Joint Optimization and Reachability Analysis in Graph Transformation Systems with Time

Szilvia Gyapay [1,2] Ákos Schmidt [1] Dániel Varró [1,3]

Department of Measurement and Information Systems
Budapest University of Technology and Economics
H-1521 Budapest, Magyar tudósok körútja 2., Hungary

Abstract

The design of safety critical systems frequently necessitates to simultaneously fulfill several logical and numerical constraints as requirements in order to deliver a functionally correct and optimal target system. In the current paper, we present a combined optimization and reachability analysis approach using the SPIN model checker [5] for problems modeled with graph transformation systems with time [3]. First, we encode graph transformation rules into transitions systems in Promela (the input language of SPIN) following [**?**,9]. Then we restrict valid execution paths to time-ordered transformation sequences by additional logical conditions. The desired reachability property (as logical condition) is used to potentially decrease the global *best cost* variable whenever a new path satisfies the property. The optimal solution for the problem is found by a single exhaustive run of the model checker encoding the numerical constraints into a dynamic LTL formula to cut off suboptimal paths violating the Branch-and-Bound heuristics.

Key words: graph transformation, optimization, model checking.

1 Introduction

Recently, the notion of *graph transformation systems (GTS) with time* has been introduced in [3] to provide formal support for modeling embedded and safety critical systems that make heavy use of concepts like timeouts, timing constraints, delays, etc. Moreover, correctness with respect to these issues is critical to the successful operation of these systems. This approach defines *globally time-ordered transformation sequences* to capture the consistency of time values attached to vertices.

Since the violation of such timing restrictions is typically critical, the formal analysis of graph transformation systems is frequently necessitated in practice to

[1] This work was supported by the Hungarian National Fund OTKA T038027

[2] Email: gyapay@mit.bme.hu

[3] Email: varro@mit.bme.hu

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

prove the functional correctness of the system by using off-the-shelf analysis tools (like model checkers or theorem provers). Several recent research activities [1,6,**?**] have been focusing on the model checking of graph transformation systems. In [9], a tool support is reported for mapping graph transformation systems into Promela (the input language of the model checker Spin [5]) based on the encoding of [**?**].

However, requirements frequently necessitate to build a system that is *simultaneously correct and optimal*, i.e., the system has to simultaneously fulfill logical and numerical conditions. For instance, a typical requirement in safety critical systems is to find the optimal path respecting all timing constraints that leads to a desirable (target) system configuration. For this purpose, we need to combine reachability analysis with optimization. A primary basis of a combined technique can be based upon [8] where optimal scheduling problems are solved by embedding Branch-and-Bound techniques into the model checker Spin [5]. The same problem has been tackled on a Petri net basis using Process Network Synthesis algorithms for optimization in [4].

In the current paper, we outline a combined reachability analysis and optimization technique for graph transformation systems with time.

(i) First, we derive a functionally equivalent transition system (TS) of the graph transformation system *without* time following [**?**] that collects all potential rule applications.

(ii) Then, additional conditions are added to each transition to restrict the execution paths traversed by the model checker to time-ordered transformation sequences.

(iii) The reachability property (i.e., the desirable target configuration) is encoded as a separate Promela process. If the desired configuration is reached on an execution path and the cost of this solution is better than the cost of the best solution found up to this certain point we decrease this global variable storing the optimal cost.

(iv) To find the optimal solution with a single exhaustive run of the model checker, the state space is pruned by Branch-and-Bound techniques for suboptimal solutions. The Branch-and-Bound technique is encoded into the property to be verified (stating that the current cost will eventually become larger than or equal to the best cost on each execution path) thus it is changing dynamically during the model checking process.

2 Optimization Problems in GTS with Time

In this section, we illustrate how to model optimization problems using graph transformation systems with time. For a motivating example, we discuss (a slightly modified version of) a job-shop scheduling problem, the Personalization Machine example discussed in [8].

2.1 The Personalization Machine Problem

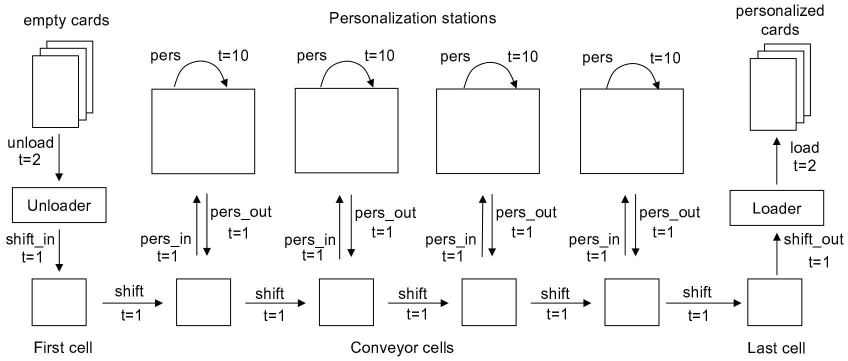

Fig. 1. Personalization machine

The Personalization Machine problem is to schedule the personalization of smart cards. In Fig. 1, a schematic overview of the personalization machine is depicted. The empty smart cards are transported by a conveyor belt (that consists of conveyor cells) to the personalization stations. The personalization process consists of the following main steps (where the costs of operations, i.e., the time they take, are also depicted in Fig. 1): (i) the Unloader puts the smart card on the first cell of the conveyor belt (using the unload and shift_in operations); (ii) the conveyor can shift a card to the next cell as a single move. If a card reaches the last cell and it has not been personalized yet, it will be sent to the empty cards and reloaded again in a later phase; (iii) the card can be taken from the belt by a free personalization station (pers_in), the personalization of the card begins immediately, i.e., the machine programs the card with personalized data (pers), and takes back to the conveyor belt (pers_out); (iv) finally, the card is removed by the Loader from the last cell of the conveyor belt (shift_out and load operations).

Our goal is to find an optimal schedule to personalize n smart cards with m personalization stations.

2.2 Defining the static structure: Type and instance graphs

The static structure of the Personalization Machine problem is modeled by using type and instance graphs which are formal graph-based representations of traditional UML class and object diagrams. Graph nodes in the type graph correspond to classes while edges correspond to associations. Attributes are frequently interpreted as special nodes. The instance graph represents the modeled system, and it is connected to the type graph by a typing homomorphism (formalizing the instance-of relation). Graph nodes in the instance graph correspond to objects, edges represent links while attributes are called slots in UML terms.

In the left part of Fig. 2, the type graph of the personalization machine problem is depicted as a UML class diagram. It consists of the nodes Card, Unloader, Loader, PersonalizationStation, and ConveyorCells. In order to model the cost of operations, all nodes (except for personalization stations) have a time attribute.

The time attributes represent logical clocks that register the time when the last operation applied on the corresponding node was completed. The current position and state of cards (and other objects) are denoted by edges and attributes with a _hold postfix. A card is considered to be processed by setting the personalized attribute to true.

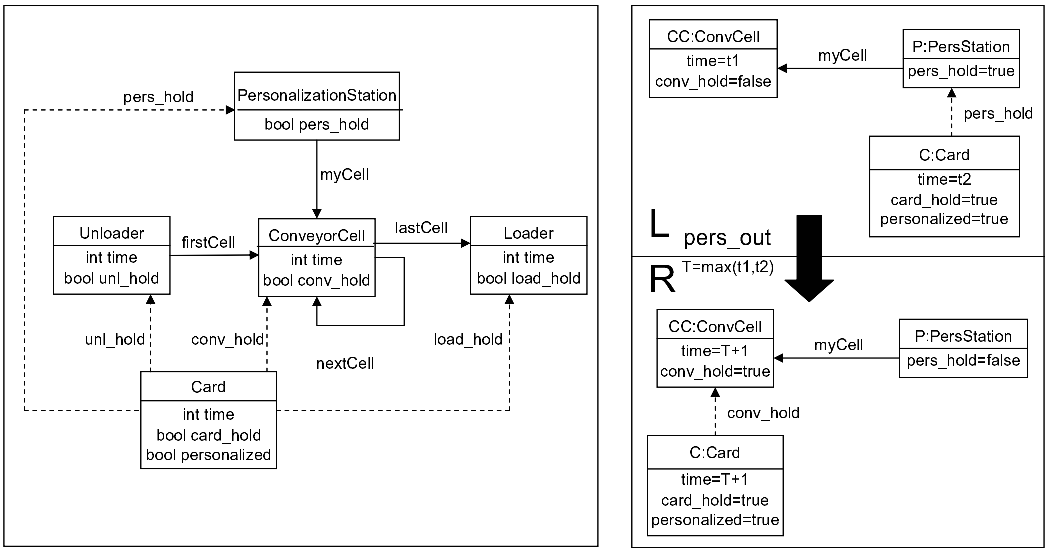

Fig. 2. Type graph and a graph transformation rule

2.3 Modeling dynamic behavior: Graph transformation with time

Graph transformation [7] provides a rule and pattern based manipulation of typed and attributed graph models by replacing a part of an instance graph matching a specific pattern with another graph. In graph transformation with time [3], a time attribute is a logical clock (typically) with non-negative integer values that can be attached to nodes. The time attribute has a distinguished role to define how the time progresses in discrete steps in a consistent way.

A *graph transformation rule* $p = (L, R)$ contains a left–hand side graph L, and a right–hand side graph R, where both graphs correspond to the type graph. Informally, L and R graphs represent the pre–conditions and the post–conditions of the rule, respectively. The *application* of a rule p to an *instance graph* G replaces a matching of L in G by an image of R. This is performed by (1) finding an occurrence of L in G, (2) removing that part of G which is matched to elements of L not belonging to R, (3) and, symmetrically, gluing the new nodes and edges to G that can be mapped to R but not to L obtaining the *derived graph* H.

A *graph transformation system (GTS)* consists of a set of graph transformation rules, while a *graph transformation* (sequence) is a sequence of consecutive rule applications selected from GTS that evolves the system from an initial graph model. A *graph transformation system with time* consists of rules that manipulate the *time* attributes in such a way that a transformation sequence derived by a GTS with time is valid if it fulfills the following axioms.

1. Local monotonicity: All time values in the occurrence of R in H are higher than any of the time values in the occurrence of L in G.

2. Uniform timestamp: All time values in the occurrence of R in H are equal. This uniform timestamp is the time of the rule application.

3. Time-ordered transformation sequences: A graph transformation sequence is called *time–ordered* if the time of consecutive rule applications in the sequence is monotonically increasing.

Informally, Axiom 1 ensures that the application of a rule takes positive time. Axiom 2 states the atomicity of rule application, i.e., all effects specified in the right–hand side are observed at the same time. Axiom 3 expresses that the system should evolve along time-ordered transformation sequences that can be executed by a central scheduler (having a global picture of the system). While Axiom 1 and 2 can be fulfilled at compile-time by the *structure* of a graph transformation rule, Axiom 3 can only be guaranteed at run-time by carefully selecting (i) the next rule to be applied and (ii) the next occurrence of the rule.

For space consideration, we introduce only one rule of the Personalization Machine problem, when the personalization machine places back the card to the conveyor belt after personalization. The pers_out rule (depicted in the right part of Fig. 2) can be applied if (1) there exists a card which has already been personalized (C.personalized=true), (2) it has not been released by the station yet (there is a pers_hold link between the card and the personalization station), and (3) the conveyor cell connected to the personalization station is empty (CC.card_hold=false).

The application of the rule yields an instance graph where the personalized card is placed back to the belt: (1) the card is released by the personalization station (P.pers_hold=false and the pers_hold link is deleted), and (2) the card is put back onto the belt (a conv_hold link is created between the cell and the card, and the corresponding _hold attributes are set), and (3) the timestamp of the card and the conveyor cell is set to max(t1,t2)+1.

Note that rule pers_out satisfies both the *Local monotonicity* and the *Uniform duration* conditions: (1) the timestamps in R are greater than the timestamps in L, and (2) all timestamps in R are uniform.

3 From GTSs with Time to Transition Systems

Transition systems (TS) are a common mathematical formalism that serves as the input specification of various model checker tools where the system is evolving by executing non-deterministic if-then-else like rules to manipulate state variables.

Graph transformation systems with time are encoded into a behaviorally equivalent TS in a two-step process. First, we collect all potential applications of graph transformation rules into transitions in the TS to handle a single graph transformation step correctly following [**?**,9]. Then valid execution paths are restricted to time-ordered transformation sequences by additional logical conditions on the time(stamp) of rule applications.

For space limitations, we only sketch the encoding of graph transformation rules into transitions in the target TS and omitting several conceptual details (such as the declaration of state variables driven by the metamodel or the initialization phase driven by the initial model) which are discussed in [**?**].

3.1 Encoding of GT rules

Potential applications of graph transformation rules are encoded into behaviorally equivalent transitions of the corresponding TS (where a transition of a TS is composed of a guard and state variable updates). Behavioral equivalence means that whenever a GT rule becomes applicable on a certain matching, the guard of a corresponding transition should evaluate to true thus the transition can be fired (and vice versa).

- All potential matchings of a GT rule are collected at compile-time into the *guards* of transitions. These guards are constituted primarily from the logical conditions imposed by the L graph and attribute conditions. Furthermore, the guards might also need to contain identification and dangling conditions as well in case of the double-pushout approach [2].
- The effects of applying a rule on a certain matching are encoded as *state variable updates*. These updates should handle if a certain graph node or edge is created or removed, or, if a new value is assigned to an attribute.

Although this compile-time preprocessing can be time-consuming, since all the potential matches of a rule have to be encountered, we only have to traverse a relatively small part of the state space for this step. This is because graph transformation rules define local modifications to the system state thus it is typically negligible when compared with the time required for traversing the entire state space during model checking.

The equivalent Promela transition[4] that encodes rule pers_out (of Fig. 2) is depicted below as a demonstration when applied on the potential matching $\{$CC $\mapsto$ cc1, P $\mapsto$ p1, C $\mapsto$ c1$\}$ (expressing that the personalization of card c1 has been completed at station p1 thus it is placed back on the conveyor cell cc1).

```
:: atomic {
   convCell[cc1] &&  conv_hold[cc1] == false && myCell[p1][cc1] &&
   persStation[p1] && pers_hold[p1] == true && pers_hold[c1][p1] &&
   card[c1] && card_hold[c1] == true && personalized[c1] == true ->
     pers_hold'[c1][p1] = false;
     conv_hold'[c1][cc1] = true;
     time'[cc1] = max(time[cc1]+1, time[c1]+1);
     time'[c1] = max(time[cc1]+1, time[c1]+1); }
```

In the guard, we check the existence of nodes and edges of corresponding types (such as, e.g., convCell[cc1] and myCell[p1][cc1]), and attribute conditions (such

[4] Note that the original technique in [**?**] uses further optimizations to reduce the number of state variables and to eliminate dead transitions which is omitted from the current paper.

as personalized[c1] == true). The state variable updates remove the pers_hold edge and create a new conv_hold edge, moreover, they update the time variable according to the axioms of GTSs with time.

3.2 Ensuring time-orderedness

The previous encoding handles the time attributes as a conventional attribute in the GTS. This way, there are execution paths in the target TS that are not time-ordered. To restrict the execution paths to time-ordered transformation sequences, several extensions are required for each transition.

We introduce a new state variable prev_time which stores the time(stamp) of the previous rule application thus it is globally accessed by all transitions on a single path. Since we aim at restricting valid execution paths to time-ordered ones, the guard of each transition is extended with a literal to check that the time of the current rule application is larger than (or equal to) the time of the previous rule application. If the guard evaluates to true (i.e., the sequence remains time-ordered) then the state variable prev_time is updated as well to store the time of the current rule application. In case of rule pers_out, the following extensions are required for the transition of the previous example.

```
max(time[cc1], time[c1]) + 1 >= prev_time && ... ->
  prev_time′ = max(time[cc1], time[c1]) + 1; ...
```

Note that since the structure of graph transformation rules trivially implies Axiom 1 and 2, only Axiom 3 has to be encoded into the Promela description.

As a summary, to restrict our investigation to time-ordered transformation sequences, changes are required to the transition system itself, but these changes are tool independent thus they can be applied to any model checker having an input language based on transition systems. Meanwhile, the techniques to be presented in Sec. 4 are specific to SPIN.

Given the transition system derived from a GTS with time, the model checker SPIN can automatically verify the validity of an arbitrary LTL formula. In this way, we are able to decide whether certain properties (requirements) hold on any *time-ordered* execution path of the GTS.

4 Combined Optimization and Reachability Analysis

To perform simultaneous optimization and verification, we restrict ourselves to reachability properties used as functional requirements (logical constraints). Thus our ultimate goal is to find an optimal transformation sequence (execution path) that leads to a desired situation defined by the reachability property. We now adapt the techniques of [8] to enable optimization in SPIN.

In a *reachability property*, we ask whether a desired situation will become true in at least one state on at least one execution path of the system. Formally, in LTL terms, a reachability property is of the form $\mathsf{G}\ \neg \mathsf{p}$, where p is the property describing the desired situation. If the property $\mathsf{G}\ \neg \mathsf{p}$ is found valid by the model checker, then the desired situation is not reachable, otherwise the desired situation is reachable, and the model checker will derive a state sequence leading to this situation as a counterexample.

In Spin, LTL formulae are translated into special Promela processes (automata) with high-priority. In fact, a reachability property can be interpreted as a special error transition (called never claim) which interrupts the run of the model checker if fired at any time and retrieves the error trace.

4.1 Updating the best cost of a solution

However, in case of optimization, the firing of such an error transition should not interrupt the model checking process, it only indicates that a (functionally) correct solution is found. Now we need to check whether the cost (i.e., the overall execution time in our case) of the current execution path satisfying the reachability property is less than the best cost of a previously found execution path. In such a case, the best cost should be decreased to store the cost of the current sequence. Finally, the model checker should continue to investigate the next execution path.

To implement this behavior, we need to rely on Spin 4.0 which supports the inclusion of embedded C code into Promela models by the following primitives: (i) c_state: to add new C variables to the Promela model; (ii) c_expr: to evaluate a C expression in a guard; and (iii) c_code: to add arbitrary C code fragments as an atomic statement to the Promela model.

(i) First we define a global variable best_cost in the Promela model using the c_state construct as follows: `c_state "int best_cost" "Hidden"` .

The scope ”Hidden” denotes that the variable best_cost will *not* be stored in the state vector and will be global to all execution runs.

(ii) The variable best_cost is initialized at the start of the verification to MAX_COST (a worst-case estimate of the cost of a schedule).

```
#define MAX_COST 1000
init {
  c_code { best_cost = MAX_COST; }}
```

(iii) Whenever a new solution is found, i.e., in the error transition(s) of the reachability property, the cost (time) of the execution path (i.e., the variable prev_time) is compared with the best_cost so far. If prev_time is smaller then we have found a better solution, so the variable best_cost is decreased and the trace is saved. The following piece of Promela code represents a reachability property reach_prop stating that each card c1, . . . cn should be personalized eventually.

```
#define reach_prop (personalized[c1] && ...  && personalized[cn])
 reach_prop ->
   c_code {
      if (now.prev_time < best_cost) {
           best_cost = now.prev_time;
```

```
                 /* Further Spin specific code to save the current trace */ }}
```

4.2 Branch-and-Bound using dynamic LTL formulae

Branch-and-Bound [10] is an approach developed for solving discrete and combinatorial optimization problems. The essence of the Branch-and-Bound approach is to enumerate all possible solutions by constructing an enumeration tree. When building the tree (i.e., the state space), we can stop considering an execution path (partial solution) if it is certain that all paths via this state (i) either lead to an *invalid solution* (logical cut) or (ii) will have *higher costs* than the best path found so far (numerical cut).

Following the guidelines of [8], we encode the Branch-and-Bound technique into the property to be verified by SPIN. We check whether the cost (i.e., the time) of an execution path will eventually become greater than or equal to the best_cost, and this property should be valid on each execution path. Formally, the corresponding LTL is $\mathsf{F}(\mathsf{higher\_cost} \lor \mathsf{G}(\neg\ \mathsf{reach\_prop}))$ where reach_prop is the desired reachability property, and higher_cost is defined as

```
#define higher_cost (c_expr { now.prev_time >= best_cost})
```

When checking $\mathsf{F}(\mathsf{higher\_cost})$, the model checker enumerates all solutions (i.e., time ordered transformation sequences leading to a desired situation) one by one, but the exhaustive traversal of the state space is reduced by *numerical cuts* whenever (i) the cost of an execution path exceeds the best_cost (i.e., $\mathsf{prev\_time} > \mathsf{best\_cost}$), or (ii) the current path is a solution with a new (more optimal) best_cost (thus $\mathsf{prev\_time} = \mathsf{best\_cost}$), or either *logical cuts* when reach_prop can never be satisfied on a certain path.

As the variable best_cost is decreased whenever a more optimal solution satisfying the reachability property is found, the LTL formula that is being checked is changing dynamically during the verification.

5 Conclusions

In the current paper, we proposed a simultaneous optimization and verification for systems modeled as GTSs with time. In this way, we are able to find the optimal time-ordered transformation sequence leading to a desired system configuration (defined by a reachability property) by using the model checker SPIN. As the next step for our current research, we aim at assessing the practical limits of the proposed technique on benchmark examples.

References

[1] Baldan, P. and B. König, *Approximating the behaviour of graph transformation systems*, in: A. Corradini, H. Ehrig, H.-J. Kreowski and G. Rozenberg, editors, *Proc.*

ICGT 2002: First International Conference on Graph Transformation, LNCS **2505** (2002), pp. 14–29.

[2] Corradini, A., U. Montanari, F. Rossi, H. Ehrig, R. Heckel and M. Löwe, "In [7]," World Scientific, 1997 pp. 163–245.

[3] Gyapay, S., R. Heckel and D. Varró, *Graph transformation with time: Causality and logical clocks*, in: A. Corradini, H. Ehrig, H.-J. Kreowski and G. Rozenberg, editors, *Proc. ICGT 2002: 1st International Conference on Graph Transformation*, LNCS **2505** (2002), pp. 120–134.

[4] Gyapay, S. and A. Pataricza, *Optimization methods for reachability analysis of Petri net models*, in: G. Tarnai and E. Schnieder, editors, *Formal Methods for Railway Operation and Control Systems (Proceedings of Symposium FORMS-2003, Budapest, Hungary, May 15-16)* (2003), pp. 53–60.

[5] Holzmann, G., *The model checker SPIN*, IEEE Transactions on Software Engineering **23** (1997), pp. 279–295.

[6] Rensink, A., *Model checking graph grammars*, in: M. Leuschel, S. Gruner and S. Lo Presti, editors, *Proc. of the 3rd Workshop on Automated Verification of Critical Systems (AVOCS 2003)*, Technical Report DSSE–TR–03–2 (2003), pp. 150–160.

[7] Rozenberg, G., editor, "Handbook of Graph Grammars and Computing by Graph Transformations: Foundations," World Scientific, 1997.

[8] Ruys, T. C., *Optimal scheduling using branch and bound with SPIN 4.0*, in: *Proc. 10th International SPIN Workshop*, LNCS **2648** (2003), pp. 1–17.

[9] Schmidt, Á. and D. Varró, *CheckVML: A tool for model checking visual modeling languages*, in: P. Stevens, J. Whittle and G. Booch, editors, *Proc. UML 2003: 6th International Conference on the Unified Modeling Language*, LNCS **2863** (2003), pp. 92–95.

[10] Winston, W. L., "Operations Research — Applications and Algorithms," Duxbury Press, Belmont, California, USA, 1994, 3rd edition.