

Behavioral consistency in multi-modeling

Tim Kräuter*, Harald König^{†*}, Adrian Rutle*, Yngve Lamo*, and Patrick Stünkel[‡]

*Western Norway University of Applied Sciences, Norway

[†]University of Applied Sciences FHDW Hannover, Germany

[‡]Haukeland Universitetssykehus, Norway

ABSTRACT Multiple heterogeneous 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 system. In such a multi-modeling setting, global consistency requirements 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) define which behavioral models in the multi-model *may interact*, (ii) specify consistency requirements as *global behavioral properties*, (iii) align the individual models by specifying *how they interact*, (iv) generate a formal specification of the *global behavior*, and finally, (v) *check* the global behavioral properties, which should be satisfied by the multi-model. Our approach is decoupled from the particular formalism used in the generated formal specification, and we currently support graph transformations (Groove) and rewriting logic (Maude).

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

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 increased productivity, and reduced 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 interrelated models is called a *multi-model* (Boronat et al. 2009), 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 denotational semantics of the software system, as it only describes the set of valid instances or states of the system. Structural models in a multi-model often contain related information. Thus, current approaches define so-called *commonalities* to explicate these relationships and keep the information consistent. Afterward, these commonalities can be used to get a comprehensive view of the global system, for example, by *merging* all models into a global view. Structural consistency can then be verified using this global view.

Nevertheless, approaches to multi-model consistency management must also include a means to maintain *behavioral con-*

JOT reference format:

Tim Kräuter, Harald König, Adrian Rutle, Yngve Lamo, and Patrick Stünkel. *Behavioral consistency in multi-modeling*. Journal of Object Technology. Vol. 22, No. 2, 2023. Licensed under Attribution 4.0 International (CC BY 4.0) <http://dx.doi.org/10.5381/jot.2023.22.2.a9>

sistency since behavioral models, like Business Process Modeling Notation (BPMN) models, are associated with execution semantics describing dynamic aspects of the system (Object Management Group 2013). For example, multi-models consisting of different interacting behavioral models are used when modeling embedded and cyber-physical systems (Vara Larsen et al. 2015).

Several approaches exist for checking the consistency of specific pairs of behavioral models. For example, consistency checking for sequence diagrams and statecharts was implemented using Petri nets (Yao & Shatz 2006) and Communicating Sequential Processes (CSP) (Küster & Stehr 2003). Moreover, the co-simulation field tackles the simulation of interacting models by composing their individual simulations. To facilitate composing simulations, the Functional Mock-up Interface (FMI) standard for simulation models and simulation tools such as Ptolemy (J. Eker et al. 2003) were developed. However, there is no approach to define and *check consistency* of *arbitrary many* behavioral models.

We propose a novel approach for consistency management of heterogeneous multi-models, which allows us to define and check *global* behavioral properties. Our approach facilitates specifying *interactions* between multiple potentially heterogeneous behavioral models, which are used to generate a specification of the global behavior. The approach is decoupled from the particular formalism used in the specification, and currently, we can generate specifications in two different formalisms. The generation of the global behavior specification is *fully automatic* and results in Graph Transformation (GT) rules (or, respectively, term rewriting rules) executable in Groove (Maude). Afterward, we can use the built-in verification mechanisms in Groove (Maude) to check the previously defined global behavioral properties.

Our approach is based on two fundamental concepts: *state* and *state-changing elements*. The state structure of each participating behavioral language must be explicitly defined to infer how global states are structured. Furthermore, state-changing elements must be identified in each participating behavioral language. Thus, state-changing elements serve as a minimal *behavioral interface* to uniformly define interactions for heterogeneous models. Our approach applies to behavioral formalisms where these two concepts can be found, which is the case for most formalisms with discrete state variables (see related work in section 6).

The proposed approach partly resembles the state of the art approaches for structural consistency. Interactions correspond to commonalities as both add necessary inter-model information. Then, a global representation of the system’s behavior/structure is constructed by composing the individual models using the interaction/commonality information. Generating a global behavior specification is similar to merging structural models into a global view. To achieve a global view for structural models, one introduces a base language in which individual models and the global view can be represented. Likewise, generating our global behavior specification is based on the two fundamental concepts of *state* and *state-changing elements*. In summary, we adapt the three steps for structural consistency management: Alignment,

Verification, and Reconciliation as proposed in (Stünkel et al. 2021) to a multi-model containing behavioral models.

This contribution builds on our previous publication (Kräuter 2021). However, we refined the behavioral consistency management workflow into five steps and introduced new key concepts such as state, state-changing elements, interactions, and behavioral relationships.

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 we can use the GT toolset Groove to check behavioral consistency (section 4). Furthermore, we discuss the potential limitations of our approach in section 5. Finally, we examine related work in section 6 and conclude in section 7.

2. Use Case

This section motivates our approach with a simplified use case in which a traffic management system is developed to guide the traffic at a T-Junction with three traffic lights. The traffic management system should control the traffic by switching between the two traffic phases highlighted in figure 1. In addition, it must fulfill the following two requirements. First, it must guarantee safe traffic by correctly changing the three traffic lights A, B, and C. 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.

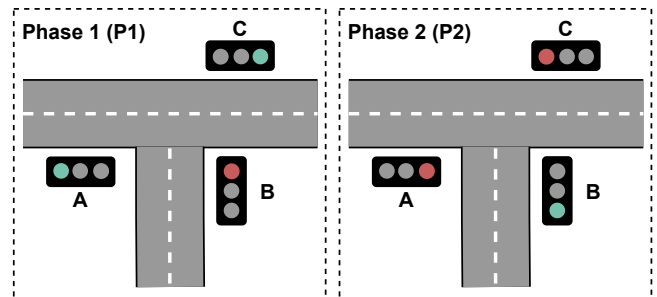


Figure 1 Traffic phases of a T-Junction

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. 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 can 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 a typical 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.

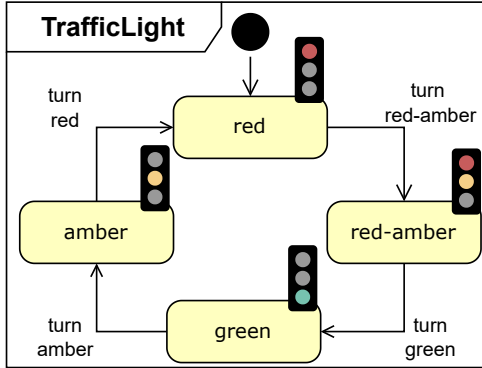


Figure 2 Traffic light state machine model

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 and utilize BPMN message and signal events to implement the communication with approaching buses.

We model two processes, one for the T-Junction and one for the Bus. Each process is modeled in its BPMN *pool*. A pool is depicted as a horizontal lane with a name on the left. Message flows (arrows with dashed lines) are only allowed between two different pools.

Figure 3 shows how a possible controller for a T-Junction behaves in the traffic management system. 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 (see top right in figure 4), which we describe together with the subprocess called phase 2 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, indicated by the default sequence flow going back to the beginning of the process.

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.

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. A BPMN model for a bus controller parameterized with the direction A or C looks nearly identical. In addition, the bus controller communicates with the phase 2 subprocess, which we only hint at in figure 4. The phase 2 subprocess has the same structure as the phase 1 subprocess but reports that A and C are red, while B is green. Similarly, it terminates if green is requested for A or C. The full model and all other models are available in (Kräuter 2023).

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 while 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 (SRM)* is shown in figure 5 using a graph-based syntax. It contains one node for each behavioral model and arrows to depict behavioral relationships, leading to possible interactions. In addition, it contains enumerations to parameterize the behavioral models. A T-Junction 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 SRM, we can define a test configuration of our traffic management system to check its requirements. Figure 6 depicts the test system configuration as an instance of the SRM. 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 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 B.

First, we would like to check the safe traffic requirement. Since we only want to check system conformance concerning

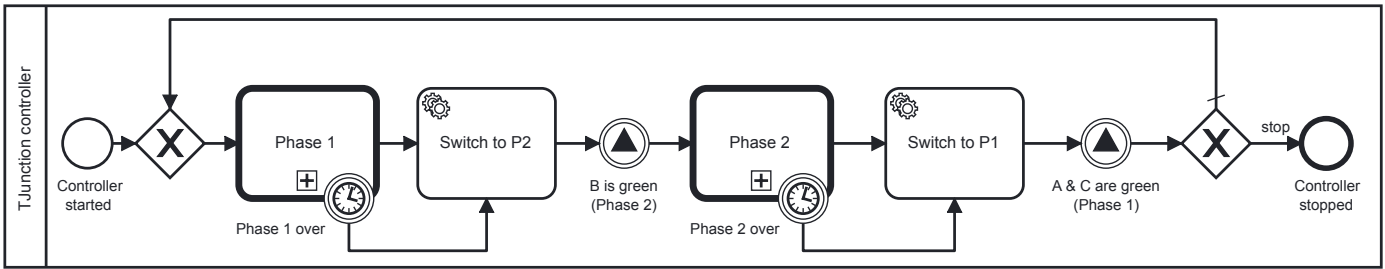


Figure 3 Model for a T-Junction controller

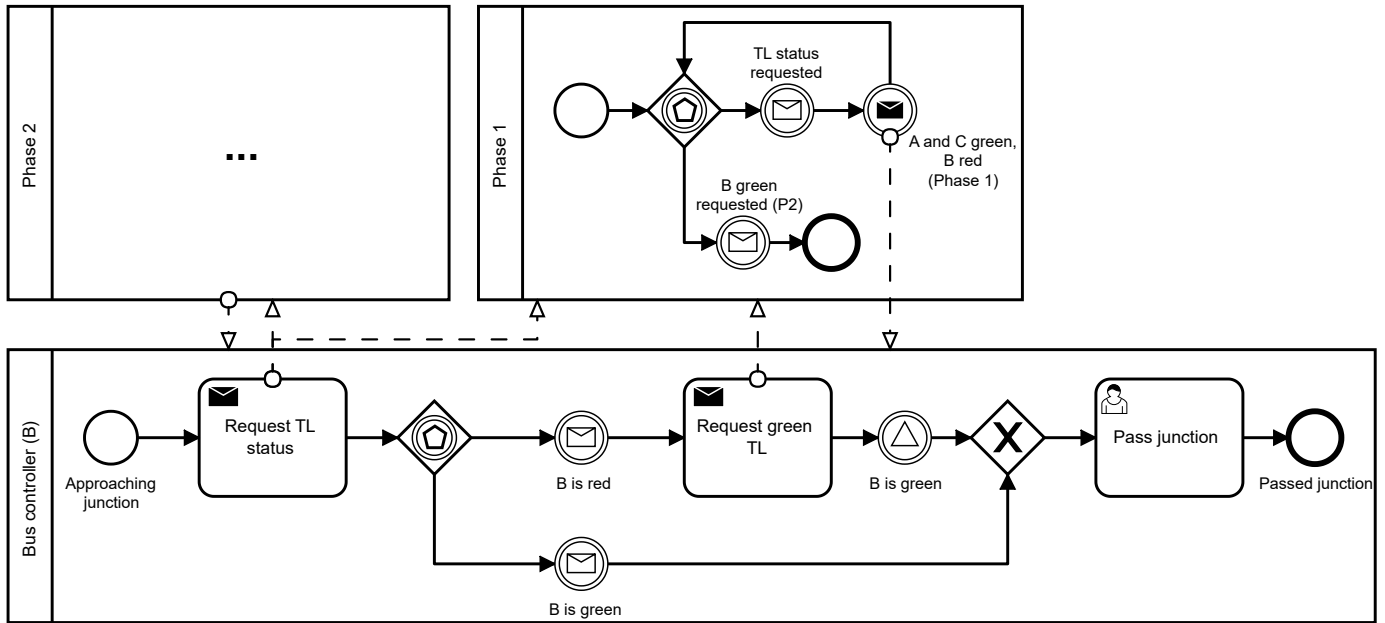


Figure 4 Model for a bus with direction B and its communication with a T-Junction

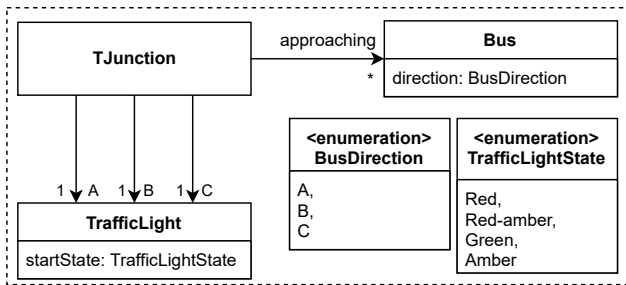


Figure 5 System relationship model of the traffic management system

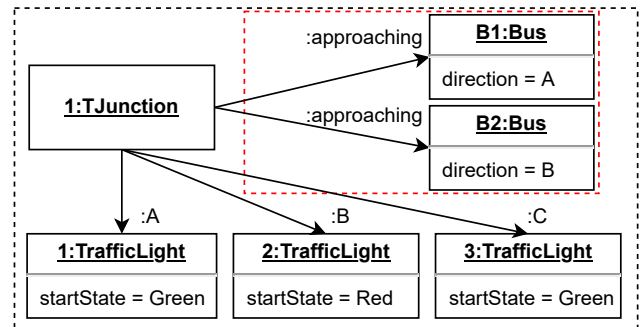


Figure 6 Test system configuration

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. Unsafe traffic occurs only in these situations since green and amber mean that cars are allowed to pass, while red (red-amber) means cars are not (not yet, respectively) allowed to pass. 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) are the following, assuming the existence of atomic propositions for each traffic light state.

$$\Box \neg ((A_{green} \vee A_{amber}) \wedge (B_{green} \vee B_{amber})) \quad (1)$$

$$\Box \neg ((C_{green} \vee C_{amber}) \wedge (B_{green} \vee B_{amber})) \quad (2)$$

If we include buses B1 and B2 in the system, we want 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), where the atomic proposition $B1_{passing}$ and $B2_{passing}$ represent that Pass Junction (see figure 4) has started but not finished yet.

$$\Box \neg (B1_{passing} \wedge (A_{red} \vee A_{red-amber})) \quad (3)$$

$$\Box \neg (B2_{passing} \wedge (B_{red} \vee B_{red-amber})) \quad (4)$$

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 SRM relating two heterogeneously typed behavioral models. In addition, the system configuration instantiates the traffic light and bus behavioral models multiple times with different parameters. Furthermore, we face the problem that the models are not independent. For example, the T-Junction controller must decide when the traffic lights A, B, and C switch states. Thus, if we were to run the models independently in parallel, the properties would be violated.

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 in the use case, 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 depend on the system configuration, i.e., the instance of the SRM 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 as a BPMN diagram.

Our approach consists of five steps.

1. We define a *System Relationship Model (SRM)* describing which behavioral models may interact.
2. We specify consistency requirements as global behavioral properties for the SRM.
3. We define *interactions* between the behavioral models using the SRM.

4. We automatically generate a specification of the specified global behavior using the interactions and the SRM.
5. Given a system configuration, we check the global behavioral properties using the generated specification.

The first three steps are marked as manual and must be completed to use our approach in a given use case. However, the last two steps are automated and reusable in any use case. In the following sections, we will describe each step in detail, highlighting what a modeler must repeatedly do for each use case and what must only be done once for each participating behavioral language. In addition, figure 18 at the end of the paper gives an overview of all the new concepts and how they are applied to the use case.

3.1. Define the system relationship model

As mentioned, a set of behavioral models might be used to describe the behavior of a software system. Each model conforms to its metamodel, corresponding to the behavioral language used to specify the model. The metamodel ensures that models specified in the corresponding languages are well-defined and machine-readable. This is crucial when automating parts of the consistency checking.

The use case utilizes state machine and BPMN models, which conform, respectively, to the metamodels of state machines (see figure 8) and BPMN (see figure 9). The metamodel of state machines is defined by a UML class diagram. In addition, the clouds depict the concrete syntax that we use to denote the models conforming to the metamodel. The traffic light model in figure 2 uses this concrete syntax.

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 transitions of a state machine. Furthermore, isolated states are not allowed.

The metamodel for BPMN (see figure 9) is defined analogously to the one of state machines. A BPMN Process contains a set of FlowNodes connected by SequenceFlows. FlowNodes and SequenceFlows are FlowElements, inheriting an id and a name. A FlowNode can be an Activity, Gateway, or Event. All special activities, gateways, and events are defined in the BPMN specification (Object Management Group 2013).

Behavioral models *interact* to realize the global system behavior. A SRM describes which behavioral models exist in the system and whether they are behaviorally related, i.e., they may interact during execution. In our approach, we define a system relationship metamodel to specify these relationships formally. The constructed SRM is use-case specific, while the metamodels for the participating languages must only be defined once.

We are using a graph-based syntax to define SRMs (see figure 5), where each node corresponds to a behavioral model (typed by a BehavioralMetamodel), while each arrow corresponds to a BehavioralRelationship (see concrete syntax depicted in clouds). For example, the SRM for the use case (see figure 5) has three behavioral relationships from TJunction to TrafficLight since a TJunction controller interacts with three

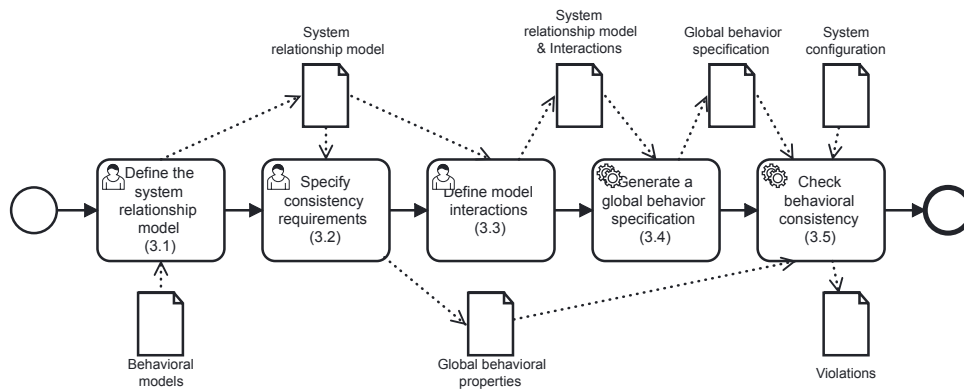


Figure 7 Behavioral consistency management workflow

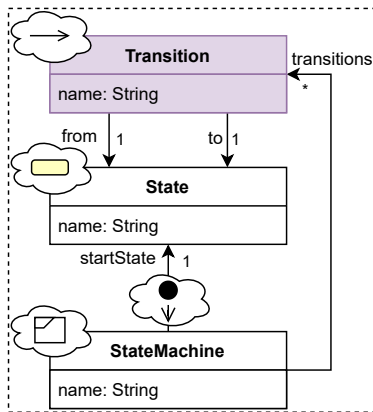


Figure 8 Finite state machine metamodel

different traffic lights A, B, and C. Furthermore, there is a behavioral relationship from TJunction to Bus because we want to check the safety properties (3) and (4). To summarize, behavioral relationships define *which* behavioral models *may* interact, while the interactions in the next step of the workflow describe *how* they interact.

In addition, we allow enumerations and attributes in SRMs. These may be used as parameters, e.g., to define the start state in a state machine (see figure 5). Different instances of the SRM can be used to analyze the global behavior of *different* system configurations by changing the parameters.

3.2. Specify consistency requirements

In this step, we specify behavioral consistency requirements as global behavioral properties. These properties are defined using temporal logic, for example, LTL as in the use case. Theoretically, any temporal logic can be used together with our approach, however, in practice, the underlying system which runs the generated specifications must support it. Furthermore, we agree with (Meyers et al. 2014) that modelers are usually unfamiliar with temporal logic. Lifting property specification to the domain-specific level is a promising idea that fits our approach. Nevertheless, for now, a modeler must define temporal logic properties.

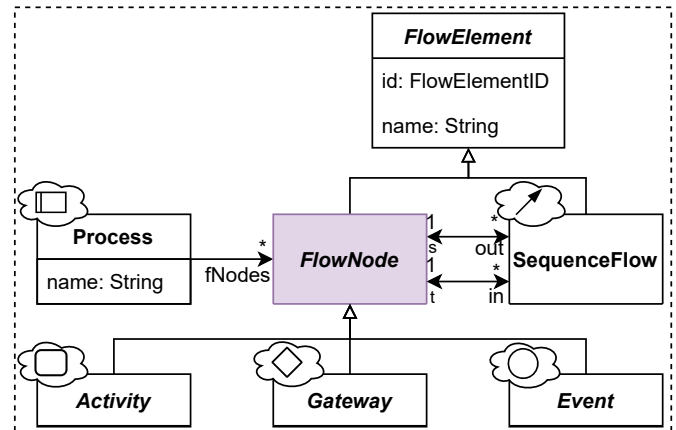


Figure 9 Simplified BPMN metamodel (Object Management Group 2013)

Temporal logic properties are built upon a set of atomic propositions which are either true or false in a given state. Each behavioral language (e.g., the BPMN) specification defines how these states are represented. Furthermore, the transitions between these states depend on the semantics of the behavioral models. In our approach, the first fundamental concept is to make *state* structure explicit. We will call the models for the state structure *snapshot metamodels* and specify them using class diagrams.

For example, in the use case, we define snapshot metamodels for state machines and BPMN. A state machine is in one state at a time, as shown in the snapshot metamodel on the left of figure 10. We are reusing the concrete syntax elements from the state machine metamodel (see figure 8) for the snapshot metamodel. In addition, each snapshot metamodel has a root element in our approach, highlighted in light blue.

The snapshot metamodel for BPMN is based on a Token distribution as described in the BPMN specifications (Object Management Group 2013) (see on the right of figure 10). The root element ProcessSnapshot has tokens and subprocesses. A Token indicates where it is located in the BPMN model using its position attribute. A valid position is the id of a FlowElement (see figure 9). Also, for the snapshot metamodel of BPMN we

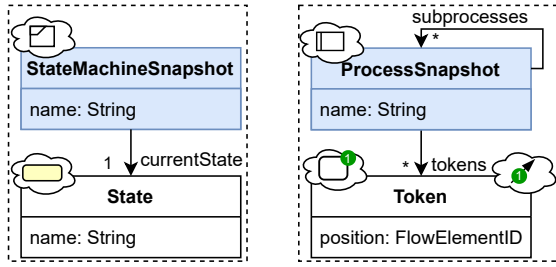


Figure 10 FSM and BPMN snapshot metamodels

reuse the concrete syntax of the BPMN metamodel. In addition, Tokens are highlighted with green bubbles in the middle of sequence flows and the top right of an activity.

Each instance of a snapshot metamodel can represent an atomic proposition since a system can either be in the state specified by this instance or not. Since instances of snapshot metamodels are essentially graphs, they can be matched to a given system state uniformly. Thus, using the snapshot metamodels to create atomic propositions, we can specify local behavioral properties for any modeling language. However, to specify global behavioral properties, we must combine the information of the SRM with the snapshot metamodels. Each behavioral model is typed by a behavioral metamodel, which has a snapshot metamodel describing its state structure. Thus, we know how states of behavioral models are represented when they are instantiated. For example, figure 11 shows how to specify the atomic propositions A_{green} and $B1_{passing}$, used in the global properties in section 2. Snapshot links connect instances of behavioral models with root element instances of the corresponding snapshot metamodel.

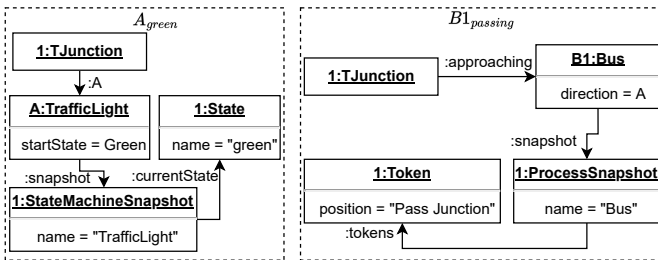


Figure 11 Atomic propositions A_{green} and $B1_{passing}$

To make formulating atomic propositions less cumbersome, one can use the concrete syntax of the individual snapshot metamodels. For example, figure 12 shows the same atomic propositions as figure 11 but uses the introduced concrete syntax.

With the defined atomic propositions as ingredients, one can use temporal logic to define global behavioral properties such as the properties (1)-(4) in section 2. It is worth noting that the defined atomic propositions are *model-specific*, meaning they exactly fit the given multi-model. Thus, property definition (including atomic propositions) is done for each use case, while snapshot metamodels for behavioral languages must only be defined once.

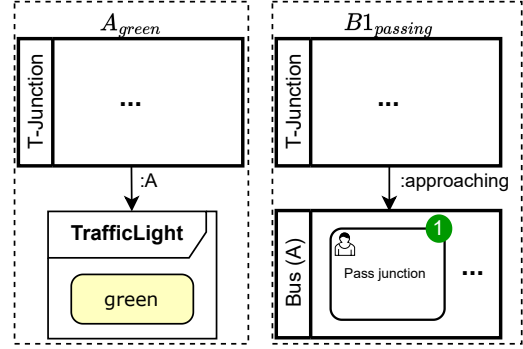


Figure 12 Concrete syntax for A_{green} and $B1_{passing}$

3.3. Define model interactions

In contrast to structural multi-modeling, we call behavioral inter-model relationships *interactions* since they carry behavioral meaning while commonalities carry structural meaning (Stünkel et al. 2021; Klare & Gleitze 2019). To specify interactions between different behavioral models, we define an interaction language given by the system relationship metamodel in figure 13. In addition, we introduce the second fundamental 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 state-changing elements of a state machine (highlighted in purple, see figure 8). Similarly, the flow nodes are the state-changing elements of a BPMN process (highlighted in purple, see figure 9)¹. The interaction of behavioral models is only possible through instances of state-changing elements. Thus, state-changing elements function as a minimal *behavioral interface* to define interactions for heterogeneous behavioral languages uniformly.

Our approach is based on the requirement that state-changing elements can be identified in metamodels for any used behavioral formalism. This requirement is not difficult to meet since behavioral modeling languages with a discrete state must have some *observable* construct to describe state changes. Inspecting other behavioral languages, such as Petri Nets or activity diagrams, shows that identifying state-changing elements (transitions and activity nodes, respectively) is unproblematic.

An Interaction has a sender, a set of receivers, and a type. Currently, there is only the synchronous InteractionType. However, more interaction types, for example, asynchronous interactions or interactions with message passing, could be added in the future. We model the role of sender and receiver in interactions to accommodate these interaction types in the future. The two roles are not needed for synchronous interactions and are hidden from the modeler in the concrete syntax used later in listing 1. Furthermore, asynchronous interactions can be modeled using two synchronous interactions with an additional behavioral model, such as a queue. Each InteractionReceiver references one BehavioralRelationship and one StateChangingElement.

¹ One exception is the event-based gateway, which is not part of the state-changing elements.

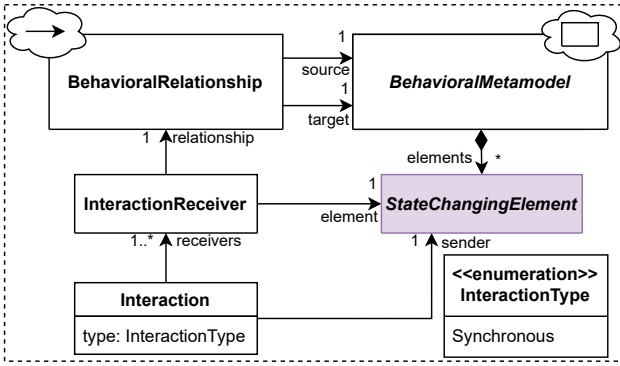


Figure 13 System relationship metamodel

Adding new interaction types only impacts steps 3.3 and 3.4 in our approach (see figure 7). Concretely, a new interaction type has to be added to the enumeration in the system relationship metamodel and must be accounted for in the concrete syntax for interactions. Then, the semantics of the new interaction type must be added to the generation of the global behavior specification in step 3.4.

The sender and the elements of the InteractionReceivers describe a state change for their behavioral models. By connecting them with a synchronous interaction, we define simultaneous state changes in one atomic step. Consequently, an interaction defines a synchronization between behavioral models. Generally, models behave in a distributed independent fashion until they reach a state-changing element that is part of an interaction. To execute these state changes, the models must then interact, i.e., synchronize as described. In addition, one can only define interactions for state-changing elements if a behavioral relationship connects their behavioral models (see constraint (5)). We use "." in constraints to navigate along associations. For example, $i.receivers$ means following all receivers links for an Interaction object, resulting in a set of InteractionReceiver objects.

$$\forall i \in Interaction : \forall r \in i.receivers : \\ r.element \in r.relationship.target.elements \wedge \\ i.sender \in r.relationship.source.elements \quad (5)$$

We allow the definition of as many interactions as desired. Two interactions are not allowed to share state-changing elements (see constraint (6)). If such a situation occurs, it must be resolved by the modeler by deleting one of the interactions or merging the two interactions into one.

$$\forall i_1, i_2 \in Interaction : \\ (i_1.receivers.element \cup i_1.sender) \cap \\ (i_2.receivers.element \cup i_2.sender) = \emptyset \quad (6)$$

In the use case, the TJunction controller interacts with the three traffic lights, A, B, and C. Listing 1 defines two interactions synchronizing TJunction controller and the traffic lights, using a textual Domain-Specific Language (DSL). The **synchronize** keyword specifies the InteractionType to be synchronous.

Each interaction first defines the *sender* state-changing element. Then *receivers* are defined by navigating along an arrow (BehavioralRelationship) to other BehavioralModels in the SRM and then specifying a StateChangingElement. All navigation along BehavioralRelationships starts from the BehavioralModel containing the *sender*, so constraint (5) is satisfied.

```

1 synchronize (TJunction.Switch_to_P1,
2             TJunction.A.turn_green,
3             TJunction.B.turn_red,
4             TJunction.C.turn_green)
5
6 synchronize (TJunction.Switch_to_P2,
7             TJunction.A.turn_red,
8             TJunction.B.turn_green,
9             TJunction.C.turn_red)

```

Listing 1 Interactions for the use case

We can explain the interactions as follows. The first interaction defines that the task Switch to P1 and three other state-changing elements synchronize. Furthermore, line 2 specifies that one of the synchronization receivers is the element *turn green* connected by the relationship A. Similarly, two other transitions of the traffic lights B and C are specified in the following two lines. Thus, the interaction defines a synchronization of a task and three traffic light transitions. The second interaction defines a synchronization for the task Switch to P2 and three traffic light transitions.

To summarize, we use the system relationship metamodel to define the relations between the behavioral models in a multi-model. Thus, the inter-relations between behavioral models in a multi-model are given by interactions and their behavioral relationships. Interactions are specific to each use case, but identifying state-changing elements must only be done once for each behavioral language.

3.4. Generate a global behavior specification

Using the SRM and snapshot metamodels, we can represent the global states of the system. However, we still need a formal specification of the global behavior to check the defined properties. The specification of the global behavior used in our approach must fulfill the following three requirements:

1. The specification must implement the semantics of each behavioral model.
2. The specification must realize the defined interactions between the behavioral models.
3. The specification semantics must allow the checking of behavioral properties for a given system configuration.

Thus, a specification in any formalism fulfilling these three requirements can be used in our approach. Consequently, one can experiment with different formalisms, for example, GTs, rewriting logic, state machines, Petri nets, or process algebras, without changing the general framework. One can then pick the most suitable formalism for the modeling scenario at hand regarding, for example, the performance of consistency checking. In section 4, we describe how we generate specifications for the GT toolset Groove.

To summarize, we generate a specification of the global system behavior. This generation takes the models and interactions as input and is fully automated to allow frequent model changes.

3.5. Check behavioral consistency

In this step, for a given system configuration, we use the generated specification of the global behavior to check the consistency. A system configuration is an instance of the SRM and is automatically translated into the formalism used in the specification. We then check the defined properties using the specification and the system configuration. This step is fully automated, such that it can be executed as many times as needed for different system configurations and properties while using the same specification.

Finally, if a consistency requirement is violated, a counterexample will be presented. We can only show counterexamples if the concrete tool, executing the generated specifications, provides them. However, most modern tools, including Groove and Maude, will provide a counterexample. Adopting the same concrete syntax to visualize the counterexample as for the atomic propositions should be ideal for helping user understanding. Uncovered inconsistencies can lead to a consistency restoration process, which is crucial but out of the scope of this paper. We describe consistency checking for the use case and its result at the end of the next section.

4. Specification of the global behavior

This section describes how GTs can be used as one possible formalism for behavioral consistency management. We utilize the Groove tool set to run the generated specifications, i.e., GT systems (Rensink 2004) for the use case and discuss the results. The successful implementation serves as a *proof of concept* for our approach. We have chosen to use the GT formalism to describe our approach because GTs provide a visual representation and allow for a clearer understanding. As an alternative to GT and Groove, we describe in (Kräuter 2023) our implementation using rewriting logic and Maude.

Both implementations utilize a *global* Higher-Order model Transformation (HOT) from the behavioral models and their interactions to GT rules (Groove) or term rewriting rules (Maude). Since the results of the Model Transformation (MT) can be regarded as MTs themselves, we say the MT is *higher-order* (Tisi et al. 2009). In the following, we describe the HOT to generate GT specifications for Groove. The HOT works similarly when generating a term-rewriting specification for Maude.

The HOT can be decomposed into two steps. The **first step** to generate a specification of the global behavior is to create GT 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 rule for a transition in a state machine changes the current state of a state machine snapshot from the source to the target of the transition.

Thus, we need *local* HOTs for each behavioral modeling language producing rules. Each local HOT only has to be implemented once by a language engineer. The HOT, metamodel, and

snapshot metamodel for a behavioral language can be shared together, for example, as a plugin, such that they can be reused in any future setting the language is needed. In addition, each local HOT must keep traces of the generated rules. Concretely, it has to save which rules originated from which state-changing elements in the behavioral model. Returning to the state machine example, we must know which transition results in which 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 is represented by two rules since it starts and then waits for an incoming message before finishing.

The **second step** is to modify the generated rules to reflect the defined interactions. Interactions define the synchronization of systems, which we encode by merging the individual rules into rules describing the global behavior. In the following section, we describe these two steps in detail using the use case as an example.

4.1. Groove specification

A GT system consists of a set of GT rules of the form $L \rightarrow R$, where the graph L is called the left-hand side and the graph R is called the right-hand side of the rule. Nodes/edges in R but not in L are added by a rule, while nodes/edges in L and R are preserved, and a rule deletes nodes/edges that are in L but not in R . Applying a GT system to a given graph, one obtains a state space where each state is a graph, and each transition is a rule application. A formal description of GT systems can, for example, be found in (Ehrig et al. 2006).

We generate typed GT systems, where the merge of the SRM and the snapshot metamodels is the type graph (Kräuter 2023). Individual rules and rules changing the global state conform to this type graph. Interactions result in global rules which change multiple parts of the global state. A global GT rule is calculated by taking the sum of all left-hand sides and right-hand sides of the individual GT rules (Baldan et al. 1999, Definition 3.2.7). Together with a system configuration, i.e., a start graph, we can obtain an executable formal specification of the global behavior. Consequently, this can be used to check behavioral consistency.

We will now explain how the GT rules are generated for the use case. To apply our approach to the use case, we need to define local HOTs from state machines and BPMN processes to GT rules.

4.1.1. State machine semantics The local HOT to generate GT rules for finite state machines is straightforward. Each transition leads to a GT rule. For example, figure 14 shows the GT rules for the transitions turn green and turn red of the traffic light model. It uses the concrete syntax introduced in figure 10 to depict state machine snapshots and their current states. We depict a GT rule by showing the graph L on the left, R on the right, and a named white arrow from L to R .

Using a traffic light snapshot with the state red as a start graph, we generate the same state space in Groove as the traffic light state machine describes. The generated GT system and the other GT systems of the use case, including further instructions regarding execution and consistency checking, can be found in

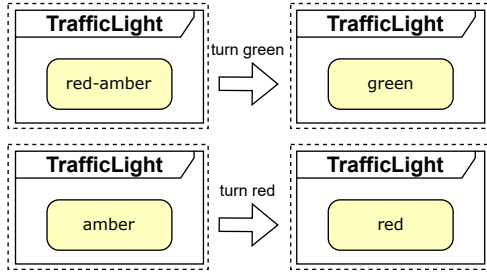


Figure 14 GT rules for *turn green* and *turn red*

(Kräuter 2023).

4.1.2. BPMN semantics The local HOT to generate GT rules for BPMN processes is challenging, and we are currently only supporting a subset of the BPMN semantics (see (Kräuter 2023)). Generally, we construct one or more rules for each flow node, i.e., type of state-changing element in a BPMN model. Furthermore, we created a comprehensive test suite to ensure the correctness of our HOT (Kräuter 2023). Figure 15 shows the GT rules for the task Switch to P1 of the TJunction controller. It uses the concrete syntax introduced in figure 10 to represent process snapshots containing tokens.

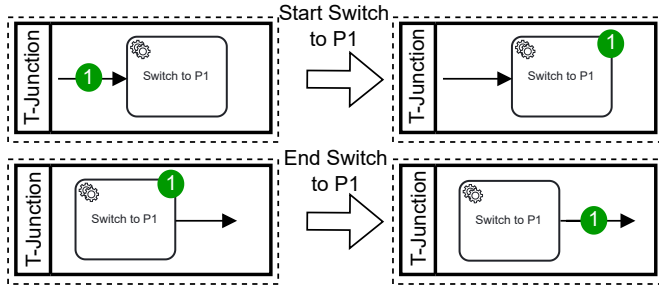


Figure 15 Example GT rules for the TJunction controller

Due to limited space, we only show the Switch to P1 rules but the artifacts of this paper (Kräuter 2023) contain the full GT system and a wiki explaining the HOT in detail. To summarize, we can generate GT systems that implement the behavioral semantics of BPMN.

4.1.3. Realizing interactions To realize interactions between the behavioral models we merge the previously generated rules as follows:

1. Rules generated from state-changing elements that are *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 rule² P_0 for the sender of the interaction and find the rules P_1, P_2, \dots, P_n for the receiver state-changing elements using the saved traces.

² If one state-changing element results in more than one rule, one can define a strategy to pick the appropriate rule.

- (b) Create a *global rule* for the rules P_0, P_1, \dots, P_n , which applies all of them at once, i.e., synchronizes the state changes of the behavioral models.
- (c) For each receiver of the interaction, instantiate the corresponding behavioral relationship from the behavioral model in P_0 to the behavioral model in P_i , for $1 \leq i \leq n$, and add it to the global rule. Thus, only behaviorally related models may interact, i.e., change their state simultaneously.

The defined interactions change the rules for switching to phases 1 and 2. Figure 16 shows the resulting rule for switching to phase 1. We decided that the interactions between the traffic lights and the TJunction controller should synchronize with the end of the task, not the start. Thus, the rule was constructed using the individual rules *turn green* and *turn red* for traffic lights (see figure 14) and the rule *End Switch to P1* (see figure 15). Exactly these rules were used since they are generated from the state-changing elements specified in the first interaction (see listing 1).

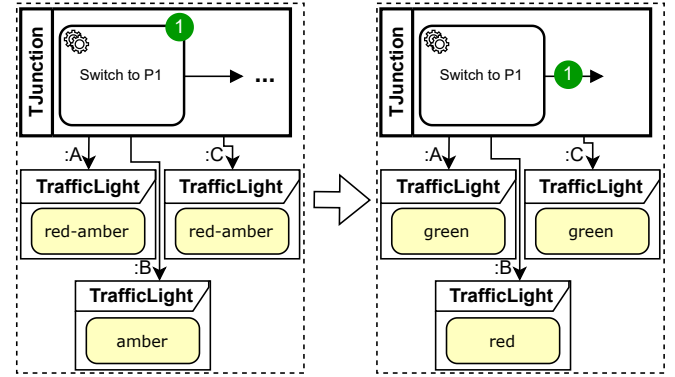


Figure 16 Global GT rule to switch to phase 1

The rule changes all traffic lights simultaneously and finishes the task. The corresponding individual rules no longer exist, so synchronization of the behavior is guaranteed. A similar rule exists for switching to phase 2, resulting from the other interaction. All other rules are left untouched while constructing the global GT system.

4.1.4. Check behavioral consistency Finally, we can generate the global state space of the system using the global GT system, which can be found in (Kräuter 2023). To check the requirements formalized by properties 1-4, we must also encode the used atomic propositions. These are specified as *graph conditions* in Groove. A graph condition in Groove is a rule that does not change elements but can be used as an atomic proposition in model checking.

4.2. Behavioral consistency in the use case

Running the obtained GT specification shows that properties (1) and (2) hold, while properties (3) and (4) do *not* hold (see artifacts in (Kräuter 2023)). The counterexamples for properties 3 and 4 show an unexpected race condition that must be handled: after the TJunction controller signals that the traffic light A

is green, bus B1 can advance to the Pass Junction activity. However, simultaneously, the TJunction controller can enter the subprocess for the next phase, which can be interrupted by the associated timer event. This can happen before bus B1 passes the junction, resulting in a violation of the consistency requirement.

The modelers have different options to handle the detected 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 to resolve the inconsistency. If buses are autonomous, this is the preferred option. For example, the T-Junction controller could wait for the bus to pass before changing the traffic lights again.

5. Discussion

In this section, we discuss two potential limitations of our approach: supporting new modeling languages and state space explosion.

5.1. Support for new modeling languages

To support a new modeling language in our approach one must do the following tasks: describe the state structure in a snapshot metamodel, identify state-changing elements in the language’s metamodel, and implement a HOT to the chosen formalism. A plugin containing these three artifacts can then be reused.

Implementing a HOT that correctly implements a new modeling language’s semantics takes time and effort. We learned that when implementing HOTs to the same underlying formalism, one gets accustomed to the formalism and naturally builds a framework to generate and test specifications in this formalism. Thus, the work needed to implement new languages decreases over time, but one must always understand the semantics of the new language to map them to the chosen formalism.

These tasks are typically done by a language engineer who may be a different person than the modeler who uses our consistency management approach. Generally, we assume two roles: language engineer and modeler. The modeler is in control of the use case and executes the manual tasks in Figure 7, while the language engineer is responsible for supporting the modeling languages which are used in the multi-model.

5.2. State space explosion

State space explosion is one of the predominant issues when applying model checking to complex systems, which are often found in real-world applications. We tested our approach using the generated GT system in Groove described earlier and the alternative implementation using rewriting logic with Maude. As one can see in table 1 and table 2, our approach is not immune to state space explosion. The tables show the states, transitions (rewrites), and average runtime of a full state space exploration in the Groove (Maude) specification. Four scenarios were benchmarked, starting with the use case multi-model

without approaching buses and then increasing the number of buses up to three. Generally, the Maude exploration time is lower despite larger state spaces due to technical differences in implementing the BPMN semantics.

To calculate the average runtime, we used the hyperfine benchmarking tool (Peter 2022) (version 1.15.0), which ran state space exploration for each scenario ten times. Timing evaluations were done with an AMD Ryzen 7700X processor and 32 GB of RAM running Maude version 3.1 (inside the Windows Subsystem for Linux) and Groove version 5.8.1. A description of how to run the benchmarks is available in (Kräuter 2023).

Use case	States	Transitions	Exploration time
No buses	168	438	~1.201 ms
1 bus	2.888	10.046	~1.647 ms
2 buses	27.880	121.554	~4.087 ms
3 buses	195.336	1.028.340	~25.137 ms

Table 1 State space exploration in Groove

Use case	States	Rewrites	Exploration time
No buses	168	664	~28 ms
1 bus	3.304	19.958	~120 ms
2 buses	35.280	279.776	~1.279 ms
3 buses	260.176	2.522.582	~12.599 ms

Table 2 State space exploration in Maude

As in our use case, one is generally not interested in a full state space exploration as in table 1 and table 2 but rather in the validity of a set of behavioral properties. Checking a property specified in LTL does not necessarily lead to a full state space exploration. Furthermore, not every property is concerned with all behavioral models. For example, properties (1) and (2) of the use case do not involve buses and can be checked on smaller state spaces, not including approaching buses. In addition, checking a set of properties can be run in parallel. If some properties are computationally expensive, they can be run on dedicated hardware, for example, during a continuous integration pipeline once a day in case a behavioral model or an interaction changes. Thus, checking the consistency of a multi-model can be seen as an additional test during MDE, which can be run locally but, in addition, is a vital part of continuous integration.

The performance of the Maude LTL model checker is comparable to the popular SPIN model checker (S. Eker et al. 2004) and thus is proven to be competitive. However, one could use different techniques to mitigate the state space explosion problem further. One technique is to abstract models further such that they only contain information relevant to the set of properties to be checked. Thus, minimal models are synchronized,

leading to smaller state spaces. However, finding correct minimal models might not be trivial.

Partial-order reduction is a well-known and effective technique to mitigate the state space explosion problem. It is currently not implemented in the Maude LTL model-checker, but there is promising work to integrate partial-order reduction into the model-checker (Farzan & Meseguer 2007), showing substantial state space reductions. The potential to reduce the state space using this technique is huge, especially for model-checking concurrent systems (Clarke et al. 2018). Our approach analyzes concurrent systems with some interaction, and thus model-checking would greatly benefit from partial-order reduction. In our opinion, partial-order reduction must be implemented to analyze models from real-world applications.

6. Related work

We organized related work into two sections. First, we discuss ad-hoc and general solutions related to the problem of behavioral consistency. Then, we relate our work to the fields of Multi-Paradigm Modeling (MPM) and co-simulation.

6.1. Ad-hoc and general solutions

The general idea of transforming different behavioral formalisms to a single formalism to reason about cross-cutting concerns is not 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. Nevertheless, these approaches only resemble ad-hoc solutions to specific combinations of two languages. They do not consider the general problem of behavioral consistency in a heterogeneous modeling scenario.

(Kienzle et al. 2019) proposes a unifying framework for the homogeneous model composition of structural and behavioral models. To combine behavioral models, they use Event structures as an underlying formalism and show how different homogeneous behavioral models can be combined. In addition, to express behavioral relationships between different models they create causal relationships, which are used during model composition. Generally, their approach is compatible with ours since we do not mandate a specific formalism. Thus, Event structures could be considered, where interactions are realized using causal relationships. Since their research challenges and resulting work items align with our work, we see our work following the same line of research. In addition, they cover the same class of behavioral formalisms as our approach (see DTDS and DEVS in the next subsection).

(Vara Larsen et al. 2015) propose the coordination framework B-COOl. In B-COOl a modeler defines behavioral interfaces for each behavioral language, which are then used to specify interactions between models. To execute the models with the specified interactions, they transform them into Clock Constraint Specification Language (CCSL) models. Their work results in plugins for GEMOC studio, which support running and debugging the models.

Both approaches (Kienzle et al. 2019; Vara Larsen et al. 2015) are similar to our approach since both have mechanisms

to define interactions between behavioral models and generate a global behavioral specification based on these interactions. Furthermore, the generation of the global behavior is achieved by transforming models to a certain formalism like the HOT in our approach. However, the generated specification does not allow to check global behavioral properties since they do not make the state structure (snapshot metamodels in our approach) of the participating models explicit. In our approach, we define the two fundamental concepts (*state* and *state-changing elements*) to check behavioral consistency when heterogeneous models interact.

6.2. Multi-paradigm modeling and co-simulation

MPM is based on the idea to model every aspect and part of a system using the most appropriate modeling formalism(s) (Amrani et al. 2021). Thus, MPM often leads to multi-modeling scenarios where models conforming to different modeling formalisms are used.

Different behavioral formalisms can be classified using two criteria: *time* and *values of state variables* (Wainer 2009). Figure 17 shows the classification where time and state variables can either be *discrete* or *continuous*.

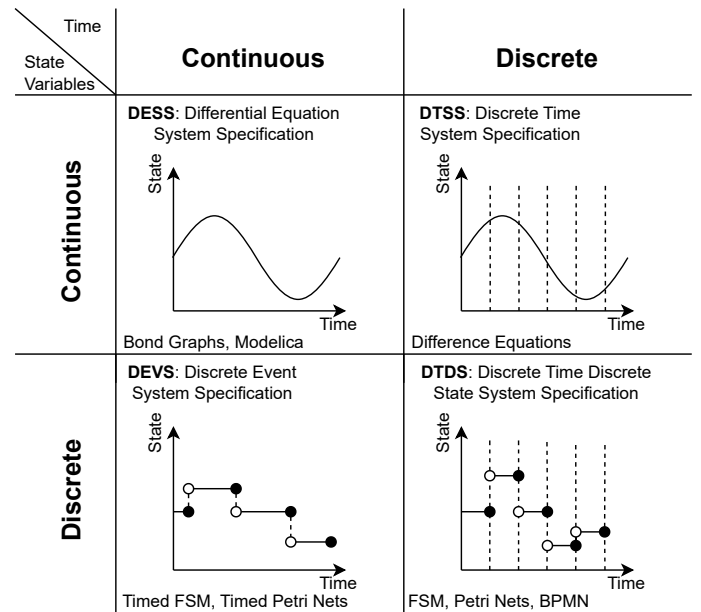


Figure 17 Classification of behavioral formalisms according to the nature of the *time* and *state variables* (adopted from (Wainer 2009; Amrani et al. 2021))

So far we have seen Discrete Time Discrete State System Specifications (DTDSs) like Finite State Machine (FSM) and BPMN in this work, where time and state variables are discrete. Furthermore, our approach can potentially be applied to Discrete Event System Specifications (DEVSSs) since the underlying formalisms support non-discrete time but verification capabilities for DEVSSs are more limited. However, we do not support formalisms with continuous state variables (top part of figure 17), since our central concept of *state-changing elements* would need to be changed. We think that supporting DEVSS

formalisms is a good trade-off between the usefulness and complexity of our approach, since DEVS covers a wide variety of formalisms (Vangheluwe et al. 2002).

Using different modeling formalisms as is the case when following the MPM paradigm makes system simulation challenging. Co-Simulation aims to solve this challenge by composing the simulation of a system’s parts into a global simulation (Gomes et al. 2019).

For example, (J. Eker et al. 2003) propose an actor-oriented co-simulation approach, where each system part is represented as an actor. An actor can communicate through its interfaces with other actors. Their approach is implemented in the tool *Ptolemy* and supports continuous time and state variables. Furthermore, the FMI³ is a co-simulation standard to exchange executable systems parts, so-called Functional Mock-up Units (FMUs). Each FMU, similar to the actors in Ptolemy, comes with an XML model to describe its interface, for example, the FMU’s exposed variables. FMUs support continuous time and state variables and are widely used in the industry.

However, with co-simulations, one can only simulate systems not check global behavioral properties.

7. Conclusion and future work

Our work represents a formalization of behavioral consistency management in a heterogeneous modeling scenario, facilitating the formulation and checking of *global* properties. Previous work either only dealt with the behavioral consistency between specific pairs of models or focused on the simulation in a heterogeneous scenario but lacked checking global properties.

Our approach is based on two fundamental concepts: *state* and *state-changing elements*. The state structure of each participating behavioral language must be explicitly defined to infer how global states are structured. Furthermore, state-changing elements serve as a minimal behavioral interface to uniformly define interactions for heterogeneous models. These two fundamental concepts can be found in most, if not every DEVS formalism. Thus, our approach can support behavioral formalisms with discrete state variables.

In future work, we plan to extend our implementation to support more behavioral modeling languages such as activity diagrams, hierarchical state machines, and the π -calculus. In addition, we aim to apply the approach to real-life industrial case studies. Furthermore, two systems often exchange data while interacting, for example, using name-passing or messaging. The exchanged data then greatly influences the future behavior of the systems. Thus, adding data transfers to interactions between heterogeneous models is an important issue left for future work. Finally, if consistency violations are found, consistency restoration must be achieved. We leave consistency restoration of behavioral models as a problem for future work.

Acknowledgments

We want to thank the anonymous reviewers for their valuable comments and helpful suggestions.

³ <https://fmi-standard.org/>

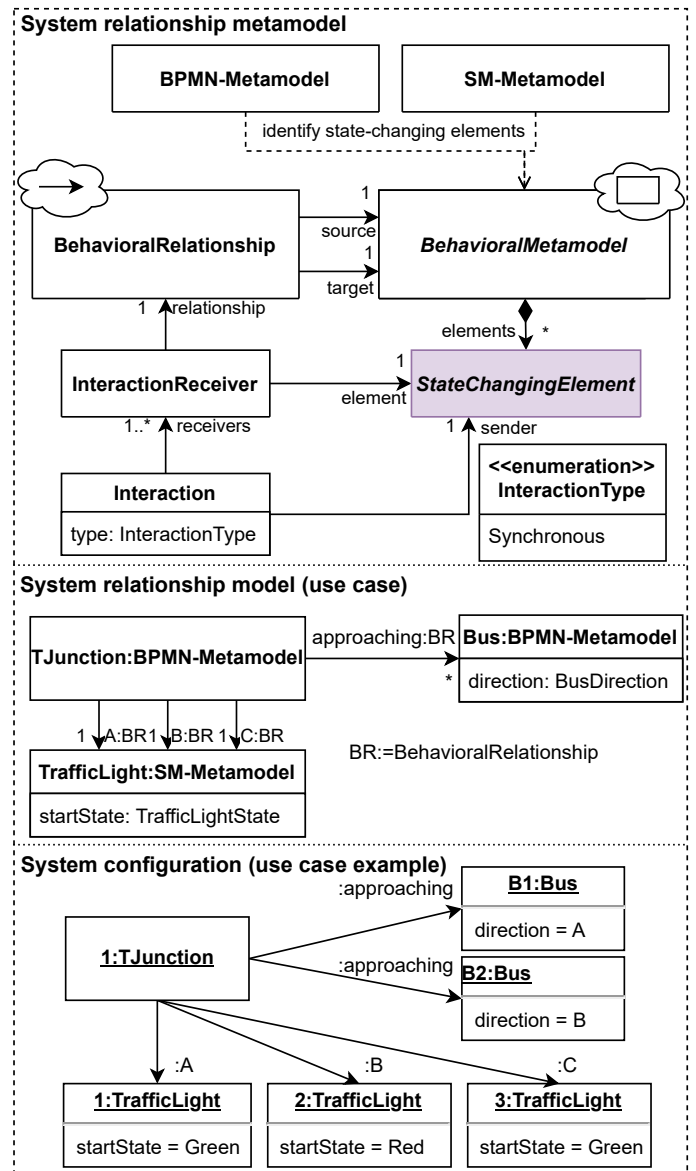


Figure 18 Overview of all concepts and their usage

References

Amrani, M., Blouin, D., Heinrich, R., Rensink, A., Vangheluwe, H., & Wortmann, A. (2021, June). Multi-paradigm modelling for cyber-physical systems: A descriptive framework. *Software and Systems Modeling*, 20(3), 611–639. doi: 10.1007/s10270-021-00876-z

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

- 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
- Clarke, E. M., Henzinger, T. A., Veith, H., & Bloem, R. (Eds.). (2018). *Handbook of Model Checking*. Cham: Springer International Publishing. doi: 10.1007/978-3-319-10575-8
- 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
- Eker, S., Meseguer, J., & Sridharanarayanan, A. (2004, April). The Maude LTL Model Checker. *Electronic Notes in Theoretical Computer Science*, 71, 162–187. doi: 10.1016/S1571-0661(05)82534-4
- 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
- Farzan, A., & Meseguer, J. (2007, July). Partial Order Reduction for Rewriting Semantics of Programming Languages. *Electronic Notes in Theoretical Computer Science*, 176(4), 61–78. doi: 10.1016/j.entcs.2007.06.008
- 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
- Gomes, C., Thule, C., Broman, D., Larsen, P. G., & Vangheluwe, H. (2019, May). Co-Simulation: A Survey. *ACM Computing Surveys*, 51(3), 1–33. doi: 10.1145/3179993
- 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)*. doi: 10.1109/MODELS-C53483.2021.00107
- Kräuter, T. (2023, February). *Artifacts ECMFA-2023*. <https://github.com/timKraeuter/ECMFA-2023>.
- 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)*.
- Meyers, B., Deshayes, R., Lucio, L., Syriani, E., Vangheluwe, H., & Wimmer, M. (2014). ProMoBox: A Framework for Generating Domain-Specific Property Languages. In B. Combemale, D. J. Pearce, O. Barais, & J. J. Vinju (Eds.), *Software Language Engineering* (Vol. 8706, pp. 1–20). Cham: Springer International Publishing. doi: 10.1007/978-3-319-11245-9_1
- Object Management Group. (2013, December). *Business Process Model and Notation (BPMN), Version 2.0.2*. <https://www.omg.org/spec/BPMN/>.
- Peter, D. (2022). *Hyperfine*. GitHub.
- Rensink, A. (2004). The GROOVE Simulator: A Tool for State Space Generation. In T. Kanade et al. (Eds.), *Applications of Graph Transformations with Industrial Relevance* (Vol. 3062, pp. 479–485). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-540-25959-6_40
- Spanoudakis, G., & Zisman, A. (2001, December). Inconsistency Management in Software Engineering: Survey and Open Research Issues. In *Handbook of Software Engineering and Knowledge Engineering* (pp. 329–380). World Scientific. 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
- Tisi, M., Jouault, F., Fraternali, P., Ceri, S., & Bézuvin, J. (2009). On the Use of Higher-Order Model Transformations. In R. F. Paige, A. Hartman, & A. Rensink (Eds.), *Model Driven Architecture - Foundations and Applications* (Vol. 5562, pp. 18–33). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-642-02674-4_3
- Vangheluwe, H., de Lara, J., & Mosterman, P. (2002). An introduction to multi-paradigm modelling and simulation..
- Vara Larsen, M. E., DeAntoni, J., Combemale, B., & Mallet, F. (2015, September). A Behavioral Coordination Operator Language (BCOoL). In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)* (pp. 186–195). Ottawa, ON, Canada: IEEE. doi: 10.1109/MODELS.2015.7338249
- Wainer, G. A. (2009). *Discrete-event modeling and simulation: A practitioner's approach*. Boca Raton: CRC Press.
- 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]*.
- 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. student 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 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 his main expertise is 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/>.