Using Shape Analysis to verify Graph Transformations in Model Driven Design

Dominik Steenken  
University of Paderborn  
Email: dominik@mail.upb.de

Daniel Wonisch  
University of Paderborn  
Email: dwonisch@mail.upb.de

Abstract—In model driven design processes, graph transformation systems are frequently used to model dynamic behaviour. Many complex models induce arbitrarily large state spaces. Since the systems they model are often safety-critical, they need to be verified. Explicit model checking fails here, since it requires the construction of the entire state space. In this paper, we present a verification technique that can handle arbitrarily large state spaces. Furthermore we show that it can easily be integrated in existing model driven design processes.

I. INTRODUCTION

In model driven design, graph transformation systems are often used to lend formal semantics to graphical specifications of behavior, e.g., in [1]. Unfortunately, systems specified in this way can easily grow to arbitrarily large sizes. To illustrate this problem, consider the following example, taken from the railway project NBP [2].

Autonomous rail vehicles, called RailCabs, move on a large railway network. They transport passengers between the stations of the network, cooperating by forming convoys or delegating tasks whenever this results in a lower overall cost. The state of the system is modeled as a graph, the behavior of the RailCabs is specified using graph production rules. These rules take graphs as input and compute a set of graphs modeling possible successor states. The upper part of Fig. 1 shows a part of a possible state of such a system, represented as a graph.

Rule application is achieved by finding a match for the left-hand side of the rule \( L \) in a graph and then replacing it with the right-hand side \( R \).

Considering that RailCabs as well as passengers may enter and leave the system at any time, and that the length of a track segment may be variable depending on the situation, it is easy to see that this system creates an arbitrarily large state space. In self-x systems, behavior models that create such arbitrarily large state spaces are common. Since these models often represent systems that could cause significant harm to their users when they fail, it is imperative to exclude this possibility by verifying the underlying graph transformation system. For the verification of arbitrarily large state spaces, there are basically two approaches.

The first approach is to limit the size of the system by imposing upper bounds on the number of elements involved, i.e., in this case, by restricting the number of RailCabs and passengers the system can accommodate. The problem with this is that realistic estimates for these numbers are difficult to make and may still result in an unfeasibly huge state space. Also, the requirements for the system may change making a new verification for even greater numbers necessary.

The second approach is to create a finite abstraction of the system, and show that the required safety properties hold regardless of the actual size of the state space.

It is clear that a successful application of the second option offers a more complete solution to the verification problem. To achieve this goal, we propose a verification technique for graph transformation systems that can check safety conditions for state spaces of arbitrary size. We base our approach on an abstraction technique taken from the concept of shape analysis for programs [3].

Summing up, our contributions in this paper are that we:

- describe the application of the summarization technique used in shape analysis to graph transformation systems (Section II),
- show how this can be used to obtain a verification process for graphical behavior specifications. This enables the checking of safety properties even for arbitrarily large systems (Section III), and
- explain how domain knowledge can be used directly, even automatically, to increase the precision of the abstraction.
The intention of this paper is to provide a comprehensive overview of our abstraction techniques. For technical details of the abstraction and its maintenance, we refer the interested reader to the technical report [4].

II. ABSTRACTION TECHNIQUE

The modeling technique used in this paper uses simple, directed, edge-labeled graphs to represent the objects in a given system, as well as their relations (Fig. 1 for an example).

The key idea leading to our abstraction technique is the observation that such graphs are remarkably similar to the objects on which the summarization technique in shape analysis is based, so-called logical structures. Essentially, these structures model sets of memory locations and interrelate them with named pointer fields, encoded using logical predicates.

Through a simple one-to-one mapping, our graphs can be converted into logical structures and vice-versa. Thus we can utilize the semantics of summarization in shape analysis to obtain abstract graphs.

As a consequence, each triple of \((\text{node}, \text{label}, \text{node})\) describing a possible edge in a given abstract graph is assigned one of the logical values 0 (false), 1 (true) and \(\frac{1}{2}\) (maybe). A 1-edge is an existing edge, a 0 edge does not exist. An edge that is assigned the value \(\frac{1}{2}\) stands for an edge that may or may not be there in a concrete graph represented by the abstract graph. Such edges are depicted using dashed lines. There are also summary nodes which represent an arbitrary, positive number of nodes. These, too, are drawn using dashed lines.

Using this simple initial idea, we abstract from the actual length of the track connections in our example, yielding the abstract graph shown in Fig. 2.

The formal semantics for such an abstract graph, called a shape can easily be obtained from those of an abstract structure in shape analysis. Intuitively, a shape \(S\) can be considered an abstraction of a graph \(G\) iff there is a surjective mapping \(f : G \to S\) such that

- whenever a node \(v\) in \(S\) is the image of more than one node in \(G\) under \(f\), then \(v\) must be a summary node, and
- for any pair of nodes in \(G\), the edge values between their images (in \(S\)) must be the same or more abstract than the edge values in \(G\), i.e. (un-)connected node must remain (un-)connected, unless they are connected by a \(\frac{1}{2}\)-edge in \(S\).

For our example we can construct such a function \(f\) by mapping the track sequences in Fig. 1 to the summary tracks that replace them in Fig. 2.

So far, we could more or less directly utilize the work done in [3] on shape analysis, making only minor adjustments to apply the principle to generic graphs. The challenge lies in finding a way to make the abstraction compatible with the dynamics of a graph transformation system, and integrating that into a model driven design process, as the next section will show.

III. VERIFICATION PROCESS

The starting point for our verification algorithm must be a graphical behavior specification for a system under construction. To this end, we adopt the UML-based behavior specification technique of MechatronicUML (MUML), used by e.g. Becker et al. in [1]. To illustrate this, Fig. 3 shows an example of a rule diagram in their modeling language. It describes the movement of a single RailCab along a track.

The crossed-out parts of the graph specify a so-called negative application condition (NAC). This means that the rule can only be applied if the NAC can not be found in the graph, i.e. the RailCab can only move forward if there is not already a RailCab there. Such diagrams can easily be converted into the graph production rules we use.

The next step is to model an initial state of the system. At this stage, the disadvantages of explicit modeling first become apparent. If we intend to translate our system into a regular graph transformation system, we need to specify a finite set of initial states. For many systems, there may be a great number of valid initial states, necessitating either a huge modeling effort or imposition of deployment restrictions for the system. In other cases, there may be no clearly defined initial state at all.

In contrast, using our shape analysis approach, we can model an initial shape which represents an infinite set of initial states, or, put differently, expresses the necessary amount of

Figure 2. A shape abstraction of the graph in Fig. 1

Figure 3. An example for a rule modeled in MUML.
Here we encounter a problem with the abstraction. Since our shapes explicitly hide structural details through summarization, it is unclear in most cases whether a rule is applicable to a given shape or not. Consider, for example, a rule \( P \) specifying the movement of a RailCab onto a switch and the shape \( S \) depicted in Fig. 2. Clearly, in the situation we intended to model (cf. the state graph in Fig. 1), the rule would not be applicable to either of the RailCabs present in the system. Yet, in the abstraction, application to \( r_1 \) seems possible, as shown in Fig. 5. Since the introduction of summarization forced us to set the values of the \( on \)-edges to \( \frac{1}{2} \), as well as to replace the tracks with a single summary track, we cannot be certain that the edges and nodes that are part of the match do or do not exist in all concrete models represented.

To be more precise, the shape \( S \) represents system states where the rule would be applicable, as well as other system states where it would not be. We need to distinguish between these two cases. This is the task of \textit{materialization}.

The term materialization is taken from [3], where it describes selective concretization of up to two concrete memory locations out of an abstracted memory location enabling the analysis of the effect of a given set of program statements. For our approach, we extended the materialization process to correctly handle arbitrary graph production rules. Since materialization is at the core of our verification technique, the following paragraphs will explain in more detail how this can be achieved.

Given a production rule and a match for it in a shape, the goal of materialization is to replace the shape with a set of more concrete shapes that allow us to precisely distinguish between all possible cases that are relevant to the application of the rule. Thus, we need to replace the shape \( S \) with a new set of shapes \( S_m := \{S_m^1, \ldots, S_m^k\} \), such that

- a graph on which \( P \) can be applied is represented by one of the shapes in \( S_m \) iff it is represented by \( S \), and
- the rule \( P \) is applicable to each shape in \( S_m \) and its match does not contain \( \frac{1}{2} \)-values

Since the cases where the rule is not applicable do not require any further action on our part, we do not need to construct materializations for those cases. Once \( S_m \) has been computed, we can apply the rule as usual and proceed with the construction of the state space. All that remains now is to compute \( S_m \).

We first observe that, if our initial match for the rule does not contain summary-nodes, replacing \textit{maybe}-edges with \textit{1}-edges as needed would yield a shape representing exactly the cases where the rule is applicable.

The situation is a bit more complicated with summary nodes. Since we do not know how many nodes these represent, for each summary node in the match there are essentially two cases.

1. The summary node represents exactly as many nodes as the match requires.
2. The summary node represents more nodes than the match requires.

The case where the summary node represents less nodes than the match requires is an instance of the case where the rule is not applicable and can thus be ignored (see above).
We will begin with the first, simpler case. Here we need to replace the summary node with the nodes from the rule that were matched to it. In order to preserve a valid abstraction, these materialized nodes must inherit the edge values from the summary node they were materialized from. Fig. 6 illustrates this.

The second case is similar, but results in a larger shape. Since the summary node represents more nodes than we need for our match, we need to account for those nodes in our materialization. This means that we need to preserve the summary node. Just as in the first case, all materialized nodes and the remaining summary node inherit their edge values from the original summary node. This potentially leads to a lot of edges, as Fig. 7 shows.

This case distinction must be made for each summary node that is part of our match. In other words, we get $2^n$ shapes in $S_m$, where $n$ is the number of summary nodes in the match. Since most rules contain no more than a few nodes and many of the produced shapes are later absorbed into already existing shapes, the effects of this exponential growth are usually limited.

When all cases have been computed for a given match, the resulting set $S_m$ is guaranteed to represent exactly those graphs represented by the original shape that allow the application of the rule at the match $m$. There is a comprehensive proof of this property available in [7], [4].

Using matching, materialization and rule application we can now apply graph production rules to shapes, yielding new shapes. Checking for embedding relations between the shapes we produce enables us to discard shapes that contain information already encoded in more abstract shapes. Thus, if the initial shape was modeled carefully, the algorithm computing the state space will eventually terminate.

Once the algorithm has terminated and has constructed the state space, determining whether the forbidden pattern occurs in one of the produced shapes is a simple matter. There are three different possible outcomes.

1. The forbidden pattern is definitely absent in all produced shapes.
2. The forbidden pattern is definitely present in at least one produced shape.
3. The forbidden pattern is maybe present in at least one produced shape.

In the first case, we can conclude that the forbidden pattern is also absent in any graph transformation system consisting of the same rules as our shape transformation system and using an initial graph that can be embedded into our initial shape. This is one of the main results for our approach so far. A proof for this claim can be found in [6].

In the other two cases, the forbidden pattern may or may not be present in the original graph transformation system. The sequence of rules and matches necessary to derive the shape containing the forbidden pattern gives us a starting point in verifying the counterexample on the concrete level. This counterexample verification is not trivial and will be the topic of future work, possibly building on previous work in this area, e.g. [8]. If we find that the counterexample is not present in the original system, we need to improve our abstraction and restart the process. This is the CEGAR approach, introduced in [9]. The next section will provide the means for doing that.

**IV. INCREASED PRECISION THROUGH DOMAIN KNOWLEDGE**

The verification algorithm, as we have presented it so far, is technically sound. However, it is not very precise. This can be easily seen e.g. by looking at the node $t_0$ in Fig. 7. Here we have a summary track with a summary $next$-loop. By our embedding definition, this means that the substructure represented by $t_0$ can take literally any form, as long as it consists of tracks connected via $next$. We also see that through production application, we get information that is more abstract than it needs to be. For example, the RailCab $r_1$ is definitely on the track segment $t'_1$. Since a RailCab can only be on one track segment at any given time, the $\frac{1}{2}$-edge from $r_1$ to $t_1$ is superfluous.

To remedy this situation we intend to use domain knowledge to increase the precision of our abstraction. As we explained in section III, our verification algorithm is used in the context of a UML-based model driven design process. Thus, we have access to a number of diagrams that contain information about the interrelation between types of objects, e.g. UML class diagrams. As an example, consider the the class diagram show in Fig. 8.

The information contained in the multiplicities invalidates many of the possible concrete graphs embedded in the shape in Fig. 7. In order to utilize this information, we now adopt
There exist methods for the automatic generation of update formulae for instrumentation predicates, detailed e.g. in [11]. These techniques work well on the program level. Deriving instrumentation update formulae for graph production rules via this method is non-trivial and might be prohibitive for large numbers of complex rules. Furthermore, the finite differencing approach in [11] does not offer the additional precision that shape constraints do. Therefore we do not employ finite differencing on the graph or shape level. However, a more elaborate comparison to finite differencing might be the subject of future work.

V. RELATED WORK

Since abstract graph transformation as a tool for the verification of infinite-state systems is in itself not a new idea, several other approaches have been developed to reach goals similar to ours. In the following, we compare our approach with those that seem closest to it.

In [1] Giese, Schilling et.al. propose an approach for the verification of arbitrarily large state system in the context of dynamic mechatronic systems. Introducing the concept of an inductive invariant of freeness of given forbidden patterns, their approach determines whether any valid graph pattern can be made invalid by a single rule application. If this is not the case, no forbidden pattern can ever occur. In contrast to operational invariants, as considered in our paper, inductive invariants therefore do not take any starting graph into account. While this property of inductive invariants imposes the advantage that no starting graph or set of starting graphs need to be supplied by the user, it may also lead to spurious counterexamples: a operational invariant of a system is not necessarily also an inductive invariant. In contrast, our approach can potentially support both cases - single starting graphs and shapes which represent infinite sets of starting graphs.

Another approach to verify arbitrarily large systems is described in [12]. There, a method for the automatic abstraction of graphs is introduced. Intuitively speaking, nodes are summarized if their neighborhood of radius \( k \in \mathbb{N} \) is the same. While this automatic abstraction greatly reduces the need for human intervention and the corresponding modal logic is preserved in the abstraction and guarantees termination, the approach does not provide much flexibility. While in our approach we can tune the abstraction very precisely using instrumentation predicates, the approach of [12] may only increase the precision along two pre-defined dimension in discrete steps. Also, there are numerous classes of graphs that do no abstract well with neighborhood abstraction, meaning that a very high \( k \) is needed to prove relevant properties. In our approach, abstraction refinement can take the structure of the entire counterexample into account but requires more human intervention.

Lastly, Baldan, Corradini and König [13] have written a series of papers in which they develop a unique approach to the verification of arbitrary large state GTSs. They relate GTSs to Petri nets and construct a combined formalism, called a petri
graph, on which they show certain properties via a technique called unfolding. This powerful method can be used to verify a great number of interesting properties about graphs, yet still it suffers from various restrictions, such as the lack of negative application conditions.

VI. CONCLUSION

The verification process described in this paper enables us to show a wide variety of safety properties for systems that induce arbitrarily large state spaces. This solves a problem that emerges in many real-world situations, such as the presented RailCab-system, but also in other cases, like generic multi-agent systems, dynamic communication protocols or software systems manipulating unbounded data structures.

Using graphs as models for our approach places us close to established modeling languages like Mechatronic UML, making integration into actual model driven design processes easy. The integration and adaptation of the basic concepts of shape analysis for our approach gives us a very high degree of flexibility in the abstraction as well as a well-understood logical foundation.

This high degree of flexibility in the abstraction also serves us well when encountering spurious counterexamples. We are capable of using all of the information in the spurious counterexample to construct additional instrumentation which then eliminates the counterexample from the analysis.

We have implemented our verification algorithm based on the logic engine TVLA. In addition to the features presented here, the implementation supports additional features, like the shape constraints described in [7]. We have tested our implementation on numerous example, with promising results.

Our verification approach is under continuous development. Work currently in progress includes the termination of our analysis (probably at the cost of precision) as well as an automatic, canonical abstraction process, further reducing the need for manual intervention.

REFERENCES


