1 Graph transformation execution semantics

This section describes one possible semantics to use in our approach to behavioral consistency management. We will use graph transformation semantics based on graph grammars, see Definition 1.

Definition 1 (Graph grammar)**.** *A graph grammar $GG = (S, P)$ consists of a start graph S and a set of production rules P [2].*

The idea of a graph grammar is to begin with a start graph and then continuously apply all possible production rules. Thus, one obtains a state space where each state is a graph, and each transition is a rule application. Rules can be applied using different approaches, such as double-pushout (DPO) [2] or single-pushout (SPO) [4]. Production rules for DPO are defined in Definition 2. Informally speaking, elements in R but not in L are added by a rule, while elements in L and R are preserved, and a rule deletes elements that are in L but not in R.

Definition 2 (Production rule)**.** *A production rule $P = L \xleftarrow{l} K \xrightarrow{r} R$ consists of graphs L, K, R, and graph morphisms $l : K \to L$ and $r : K \to R$ [2].*

The first step to generate global execution semantics is to generate a set of production rules for each behavioral model contained in the multi-model. Each set of rules must describe the behavior of the given behavioral model by manipulating instances of the snapshot metamodels. For example, a production rule for a transition in a state machine changes the current state of a corresponding state machine snapshot from the source to the target state of the transition.

Thus, we need a *model transformation* to a set of graph transformation rules for each behavioral language. However, each model transformation can be implemented once by an Model-driven engineering (MDE) tool developer and made available, for example, as a plugin, such that it can then be reused in any future setting the language is needed. In addition, each model transformation must keep traces of the generated set of rules. Concretely, it has to save which rules originated from which state-changing elements in the behavioral model. Coming back to the state machine example, we must know which transition results in which production rule. In general, multiple rules may be associated with one state-changing element of a behavioral model. For example, a receive task in a Business Process Modeling Notation (BPMN) process cannot be represented by only one rule, since it starts and then waits for an incoming message before finishing.

We apply the following steps to merge the obtained rules into rules describing the global execution semantics.

1. All rules generated from state-changing elements not part of interactions remain unchanged and are added to the global rule set.

2. For each interaction, we do the following:

   (a) Find the corresponding production rule[1] P_0 for the interaction host of the interaction and find the rules $P_1, P_2, \ldots P_n$ for the state-changing elements taking part in

[1] If one state-changing element results in more than one rule, one can define a strategy to pick the appropriate rule. For example, if one wants to synchronize with the start of a BPMN task, one can use a strategy to pick that rule.

the interaction using the saved traces.

(b) Calculate the parallel production [1, Definition 3.2.7] $P_0 + P_1 + \ldots + P_n$ for the found rules. Intuitively, applying a parallel production can be seen as applying multiple rules simultaneously, i.e., synchronizing the state changes of the behavioral models.

(c) For each part of the interaction, add a behavioral relationship from the behavioral model in P_0 to the behavioral model in P_i, for $0 < \mathrm{i} \leq \mathrm{n}$. The behavioral relationship is essentially a copy of the behavioral relationship referenced by that interaction part.

Together with a system configuration, the set of global rules results in a graph grammar generating a state space. Each state in this state space is an instance of the global snapshot metamodel. Consequently, this can be used to check the behavioral consistency of the multi-model.

We will now explain how the graph transformation execution semantics are generated for the use case. To apply our approach to the use case, we need to define model transformations from state machines and BPMN processes to graph transformation rules.

1.1 State machine semantics

The generation of production rules to implement finite state machine execution semantics is straightforward. Each transition leads to a production rule. For example, Figure 1 shows the production rules for the transitions turn red-amber and turn green of the traffic light model. It uses the concrete syntax introduced in the paper to depict state machine snapshots and their current state. We depict a production rule by showing its graph L on the left, pointing with a named white arrow to its graph R on the right.

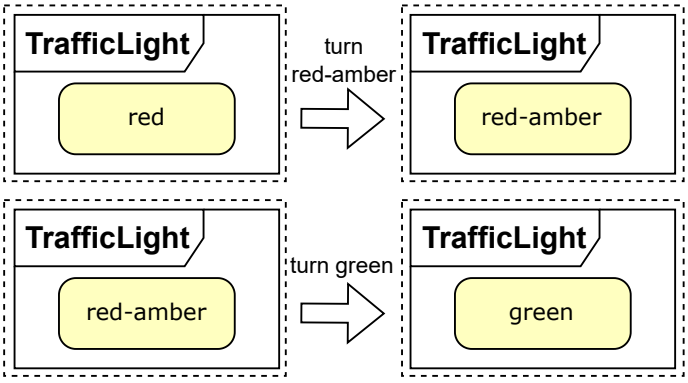

Figure 1: Production rules for *turn red-amber* and *turn green*

Using a traffic light snapshot with the state red as a start graph results in a graph grammar[2], which will generate the same state space as the traffic light state machine. We

[2]All generated graph grammars, including further instructions regarding execution and consistency checking, can be found at `https://github.com/timKraeuter/Towards-behavioral-consistency-in-multi-modeling`.

evaluated our approach by executing the graph grammar with the graph transformation tool Groove [3, 5]. However, in the future, we can also experiment with other graph transformation tools since rule-generation can be adapted without much effort.

1.2 BPMN semantics

The generation of production rules for BPMN processes is challenging, and we are currently only supporting a subset of the BPMN semantics, similar to the extent supported by [6]. Generally, we construct one or more rules for each flow node contained in a BPMN process. Figure 2 shows production rules for some flow nodes of the T-Junction controller. It uses the concrete syntax introduced earlier to represent process snapshots containing tokens.

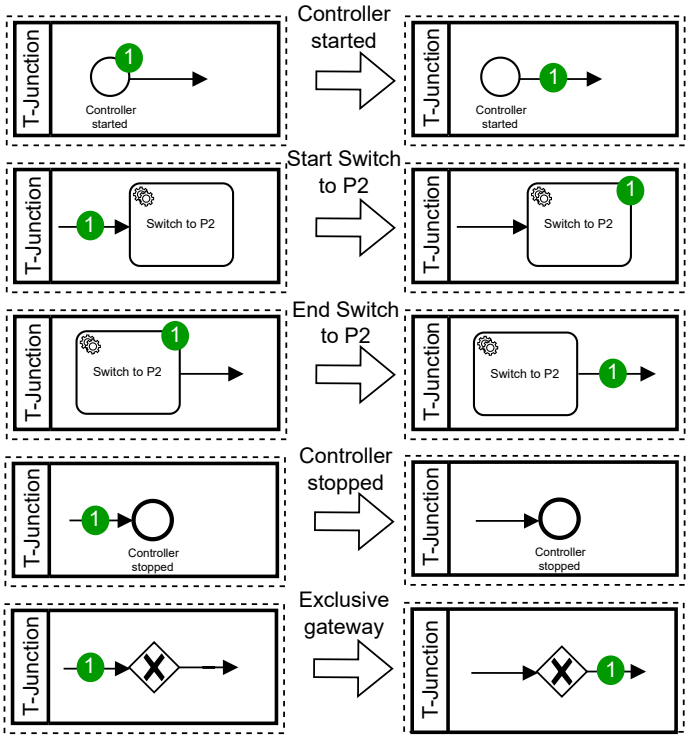

Figure 2: Production rules for the T-Junction controller

However, we cannot show all possible rules resulting from our BPMN semantics implementation. Especially the implementation of signal events, message events, event-based gateways, and subprocesses needs further explanation but does not matter for the overall approach presented in this paper, as long as it conforms to the BPMN semantics. We compared the state space resulting from our implementation with the possible token flows realizable using the *bpmn-js token simulation*[2], which claims to conform to the BPMN semantics. This resulted in finding bugs in the bpmn-js simulator and our rule generator.

Our semantics is inspired by [6] but differs in some aspects. One key difference is that they define a fixed rule set that describes the semantics of BPMN models, while we generate a specific rule set for each model to enable interaction with specific parts of each model.

Following this procedure, we can generate production rules that implement the behavioral semantics of the BPMN models used in the use case[2].

1.3 Global execution semantics

The interactions we have defined change the rules for switching to phases 1 and 2. Figure 3 shows the resulting rule for switching to phase 1. We decided that the interactions between the traffic lights and the T-Junction controller should synchronize with the end of the task, not the start.

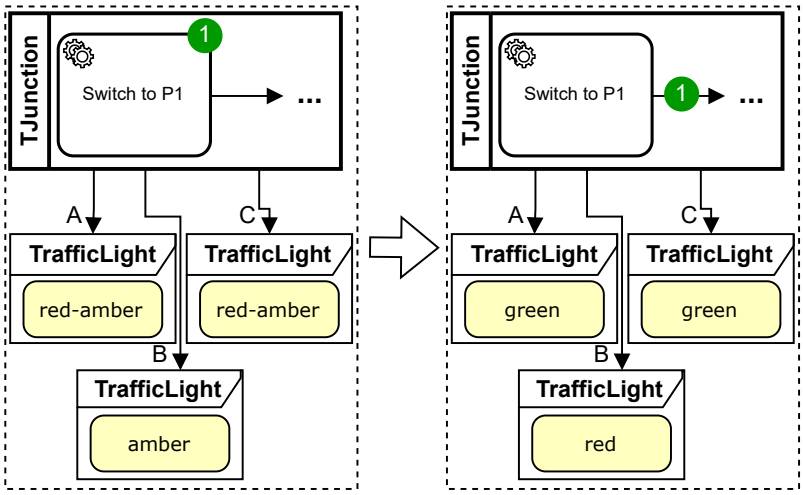

Figure 3: Parallel production rule to switch to phase 1

The rule changes all traffic lights simultaneously and finishes the task. The corresponding individual rules do not exist anymore, such that a synchronization of the behavior is guaranteed. A similar rule exists for switching to phase 2, resulting from the second interaction. All other rules are left untouched during the construction of the merged graph grammar.

Finally, we can generate the global state space of the system using the merged graph grammar[2].

1.4 Global consistency checking

To check the requirements formalized by properties 1-4, we must ensure the generated state space contains the needed atomic propositions. We formulate all atomic propositions as instances of M_s^+ and implement them as graph conditions in Groove. Graph conditions in Groove are rules which do not change any elements and can be used for model checking.

Properties 1 and 2 hold, while properties 3 and 4 do not hold[2]. The counterexamples for properties 3 and 4 show an unexpected race condition that must be handled. After the T-Junction controller signals that the traffic light A/B is green, bus B1/B2 can advance to the Pass Junction activity. However, at the same time, the T-Junction controller can enter the subprocess for the next phase, which can be interrupted by the associated timer event. This can happen before the bus B1/B2 passes the junction, resulting in an invalid state.

The system developers have different options to handle the uncovered inconsistencies. One option is to keep the models unchanged and pay special attention to the found race condition during system implementation. This can be an acceptable solution since the Pass Junction activity is also modeled as a user activity, i.e., the bus driver decides when to cross the T-Junction. Furthermore, tolerating inconsistencies can be a viable option in MDE [7].

Another option is to change the models until the inconsistency is resolved. For example, the T-Junction controller could wait for the bus to pass before changing the traffic lights again. Currently, we are using discrete-time execution semantics for the use case. However, it already shows that continuous-time execution semantics might be needed in some scenarios.

References

[1] Paolo Baldan et al. "Concurrent Semantics of Algebraic Graph Transformations." In: *Handbook of Graph Grammars and Computing by Graph Transformation*. Vol. 3. World Scientific, Aug. 1999, pp. 107–188. DOI: 10.1142/9789812814951_0003.

[2] H. Ehrig et al. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 3-540-31187-4.

[3] Amir Hossein Ghamarian et al. "Modelling and Analysis Using GROOVE." In: *International Journal on Software Tools for Technology Transfer* 14.1 (Feb. 2012), pp. 15–40. ISSN: 1433-2779, 1433-2787. DOI: 10.1007/s10009-011-0186-x.

[4] Michael Löwe. "Algebraic Approach to Single-Pushout Graph Transformation." In: *Theoretical Computer Science* 109.1-2 (Mar. 1993), pp. 181–224. ISSN: 03043975. DOI: 10.1016/0304-3975(93)90068-5.

[5] Arend Rensink. "The GROOVE Simulator: A Tool for State Space Generation." In: *Applications of Graph Transformations with Industrial Relevance*. Ed. by John L. Pfaltz, Manfred Nagl, and Boris Böhlen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 479–485. ISBN: 978-3-540-25959-6.

[6] Pieter Van Gorp and Remco Dijkman. "A Visual Token-Based Formalization of BPMN 2.0 Based on in-Place Transformations." In: *Information and Software Technology* 55.2 (Feb. 2013), pp. 365–394. ISSN: 09505849. DOI: 10.1016/j.infsof.2012.08.014.

[7] Nils Weidmann, Suganya Kannan, and Anthony Anjorin. "Tolerance in Model-Driven Engineering: A Systematic Literature Review with Model-Driven Tool Support." In: *arXiv:2106.01063 [cs]* (June 2021). arXiv: 2106.01063 [cs].