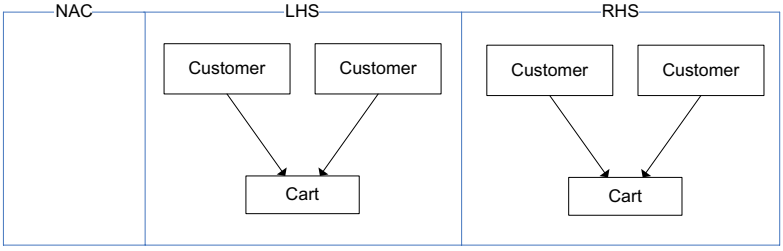


Fig. 12. Example property as a rule

In this example, we consider two possible nodes, and thus we can only create a conjunction. Indeed, if we consider the first node of type `Customer` and all its propositions, this involves the second node, which is also the only other node. Once we include its propositions, we have already considered all the combinations. `LTL.always` and `LTL.negation` are equivalent to operators $\Box$ and $\neg$, respectively.

```
fun Property1() returns boolean=
   LTL.temporalProperty(
      Property.createObservableDictionary(
         Property.createObservableKey("p",instance.customers[0].cart_outgo!=null &&
                                  instance.customers[1].cart_outgo!=null &&
                  instance.customers[0].cart_outgo==instance.customers[1].cart_outgo )
       ),
       LTL.always(LTL.negation(LTL.prop("p")))
    );
```

Fig. 13. Property of Figure 12 rendered in BIR

For reachability properties, the state "to be reached" is specified as a special graph, which is translated into a variable $G'$ of type `graph`. Then:

- $\forall n \in G$, *we consider the active variables* $v_1, \ldots, v_h$ *of type* $n$
- $\forall n' \in G'$, *if* $n'$ *has the same type as* $n$, *we consider the active variables* $v_1, \ldots, v_k$ *of type* $n$
  - $\forall v_i \in \{v_1, \ldots, v_h\}, v'_j \in \{v_1, \ldots, v_k\}$, *we create a proposition* $p_i j$ *that compares* $v_i$ *and* $v'_j$
- *We create ST as a disjunction of conjunctions, where each conjunction is formed by exactly one proposition for each attribute and association of nodes in G. In the end, we generate the LTL expression* $\neg(\Box(\neg\ ST))$

### *5.1 Case studies*

To compare our approach against existing ones, we implemented three different case studies: (1) the shopping cart example, introduced in Section 4, (2) the concurrent append example, and (3) the dining philosophers problem, presented in [25]. Our experiments were run on a 1.66 GHz Pentium IV processor with 512 MB of memory. As for CheckVML, we use the results in [25], where they used a 3GHz Pentium IV processor with 1 GB of memory, and as for GROOVE, we used the results provided by the GROOVE group (obtained on GROOVE 1.4.2 with a 3.2GHz processor with 500MB of memory).

Table 1 summarizes the results obtained with our approach and compares them against CheckVML and GROOVE. The table shows that for the dining philosophers, our approach is weaker than the other two and this is due to the use of many auxiliary variables to make the matching phase as general as possible. The table also says that our approach in most of the cases uses

| Example | | | Groove | | | | CheckVML | | | | Our Approach | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | State | Tran | Mem | Sec | State | Tran | Mem | Sec | State | Tran | Mem | Sec |
| Dining Philosopher | # entities | 3 | 17 | 35 | 0.1 | 0.1 | 57 | 125 | 2.6 | 0.2 | 133 | 820 | 6 | 1.3 |
| | | 4 | 45 | 124 | 0.1 | 0.1 | 181 | 554 | 2.6 | 0.2 | 481 | 3731 | 9.7 | 4.3 |
| | | 5 | 117 | 403 | 0.2 | 0.2 | 603 | 2397 | 2.6 | 0.2 | 2321 | 19147 | 42 | 28.5 |
| | | 8 | 3261 | 17984 | 0.6 | 2.2 | 25961 | 171058 | 8.8 | 0.6 | Out of memory | | | |
| | | 12 | 347337 | 2873308 | 72.6 | 367.6 | Out of memory | | | | | | | |
| Concurrent Append | # entities | 2:3 | 57 | 94 | 0.2 | 0.2 | 22 | 169 | 2.6 | 0.5 | 131 | 2159 | 1.2 | 0.4 |
| | | 2:5 | 145 | 290 | 0.4 | 0.3 | 86 | 395 | 2.6 | 1.1 | 221 | 4626 | 2.1 | 0.6 |
| | | 3:5 | 1125 | 3161 | 0.6 | 1.2 | 3311 | 5764 | 37 | 40 | 2654 | 66816 | 4.3 | 6.2 |
| | | 3:7 | 2617 | 7766 | 1 | 2.2 | Out of memory | | | | 3402 | 89459 | 12.4 | 8.6 |
| | | 4:8 | 31104 | 116642 | 28.3 | 30.8 | | | | | 41364 | 1117543 | 63.2 | 112.6 |
| Shopping Example | | | 8584 | 23196 | 5.9 | 6.1 | Not Available | | | | 3816 | 141978 | 8.9 | 7.5 |

Table 1
Comparison: *Mem* is consumed memory in MB, *Sec* is the time in seconds, *entities#* is the number of philosophers, and *(x:y)* is the number of appends and cells.

more memory and generates more states and transitions than GROOVE, but when attributed graphs are considered, like in the shopping cart example, it performs better in terms of number of states, dealing natively with attributes.

Compared to ChechVML, our approach is more efficient in all the cases but the dining philosophers. Unfortunately, CheckVML is not available now, therefore we cannot test the shopping cart example, but, as the authors mention in [25], CheckVML has problems with dynamic cases.

Although the main idea of our approach is similar to CheckVML, there are many differences in the details. One of them, which highly impacts the performance of our approach, is the way we manage dynamic nodes and edges. As we mentioned before, we store a graph as a record in BIR and we use a linear array for each node type in this record. Moreover, the use of the navigation support provided by Bogor to handle associations and attributes further improves the efficiency of the proposal.

As for the properties we can check, we implemented the examples in [17,29]. In these works, the authors describe a formal semantics for dynamic meta modeling with graph transformation systems and, as an example, they mention a formal semantics for activity diagrams based on token flow semantics to model and analyze workflows. They use GROOVE and define a liveness property based on token semantics and, at the end, they check whether activity diagrams are live. As they mention, their approach only checks whether the tokens in all the paths reach the final node in the activity diagram, but they cannot check the liveness property of a single activity node. Again, we implemented this example with our approach and besides being able to check liveness for a complete activity diagram, we also checked the liveness of all the nodes in the model.

## 6 Conclusions and Future Work

The paper presents an efficient approach for the automatic verification of graph transformation systems with Bogor. The approach supports both at-

tributed typed graphs and layered graph transformation systems. The properties against which we check the transformation system can be expressed in LTL directly or as special-purpose transformation rules.

As future work, we are completing the implementation of a prototype analysis framework for graph transformation rules based on AGG and Bogor. Moreover, we want to exploit the extensibility of Bogor and customize the generation of the state space to only consider the states that contain "different" graphs with the obvious benefits of better performances and less memory use.

The approach can also be quite naturally extended to hierarchical graph transformation systems [12]. Indeed, the approach already supports inheritance and this extension can be combined with layered graph transformation systems to assign different rules to different levels of the hierarchy. This would further improve the average performance of our model checking approach. In the worst case, the model checker would explore the entire state space both in the flat case and in the modular one. In many situations, however, if the model presents different levels of abstraction, the model checker does not explore the entire space, but it can work incrementally by traversing the different levels.

## References

[1] AGG,tfs.cs.tu-berlin.de/agg/.

[2] Baldan, P., Corradini, A., and König, B., "Verifying Finite-State Graph Grammars: An Unfolding-Based Approach", In Proc. of International Conference on Concurrency Theory (CONCUR ), 83–98, 2004.

[3] Baldan, P. and König, B., "Approximating the Behaviour of Graph Transformation Systems", In Proc. of First International Conference on Graph Transformation (ICGT ), 14–29, 2002.

[4] Baresi, L., Ghezzi, C. and Motolla, L., "On Accurate Automatic Verification of Publish-Subscribe Architectures", In Proc. of the International Conference on Software Engineering (ICSE07), pages 199-208, IEEE Computer Society, 2007.

[5] Baresi, L. and Heckel, R., "Tutorial Introduction to Graph Transformation: A Software Engineering Perspective", In Proc. of the First International Conference on Graph Transformation (ICGT), vol. 2505 of Lecture Notes in Computer Science, 402–429, 2002.

[6] Baresi, L., Heckel, R., Thöne, S. and Varró, D., "Modeling and Validation of Service Oriented Architectures: Application vs. Style", In Proc. of European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, 68–77, 2003.

[7] Baresi, L. and Spoletini, P., "On the Use of Alloy to Analyze Graph Transformation Systems", In Proc. of Third International Conference on Graph Transformations, (ICGT), 306–320, 2006.

[8] Bogor Extensions for LTL Checking, projects.cis.ksu.edu/projects/gudangbogor/.

[9] Bottoni P., Taentzer G., and Schürr A., "Efficient Parsing of Visual Languages based on Critical Pair Analysis and Contextual Layered Graph Transformation", Proc. of the 2000 IEEE International Symposium on Visual Languages, IEEE Computer Society, 59-68, 2000.

[10] Corbett,J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Pasareanu, C. S., Robby and Zheng, H., "Bandera:Extracting Finite-State Models From Java Source Code", In Proc. of the 2nd Int. Conf. on Software engineering, 439–448, 2000.

[11] Dotti, F. L., Foss, L., Ribeiro, L. and Santos, O.M., "Verification of Object-Based Distributed Systems", In Proc. of 6th International Conference on Formal Methods for Open Object-based Distributed Systems, 261–275, 2003.

[12] Drewes F., Hoffmann B., and Plump D., "Hierarchical Graph Transformation", Proceedings of the Third International Conference on Foundations of Software Science and Computation Structures, Springer-Verlag, 98-113, 2000.

[13] Ehrig, H., Engels, G., Kreowski, H. and Rozenberg, G. (eds.), "Handbook on Graph Grammars and Computing by Graph Transformation", vol. 2: Applications, Languages and Tools. World Scientific, 1999.

[14] GROOVE, groove.sourceforge.net/groove-index.html.

[15] Gyapay, S., Schmidt, Á. and Varró, D., "Joint Optimization and Reachability Analysis in Graph Transformation Systems with Time", In Proc. GT-VMT 2004, International Workshop on Graph Transformation and Visual Modelling Techniques, vol. 109 of ENTCS, 137–147, 2004.

[16] Gyapay-Varró, S. and Varró, D., "Optimization in Graph Transformation Systems Using Petri Net Based Techniques", In Proc. of the Workshop on Petri Nets and Graph Transformations, 2006.

[17] Hausmann, J.H., "Dynamic Meta Modeling: A Semantics Description Technique for Visual Modeling Languages", Ph.D. Thesis, University of Paderborn, Germany, 2005.

[18] Hausmann, J. H, Heckel, R. and Taentzer, G., "Detection of Conflicting Functional Requirements in a Use Case-Driven Approach: A Static Analysis Technique Based on Graph Transformation", In Proc. of Int. Computer Softawre Engineering ( ICSE), 105–115, 2002.

[19] Heckel, R., "Compositional Verification of Reactive Systems Specified by Graph Transformation", In Proc. of Fundamental Approaches to Software Engineering (FASE), 138–153, 1998.

[20] Holzmann, G. J., "The Model Checker Spin", IEEE Transaction Software Engineering, vol. 23(5), 279–295, 1997.

[21] Jackson D., "Software Abstractions: Logic, Language, and Analysis", The MIT Press, 2006.

[22] Kastenberg, H., "Towards Attributed Graphs in GROOVE", In Workshop on Graph Transformation for Verification and Concurrency, Volume 154 of Electronic Notes in Theoretical Computer Science, 47–54, 2005.

[23] Kuske, S., "A Formal Semantics of UML State Machines Based on Structured Graph Transformation", In Proc. of UML 2001, volume 2185 of Lecture Notes in Computer Science. Springer-Verlag, 2001.

[24] Rensink, A., "The GROOVE Simulator: A Tool for State Space Generation", In Applications of Graph Transformations with Industrial Relevance (AGTIVE), vol. 3062 of Lecture Notes in Computer Science, 479–485, 2004.

[25] Rensink, A., Schmidt, Á. and Varró, D., "Model Checking Graph Transformations: A Comparison of Two Approaches", In Proc. of Second International Conference on Graph Transformation (ICGT), vol. 3256 of LNCS, 226–241, 2004.

[26] Robby, Dwyer, M. and Hatcliff, J., "Bogor: An Extensible and Highly-Modular Software Model Checking Framework", In Proc. of the 9th European software engineering Confference, 267–276, 2003.

[27] Schmidt, Á., "Model Checking of Visual Languages", Master's Thesis, Budapest University of Technology and Economics, Hungary, 2004.

[28] Schmidt, Á. and Varró, D., "CheckVML: A Tool for Model Checking Visual Modeling Languages", In Proc. of 6th International Conference on the Unified Modeling Language (UML), vol. 2863 of LNCS, 92–95, 2003.

[29] Soltenborn, C., "Analysis of UML Workflow diagrams with dynamic Meta Modeling Techniques", Master's Thesis, University of Paderborn, Germany, 2006.