Integrating Graph Transformations and Modal Sequence Diagrams for Specifying Structurally Dynamic Reactive Systems

Sabine Winetzhammer[1], Joel Greenyer[2], and Matthias Tichy[3]

[1] sabine.winetzhammer@uni-bayreuth.de Chair of Applied Computer Science 1 - Software Engineering, Universität Bayreuth, Universitätsstraße 30, 95440 Bayreuth, Germany.
[2] greenyer@inf.uni-hannover.de Software Engineering Group, Leibniz Universität Hannover, Welfengarten 1, 30167 Hannover, Germany.
[3] matthias.tichy@cse.gu.se Software Engineering Division, Chalmers | University of Gothenburg, 412 96 Gothenburg, Sweden.

Abstract. Software-intensive systems, for example service robot systems in industry, often consist of multiple reactive components that interact with each other and the environment. Often the behavior depends on structural properties and relationships among the system and environment components, and reactions of the components in turn may change this structure. Modal Sequence Diagrams (MSDs) are an intuitive and precise formalism for specifying the interaction behavior among reactive components. However, they are not sufficient for specifying structural dynamics. Graph transformation rules (GTRs) provide a powerful approach for specifying structural dynamics. We describe an approach for integrating GTRs with MSDs such that requirements and assumptions on structural changes of system resp. environment objects can be specified. We prototypically implemented this approach by integrating ModGraph with ScenarioTools. This allows us not only to specify MSDs and GTRs in Eclipse, but also to simulate the specified behavior via play-out.

Keywords: scenario-based specification, reactive systems, embedded systems, automotive, simulation, validation, testing

1 Introduction

In many areas, such as industry and transportation, we find increasingly complex, interconnected, software-intensive systems. In industry, for example, service robots support workers and decentralized control components control complex production processes; advanced driver assistance systems in cars rely on the inter-vehicle communication to realize collision avoidance or vehicle platooning.

As an example, Fig. 1 shows an autonomous robot transport system in a production plant. Workers at assembly stations can order items to be delivered

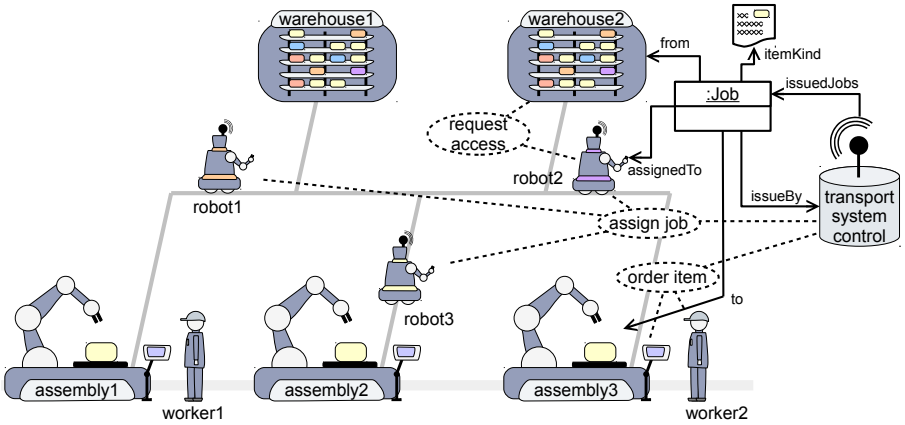

Fig. 1. Example of an autonomous robot transport system in a production plant

to them by a transport system. Upon receiving an order, the transport system control assigns a job to a robot, which executes it by requesting access to the given location (a warehouse), picking up the item, and delivering it.

These systems often consist of multiple, physically distributed mechatronic components that comprise hardware, mechanical parts, and software. It is the software which mainly realizes the systems' complex functionality. The software processes environment events, performs the coordination of the components, the interaction with the users, and it acts on the physical environment via actuators. We therefore view these systems as distributed reactive systems.

The challenge in the design of these systems is that the requirements often span multiple components, and components may have to satisfy multiple requirements at the same time. To exemplify this, consider a worker that orders an item: the worker inputs the order via a terminal at the assembly station, the terminal then notifies the transport system control, which then assigns a job to a service robot, etc. At the same time, the transport system may receive notification of a robot's malfunction and must notify service personnel.

Moreover, the requirements often relate to the system's structure, which can be its physical structure or logical structures within or shared among its software components. For example, which robot the transport system control assigns a job to depends on the robot's availability and proximity to the pick-up location (physical structure). Which warehouse the robot requests access to depends on the job it received (logical structure). In turn, reactions of the software can change physical or logical structures. For example, when ordering the robot to move to a certain location, we can assume that it will eventually arrive there (physical structure). An example for changes in the logical structure would be the transport system control creating a job object and assigning it to a robot.

We propose to specify these systems using Modal Sequence Diagrams (MSDs), a formal interpretation of UML sequence diagrams [10] based on the

concepts of Live Sequence Charts (LSCs) [4]. MSDs allow us to formally, but intuitively specify sequences of events between system and environment components that may, must, or must not happen. One advantage of this formalism is that the specifications can be executed via the *play-out* algorithm [11,12]. We recently extended MSDs and the play-out algorithm to not only consider *requirements* on what the system must to do, but also to support *assumptions* on what will and will not happen in the system's environment [3]. Further extensions allow us to express simple structural changes - like changes of attribute values - complex structural changes, however, cannot be modeled adequately.

In this paper, we therefore propose integrating graph transformation rules (GTRs) with MSDs to eliminate this drawback. We explain the semantics and the extension of the play-out algorithm by the help of an illustrative example. The main idea of the integration is straightforward: use GTRs to model side-effects that messages have on the system structure. However, our integration goes further: GTRs can also constrain in which structural contexts the system is allowed to perform certain actions (requirements) and in which structural contexts certain events can occur in the environment (assumptions). We implemented our approach prototypically by integrating ModGraph[4] [19], a tool for modeling and executing GTRs, and ScenarioTools[5] [3], a tool suite that supports the modeling and play-out of MSDs.

The resulting modeling and analysis approach supports an iterative and incremental specification of message-based interaction behavior and structural system reconfiguration behavior. The advantage of the scenario-based approach is that adding single scenarios to a specification can extend as well as constrain previously specified behavior [13]. Integrating GTRs adds intuitive means for expressing structural changes. The declarative style of specifying rules with object-patterns as pre- and post-conditions, combined with the graphical, color-coded notation, makes complex changes on the object system easy to understand.

This paper is structured as follows. Section 2 provides the foundations. Section 3 then describes the concepts of the integration, and Sect. 4 describes the tool integration. We discuss related work in Sect. 5 and conclude in Sect. 6.

2 Foundations

In the following, we describe the basics of MSDs and graph transformation rules.

2.1 Modal Sequence Diagrams

MSDs [10] are a formal interpretation of UML sequence diagrams, based on the concepts of LSCs [4,12]. An MSD specification consists of a set of MSDs. MSDs can be either *existential* or *universal*. Existential MSDs describe sequences of events that must be possible to occur, universal MSDs describe properties that must hold for all sequences of events. Here, we focus on universal MSDs only.

[4] `http://btn1x4.inf.uni-bayreuth.de/modgraph/homepage`
[5] `http://scenariotools.org`

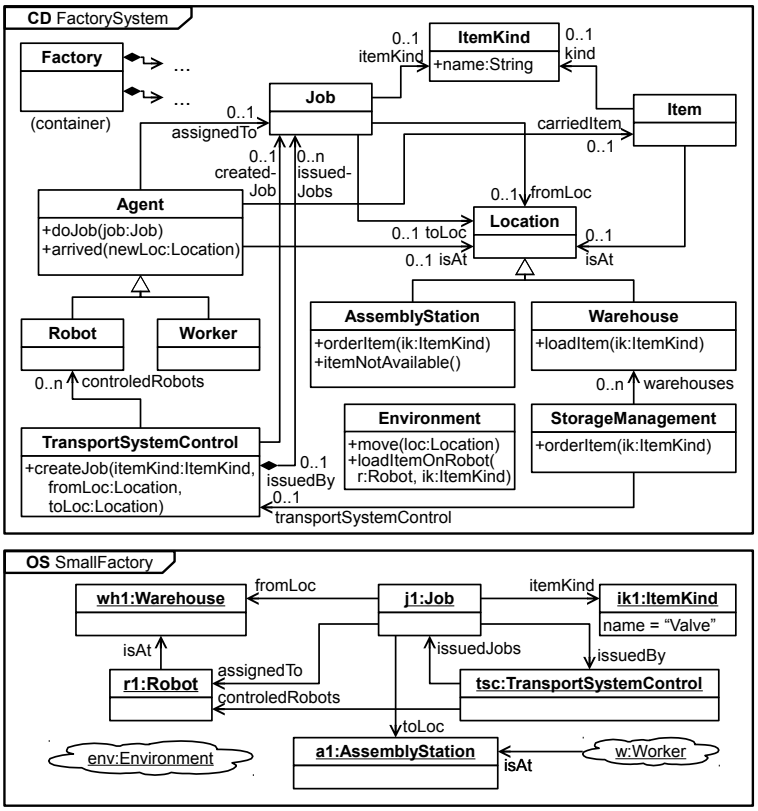

Fig. 2. Class diagram of production plant systems and an object diagram of a small instance system (cf. Fig. 1)

The lifelines of an MSD represent *objects* in an *object system*. The objects are either controllable *system objects* or uncontrollable *environment objects*. The set of environment objects is also called the *environment*; the set of system objects is also called the *system*.

We consider the object system to be a valid instance of a class model that can define associations and attributes. Objects then carry attribute values according to the attribute definitions and there can exist *links* among the objects according to the associations. As an example, Fig. 2 shows the class diagram of our factory system and a possible object system; the object system represents a very simple plant with one assembly station, one robot, and one warehouse. Environment objects have a cloud-like shape; system objects have a rectangular shape.

The objects can interchange *messages*. A message has a sending and receiving object and refers to an operation that must be defined by the receiving object's class. Here we consider only *synchronous* messages where the sending and receiving together is a single *event*, also called *message event*.

Lifelines of the MSDs each represent an object in the object system. A message in an MSD, also called a diagram message, represents a message event in the object system. The diagram message has a sending and receiving lifeline and refers to an operation.

A diagram message has a *temperature* and an *execution kind*. The temperature can be either *hot* (red arrow, labeled h) or *cold* (blue arrow, labeled c); the execution kind can be either *monitored* (dashed arrow, labeled m) or *executed* (solid arrow, labeled e). Intuitively, messages that are *monitored* may occur, while messages that are executed must eventually occur. If a message is hot, it means that when a point is reached in the scenario where this message is expected, no other event that is expected at another point in the scenario is allowed to occur.

In order to explain the message temperature and execution kind in more detail, we must first introduce the concepts of *unification*, *active MSDs* and the *cut*: We say a diagram message can be *unified* with a message event if its sending and receiving lifeline represent the sending and receiving object of the message event and the diagram message and the message event both refer to the same operation. When an event occurs in the system that can be unified with the first message in an MSD, an *active MSD* is created. As further events occur that can be unified with the subsequent messages in the diagram, the active MSD progresses. This progress is represented by the *cut*, which marks for every lifeline the locations of the messages that were unified with the message events. If the cut reaches the end of an active MSD, the active MSD is terminated.

The semantics of the messages temperature and execution kind is as follows. If the cut is in front of a message on its sending and receiving lifeline, the message is *enabled*. If a hot message is enabled, the cut is also *hot*. Otherwise the cut is *cold*. If an executed message is enabled, the cut is also *executed*. Otherwise the cut is *monitored*. A *violation* of an MSD occurs if a message event occurs that can be unified with a message in the MSD that is not currently enabled. If the cut is hot, it is a *safety violation*; if the cut is cold, it is called a *cold violation*. Safety violations must never happen, while cold violations are allowed to occur and result in terminating the respective active MSD. If the cut is executed, this means that the active MSD must progress and it is a *liveness violation* if it does not. Instead, an active MSD is not required to progress in a monitored cut.

A (universal) MSD *accepts* an infinite sequence of message events in an object system, also called a *run* of an object system, if it does not lead to a safety or liveness violation of that MSD. An object system *satisfies* an MSD specification (consisting of a set of universal MSDs), iff all possible runs of the object system are accepted by all universal MSDs. We assume that at some point the specification will be implemented by a software *controller* for the system objects. This controller can be a single, centralized control program for all system objects, or it can be a set of distributed controllers, e.g., one controller per system object. We say that a controller for the system objects *implements* an MSD specification if the closed system formed of the system controller with any possible environment are accepted by all universal MSDs in the specification. Additionally it is

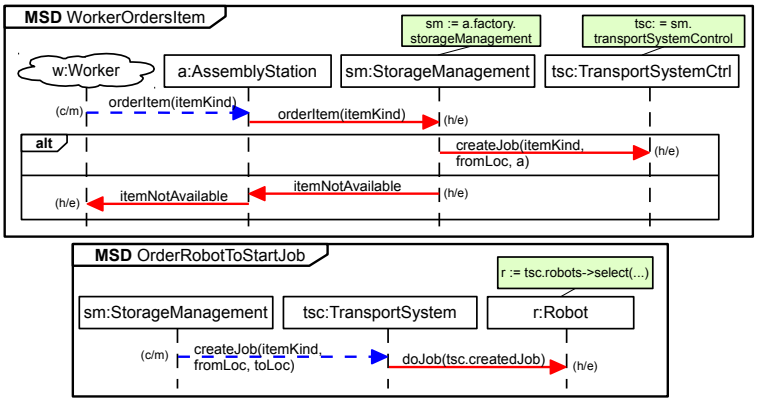

Fig. 3. MSDs to order an item.

assumed that the system is always fast enough to take any finite number of steps before the next environment event occurs [12]. Note that an MSD specification can contain contradictions and then no implementation exists [9,2,7,8].

For more details on the MSD semantics, we refer to Harel and Maoz [10]. Note, however, that our interpretation of the message modalities differs slightly from the original definition where hot messages also encode the liveness requirement (must eventually occur). In our interpretation, the execution kind defines whether a message may or must eventually occur. Hot messages are typically also executed and cold messages are monitored, but there are also cases where hot monitored messages (may occur but must not be violated) or cold executed ones (must eventually occur but may be violated) are used.

As an example for an MSD consider the MSD WorkerOrdersItem in Fig. 3. It says that when a worker tells the assembly station to order an item of a particular kind, the assembly station must send an order to the storage management. Then the storage management can either reply that the item is not available; in that case the assembly station must then forward this information to the worker. Alternatively, the storage management can command the transport system to create a job (for some robot) to pick up an item of the given kind a certain location and deliver it to another location. When the message `createJob` is sent, this activates a copy the MSD OrderRobotToStartJob which requires that then a robot is ordered to do the job[6].

This example MSD also introduces several advanced concepts. First, it is possible that lifelines do not only represent one particular object, but they can be *symbolic* and represent any object of a certain class [12, Chap. 7]. As events occur between certain objects, lifelines can be *bound* dynamically to objects. The

[6] Here the job sent to the robot is the one that the transport system control points to via its createdJob link. We assume that this link points to the job that was created last. However, how we model the creation of a job will be explained in Sect. 3, where we also introduce a more elegant way of assigning the new job to the robot.

sending and receiving lifelines of the first message are bound during the unification of the first message with a message event. The objects that the remaining lifelines are bound to are specified by *binding expressions* that are attached to the lifelines. In our case, these expressions are OCL expressions where lifelines names can be used as variables. For more details we refer to Brenner et al. [3]. Note that there can be several active copies of the same MSD with different lifeline bindings, or with the same lifeline bindings, but then with different cuts.

The second advanced concept is that messages can have *parameters*. A list of parameters that a message has is defined by the operation of the message. Parameters can have a primitive type, e.g. Boolean, integer, string, or they can by typed by classes. A message event must carry *values* for each parameter that the operation defines, which are thus concrete primitive values or, in the case that the parameter is typed by a class, pointers to objects. A diagram message in an MSD can specify values for message parameters, either by defining constant values or by referring to lifeline names, or other variables.

For example, by referring to the lifeline a in the MSD WorkerOrdersItem, we specify that the destination of the transport job should be the assembly station where the worker placed the order initially. (See that the third parameter of the operation TransportSystemControl.createJob(...) is toLoc.)

An MSD can also contain further variables, called *diagram variables*, which are only visible in the scope of an active MSD. They can be *bound* or *unbound* if no value was yet assigned to them. In the MSD WorkerOrdersItem, for example, the variable itemKind specifies the parameter value for the two `orderItem` messages. Initially, the variable is unbound and in that case the diagram message can be unified with any `orderItem` message sent between a worker and an assembly station, regardless which item kind object it carries as parameter value. After unification, the variable itemKind is bound to the item kind object carried by the unified message event. For the next `orderItem` message sent from the assembly station to the storage management, the diagram variable itemKind is bound and, in that case, the diagram message can only be unified to a message event when the carried parameter value matches the specified value.

If a message event occurs that can be unified with the diagram message, but only carries a parameter value that does not match the specified value, this is a violation of the MSD (cold violation or safety violation, depending on the cut temperature). In the MSD WorkerOrdersItem, this means that the item kind transmitted to the storage management (msg. 2) and the item kind for the creation of the job (msg. 3) must be the same item kind as originally sent by the worker to the assembly station (msg. 1). For more details on message parameters we refer to Harel and Marelly [12, Chap. 7] and Brenner et al. [3].

The third advanced concept is the alt-fragment, which allows us to specify decisions or non-deterministic choices. Here there is a non-deterministic choice whether to create a job or to reply that an item is not available. What this decision depends on can be modeled in another MSD that, for example, checks whether an item of that kind is available in a warehouse. We omit this for brevity.

An MSD specification can be executed by the *play-out* algorithm, which provides an operational semantics to MSD/LSC specifications [12,15]. It roughly works as follows: when an environment event occurs that activates or progresses one or multiple MSDs into cuts where executed system messages are enabled, then a system event is executed that can be unified with one of the enabled executed system messages and does not lead to a safety violation.

We recently extended the play-out algorithm to execute not only MSD specification consisting of MSDs that describe what the system objects are *required* to do, but we also support *assumption MSDs* that describe assumptions on what the environment can, will or will not do. We can think of the set of assumption MSDs, also called *environment assumptions*, as the dual to the requirements: a system is expected to satisfy its requirements as long as the environment satisfies the assumptions [8]. This extension of play-out is implemented in ScenarioTools [3]. We give an example of an assumption MSD in Sect. 3.

The ScenarioTools play-out supports messages that can have simple side-effects on the objects in the object system. For example, by convention, if a class defines an attribute a:⟨Type⟩ and an operation setA(a:⟨Type⟩) (with a parameter of the same type), then message events referring to that operation will change the attribute value of the receiving object according to the value carried by the message event. This also works for single-valued references. Maoz et al. describe an implementation of the play-out algorithm that supports the creation of objects [16]. Complex changes, for example, the creation of a job object as shown in Fig. 2, with its links to other objects, are currently very difficult to express; they require one message per creation of an object or link.

2.2 Graph Transformation Rules

Graph transformation rules (GTRs) [6] describe changes on a typed graph in a declarative way. Since software models can be considered graphs, typed by their meta-model, GTRs can be used to describe changes on models.

An existing graph, called *host graph*, is changed into a *target graph* using a graph transformation rule, which consists of a left-hand and a right-hand side as shown on the left of Fig. 4. They are marked with LHS and RHS, respectively. The figure shows the GTR arrived that describes the movement of an agent from one location oldLoc to another location newLoc.

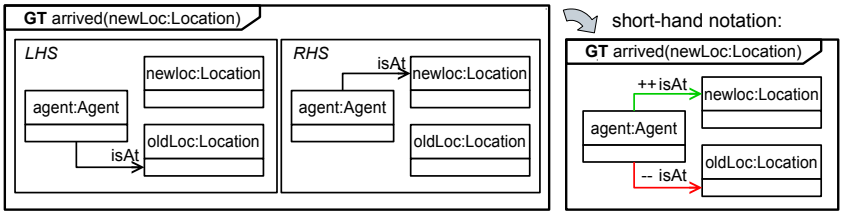

Fig. 4. Two representations of a GTR for an agent arriving at a new location

The left-hand side defines a pattern for which a *match*, an isomorph subgraph, needs to exist in the host graph in order to apply the rule. The right-hand side defines the replacement to be performed on the host graph that changes it into the target graph. Hence, the two sides of the rules can be interpreted as follows: (1) nodes and edges occurring on the left-hand and right-hand side are kept in the host graph, (2) nodes and edges occurring on the left-hand but not the right-hand side are removed from the host graph, and (3) nodes and edges occurring only on the right-hand side are added to the host graph.

In our example, the left hand side requires the agent to be at a location. The right-hand side defines that the agent must be at another location after the transformation.

We use a short-hand notation for GTRs as shown on the right of Fig. 4. Elements marked red and with "−−" belong to the right-hand side, element marked green and with "++" belong to the left-hand side. Unmarked elements belong to both sides.

There exists a range of tools that support the modeling and execution of graph transformations. They often add concepts like positive and negative *conditions*. Positive conditions are additional conditions that must hold in order to apply the rule. Conversely, negative conditions, also called *negative application conditions* (NACs), must not hold in order to apply the rule. Conditions can be specified using additional graph patterns or expressions, for example in OCL.

ModGraph [20] is a tool for model-driven software engineering with GTRs. It is based on and built for the Eclipse Modeling Framework (EMF) [18]. The vision of ModGraph is to provide a model-driven software engineering tool that combines the advantages of EMF, Xcore[7] and ModGraph's GTRs. EMF, with its meta-modeling language Ecore, supports the modeling of object-oriented structures. Xcore is a textual language for Ecore, extended with the programming language Xbase. On top, ModGraph's GTRs provide a higher level of abstraction for operations that involve complex matching and transformation.

A ModGraph GTR implements an operation defined in an Ecore or Xcore class model. A rule comprises a rule pattern in short-hand notation (as shown in Fig. 4) and, optionally, textual pre- and post-conditions and graphical NACs. If the operation is called on an object, the rule, if applicable, will be applied. If the rule is not applicable, an exception is thrown.

A graph pattern can consist of several kinds of nodes. First, there is a special node, called the *current node*, which is named this. This node represents the object on which the operation is called. When the operation is called on an object, this node is *bound* to the called object, which means that, in order to apply the rule, a match of the LHS-pattern must be found in the model where the this-node maps to the called object.

Also other nodes in the rule can have a pre-defined binding. If a node's name equals the name of an accordingly typed parameter of the operation, these nodes, when the operation is called, will be bound to the objects that are provided as parameter values by the call. Again the match for the rule's LHS must respect

[7] `http://wiki.eclipse.org/Xcore`

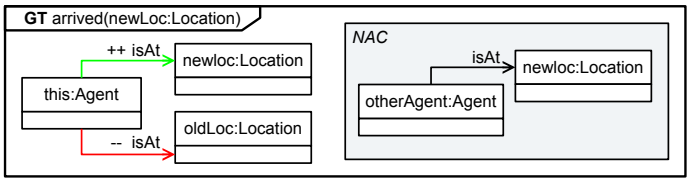

Fig. 5. The GTR for Agent.arrived(newLoc:Location) with a current node and NAC

these pre-defined node bindings. Parameter names can also be used in conditions and nodes with pre-defined bindings can also appear in graphical NACs.

All other nodes are *unbound* and can be mapped to any object in a match.

Figure 5 shows a modified version of the GTR arrived. We suppose that arrived(...) is an operation of the class Agent. The agent node is now the this-node. The node newLoc has a pre-defined binding due to the operation's corresponding newLoc-parameter. The node oldLoc is unbound and will be bound to whatever location the agent is at the time the operation is called. The figure also shows a NAC that says that the rule can only be applied when there is currently no (other) agent at the new location. This expresses that, in our factory example, only one robot may be at a warehouse or assembly station at a time; we can think of each location having only one loading/unloading apparatus.

Technically, for execution, ModGraph GTRs are transformed into Java code or Xcore operations. The transformation to Xcore enables the indirect interpretation of the GTRs [20].

3 Integration of MSDs and GTRs

The basic idea of our integration of MSDs and GTRs is straightforward. As before, we use GTRs to describe implementations of operations. As message events occur during a system run, GTRs are executed as side-effects. More specifically, for each message event referring to an operation that is implemented by a GTR, that GTR is executed. The execution is synchronous, which means that the next message event occurs only after the execution of the GTR is completed.

In addition, GTRs can also constrain the allowed sequences of events: We define that, if the *precondition* for applying a GTR is *not satisfied*, that is, there is no match for the LHS, a positive precondition is not satisfied, or there is a match for a NAC, then this implies that *the corresponding event must not occur*. In other words, an occurrence of an event that demands the execution of an inexecutable GTR leads to a *safety violation*. If the event is a message sent by a system object, then it is a safety violation of the *requirements*; if it is a message sent by an environment event, it is a safety violation of the *assumptions*.

In the following, we illustrate the integration by two examples:

As a first example, consider the two GTRs that implement the operations TransportSystemControl.createJob(...) and Agent.doJob(job:Job) shown in Fig. 6.

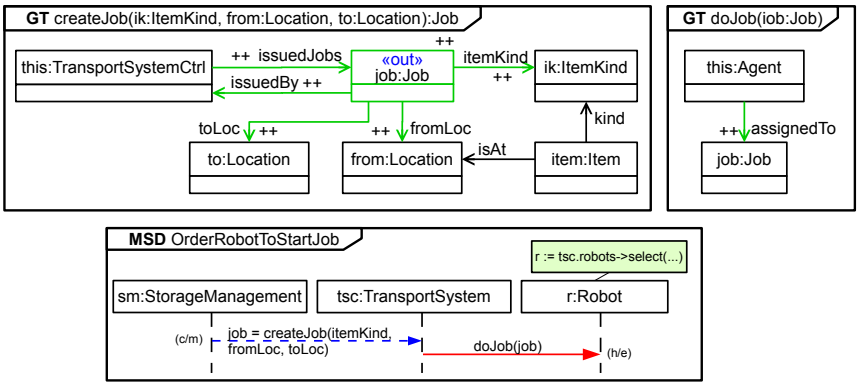

Fig. 6. GTRs for creating a job and assigning it to an agent with a more elegant version of MSD OrderRobotToStartJob

While the second could be modeled equally with a message referring to an operation Agent.setAssignedTo(job:Job) (see the convention for set-messages explained in Sect. 2.1), the structural change intended by TransportSystemControl.create-Job(...) is much more elaborate and the GTR provides a concise, visual way for modeling the creation of a job object and the setting of the all the links.

Furthermore, the LHS of the rule also contains an item node. This node will not be connected to the job via any link—its only purpose is to constrain the application of the rule in such a way that the rule will be applied only if at least one item of the specified kind is at the the specified pick-up location. If this is not the case, sending the respective message event would be a safety violation of the requirements.

We furthermore extend the integration so that now an operation's return value can be assigned to a MSD diagram variable. We extend the example so that now the operation TransportSystemControl.createJob(...) returns the newly created job. In the MSDs, we then use the return value. In the new version of the MSD OrderRobotToStartJob as shown on the right of Fig. 6, we use the reference to the newly created job to more easily model that the newly created job must be assigned to a robot (cf. Fig. 3). This way, we no longer require the association TransportSystemControl.createdJob to point to the newly created job (see the class diagram in Fig. 2).

Figure 7 shows the MSD RobotMoveToPickUpLocation. It specifies that the Robot, after being ordered to perform the job, must move to the pick-up location as indicated by the job (Job.fromLoc). This is modeled as a message to the environment, which abstracts from the robot's software controller ordering it's drives to physically move to the location. The arrival is modeled as a message from the environment to the robot, which abstracts from the robot's sensors telling the robot that it arrived at the desired location.

Upon arrival at that location, which is a warehouse that will be bound to the w:Warehouse lifeline, the robot must order the warehouse to load an item of

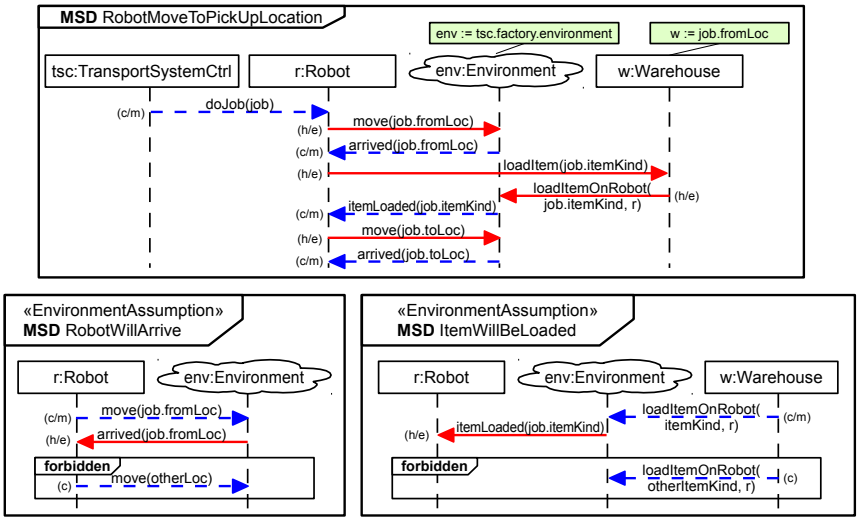

Fig. 7. MSD for a robot to execute a job

the kind specified by the job onto the robot. Again, we abstract by a message to the environment that the warehouse's software orders some physical/mechanical loading mechanism (maybe even a human worker) to load an item onto the robot. Also, the effective loading of the item onto the robot, which will be recognized by a sensor of the robot, is again modeled as a message from the environment to the robot. After the item is loaded, the robot moves to the destination as specified by the job (Job.toLoc). The unloading of the item is modeled in another MSD that we omit here for brevity.

There are two aspects about the process modeled in the MSD RobotMoveToPickUpLocation that are not expressed in this diagram.

The first missing aspect is that arriving at a certain location is a *spatial change* of the robot in the factory. It should be accompanied with a structural change in the object system. We model this with the GTR arrived that we already discussed previously (see Fig. 5). Note that, due to the NAC, this rule is only applicable if no other agent is currently at the target location. Since `arrived(...)` is an environment message (sent by an environment object), an occurrence of that message in this case would lead to a safety violation of the environment assumptions. It means that we assume that this will never happen.

Extending the play-out algorithm to consider the safety properties implied by GTRs is conceptually quite simple: The play-out algorithm selects only events for execution that do not lead to safety violations in any MSDs. Now, additionally, we only need to check that events selected for execution do not violate an application precondition of a corresponding GTR. The technical dimension for realizing this in our tool environment is a little more involved, as will be explained in Sect. 4.

Second, the diagram RobotMoveToPickUpLocation does not model that we *assume* that when a robot moves to a location, it will eventually arrive there. That is, when the third message in RobotMoveToPickUpLocation is enabled, the environment could also decide that the robot arrives at a different location, which would lead to a cold violation of the diagram. Also, it may never arrive anywhere, i.e., the environment will not send any `arrived(...)` message. In both cases, the MSD RobotMoveToPickUpLocation will not progress.

To express that we assume that the robot will also arrive at the location that it moves to, we need the assumption MSD RobotWillArrive as shown on the bottom left of Fig. 7. It models that if a robot starts moving to a certain location, it will eventually arrive at that location. The forbidden message says that if the robot decides to move to another location before arriving at the previously indicated location, we do not assume that it will arrive at the previously indicated location. The idea behind the assumption MSD ItemWillBeLoaded is very similar.

4 Integrating ScenarioTools and ModGraph

In the following, we describe how we implement the integration of MSDs and GTRs by integrating the tools ModGraph and ScenarioTools.

The interaction between both tools is shown in Fig. 8. In ScenarioTools MSD specifications are modeled in UML, using the Papyrus editor (see step 1 in Fig. 8). UML is extended with a profile to add modalities to sequence diagram messages, for example. The UML class model is then transformed into an Ecore class model (step 2), from which an object system can be instantiated (step 5). Based on the object system, ScenarioTools can interpret the MSDs and perform play-out (step 6) [3].

When integrating ModGraph with ScenarioTools, before performing play-out, we model GTRs and compile them into an executable Xcore model. The basis for modeling GTRs with ModGraph is the Ecore model created in step 2. The behavior of the operations in the Ecore class model can be specified by GTRs (step 3). These GTRs are then compiled into an Xcore model (step 4). The Xcore implementation of the GTRs can now be called by ScenarioTools when corresponding message events are executed during play-out.

In the Xcore model, for each GTR, two Xcore operations are generated, a *check-operation* and a *do-operation*. The check-operation is used to check the precondition for the applicability of the rule; the do-operation executes the transformation. When the ScenarioTools play-out selects possible messages events for execution, it first calls the check-operation. Only if this message returns a valid match of the precondition, play-out may choose to safely execute the corresponding message event. Otherwise, as described in Sect. 3 executing the message leads to a safety violation.

One limitation of our tool integration is that currently ScenarioTools only supports messages with one parameter. We plan to extend ScenarioTools so that multiple parameters will be supported. For realizing our example with the current limitation, we use multiple messages to transmit each parameter individ-

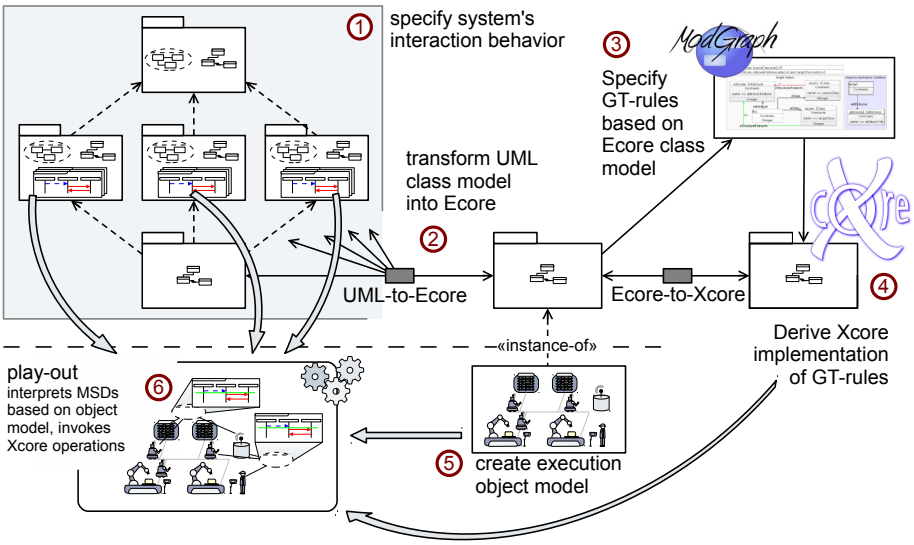

Fig. 8. Overview of the ScenarioTools-ModGraph-integration

ually. This complicates the current example implementation, but conceptually, the tool integration demonstrates a successful integration of the two modeling paradigms.

5 Related Work

While there is extensive work on scenario-based specification and analysis approaches based on LSCs/MSDs or other kinds of sequence diagrams, e.g. STAIRS [14], to the best of our knowledge, none of the them rigorously supports the reconfiguration of the participating objects or components at run-time.

Thus, we will in the following discuss two different approaches that combine models for structural reconfiguration behavior and message-based interaction behavior.

The MechatronicUML [1] is a design method for self-adaptive mechatronic systems. This method consists of a family of languages for modeling real-time behavior and architectural reconfiguration [17]. The behavior of the components is specified using state machines with real-time annotations. The architectural reconfiguration is specified using graph transformations on the component structure. Similar to our approach, the execution of graph transformation changes the structure of the active components and their behavior. However, in MechatronicUML, the message based interaction is defined by intra-component state machines and not inter-component scenario models. For the early design of complex interaction behavior, the latter are much more intuitive.

Diethelm et al. [5] take a complementary approach for the combination of scenarios and graph transformation. They use a set of simple graph transfor-

mation scenarios as input and synthesize a state machine which contains the graph transformations in the states. The basic idea is that all similar graph transformations are mapped to a common state in the state machine. An additional difference to our approach is that they do not consider that the graph transformations can change the object structure, which in turn would affect the execution of the scenarios as in our approach.

6 Conclusion and Future Work

In many software-intensive systems, there is a tight interdependency between the message-based interaction of its components and the structural dynamics of the system. In order to intuitively, yet precisely design such systems, we presented an approach that integrates scenario-based specifications using MSDs with graph transformation. MSDs support an incremental refinement and extension of the message-based interaction behavior and GTRs offer easy to understand, declarative, pattern-oriented means for expressing structural change. The integration of the two formalisms works in two ways: structural transformations are executed as side effects of messages, but GTRs can also constrain when certain actions can be performed.

One interesting direction of future research is how to systematically and efficiently analyze the resulting specification for realizability. Simulation via play-out is of course a first method to search for contradictions, but one can hardly be sure to simulate all possible sequences of events in all structural configurations. We are working on an extension of the ScenarioTools realizability-checking capabilities [7] to be able to explore different object system reconfigurations.

Acknowledgments

We thank Fabian Schmidt for his work on the factory example.

References

1. Becker, S., Dziwok, S., Gerking, C., Schäfer, W., Heinzemann, C., Thiele, S., Meyer, M., Priesterjahn, C., Pohlmann, U., Tichy, M.: The MechatronicUML design method – process and language for platform-independent modeling. Tech. Rep. tr-ri-14-337, Heinz Nixdorf Institute, University of Paderborn (Mar 2014), version 0.4
2. Bontemps, Y., Heymans, P.: From live sequence charts to state machines and back: A guided tour. Transactions on Software Engineering 31(12), 999–1014 (2005)
3. Brenner, C., Greenyer, J., Panzica La Manna, V.: The ScenarioTools play-out of modal sequence diagram specifications with environment assumptions. In: Proc. 12th Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2013). vol. 58. EASST (2013)
4. Damm, W., Harel, D.: LSCs: Breathing life into message sequence charts. In: Formal Methods in System Design. vol. 19, pp. 45–80. Kluwer Academic (2001)

5. Diethelm, I., Geiger, L., Maier, T., Zündorf, A.: Turning collaboration diagram strips into storycharts. Florida, Orlando, USA (2002)
6. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Berlin, illustrated edition edn. (2006)
7. Greenyer, J., Brenner, C., Cordy, M., Heymans, P., Gressi, E.: Incrementally synthesizing controllers from scenario-based product line specifications. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. pp. 433–443. ESEC/FSE 2013, ACM, New York, NY, USA (2013)
8. Greenyer, J., Kindler, E.: Compositional synthesis of controllers from scenario-based assume-guarantee specifications. In: Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P. (eds.) Proc. 16th Int. Conf. on Model-Driven Engineering Languages and Systems (MODELS 2013), Lecture Notes in Computer Science, vol. 8107, pp. 774–789. Springer Berlin Heidelberg (2013)
9. Harel, D., Kugler, H.: Synthesizing state-based object systems from LSC specifications. In: Foundations of Computer Science. vol. 13:1, pp. 5–51 (2002)
10. Harel, D., Maoz, S.: Assert and negate revisited: Modal semantics for UML sequence diagrams. Software and Systems Modeling (SoSyM) 7(2), 237–252 (May 2008)
11. Harel, D., Marelly, R.: Specifying and executing behavioral requirements: The play-in/play-out approach. Software and System Modeling (SoSyM) 2(2), 82–107 (2002)
12. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer (August 2003)
13. Harel, D., Marron, A., Weiss, G.: Behavioral programming. Commun. ACM 55(7), 90–100 (Jul 2012)
14. Haugen, Ø., Husa, K., Runde, R., Stølen, K.: STAIRS towards formal design with sequence diagrams. Software & Systems Modeling 4(4), 355–357 (2005)
15. Maoz, S., Harel, D.: From multi-modal scenarios to code: Compiling LSCs into AspectJ. In: Proc. 14th Int. Symp. on Foundations of Software Engineering. pp. 219–230. SIGSOFT '06/FSE-14, ACM, New York, NY, USA (2006)
16. Maoz, S., Harel, D., Kleinbort, A.: A compiler for multimodal scenarios: Transforming LSCs into AspectJ. ACM Trans. Softw. Eng. Methodol. 20(4), 18:1–18:41 (Sep 2011)
17. Priesterjahn, C., Steenken, D., Tichy, M.: Timed hazard analysis of self-healing systems. In: Javier Camara, R.d.L., Ghezzi, C., Lopes, A. (eds.) Assurances for Self-Adaptive Systems, Lecture Notes in Computer Science, vol. 7740, pp. 112–151. Springer Berlin Heidelberg (2013)
18. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. Addison-Wesley, Boston, MA, 2 edn. (2009)
19. Winetzhammer, S.: ModGraph – generating executable EMF models. In: Margaria, T., Padberg, J., Taentzer, G., Krause, C., Westfechtel, B. (eds.) Proc. 7th Int. Workshop on Graph Based Tools (GraBaTs'12). Electronic Communications of the EASST, vol. 54, pp. 32–44. EASST, Bremen, Deutschland (September 2012)
20. Winetzhammer, S., Westfechtel, B.: Compiling graph transformation rules into a procedural language for behavioral modeling. In: Pires, L.F., Hammoudi, S., Filipe, J., das Neves, R.C. (eds.) Proc. 2nd Int Conf. on Model-Driven Engineering and Software Development (MODELSWARD 2014). pp. 415–424. SCITEPRESS Science and Technology Publications, Portugal, Lisbon, Portugal (2014)