Towards behavioral consistency in multi-modeling

Tim Kräuter[*], **Harald König**[*†], **Adrian Rutle**[*], **Yngve Lamo**[*], **and Patrick Stünkel**[‡]
[*]Western Norway University of Applied Sciences, Bergen, Norway
[†]University of Applied Sciences, FHDW, Hannover, Germany
[‡]Haukeland Universitetssykehus, Bergen, Norway

ABSTRACT Multiple interacting systems are needed to realize the requirements of complex domains. Describing the interactions between these systems and checking their global behavioral consistency is a general, well-known challenge in software engineering. To address this challenge, model-driven software engineering utilizes abstract representations of the constituting systems and their interactions, resulting in a *multi-model* representing the overall software. In such a multi-modeling setting, global consistency rules must be satisfied by a set of heterogeneously typed models to guarantee a desired *global behavior*. In this paper, we propose a novel approach for behavioral consistency management of heterogeneous multi-models. The approach introduces a workflow in which we (i) align the individual models and specify their *interactions*, (ii) generate a *global execution semantics* for the multi-model, and finally, (iii) define and check *global properties* which should be satisfied by the multi-model. Although our approach is independent of a particular formalism as an underlying global execution semantics, the current implementation utilizes graph transformations for this purpose.

KEYWORDS Global behavioral consistency, Consistency verification, Multi-modeling, Graph transformation, Heterogeneous models

1. Introduction

Model-driven engineering (MDE) addresses the increasing complexity of software systems by employing models to describe the different aspects of the system. In this way, MDE promotes a clear separation of concerns and raises the abstraction level throughout the entire development process (France & Rumpe 2007). These models are then used to generate portions of the system leading to an increase in productivity and reduction of errors (Brambilla et al. 2017). As multiple interacting systems are needed to realize the requirements of complex domains, a set of corresponding models would be needed to represent these systems and their interactions. Such a collection of inter-related models is referred to as a *multi-model* (Boronat et al. 2009; Stünkel et al. 2021), which is usually heterogeneous, meaning it consists of models conforming to different modeling languages. Models in a multi-model contradicting each other can lead to problems during development, system generation, and system execution. Consequently, continuous multi-model consistency management during the development process is a significant issue for multi-models (Spanoudakis & Zisman 2001; Cicchetti et al. 2019).

Recent research describes methods to check the structural consistency of a multi-model (Stünkel et al. 2021; Klare & Gleitze 2019). Structural models, like UML class diagrams, describe structural aspects of systems, i.e., domain concepts and relations between these concepts. This is usually referred to as the static semantics of the software system as it only describes the set of valid instances or states of the system. Nevertheless, approaches to multi-model consistency management must also include a means to maintain *behavioral consistency* since behavioral models, like Business Process Modeling Notation (BPMN), are associated with execution semantics describing dynamic aspects of the system (Object Management Group 2017, 2013).

Several approaches exist for checking the consistency of pairs of behavioral models. For example, consistency checking for sequence diagrams and statecharts was implemented using Petri nets (Yao & Shatz 2006) and Communication Sequential Processes (CSP) (Küster & Stehr 2003). Moreover, some

JOT reference format:
Tim Kräuter, Harald König, Adrian Rutle, Yngve Lamo, and Patrick Stünkel. *Towards behavioral consistency in multi-modeling*. Journal of Object Technology. Vol. vv, No. nn, 2022. Licensed under Attribution 4.0 International (CC BY 4.0) http://dx.doi.org/10.5381/jot.2022.vv.nn.aa

approaches for model simulation in heterogeneous scenarios have been developed, such as Ptolemy (Eker et al. 2003; Lee 2010) and GEMOC studio (Deantoni 2016; Vara Larsen 2016). However, current approaches either only allow for consistency checking in a binary case or do not allow for definition and checking of global behavioral properties.

We propose a novel approach for consistency management of heterogeneous multi-models, which allows us to define and check *global* behavioral properties. Our approach allows specifying *interactions* between potentially heterogeneous behavioral models, which are in turn used to generate global execution semantics. Although our approach is independent of a particular formalism as an underlying global execution semantics, the current implementation utilizes graph transformations for this purpose (see section 4).

The remainder of this paper is structured as follows. We introduce a simplified use case (section 2) before explaining our behavioral consistency management approach in detail (section 3). Afterward, we show how our approach can use graph transformation as execution semantics (section 4). Finally, we discuss related work in section 5 and conclude in section 6.

2. Use Case

This section motivates our approach by a simplified use case in which a traffic management system is developed to guide the traffic at a T-Junction with three traffic lights. It should lead the traffic by switching between the two traffic phases highlighted in Figure 1. In addition, the system must fulfill the following two requirements. First, it must guarantee safe traffic by changing the three traffic lights, A, B, and C, correctly. Second, it should prioritize arriving buses, i.e., switch the traffic lights quicker than usual to let an approaching bus pass (early green). This so-called bus priority signal is a widely implemented technique to improve service and reduce delays in public transport.

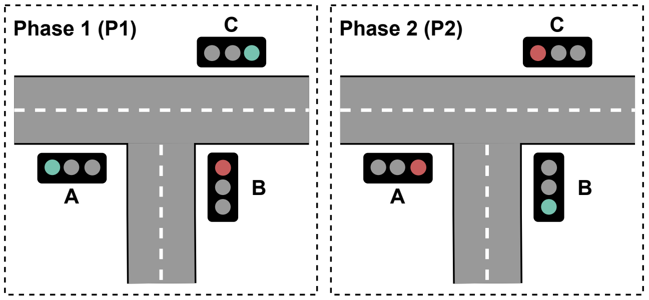

Figure 1 Traffic phases of a T-Junction

To develop the behavior of the traffic management system, we follow an MDE approach. First, we model the behavior of a traffic light as a Unified Modeling Language (UML) state machine, and then we use BPMN to model the different traffic phases of the T-Junction, including the prioritization of approaching buses.

Using different behavioral modeling languages in the use case has two reasons. First, two software development teams might work on the system in parallel but prefer different modeling languages. Second, each team is free to choose the most appropriate modeling language for defining their part of the system. In this use case, the behavior of a traffic light and a T-Junction differs significantly in complexity and requirements, resulting in the use of two different behavioral modeling languages, namely UML state machines, and BPMN.

The behavior of a traffic light is straightforward since it uses only three colors to guide the traffic. Figure 2 shows the typical European[1] traffic light that switches from red to red-amber, green, amber, and back to red. The start state of the traffic light in Figure 2 is red but can be any of the four possible states.

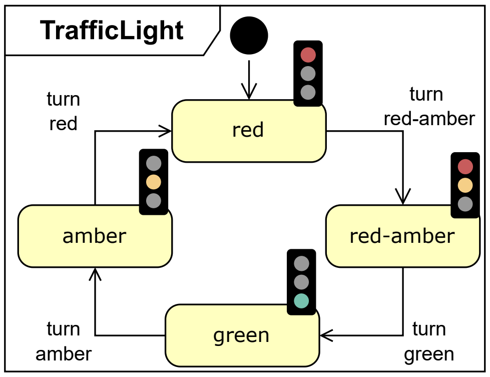

Figure 2 Traffic light state machine model

However, the T-Junction's behavior is more complex since it should coordinate the three traffic lights and communicate with approaching buses to implement bus priority. Consequently, we are using BPMN to model this aspect of the system's behavior since we can utilize BPMN message and signal events to implement the communication with approaching buses.

We model two processes, one for the T-Junction and the Bus. Each process is modeled in its own BPMN *pool*. A pool is depicted as a vertical lane with a name on the left and allows to draw message flows (arrows with dashed lines) between two different pools.

Figure 3 shows how a possible controller for a T-Junction behaves in the traffic management system[2]. When a T-Junction controller is started, we assume that the traffic lights are showing the colors according to phase 1 (see Figure 1). Thus, the controller enters a subprocess called phase 1, which we describe later. However, when a fixed amount of time has passed, the subprocess is interrupted by the attached timer boundary event. Then, the controller executes the next activity and switches to phase 2. The controller will pass a throwing signal event before entering a subprocess for phase 2 and repeat the same steps. This signal event represents a broadcast to all buses waiting for traffic light B to become green. After switching back from phase 2 to phase 1 and signaling that traffic lights A and C are green, the controller can stop or execute the described steps again. Typically, the controller does not stop, which is indi-

[1] Traffic lights in other parts of the world might not show red and amber simultaneously before switching to green.

[2] All models and their source files can be found at https://github.com/timKraeuter/ECMFA-2022.

cated by the default sequence flow going back to the process beginning.

Figure 4 shows the communication of a bus with the subprocess phase 1. The BPMN model and communication for phase 2 of the controller can be defined accordingly [2].

The phase 1 model uses an event-based gateway to respond to two different kinds of messages. First, the traffic light status can be requested, which is answered by sending a message declaring that the traffic lights A and C are green while B is red. Moreover, early green for traffic light B can be requested. This request ends the subprocess, and the controller immediately switches to phase 2 (see Figure 3), which results in the traffic light B turning green.

The bottom of Figure 4 shows the controller for a bus parameterized with direction B. It will first request the traffic light status to determine if traffic light B is green. If it is green, the bus can pass the junction. However, if it is red, the bus requests to change B to green and waits for a signal that the controller has changed the traffic light. After receiving the signal, the bus passes the junction. In addition, the bus controller also communicates with the phase 2 subprocess, which we only hint at in Figure 4[2]. A BPMN model for a bus controller parameterized with the direction A or C looks nearly identical.

Having developed behavioral models for the system, we want to check the previously stated *safe traffic* requirement while buses are prioritized. We can lower the overall development cost if we find bugs related to these requirements as early as possible during system development. However, the traffic light model is currently not related to the T-Junction and bus models, which is a problem since the T-Junction is supposed to control the traffic lights, for example, when it switches between the two traffic phases. In addition, the system has to manage multiple slightly different instances of the behavioral models. For example, there are three traffic lights at one T-Junction starting in different states, i.e., showing different colors and buses approaching the T-Junction from one of the three directions. Consequently, we need a model of the system to allow us to define interactions between the models and configure instances of the behavioral models contained in the multi-model.

The resulting model called the *system relationship model* is shown in Figure 12 using a class diagram-like syntax. It contains one class for each behavioral model and associations to depict behavioral relationships, leading to possible interactions. In addition, it contains enumerations to parameterize the behavioral models.

A TJunction has three associated TrafficLights, A, B, and C, and a set of currently approaching Buses. A TrafficLight has four possible TrafficLightStates and an attribute to define its startState. A Bus has a direction that indicates which TrafficLight of the T-Junction it is approaching.

Finally, using the system relationship model, we can define a test configuration of our traffic management system to check its requirements. Figure 6 depicts the test system configuration, an instance of the system relationship model. First, it contains three instances of the traffic light behavioral model, representing the three traffic lights, **A**, **B**, and **C**. Second, it contains an instance of the T-Junction behavioral model, which is connected to the three traffic lights, and two instances of the bus behavioral model. Thus, the test system configuration describes a system that controls one T-Junction with three traffic lights and two buses approaching from directions A and C.

First, we would like to check the safe traffic requirement. Since we only want to check system conformance concerning the two traffic phases, we do not need to include the two buses depicted in the red dotted square in Figure 6 in the analysis. We cannot simply assert that the system is either in phase 1 or phase 2 since there are intermediate states during the transition between the two phases, which are allowed. By consulting Figure 2, we can, for example, expect a state in which traffic lights A and C are amber, and traffic light B is red-amber before reaching phase 2. However, we can define *safe traffic* as the absence of *unsafe traffic*, which is easier to define.

For the T-Junction, unsafe traffic occurs if traffic light A is green or amber and traffic light B is green or amber simultaneously. In addition, the same state combinations are forbidden for traffic lights B and C. We can formalize the consistency requirements as safety properties in Linear temporal logic (LTL), i.e., states that should never be reached. The resulting global properties 1 and 2[3] are the following:

$$\Box\neg((A_{green} \lor A_{amber}) \land (B_{green} \lor B_{amber})) \tag{1}$$
$$\Box\neg((C_{green} \lor C_{amber}) \land (B_{green} \lor B_{amber})) \tag{2}$$

If we include buses B1 and B2 in the system, we would like to check that they cannot pass when their traffic light is red or red-amber. Concretely, this means the Pass Junction activity should not execute while the corresponding traffic light is red or red-amber. We formalize these requirements again by using LTL safety properties 3 and 4[4].

$$\Box\neg(B1_{passing} \land (A_{red} \lor A_{red-amber})) \tag{3}$$
$$\Box\neg(B2_{passing} \land (B_{red} \lor B_{red-amber})) \tag{4}$$

However, to check the global properties, we must execute the system with the behavior specified in the behavioral models according to the test configuration. This is not straightforward since the multi-model of the use case consists of a system relationship model relating two heterogeneous behavioral models. In addition, the system configuration instantiates two of the behavioral models multiple times with different parameters. Furthermore, we face the problem that the models are not independent of each other. For example, the T-Junction controller must control when the traffic lights A, B, and C switch their states. Thus, if we were to run the models independently in parallel, the properties would be violated.

As far as we can tell, current approaches can simulate a set of interacting heterogeneous behavioral models. However, they do not allow multiple instances of a behavioral model (without duplicating it) nor provide concrete means to define and check

[3] We assume the existence of atomic propositions for each traffic light showing the color green or amber. The propositions are formalized later.

[4] The atomic proposition $B1_{passing}$ / $B2_{passing}$ represents that Pass Junction (see Figure 4) has started but not finished yet, i.e., there is a token in the activity when using BPMN token semantics (Object Management Group 2013).

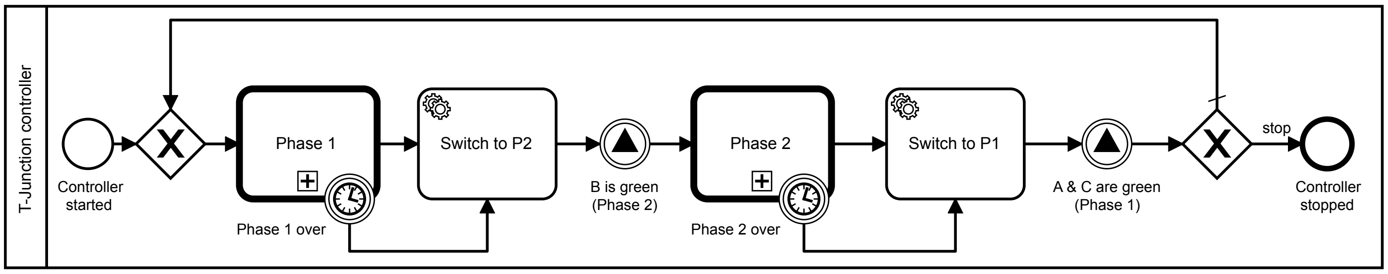

Figure 3 Model for a T-Junction controller

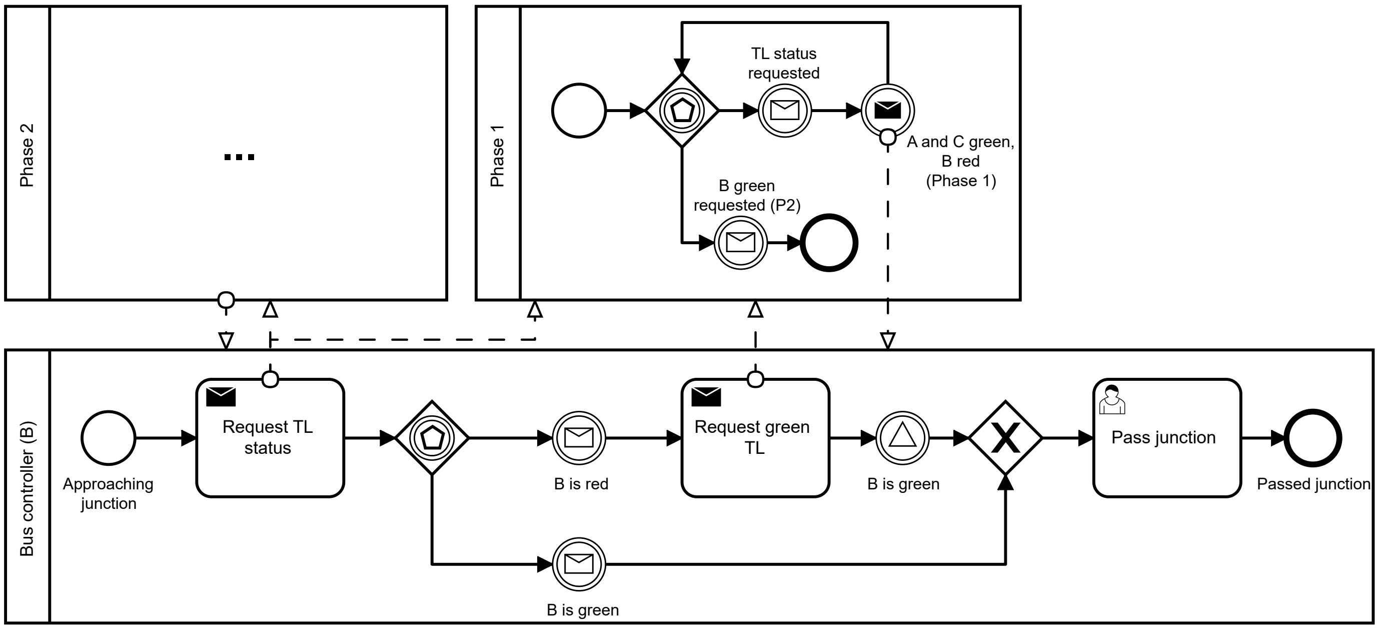

Figure 4 Model for a bus with direction B and its communication with a T-Junction

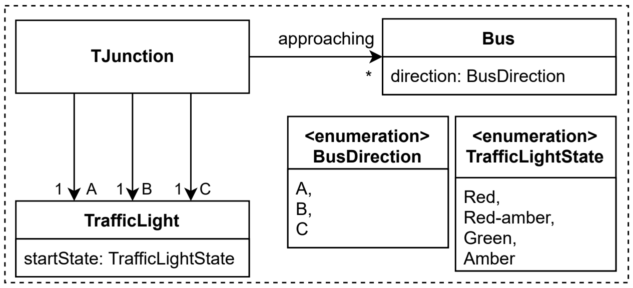

Figure 5 System relationship model of the traffic management system

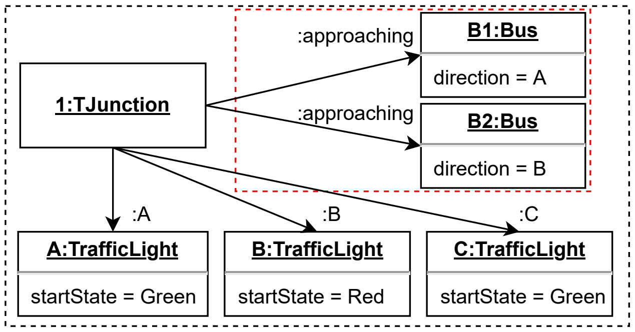

Figure 6 Test system configuration

global behavioral properties. Thus, they are not capable of checking multi-model behavioral consistency. A multi-model is behaviorally consistent if it satisfies all of its behavioral properties. A behavioral property is given in temporal logic, for example, LTL earlier, and is characterized as *local* if it constrains only one model and as *global* if it spans two or more models in a multi-model. Furthermore, global properties are dependent on the system configuration, i.e., the instance of the system relationship model used. In the remainder of this paper, we will describe our approach to address behavioral consistency in multi-modeling and apply it to this use case.

3. Behavioral consistency management

Figure 7 depicts our approach to behavioral consistency management using a BPMN diagram. We leave potential consistency restoration as a problem for future work.

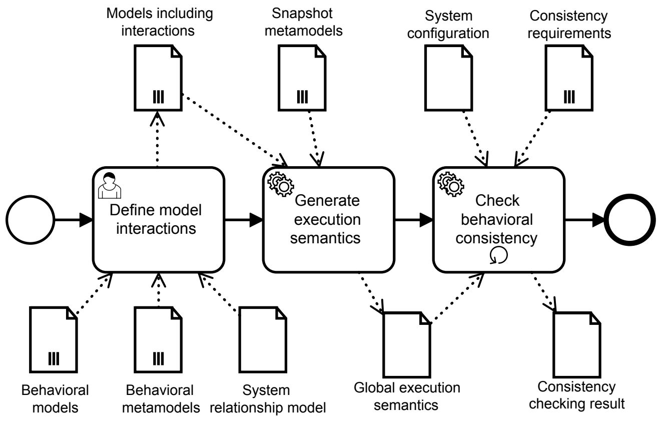
Models including interactions
Snapshot metamodels
System configuration
Consistency requirements
Define model interactions
Generate execution semantics
Check behavioral consistency
Behavioral models
Behavioral metamodels
System relationship model
Global execution semantics
Consistency checking result

Figure 7 Behavioral consistency management workflow

Our approach can be summarized in three steps. First, we define interactions between the behavioral models, which describe parts of a system. Second, based on these interactions and the behavioral models, we generate execution semantics for the global system. Finally, given a system configuration and consistency requirements, we can check these requirements using the generated global execution semantics.

We will now describe our approach in detail by introducing the different model types and explaining each step separately.

3.1. Prerequisites

As prerequisites, we assume a set of behavioral models, including their metamodels, see Figure 8. In addition, we require a snapshot metamodel for each behavioral metamodel used. A snapshot metamodel should represent the structure of a state for a given behavioral language, defined in a metamodel. For example, each state in a Petri net is defined by a set of tokens distributed over its places.

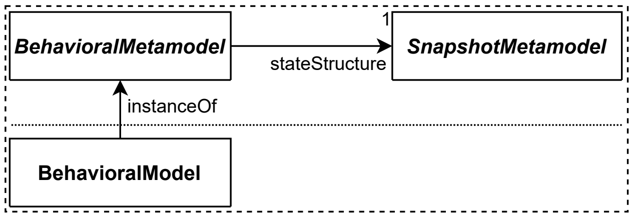

Figure 8 Behavioral metamodels and snapshot metamodels

Furthermore, we will later introduce the system relationship model, which is used to define behavioral relationships between the behavioral models. We will now explain the different model types in detail and give examples of them for the use case.

3.1.1. Metamodels

Each behavioral model must conform to a metamodel (see Figure 8). A metamodel can be defined by a class diagram/grammar for graphical/textual modeling languages. The metamodel ensures that the corresponding models are machine-readable, which is crucial when automating parts of the consistency checking.

The use case utilizes state machine and BPMN models. Figure 9 shows the metamodel for finite state machines (ignore the purple coloring for now). It is defined by a UML class diagram with additional information depicted in clouds. The clouds define a concrete syntax for state machines inspired by UML statecharts. A model in this concrete syntax, such as the traffic light model (see Figure 2), can automatically be translated to abstract syntax and typed in the metamodel.

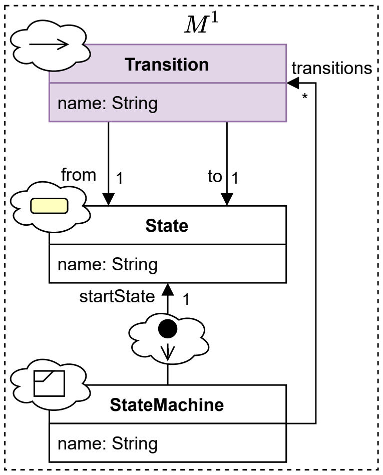

Figure 9 Finite state machine metamodel M^1

A StateMachine has a startState and transitions, whereas each Transition connects two States. The states of a state machine are not explicitly modeled but can be derived from the states connected by the transitions of a state machine.

Similarly, we must define a metamodel for the BPMN diagrams. Figure 10 shows a simplified metamodel for BPMN (ignore the purple coloring for now). Once more, we have attached concrete syntax elements.

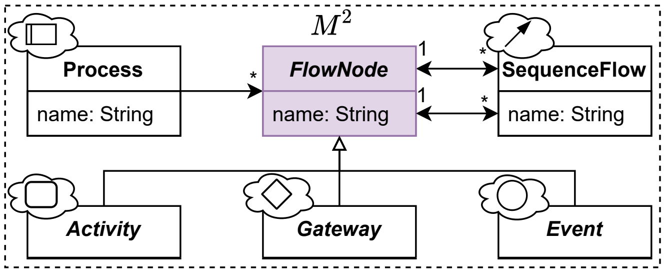

Figure 10 Simplified BPMN metamodel M^2 (Object Management Group 2013)

A BPMN Process contains a set of FlowNodes connected by SequenceFlows. A FlowNode can be an Activity, Gateway, or Event. Possible BPMN Gateways are shown in Figure 11.

Similarly, specific activities and events are defined in the full BPMN metamodel (Object Management Group 2013, Figure 10.6/10.69). The essential activities are manual tasks, service tasks, send/receive tasks, and call activities. In the case of events, start events, message events, timer events, and end events are the most used.

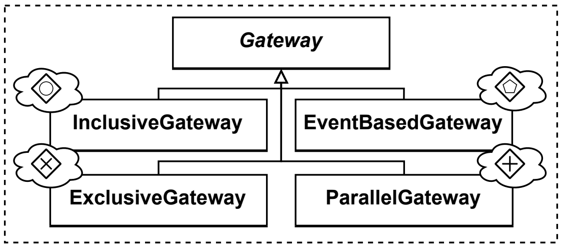

Figure 11 BPMN gateways (Object Management Group 2013)

3.1.2. System relationship model We assume a set of behavioral models describing the system, which *interact* to realize the global system behavior. A system relationship model describes which behavioral models exist in the system and how they are related. The system relationship *metamodel* is depicted in Figure 12.

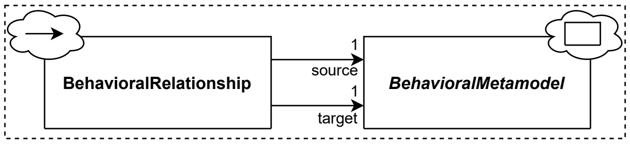

Figure 12 System relationship metamodel (without attributes and enumerations)

Behavioral models typed in BehavioralMetamodels can be connected by BehavioralRelationships to allow for interactions. We are using a UML class diagram-like syntax to define system relationship models, where each class corresponds to a behavioral model (typed in a BehavioralMetamodel), while each association corresponds to a BehavioralRelationship (see concrete syntax depicted in clouds). Thus, a system relationship model does *not* describe the structure of a system but rather behavioral relationships between the behavioral models of a system. The behavioral relationships will be crucial to define interactions between the models in the first step of our approach.

In addition, we allow enumerations and their use as attributes for behavioral models in the system relationship model. This helps not to duplicate behavioral models because we can use the attributes as parameters, for example, to define the start state in a state machine (see Figure 5). Different instances of the system relationship model can be used to analyze the global behavior of *different* system configurations by changing the attribute values and behavioral model instances.

The system relationship model in the use case (see Figure 5) contains three classes corresponding to the traffic light, T-Junction, and Bus behavioral models. A T-Junction has three relationships with the traffic light, which allows the T-Junction to interact with up to three traffic lights. In addition, we can analyze different scenarios by adding buses approaching from particular directions.

3.1.3. Snapshot metamodels The provided metamodels are used to structurally define all possible models by requiring a typing from each model to its metamodel. In addition, we can add constraints to restrict all typed models further.

However, we need additional or different concepts from those in the metamodel to describe the structure of instances of the behavioral models during execution. Consequently, we introduce a new model type called the *snapshot metamodel*, which defines the structure of a state for a given formalism (see Figure 8).

The snapshot metamodel for a state machine is straightforward since a state machine can only be in one state at a time, as shown in Figure 13. Since we are only interested in the current state and not the history of states, we call this a *state machine snapshot*. We are reusing the concrete syntax elements from the metamodel level. In addition, each snapshot metamodel has a root element in our approach, highlighted in light blue in Figure 13.

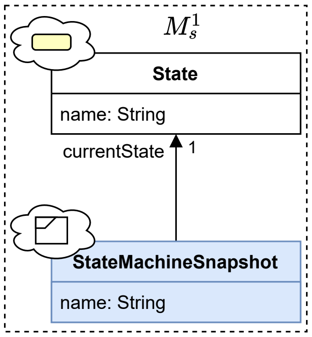

Figure 13 Finite state machine snapshot metamodel M_s^1

We define the snapshot metamodel for BPMN using tokens as suggested by the semantics description in (Object Management Group 2013). Thus, Figure 14 shows that a *snapshot* of a process is a set of Tokens. If the set of Tokens is empty, the state Terminated is derived. Otherwise, the process is still Running. Furthermore, ProcessSnapshots can also have subprocesses with associated Tokens. A Token indicates where it is located in the BPMN using its position attribute. A valid position is a sequence flow or a started but unfinished activity.

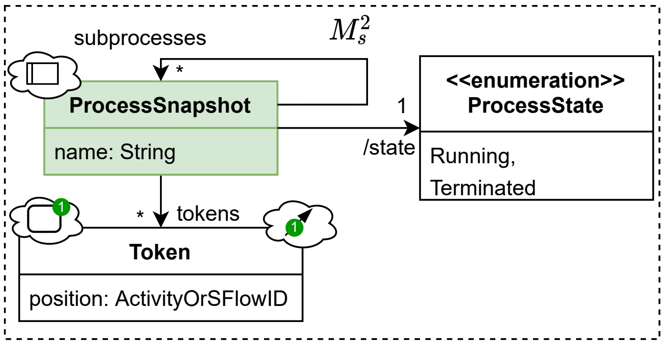

Figure 14 BPMN snapshot metamodel M_s^2

ProcessSnapshots are visualized using the concrete syntax of the underlying process. In addition, Tokens are highlighted with green bubbles in the middle of sequence flows or the top right of an activity. The root of the BPMN snapshot metamodel is the ProcessSnapshot, highlighted in light green.

3.2. Define model interactions

Structural models in a multi-model might contain related information. Thus, current approaches define so-called *commonalities* to explicate these relationships and keep the information consistent (Stünkel et al. 2021; Klare & Gleitze 2019).

To check the behavioral consistency of a system, we are primarily interested in the global system behavior. However, global behavior depends not only on the local behavior of the models but also on their interactions. Consequently, we call these inter-model relationships *interactions* since they carry behavioral meaning while commonalities carry structural meaning (Kräuter 2021).

To specify interactions between different behavioral models, we defined an interaction language given by the interaction metamodel in Figure 15. The interaction metamodel uses concepts from the system relationship metamodel (see Figure 12) since interactions can only be specified between behavioral models connected by BehavioralRelationships.

In addition, we introduce the concept of *state-changing elements*. Each BehavioralMetamodel specifies a set of StateChangingElements. For example, a state machine defines states and transitions, but only the transitions describe how the states in a state machine change. Thus, the transitions are the StateChangingElements of a state machine (highlighted in purple, see Figure 9). Similarly, the flow nodes are the StateChangingElements of a BPMN process (highlighted in purple, see Figure 10). The interaction of behavioral models is only possible through instances of the StateChangingElements.

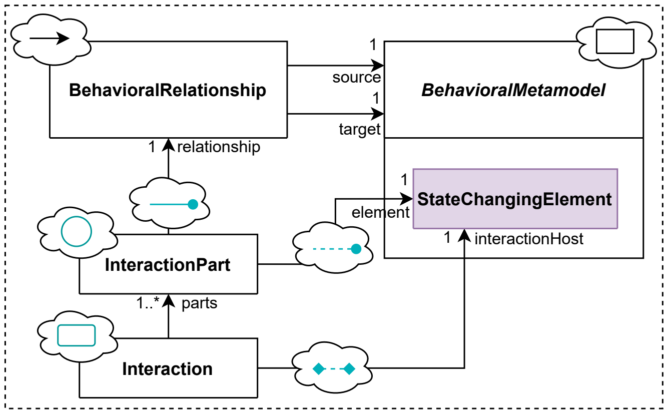

Figure 15 Interaction metamodel

An Interaction has an interactionHost and a set of interaction parts. Each InteractionPart references one BehavioralRelationship and one StateChangingElement. Again, we attach some concrete syntax elements to the interaction metamodel, which we use to define interactions later.

The interactionHost and the elements of the InteractionParts describe a state change for their concrete behavioral model. By connecting them with an interaction, we define that all state changes must happen simultaneously in one atomic step. Consequently, the interactions define synchronizations of the behavioral models. In addition, each InteractionPart must be based on a BehavioralRelationship in the system relationship model, which must have the behavioral model containing the *interactionHost* as a source and the behavioral model containing the StateChangingElement of the InteractionPart as a target. Thus, one can only define interactions for behavioral models connected by behavioral relationships.

We allow the definition of as many interactions as desired. The interactions restrict the local behavior of the models since some parts of its behavior must synchronize, i.e., certain behavior cannot happen locally anymore. If two interactions share elements that should be synchronized, the interactions do not influence each other, such that either of them can happen.

Currently, interactions only specify synchronization and not causality between models, i.e., this state change should happen before/after the other one. However, causality between state changes can be modeled using only synchronization. In the future, one could support causality directly, effectively introducing *syntactic sugar* for interactions.

In the use case, the T-Junction controller interacts with the three traffic lights, **A**, **B**, and **C**. Figure 16 depicts this interaction (cyan-colored parts) synchronizing the traffic lights with the T-Junction controller. It has three InteractionParts specifying the synchronization with each traffic light. We are using the concrete syntax of interactions, defined in the interaction metamodel (see Figure 15).

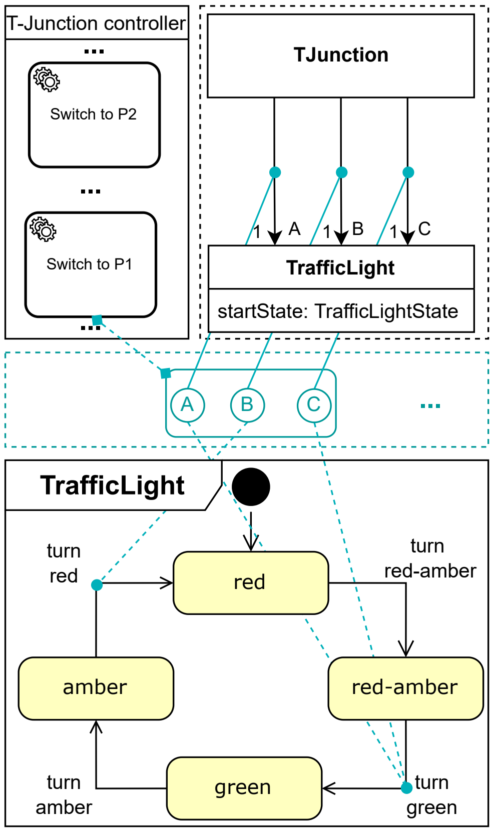

Figure 16 One interaction for the use case

We can explain the interaction as follows. The connection of the interaction to the task Switch to P1 defines that the execution of the task in a T-Junction controller should be part of a synchronization. Switch to P1 is the interaction host and its behavioral model the TJunction is connected with the traffic lights in the system relationship model. Furthermore, the three bubbles and their connections to the two other models state which transitions of which traffic lights should participate in the synchronization. For example, bubble **A** determines that a TrafficLight connected by an A-typed behavioral relationship should be present and turn green simultaneously with the execution of the task. A corresponding interaction exists for the activity Switch to P2 but is not shown in Figure 16.

3.3. Generate execution semantics

To execute the models, we must be able to represent the global system state which is composed of the individual states of each behavioral model. We call this model the *global snapshot metamodel*.

To calculate the global snapshot metamodel, one must merge all snapshot metamodels with the system relationship model. The first step is to add all model elements to one global model, i.e., calculating their disjoint[5] union.

The second step is adding inheritance relations to connect behavioral models in the system relationship model with their corresponding snapshot metamodels. Each behavioral model is typed in a behavioral metamodel, which has a snapshot metamodel (Figure 8) structurally describing its state. Thus, we know how the states of each behavioral model are structured by referring to its metamodel. We make this knowledge explicit by adding an inheritance relation for each behavioral model to the root of the corresponding snapshot metamodel. The resulting artifact, the *global snapshot metamodel*, describes all possible states of the global system since each behavioral model inherits a description of its state.

Figure 17 depicts the global snapshot metamodel M_s^+ for the use case, describing the states of the traffic management system. We added three inheritance relations to construct M_s^+ for the use case and omitted unchanged parts from the system relationship model and the individual snapshot metamodels.

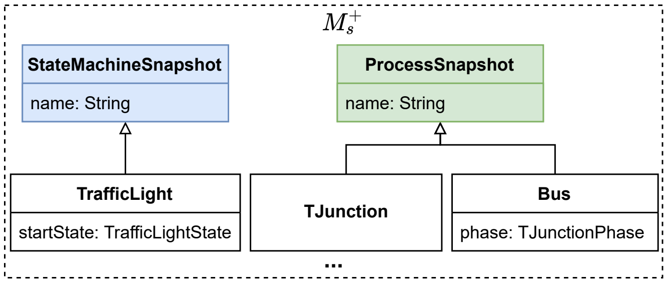

Figure 17 Global snapshot metamodel M_s^+ for the use case

With the global snapshot metamodel, we can represent the global states of the system, but we still need execution semantics to generate a state space to check the properties. The execution semantics used in our approach must fulfill the following three requirements:

[5] Possible conflicts due to the naming of model elements can be resolved by user input or a fixed strategy, for example, prefixing model elements with the model name.

1. The execution semantics must incorporate each behavioral model.

2. The execution semantics must reflect the defined interactions between the behavioral models.

3. The execution semantics must generate a state space, where each state is an instance of M_s^+, given a system configuration (instance of the system relationship model).

Generally, our approach is independent of the underlying execution semantics if it fulfills the three requirements. Thus, one can experiment with different underlying semantics, for example, graph transformations, state machines, Petri nets, or process algebras, without changing the general framework. One can then pick the most suitable semantics for the modeling scenario at hand regarding, for example, consistency checking performance.

To summarize, we obtain execution semantics, which describes the global behavior of the system. Generating these execution semantics from the models and interactions must be fully automated to allow frequent model changes. Later in the paper, we introduce execution semantics based on graph transformations and apply them to the use case.

3.4. Check behavioral consistency

We use the previously constructed execution semantics for consistency checking to generate a state space based on a system configuration. However, we also need atomic propositions interpreted on the global states to define and check global properties.

Atomic propositions must be formulated uniformly for all used behavioral models. We propose to use the global snapshot metamodel M_s^+ to formulate atomic propositions by providing an instance of it for each atomic proposition. Figure 18 shows how to formulate the atomic propositions A_{green} and $B1_{passing}$, used in the global properties stated earlier. Varying propositions with more or less complexity can be formulated using this approach.

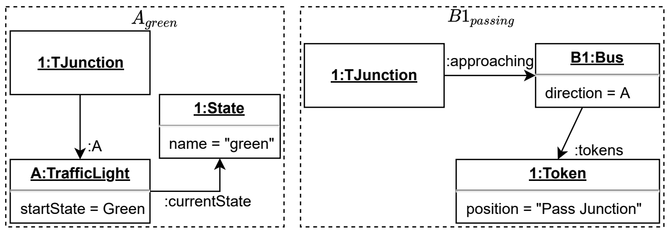

Figure 18 Atomic propositions A_{green} and $B1_{passing}$

Since each state is an instance of M_s^+, we can interpret the constructed atomic propositions on the states. An atomic proposition is true for a state if the corresponding instance of M_s^+ contains the atomic proposition. Then, we can check consistency requirements on the generated state space. This step should be fully automated, such that it can be executed as many times as needed (indicated by the loop in Figure 7) for different system configurations and consistency requirements (including atomic propositions).

Alternatively, to make formulating atomic propositions less cumbersome, one can use the concrete syntax of the individual snapshot metamodels to formulate atomic propositions. For example, Figure 19 shows the same atomic propositions as Figure 18 but uses the introduced concrete syntax.

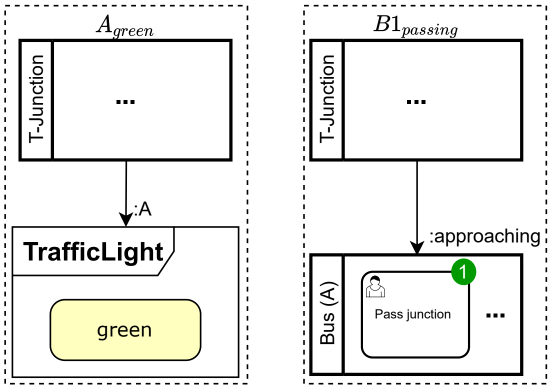

Figure 19 Concrete syntax for A_{green} and $B1_{passing}$

Finally, if a consistency requirement is violated, a counterexample based on the state space should be presented to the user. An unsuccessful consistency check leads to a consistency restoration process, which is crucial but out of scope for this contribution. We describe consistency checking for the use case and its result in the next section. In addition, the appendix contains Figure 23, which gives an overview of the introduced concepts and their usage.

4. Graph transformation execution semantics

This section describes one possible semantics to use in our approach for 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 (Ehrig et al. 2006).*

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) (Ehrig et al. 2006) or single-pushout (SPO) (Löwe 1993). 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 $l : K \rightarrow L$ and $r : K \rightarrow R$ (Ehrig et al. 2006).*

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 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 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[6] 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 the interaction using the saved traces.

   (b) Calculate the parallel production (Baldan et al. 1999, 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 $0 < i \leq n$. 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 executions 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.

[6] 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.

4.1. 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 20 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 Figure 13 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.

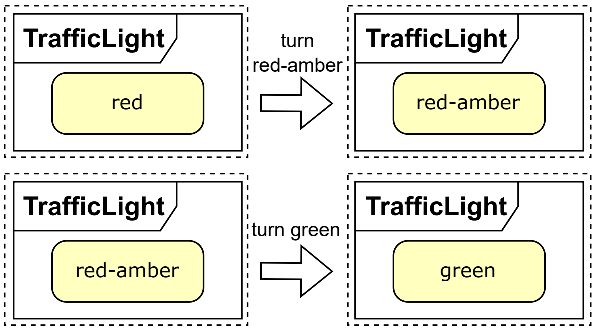

Figure 20 Production rules for *turn red-amber* and *turn green*

Using a traffic light snapshot with the state red as a start graph results in a graph grammar[7], which will generate the same state space as the traffic light state machine. We evaluated our approach by executing the graph grammar with the graph transformation tool Groove (Ghamarian et al. 2012; Rensink 2004). However, in the future, we can also experiment with other graph transformation tools since rule-generation can be adapted without much effort.

4.2. 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 (Van Gorp & Dijkman 2013). Generally, we construct one or more rules for each flow node contained in a BPMN process. Figure 21 shows production rules for some flow nodes of the T-Junction controller (see Figure 3). It uses the concrete syntax introduced earlier to represent process snapshots containing tokens.

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 *bmpn-js token simulation*[7], 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 (Van Gorp & Dijkman 2013) but differs in some aspects. One key difference is that they

[7] All generated graph grammars, including further instructions regarding execution and consistency checking, can be found at https://github.com/timKraeuter/ECMFA-2022.

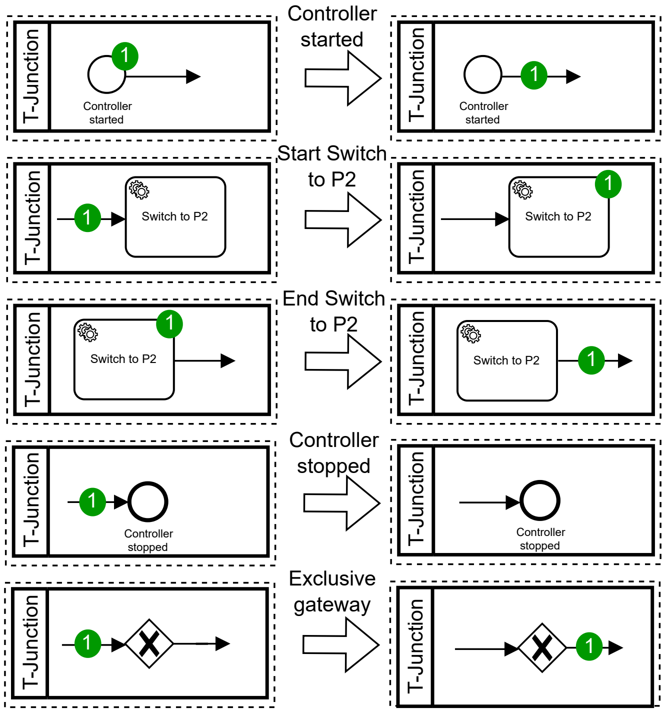

Figure 21 Production rules for the T-Junction controller

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[7].

4.3. Global execution semantics

The interactions we have defined change the rules for switching to phases 1 and 2. Figure 22 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.

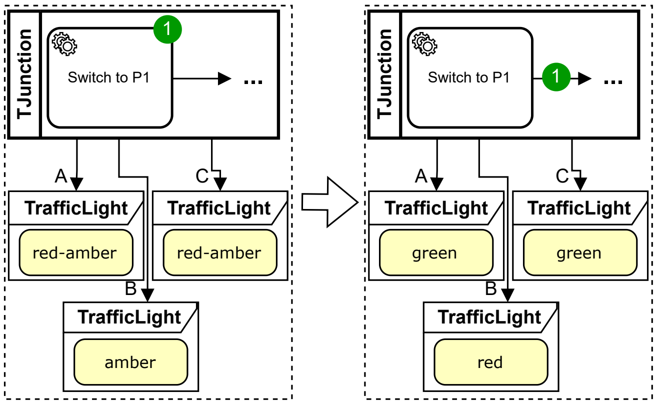

Figure 22 Parallel production rule to switch to phase 1

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[7].

4.4. 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[7]. 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 (Weidmann et al. 2021).

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.

5. Related work

This contribution builds on our previous publication (Kräuter 2021). However, we refined the behavioral consistency management workflow by introducing new concepts such as the system relationship model and snapshot metamodel. Moreover, we created an interaction language and outlined a general approach to define atomic propositions, leading to global behavioral properties. In addition, our previous work did not support BPMN models.

The general idea of transforming different behavioral formalisms to a single semantics domain to reason about cross-cutting concerns is nothing new, see, e.g. (Engels et al. 2001). For example, (Küster & Stehr 2003) developed consistency checking for sequence diagrams and statecharts based on CSP, while (Yao & Shatz 2006) used Petri nets for the same scenario. Furthermore, (Cunha et al. 2011) analyze sequence diagrams in the context of embedded systems using a transformation to Petri nets. Nevertheless, all approaches focus on a single modeling language or a fixed combination of two languages. Consequently, they do not consider the general problem of behavioral consistency in a heterogeneous modeling scenario.

(Kienzle et al. 2019) proposes a unifying framework for homogeneous model composition. To combine behavioral models, they use Event structures (Winskel 1987) as an underlying formalism and show how different homogeneous behavioral models can be combined. Nonetheless, they do not apply their approach to heterogeneous models. Generally, their approach is compatible with ours since we do not mandate a specific formalism in our approach. However, we have chosen graph production rules as the first formalism to use in our approach since they operate on a higher abstraction level. In addition, it was shown that many formalisms, for example, the π-calculus (Gadducci 2007), can be implemented using graph production rules.

(Deantoni 2016) tackles the problem of dealing with relationships between heterogeneous behavioral models. They coordinate the different models using a dedicated coordination language (Vara Larsen 2016), similar to the interactions we define between behavioral models. Their work also results in plugins for GEMOC studio, which support running and debugging the models. (Eker et al. 2003; Lee 2010) propose an actor-oriented solution to the model composition problem in the presence of heterogeneity. Their approach results in the tool Ptolemy (Ptolemaeus 2014).

However, both approaches focus on simulation rather than consistency or model-checking. As far as we can tell, neither provide concrete means to define atomic propositions and check global behavioral properties.

6. Conclusion and future work

Our work represents the first general fully-formalized approach to behavioral consistency management in a heterogeneous modeling scenario, enabling formulating and checking *global* properties. Previous work either only dealt with the behavioral consistency between specific pairs of models (Yao & Shatz 2006; Küster & Stehr 2003) or focused on the simulation in a heterogeneous scenario (Deantoni 2016; Vara Larsen 2016; Eker et al. 2003; Lee 2010).

Our approach follows three key steps (see Figure 7). First, we align the behavioral models by defining their interactions. Then, we generate global execution semantics based on the defined interactions while preserving the original behavior of the models. Finally, we can check the global consistency of the system in different configurations with varying consistency requirements using the generated semantics. Although our approach is independent of a particular formalism as an underlying global execution semantics, the current implementation utilizes graph transformations for this purpose.

Since we could not fully describe our formalization of the BPMN semantics using graph transformations, we plan to do so in a future publication, including a thorough comparison with other formalizations such as (Van Gorp & Dijkman 2013) and (Dijkman et al. 2008).

It is also interesting to extend the graph transformation execution semantics to include other behavioral formalisms such as activity diagrams, hierarchical state machines, and the π-calculus. Especially integrating the π-calculus, which was formalized using graph grammars in (Gadducci 2007), will be beneficial since it is profoundly different from the currently supported formalisms.

References

Baldan, P., Corradini, A., Montanari, U., Rossi, F., Ehrig, H., & Löwe, M. (1999, August). Concurrent semantics of algebraic graph transformations. In *Handbook of Graph Grammars and Computing by Graph Transformation* (Vol. 3, pp. 107–188). World Scientific. doi: 10.1142/9789812814951_0003

Boronat, A., Knapp, A., Meseguer, J., & Wirsing, M. (2009). What Is a Multi-modeling Language? In A. Corradini & U. Montanari (Eds.), *Recent Trends in Algebraic Development Techniques* (Vol. 5486, pp. 71–87). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-642-03429-9_6

Brambilla, M., Cabot, J., & Wimmer, M. (2017). *Model-driven software engineering in practice* (Second ed.) (No. 4). San Rafael, Calif.: Morgan & Claypool Publishers.

Cicchetti, A., Ciccozzi, F., & Pierantonio, A. (2019, December). Multi-view approaches for software and system modelling: A systematic literature review. *Software and Systems Modeling*, *18*(6), 3207–3233. doi: 10.1007/s10270-018-00713-w

Cunha, E., Custodio, M., Rocha, H., & Barreto, R. (2011, November). Formal Verification of UML Sequence Diagrams in the Embedded Systems Context. In *2011 Brazilian Symposium on Computing System Engineering* (pp. 39–45). Florianopolis, Brazil: IEEE. doi: 10.1109/SBESC.2011.18

Deantoni, J. (2016, April). Modeling the Behavioral Semantics of Heterogeneous Languages and their Coordination. In *2016 Architecture-Centric Virtual Integration (ACVI)* (pp. 12–18). Venice, Italy: IEEE. doi: 10.1109/ACVI.2016.9

Dijkman, R. M., Dumas, M., & Ouyang, C. (2008, November). Semantics and analysis of business process models in BPMN. *Information and Software Technology*, *50*(12), 1281–1294. doi: 10.1016/j.infsof.2008.02.006

Ehrig, H., Ehrig, K., Prange, U., & Taentzer, G. (2006). *Fundamentals of algebraic graph transformation (monographs in theoretical computer science. An EATCS series)*. Berlin, Heidelberg: Springer-Verlag.

Eker, J., Janneck, J., Lee, E., Jie Liu, Xiaojun Liu, Ludvig, J., … Yuhong Xiong (2003, January). Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, *91*(1), 127–144. doi: 10.1109/JPROC.2002.805829

Engels, G., Küster, J. M., Heckel, R., & Groenewegen, L. (2001, September). A methodology for specifying and analyzing consistency of object-oriented behavioral models. *ACM SIGSOFT Software Engineering Notes*, *26*(5), 186–195. doi: 10.1145/503271.503235

France, R., & Rumpe, B. (2007, May). Model-driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering (FOSE '07)* (pp. 37–54). Minneapolis, MN, USA: IEEE. doi: 10.1109/FOSE.2007.14

Gadducci, F. (2007). Graph rewriting for the π-calculus. *Mathematical Structures in Computer Science*, *17*(3), 407–437. doi: 10.1017/S096012950700610X

Ghamarian, A. H., de Mol, M., Rensink, A., Zambon, E., & Zimakova, M. (2012, February). Modelling and analysis using GROOVE. *International Journal on Software Tools for Technology Transfer*, *14*(1), 15–40. doi: 10.1007/s10009-011-0186-x

Kienzle, J., Mussbacher, G., Combemale, B., & Deantoni, J. (2019, October). A unifying framework for homogeneous model composition. *Software & Systems Modeling*, *18*(5), 3005–3023. doi: 10.1007/s10270-018-00707-8

Klare, H., & Gleitze, J. (2019, September). Commonalities for Preserving Consistency of Multiple Models. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)* (pp. 371–378). Munich, Germany: IEEE. doi: 10.1109/MODELS-C.2019.00058

Kräuter, T. (2021). Towards behavioral consistency in heterogeneous modeling scenarios. In *2021 ACM/IEEE international conference on model driven engineering languages and systems companion (MODELS-C)* (pp. 666–671). doi: 10.1109/MODELS-C53483.2021.00107

Küster, J., & Stehr, J. (2003). Towards explicit behavioral consistency concepts in the UML. In *Proceedings of 2nd ICSE workshop on scenarios and state machines: Models, algorithms, and tools (portland, USA).*

Lee, E. A. (2010). Disciplined heterogeneous modeling. In *Proceedings of the 13th international conference on model driven engineering languages and systems: Part II* (pp. 273–287). Berlin, Heidelberg: Springer-Verlag.

Löwe, M. (1993, March). Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, *109*(1-2), 181–224. doi: 10.1016/0304-3975(93)90068-5

Object Management Group. (2013, December). *Business Process Model and Notation (BPMN), Version 2.0.2.* https://www.omg.org/spec/BPMN/.

Object Management Group. (2017, December). *Unified Modeling Language, Version 2.5.1.* https://www.omg.org/spec/UML.

Ptolemaeus, C. (Ed.). (2014). *System design, modeling, and simulation: Using Ptolemy II* (1. ed., version 1.02 ed.). Berkeley, Calif: UC Berkeley EECS Dept.

Rensink, A. (2004). The GROOVE simulator: A tool for state space generation. In J. L. Pfaltz, M. Nagl, & B. Böhlen (Eds.), *Applications of graph transformations with industrial relevance* (pp. 479–485). Berlin, Heidelberg: Springer Berlin Heidelberg.

Spanoudakis, G., & Zisman, A. (2001, December). Inconsistency Management in Software Engineering: Survey and Open Research Issues. In (pp. 329–380). doi: 10.1142/9789812389718_0015

Stünkel, P., König, H., Lamo, Y., & Rutle, A. (2021, July). Comprehensive Systems: A formal foundation for Multi-Model Consistency Management. *Formal Aspects of Computing*. doi: 10.1007/s00165-021-00555-2

Van Gorp, P., & Dijkman, R. (2013, February). A visual token-based formalization of BPMN 2.0 based on in-place transformations. *Information and Software Technology*, *55*(2), 365–394. doi: 10.1016/j.infsof.2012.08.014

Vara Larsen, M. (2016). *BCOol : The behavioral coordination operator language* (PhD Thesis). Université Nice Sophia Antipolis.

Weidmann, N., Kannan, S., & Anjorin, A. (2021, June). Tolerance in Model-Driven Engineering: A Systematic Literature Review with Model-Driven Tool Support. *arXiv:2106.01063 [cs]*.

Winskel, G. (1987). Event structures. In G. Goos et al. (Eds.), *Petri Nets: Applications and Relationships to Other Models of Concurrency* (Vol. 255, pp. 325–392). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/3-540-17906-2_31

Yao, S., & Shatz, S. (2006, November). Consistency Checking of UML Dynamic Models Based on Petri Net Techniques. In *2006 15th International Conference on Computing* (pp. 289–297). Mexico city, Mexico: IEEE. doi: 10.1109/CIC.2006.32

About the authors

Tim Kräuter is a Ph.D. research fellow at the Western Norway University of Applied Sciences, Bergen, Norway. His primary research is on integrating heterogeneous behavioral models in multi-model-driven software engineering. Previously he worked as a software developer at SET GmbH in Germany and acquired a master of science in Information Engineering at the University of Applied Sciences, FHDW Hannover, Germany. You can contact the author at tkra@hvl.no or visit https://timkraeuter.com/.

Harald König is a professor for Computer Science at the University of Applied Sciences, FHDW Hannover, Germany, and an Adjunct Professor at the Department of Computer Science, Electrical Engineering and Mathematical Sciences at the Western Norway University of Applied Sciences, Bergen, Norway. Before entering academia, he worked at SAP in Walldorf and received his Ph.D. in pure Mathematics from Leibniz University in Hannover, Germany. You can contact the author at harald.koenig@fhdw.de.

Adrian Rutle is a professor at the Department of Computer science, Electrical engineering and Mathematical sciences at the Western Norway University of Applied Sciences, Bergen, Norway. Rutle's main interest is applying theoretical results from the field of model-driven software engineering to practical domains and has his main expertise in the development of formal modeling frameworks for domain-specific modeling languages, graph-based logic for reasoning about static and dynamic properties of models, and the use of model transformations for the definition of the semantics of modeling languages. His recent research focuses on multilevel modeling, model repair, multi-model consistency management, modeling and simulation for robotics, digital fabrication, smart systems, and applications of machine learning in model-driven software engineering. You can contact the author at aru@hvl.no.

Yngve Lamo is a professor at the Department of Computer science, Electrical engineering and Mathematical sciences at the Western Norway University of Applied Sciences, Bergen. Lamo holds a Ph.D. in Computer Science from the University of Bergen, Norway. His research interests span from formal foundations of Model Based Software Engineering to its applications, especially in Health Informatics. He is currently applying MDSE to create Adaptive Software Technologies for mental Health. You can contact the author at yla@hvl.no.

Patrick Stünkel is currently working as a postdoctoral researcher at Haukeland University Hospital in Bergen, Norway, where he is working on applying process mining, workflow modeling, and optimization techniques in digital pathology. Before that, Patrick did his Ph.D. at Western Norway University of Applied Sciences on the topics of semantic interoperability and consistency management among heterogeneous software models. You can contact the author at Patrick.Stunkel@helse-bergen.no or visit https://past.corrlang.io/.

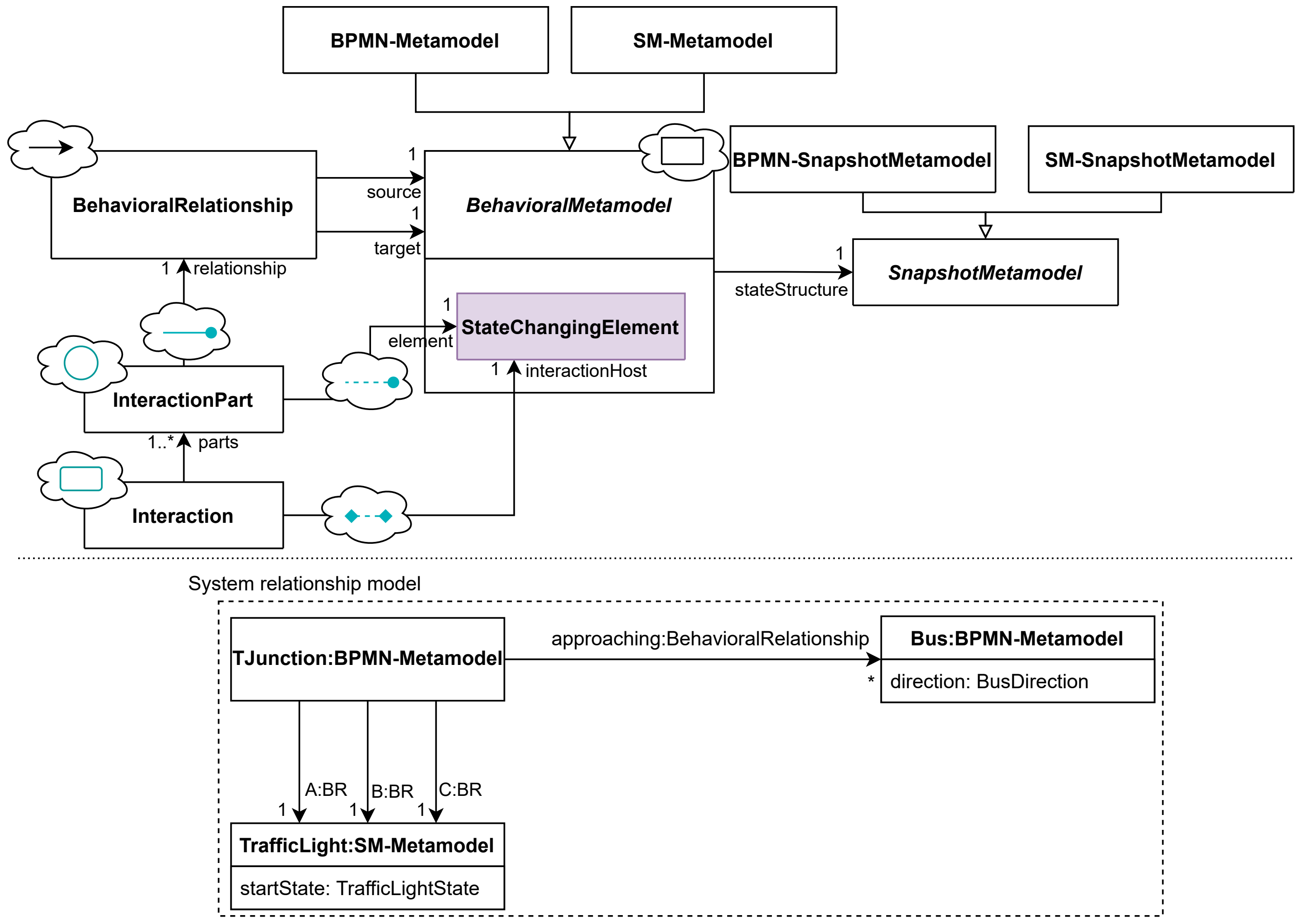

Figure 23 Overview of all concepts and their use in the system relationship model