Heterogeneous behavioral model composition using graph grammars

Tim Kräuter, Adrian Rutle, and Yngve Lamo

Høgskulen på Vestlandet
Bergen, Norway
tkra@hvl.no, aru@hvl.no, yla@hvl.no

1 Introduction

A common approach to handle the increasing complexity of software systems is Model-driven engineering (MDE). MDE envisions a clear separation of concerns by modeling each aspect of a system separately [2]. However, these individual models must be composed to execute the system or reason about global properties [5].

Our contribution addresses the problem of how the individual models can be composed to describe the entire software system, even if the used models are heterogeneous, i.e., do not conform to the same modeling language. In particular, we address the composition of behavioral models since its structural counterpart has been the focus of a significant amount of research already, for example, in [5, 6, 10]. We further limit the behavioral models to those describing discrete behavior such as finite state machines, Business Process Modeling Notation (BPMN) diagrams, process algebras, and Petri nets.

Our contribution can be summarized as follows. Given a collection of behavioral models describing one system coordinated using our two types of interactions, we can construct a graph grammar (GG) representing the system's behavior. One can then analyze the state space generated by the GG to reason about global properties and analyze possible execution paths of the system. This contribution extends our work in [7] by adding asynchronous communication and an example showing how to implement hierarchical composition.

An approach with similar goals uses event structures as an underlying formalism and event scheduling to model behavioral relations [5]. However, it has currently only been applied to compose homogeneous models. We will now explain our contribution in more detail.

2 Heterogeneous behavioral model composition

We assume that each behavioral model describes a component of the overall system running independently and in parallel when it is not interacting with other components. Each component, i.e., behavioral model, should interact at least once with a different model to contribute to the composite system behavior.

Our approach supports two types of interactions between behavioral models: *synchronous* and *asynchronous* communication. One can use synchronous communication to model that one component acquires a resource realized in a separate model since it is shared across the entire system. Asynchronous communication can, for example, be used to model that two components communicate using a messaging system or by writing/consuming files in a shared file system. Defining synchronous communication leads to two components becoming active simultaneously, while asynchronous communication only requires one component to be active.

Our model composition approach can be separated into two steps. The **first step** is to define interactions among a set of behavioral models using synchronous and asynchronous communication. However, not any element in a behavioral model can be part of an interaction. For example, it does not make sense to define that a state in a state machine should communicate with a transition in a Petri net since a state represents static information. Consequently, only certain elements in each behavioral model, such as transitions in state machines and Petri nets, should be used when defining interactions between behavioral models. One can achieve these restrictions by creating and aligning the metamodels of the respective behavioral models, as described in [7].

In the **second step** of our approach, we will use GGs to realize the behavioral semantics of the individual models and their combination as defined by the interactions. However, one could also choose a different formalism than GGs in this step.

For the second step, there has to be an implementation of each behavioral formalism using GGs. Implementations for finite state machines and Petri nets were, for example, defined in [7]. An implementation of the π-calculus process algebra using GGs is described in [3], while [9] illustrates how to execute workflow models using graph transformations. Consequently, we can (automatically) map each model to a GG representing that model's behavior. Afterward, all GGs are combined into one respecting the interactions defined earlier.

The start graph of the GG describing the composite system is given by the sum of all start graphs of the individual GGs. The set of GG rules for the composite system depends on the defined interactions between the models because the rules corresponding to these elements will be combined. All non-related GG rules are kept unchanged for the composite system.

Rules related by synchronous communication are amalgamated into one *parallel production rule*, formally defined as a coproduct construction using category theory (CT) in [1, Def. 3.2.7]. This forces both rules to be executed simultaneously in one rule application step. An example of a rule created by synchronous communication can be seen on the left of Figure 1. Synchronous rules can have two or more participants from distinct behavioral models.

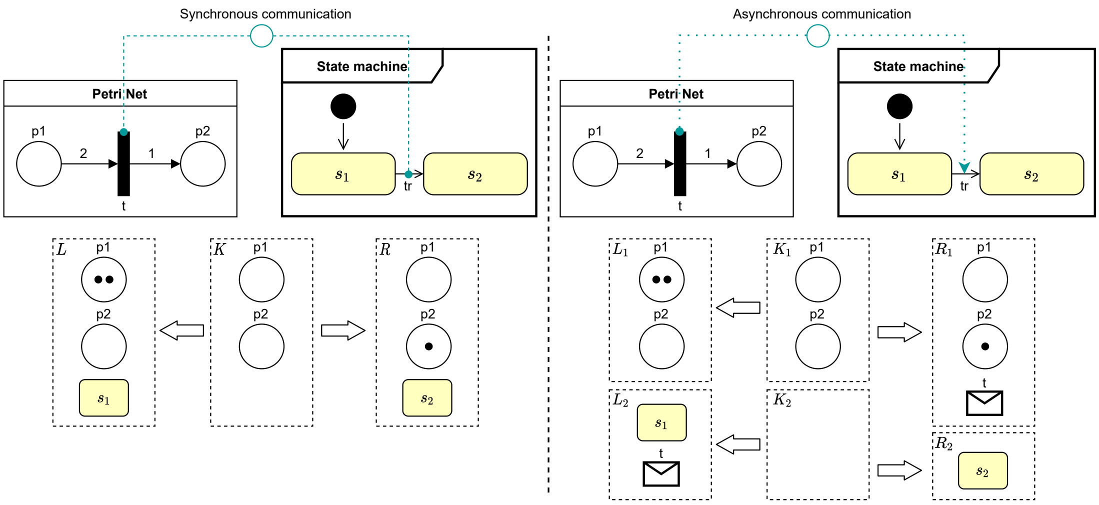

Figure 1: GG rules for synchronous and asynchronous interactions

Asynchronous communication will lead to rules being enriched, not merged. An asynchronous interaction has a direction and links only two model elements. It will lead to the creation and consumption of a unique message node. An example can be seen on the right of Figure 1. One could also think about asynchronous messages from one model to a set of models, creating multiple messages instead of only one message. Asynchronous communication influences the global system behavior by forcing two previously unrelated rules to be executed in a particular order.

In addition, we are working towards including a structural model into our approach, which describes how many instances of each behavioral model exist and which of these instances are communicating with whom. This is useful because we do not want to duplicate behavioral models only to model that two instances exist in our system. For example, if we have a state machine representing that a resource is acquired or not acquired, we do not want to duplicate this model for every resource in our system. However, including a structural model makes rule generation more complex since the objects representing instances of behavioral models have to be taken into account.

As a simple example, we want to show how we can use our approach to implement the hierarchical composition of two behavioral models. Often one wants to specify the behavior of one part of a model in more detail using a separate sub-model.

In Figure 2, we have a state machine and a BPMN process model where the behavior of the state `Processing` is described by a BPMN process model. When the state `Processing` is entered, a BPMN process should be started, and the state can only be excited when this process is finished. We can achieve this by synchronizing transition b with the *start event* and transition c with the *end event* in the BPMN process model, as highlighted by the cyan connections in Figure 2.

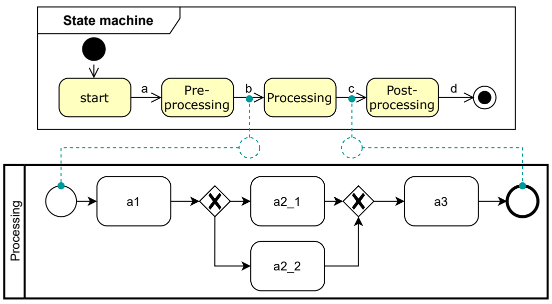

Figure 2: Hierarchical composition of a state machine and a BPMN process model

We implemented the hierarchical composition example in Groove[1]. Groove is a toolset that supports creating and executing GGs, among other things [4, 8]. We manually constructed the GG, which describes the behavior of the composite system, and executed it to check its validity.

References

[1] Paolo Baldan, Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, and Michael Löwe. Concurrent semantics of algebraic graph transformations. In *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 3, pages 107–188. World Scientific, August 1999.

[2] Robert France and Bernhard Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering (FOSE '07)*, pages 37–54, Minneapolis, MN, USA, May 2007. IEEE.

[1] `https://github.com/timKraeuter/NWPT-2021/tree/main/groove`

[3] Fabio Gadducci. Graph rewriting for the π-calculus. *Mathematical Structures in Computer Science*, 17(3):407–437, 2007.

[4] Amir Hossein Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon, and Maria Zimakova. Modelling and analysis using GROOVE. *International Journal on Software Tools for Technology Transfer*, 14(1):15–40, February 2012.

[5] Jörg Kienzle, Gunter Mussbacher, Benoit Combemale, and Julien Deantoni. A unifying framework for homogeneous model composition. *Software & Systems Modeling*, 18(5):3005–3023, October 2019.

[6] Heiko Klare and Joshua Gleitze. Commonalities for Preserving Consistency of Multiple Models. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 371–378, Munich, Germany, September 2019. IEEE.

[7] Tim Kräuter. Towards behavioral consistency in heterogeneous modeling scenarios. In *To Appear in: Proceedings of the 24rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '21, 2021.

[8] Arend Rensink. The GROOVE simulator: A tool for state space generation. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance*, pages 479–485, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[9] Adrian Rutle, Wendy MacCaull, Hao Wang, and Yngve Lamo. A metamodelling approach to behavioural modelling. In *Proceedings of the Fourth Workshop on Behaviour Modelling - Foundations and Applications - BM-FA '12*, pages 1–10, Kgs. Lyngby, Denmark, 2012. ACM Press.

[10] Patrick Stünkel, Harald König, Yngve Lamo, and Adrian Rutle. Comprehensive Systems: A formal foundation for Multi-Model Consistency Management. *Formal Aspects of Computing*, July 2021.