Visual Specification of Structural and Temporal Properties

Holger Giese and Florian Klein†
Software Engineering Group, Department of Computer Science,
University of Paderborn, D-33098 Paderborn
hgjklein@uni-paderborn.de

ABSTRACT

The UML has become the de-facto standard in software engineering. Due to the visual nature and accessibility of its structural diagrams, it is widely accepted as the tool of choice for structural modeling. However, for specifying structural properties that go beyond cardinalities, the UML only provides a textual specification language, the OCL. For mixed structural and temporal properties, only proprietary combinations of OCL with temporal logic exist today. The intricate nature of both OCL and temporal logic already causes problems for many software engineers. When communicating with people without a computer science background, e.g., domain experts, employing OCL, any dialect of temporal logic, or a mix of both is usually impracticable. In this paper, we propose a visual language for specifying requirements including structural as well as temporal aspects. Based on an extension of Story Patterns, our approach will allow specifying scenarios that contain requirements concerning structural dynamics within Fujaba. In addition, we present a scheme for turning a specification into a powerful behavioral monitor, enabling us to verify dynamic structural properties of models at run-time or in a model checker.

1. INTRODUCTION

The popularity of the UML is in large part due to its visual nature and accessibility of its structural modeling concepts. However, for specifying more detailed structural properties, the UML only provides a textual specification language, the OCL [20]. The writing of OCL properties requires that the developer translates his/her concrete ideas about the required structural properties from the familiar structural view in form of class and object diagrams into an often intricate textual syntax. When reading OCL, a complicated and error prone translation in the opposite direction is required.

This mental mapping problem of textual OCL is already problematic in most standard software engineering environments, where OCL is therefore rarely employed. Important structural properties are often not documented, and information to this effect is lost in the course of the development process, as no tool besides natural language seems to be able to capture them, at least not economically.

For temporal logic such as LTL or CTL [7], the situation is even graver. As reported in [8], developers (even experts) have serious problems handling the intricate nature of these logics. Even in projects with very well trained experts, employing them is often impossible, as the resulting property specifications will usually be unintelligible to domain experts from other disciplines that need to participate in the effort.

For example, this problem becomes a serious hinderance when software engineers develop the software for complex mechatronic systems which also involve complex control engineering, mechanical engineering, and electrical engineering. As part of the trend towards more intelligent, efficient, and flexible mechatronic systems, dynamic software architectures which permit structural adaptation at run-time are beginning to displace static architectures and models. While this permits building systems that change in response to current needs, designing and validating such adaptable systems poses new challenges to software engineers, as the involved structural and temporal aspects are closely intertwined.

In this paper, we discuss how visual languages can be used for specifying structural as well as dynamic properties. First, we show how Story Decision Diagrams (SDD) [12] can be used to capture structural requirements. SDD are an extension of Story Patterns [17], combining the intuitive concept of matching structural patterns with decision diagrams, which foster a consecutive if-then-else decomposition of complex properties into comprehensible smaller ones. We then introduce Timed Story Scenario Diagrams (TSSD), a new notation inspired by the Visual Timed Event Scenario approach [1], as a way of capturing dynamic properties. They provide conditional timed scenarios describing the partial order of specific structural configurations. In addition, we present a scheme for turning specifications into powerful monitors which enable the verification of models w.r.t. dynamic structural properties using a model checker that supports structural evolution.

After reviewing and discussing the current state of the art in Section 2, we introduce our application example from the mechatronic domain in Section 3. The concepts for modeling structural properties are introduced in Section 4 and combined with the ones for modeling temporal properties in Section 5. By providing a mapping from the property specification to operational detector behavior in Section 6, we can demonstrate the practical validation of these properties. Finally, the paper provides a conclusion and an outlook on planned future work.
2. RELATED WORK

Visual Structural Properties. Constraint diagrams [15] visualize constraints as restrictions on sets using Euler circles, spiders and arrows. To compensate for the decrease in expressive power w.r.t. the OCL, constraint trees [16] combine them with the idea of parsing an OCL statement into a tree, replacing only selected constraints with constraint diagrams. The downside is that while quantification on sets is intuitive, structural constraints quickly result in intricate, visually complex diagrams with little relation to the original UML specification.

VisualOCL [3] is an approach that focuses on mapping OCL syntax to a visual format as closely as possible, thus facilitating the parsing of structural constraints. Based on the theory of graph grammars, Story Patterns (cf. [17]), an extension of UML Object Diagrams, are an alternative approach which can also be used for specifying constraints. Like most approaches that extend UML Structure Diagrams, they are very accessible, but in turn have deficits when it comes to quantification and negation.

Visual Temporal Properties. Several notations for scenarios as a means to visually describe temporal behavior have been proposed: UML 1.x sequence diagrams or message sequence charts have been employed to specify and check timed properties (cf. [19]). However, they are usually considered as not expressive enough, as only a set of runs or one specific run of the system, but no conditional properties, can be described. Therefore, the interpretation w.r.t. the system is usually unclear. This limitation has been tackled by a number of approaches such as live sequence charts (LSC) [14] or triggered message sequence charts (TMSCs) [24], which add the ability to describe conditional behavior in a sequence diagram style notation. To some extent, these enhancements found their way into UML 2.0 sequence diagrams (cf. [21, p. 444] assert block).

Other approaches such as the Visual Timed Event Scenario approach [1] focus on scenarios for pure events, rather than the interaction of predefined units. Therefore, they provide a more intuitive notion of temporal ordering than sequence diagrams, which require specifying a sequence of interactions that “enforces” this ordering.

Another related thread of research are specification patterns for temporal properties. As outlined in [8], the overwhelming number of temporal properties can be covered using a rather small set of specification patterns. This idea has been extended and applied to real-time systems in [18]. However, all these scenario-based or specification pattern-based approaches focus on the purely temporal aspect of behavior, abstracting from its structural aspects. Statements concerning the required temporal behavior of expressive structural properties are not supported.

Most approaches which permit combining structural and temporal properties are extensions of the OCL towards the description of dynamics. Through the introduction of additional temporal logic operators in OCL (e.g., eventually, always, or never), modelers are enabled to specify required behavior by means of temporal restrictions among actions and events, e.g., [4]. Temporal extensions of the OCL that consider real-time issues have been proposed for events in OCL/RT [6] and for states in RT-OCL [10]. As temporal logic alone already causes an even more demanding mapping problem (cf. [8]), integrating the OCL and some temporal logic concepts at the textual level does not yield a sufficiently comprehensible solution.

Story Diagrams [9] extend UML Activity Diagrams with Story Patterns to provide them with operational semantics. Though visually similar to TSSDs, their purpose is different: They strive to specify exactly how something happens, while TSSDs focus on mechanisms to specify what and when should result.

3. APPLICATION EXAMPLE

Motivation. The RailCab R&D project is developing a system of autonomous shuttles travelling on a railway network, with the intent to combine the advantages of railways and automobiles, providing fast, safe, energy-efficient and convenient individual transportation. In order to achieve significant improvements over existing systems, the project combines traditional mechanical and electrical engineering with software engineering techniques. The project is representative of a new class of advanced mechatronic systems [5] using sophisticated control and coordination techniques such as structural adaptation, ad-hoc collaboration, or self-optimization in complex real world situations. The promise of more intelligent, efficient, and flexible systems has led to an increased interest in such mechatronic systems, notably in the automotive sector. However, these improvements come at a cost, as designing the required more complex software poses new challenges to software engineers. Advanced mechatronic systems typically run concurrently and with real-time requirements, are often distributed and heterogeneous, the relevant context for decisions is often characterized by complex structural properties, and their physical nature makes them safety-critical almost by default. Approaches for handling the additional levels of complexity and verifying system safety are thus required.

Throughout this paper, we will use an example that is inspired by the RailCab project. In previous work, we have used related examples to demonstrate the compositional verification of real-time coordination patterns [13], modular system coordination using social structures, and the verification of safety properties that are inductive invariants of the system [2]. Here, we focus on specifying the associated structural and behavioral system requirements in a manner that is expressive, accessible to domain experts, and yet operational and compatible with existing model checking and verification techniques.

Structure. The railway network is modeled as a graph of small track segments, each about as large as a shuttle. Tracks are unidirectional, they have one or two (branch) successors and are successor to one or two (join) tracks. Shuttles are located on one track and may have next relationships with other tracks to indicate where they are travelling.

Tracks are monitored by associated controllers. Shuttles can execute a registration pattern with a controller. The registration pattern is a real-time coordination pattern which ensures that a shuttle keeps the controller informed about its exact position and is in turn informed about the position of all other shuttles in the controller’s area of responsibility in regular intervals. This pattern is the foundation upon which another coordination pattern, the convoy pattern, operates. This pattern ensures that two shuttles in close proximity safely coordinate their behavior, which provides shuttles with the ability to reduce drag by forming
contact-free convoys. Figure 1 provides an overview of these elements.

![Diagram](image)

Figure 1: The elements of the shuttle system

The primary requirement we are considering is the absence of accidents. As the continuous control aspects are (correctly) encapsulated in the coordination patterns, we can analyze the safety of the system on a discretized world model by checking whether the correct coordination patterns exist in all specific instance situations, i.e. evaluating the structural correctness of the system.

Properties. We now derive the properties we will formalize below. First of all, no two shuttles may share a track, as this would correspond to a collision. In order to make shuttle behavior predictable, shuttles need to mark the next two tracks they will use. Furthermore, if there is a shuttle right in front of another, the shuttles have to execute the convoy pattern in order to avoid collisions. As the convoy pattern depends on the registration pattern, both shuttles need to be registered with the same controller beforehand. Therefore, shuttles are required to register with all available controllers for their current position. To avoid problems when moving from one controller’s area to another, these areas overlap - we require that for each shuttle, there always exists a controller that covers both the shuttle and its two next tracks. Finally, we impose a structural constraint that the system contains no dead ends and all tracks are reachable from any other track.

4. STRUCTURAL PROPERTIES

The fundamental abstraction that our approach is based upon is the idea of interpreting instance situations of an object-oriented system as graphs. Informally, this seems intuitively plausible, as UML Object Diagrams as a common way of describing instance situations already have a graph-like structure. More specifically, we map each object to a node and each attribute/association to an edge of a graph-like structure. More specifically, we map each object to a node and each attribute/association to an edge of a graph-like structure. The theory of graph transformation systems (cf. [23]) then provides the formal semantics that are typically missing from UML-based notations, which allows reasoning about states and behavior of object-oriented systems modeled using a visual notation.

Story Patterns are an extended type of UML object diagram (cf. [17]) that allow expressing properties and transformations, especially structural changes. They consist of a precondition, the left hand side (LHS), and a postcondition, the right hand side (RHS). It is possible to define negative application conditions by crossing out elements of the diagram; however, it is not possible to forbid groups of elements or forbid only elements with specific associations.

Story Patterns without side effects, i.e. with identical LHS and RHS, can be used to describe and allow testing for system properties. E.g., the Story Pattern in Fig. 2 matches if a shuttle’s on and next associations point to adjacent tracks in the proper order. A translation into OCL is provided below the figure. For our example, we would like this property to be a positive invariant of the system that is true for all shuttles. However, there is no way to make this explicit in the pattern.

![Diagram](image)

Figure 2: Story pattern: a simple positive invariant

In [2], we used Story Patterns to specify invariants of the system that represented forbidden states (accidents, hazards), which could then be formally verified. This required the implicit convention that all patterns represented negative invariants of the system, which could not be indicated explicitly. The resulting restriction to negative invariants entailed the use of untintuitive multiple negations, i.e. representing a required element as a forbidden element of a forbidden pattern.

Story Decision Diagrams (SDD) [12] are an extension of Story Patterns that allow expressing more complex properties while retaining or surpassing the intuitiveness of the original visual notation. The most significant enhancements they provide are quantors, implication, alternatives, negation of complex properties, and a concept for modularity.

An SDD is a directed acyclic graph (DAG). Each node contains a Story Pattern without side effects that specifies some property. The nodes are processed starting from the root node. Each node in the DAG can essentially be seen as a local if-then-else decision for a binding. If a match is found, we follow the solid then connector; if no match is found, we follow the dashed else connector. New class and association bindings, i.e. successfully matched elements, are added to the current binding and propagated to subsequent elements. There are two special leaf nodes, (1) signifying true and (0) signifying false. When a binding reaches a leaf node, it evaluates to true or false, respectively.

Figure 3 encodes the requirement that for any three consecutive Tracks (\(\triangledown\)) there must be a new Controller (\(\exists\)) supervising them all. The forall quantifier indicates that every binding for the first node needs to reach a (1) node eventually, whereas the existential quantifier only requires that one valid binding exists.

Figure 4 marks the existence of an accident (two Shut- tles on the same Track) as an undesired instance situation. The negation is modeled by switching the then and else connectors, i.e. a match leads to failure and no match leads to success. There are only positive nodes in the patterns, which facilitates their interpretation. Complex negative conditions can be expressed by chaining the corresponding nodes.

\(^1\)Color is optional and only encodes redundant information.
Finally, SDDs provide modularity through the concept of **Embedded SDD (ESDD)** that encode nontrivial properties in a reusable fashion. ESDDs behave like patterns with free variables that can be bound depending on the respective current context. By inserting the ESDD into an SDD and binding concrete node instances to these variables, the ESDD can then be employed as a compact notation for the specified property. It is also possible to have nested ESDDs. This quite naturally leads to recursively defined patterns. Figure 5 requires that a Shuttle executes a correct ConvoyPattern with a Shuttle on its immediate next Track, and does not execute a correct pattern if the other Shuttle is not on one of its next Tracks. The existence of a correct ConvoyPattern is encoded by the ESDD Convoy in Fig. 6.\(^2\) It requires a common controller and matching RegistrationPatterns.

5. **TEMPORAL PROPERTIES**

So far, we have used graphs to express complex static properties. It is tempting to apply the same approach to temporal properties, e.g., in order to describe the structural adaptation of a system. In addition to the outlined concepts for structural properties, we then need to consider their occurrence and temporal ordering.

The temporal behavior of a system can be described as a sequence of states. Extending our former considerations, each state is described as a graph. The identity of nodes and edges is preserved between two system states, while attribute values are anonymous.

\(^2\)Standard leaf nodes can be omitted.

---

**5.1 Timed Story Scenario Diagrams**

**Observations.** The idea behind *timed story scenario diagrams* (TSSD) is to use the ordering of structural observations in order to specify valid orderings and temporal properties. Each such observation is made by an *observation node* containing an SDD.

**Edges.** Two structural observations can be related to each other using temporal ordering and constraint edges:
Figure 7: TSSD Syntax - A Shuttle registers with a Registry

(a) The temporal ordering edge \( (A \longrightarrow B) \) denotes that observation \( A \) is made before observation \( B \). Note that this includes that observation \( A \) and \( B \) may occur at the same point in time.

(b) The until temporal ordering edge \( (A \nexists\longrightarrow B) \) denotes that observation \( A \) is observed before observation \( B \) and that the structural property \( C \) is valid for all states between the observation \( A \) and the observation \( B \). If observations \( A \) and \( B \) occur at the same time, structural property \( C \) is never evaluated and does not need to be fulfilled.

(c) In addition to the temporal ordering edges, time constraint edges with lower bound \( a \) and upper bound \( b \) \( (A \nexists\longrightarrow [a..b] B) \) constrain the permitted time difference between the occurrence of two observations \( A \) and \( B \).

AND, OR, NOT. Observations are only partially ordered. If the TSSD branches, both paths progress independently and in parallel. If an observation node has multiple ingoing temporal ordering edges, all preceding observations need to occur before the observation is considered (AND).

The final node ✗ indicates a successful match. More formally, a system trace \( \pi \) fulfills a TSSD if ✗ has been reached for a prefix of \( \pi \). By using multiple ✗ nodes on independent paths, disjunction (OR) can be expressed.

The final node ◐ indicates a violation. Replacing ✗ with ◐ turns a path into a forbidden scenario (NOT), just as switching the leaf nodes in an SDD is used to express negation. If a trace reaches both types of final node at the same time, failure ◐ takes precedence over success ✗, resulting in a violation.

Triggers. However, TSSDs are not limited to simply recognizing and chronicling observations over time. Arbitrary initial fragments of a TSSD can be combined into a trigger. As long as the trigger sequence has not been completely detected, the resulting TSSD provides no constraint for the correct temporal behavior. However, once the trigger has been detected, the remaining elements of the TSSD define temporal properties which have to be fulfilled.

Figure 7 is a basic TSSD presenting the key elements of the syntax. When a Shuttle is approaching a Controller’s supervised area, they have between 0 and 100 time units to launch a RegistrationPattern. In the mean time, the Shuttle must not yet have entered the critical area, which is indicated by the (forbidden) state on the transition.

Subscenarios. Modularity is again of paramount importance for practical scalability. We therefore provide the ability to invoke a subscenario (see Fig. 8) that is defined elsewhere (see Fig. 9). In the example fragment, we use the subscenario to abstract from the details of convoy formation. Invocation works just like ESDD evaluation. However, as a subscenario might create bindings that are needed later on in the scenario, the invocation itself takes place inside a λ-node that allows exporting arbitrary bindings (e.g. pattern $\rightarrow c1$) back from the subscenario.

Quantification. A subtle aspect of TSSD is that they provide three different levels of quantification. First of all, observations are matched by SDDs and can thus be structurally equivalent but distinct instances of the same pattern. This is very different from typical event- or message-based approaches that do not consider structure and cannot differentiate between multiple (concurrent) instances of the same event. Consider a simple scenario where a student’s undergraduate program (trigger) requires her to choose a class and eventually complete it. Assume the class and its completion are existentially quantified in the SDDs; we therefore bear with her as she starts and abandons several classes un-
til she finally completes one. Multiple matches thus become part of a set of alternative bindings, only one of which needs to succeed. Secondly, we quantify over triggers, as seen in Fig. 7. We want the scenario to be triggered and successfully completed for all cases when a new matching Shuttle-Controller-pair is detected. Finally, we quantify over time: we require untriggered scenarios to match at least once (existential TSSD), whereas triggered scenarios always need to be fulfilled (universal TSSD).

5.2 Formal Semantics

Clear and intuitive semantics are of paramount importance for temporal properties. It is well known that temporal logics such as LTL or CTL [7] are hard to understand and even harder to write for any non-trivial property (cf. [8]).

If we want to be able to specify that a specific detected structural constellation will result in a different specific structural constellation within a certain time bound (see Figure 7), we not only have to consider individual states in form of a graph, but all sequences of states in form of graphs as generated by a graph transformation system (GTS) representing our system model. Graph-Interpreted Temporal Logic [11] provides a propositional temporal logic for such systems.

For a path π, we denote its potentially infinite length by l(π) and refer to its i-th state graph as π[i]. A function T(π, i) denotes the time when the i-th state has been reached and thus provides a concrete notion of time.

We can exploit the DAG property of all TSSDs to define their semantics. We propagate the π and bindings X between variables and instances that have been created by previous observations along the temporal and constraint edges to determine the valid bindings for each node. Then, the semantics for a node n for time offset t, (written [n]π,t) is given by those pairs (tn, Xn) of time tn and bindings Xn where tn fulfills certain conditions and Xn are valid bindings for n’s SDD sddn that extend the bindings generated by n’s direct predecessor nodes.

(tn, Xn) ∈ [n]π,t

iff (a) a set of pairs (tn′, Xn′) for all regular predecessor nodes N = {n′[n′ → n] ∨ n′[n′ → n]} exists whose combined bindings permit matching the SDD sddn of node n:

∀n′ ∈ N: ∃(tn′, Xn′) ∈ [n]π,t ∈ Xn ∈ sddn |comb(π,n′∈N)|,

with comb(\{X1, ..., Xn\}) the combination of the different bindings. In addition (b), the time offset ton is uniquely determined for the ton and ton = max{tn|n′ ∈ N} by the following conditions: (1) The node n cannot be fulfilled earlier by the same binding.

∀t′: ton ≤ t′ < ton : Xn ∩ [sddn]π,t′ = ∅

(2) In case of an until edge n′[n′ → n], the SDD for n′ must be satisfied until n is reached.

∀n′, n′[n′ → n] ∨ n ∀t′ : ton ≤ t′ < ton : [sddn′]π,t′ = ∅

(3) For a time constraint n′[\sum_{i=0}^{b} n[i → n]], the chosen time offsets must satisfy [a, b].

∀n′, a, b, n′[a → be n] : a ≤ [T(π, ton) − T(π, ton)] ≤ b

If no predecessor node exists (pred(n) = 0), we only require that the detection of the binding is recognized as early as possible. If ton is not zero, we thus require in this case:

∀t, ton ≤ t < ton : Xn ∩ [sddn]π,t = ∅

A pair (t, X) is a solution for a TSSD td for a path π and the time offset t, iff a final success node n of the TSSD exists with (t, X) ∈ [n]π,td such that for all X and (t′, X′) ∈ [n]π,td, hold that t′ > t or X[td∈π] ∩ X[td∈π] = ∅. We write [td]π,td to denote all solutions, i.e., all successful bindings reaching a success node before any failure node is reached.

Given this definition for [td]π,td, we can further define the semantics for universal and existential TSSD for a given GTS M and TSSD td with free variables free(td) as follows:

- A GTS M fulfills an existential TSSD φ = ∃td iff there exists a trace π, a time offset t, and an initial binding ξ ∈ X[free(td)] for which td can be satisfied: ∃π ∈ [M], t ∈ [0, l(π)], ξ ∈ X[free(td)] : [td]π,td(ξ) ≠ ∅.

- A GTS M fulfills the universal TSSD ψ = ∀td : td iff all traces π, all time offsets t, and all initial bindings ξ ∈ X[free(sdd)] holds that a matching binding for td has been found, then for the same binding, td will also eventually become true:

∀t(π, X) ∈ [td]π,td(ξ) : [td]π,td(ξ) ≠ ∅.

6. PROPERTY DETECTORS

Specifications using SDDs and TSSDs are not merely a tool for communicating and reasoning about structural and temporal properties, but can be used for verification or run-time monitoring the specified properties by running a property detector in parallel with the system. There are two fundamentally different ways to implement such a detector open to us. The first possibility is to start from the graph-based semantics, translate the property detector into a GTS and execute it inside a graph model checker, which is useful for verification and evaluating prototypes. The second, more efficient possibility is to use Fujaba to generate the detector as a Java or C++ program capable of monitoring an application. Here, we focus on the first option, which enforces a more rigorous approach, for assessing the feasibility of such detectors. Code generation is discussed in [12].

GROOVE is a GTS model checker [22], capable of simulating GTS and generating state spaces. For a GTS specifications, GROOVE can compute all reachable states of the transformation system, optionally bounded by the occurrence of a forbidden graph. We have previously developed a Fujaba plugin for exporting Story Patterns from Fujaba into GROOVE. While exporting SDDs is much more complex, we used this foundation to manually derive story detectors for evaluation.

6.1 Structural Properties

For structural properties, we have the choice of computing all alternative bindings exhibiting the property or just one set of bindings that fulfills the SDD. Pragmatically, we have chosen to stop the search as soon as the first solution is
found. A single SDD is transformed into many small graph rules (the ConvoyMode SDD resulted in 44 rules) following a repetitive pattern that could easily be automated, especially since the largest part of the rules are identical for all SDDs.

The basic algorithm works by implicitly traversing the SDD from its root to the leaves. On the way up, markers are used to store bindings and activate and inhibit the appropriate nodes. When a marker reaches a leaf, it is in turn marked up with the result, which is then propagated back down towards the root, along with a set of witnesses. Once a result marker reaches the root, the evaluation of the SDD is complete and all markers and results are cleared. There is only one type of marker; markers are identified by their position relative to the root element and connected through 1 and 0 edges, corresponding to then and else connectors. Each marker only stores the additional variables it binds; the complete binding is thus defined by the path from the marker to the root element. Figure 10 shows a snapshot of the matching process for the property from Fig. 2 – the shuttle to the left has already been marked as correct; the detector will next try to bind tracks t1, t2, t3 using the rule in Fig. 11 for the shuttle on the right.

Rule priorities are very important for the correct execution. (A) Cleanup has the highest priority. It is triggered by a result marker at the root element. (B) Next come the immediate propagation rules, i.e. success for existential and failure for universal nodes. Here, nodes closest to root have the highest priority to ensure immediate propagation. (C) Then the rule groups containing the main part of the rules follow. Here, nodes that are farther away from the root node have higher priority to ensure depth-first traversal. Inside each group, there are three rules per node: (2) if no matching 1 marker is present, the then rule tries to match the actual pattern and create a 1 marker (see e.g. Fig. 11). (2) Rules creating result markers at leaf nodes operate at this level as well. (1) The else rule creates a 0 marker if no 1 or 0 marker is present. (0) The return rule propagates failure (success) for existential (universal) nodes. (C') Nodes containing ESDDs have a fourth rule with highest priority (3) that triggers the ESDDs by creating a root marker with a binding for the free variables. The (2) then rule then additionally checks the result marker of the ESDD. The whole ESDD has higher priority than the current SDD.

When all rules were translated, we created a set of correct and incorrect instance graphs (see Fig. 12) and ran the SDD property detectors on them. All examples were evaluated correctly, yielding the expected results. Notwithstanding the large number of rules, evaluation was fast and efficient, as the individual rules were mostly small and only a limited number of them was active at any one time.

6.2 Temporal Properties

At the operational level, verifying temporal properties is remarkably similar to verifying structural properties with ESDDs. Whenever a positive result has been propagated down to an observation’s root marker, new root markers are created for all subsequent observations and the bindings are copied from the witness to the new root markers. However, the successful observation itself also remains active, ready to create new bindings. In Fig. 13, the scenario in Fig. 7 has just reached the success node. The result will now be propagated back down the F edges to the trigger, and all root markers will be deleted.

In our experiments, we chose not to implement the time constraint, which would have required an unwieldy auxiliary construction as GROOVE does not support time natively. However, we added a marker representing a time step that
inhibits all SDD rules in order to give the shuttle the opportunity to actually move between subsequent matching attempts of the SDD rules.

7. CONCLUSION AND FUTURE WORK

We have presented a visual approach for the specification of temporal and structural properties. We have shown how extensions of UML object diagrams can be employed for the description of complex structural conditions. Based on these notations, we have introduced timed scenarios as a natural way of specifying the temporal order and time constraints for a sequence of observations. The approach thus combines the specification of detailed structural properties and requirements concerning structural dynamics using a clear and intuitive visual notation. The presented formalization provides the required solid foundation for the soundness of the approach. The operational realization as Story Detectors that detect the specified properties based on the GTS model checker GROOVE show that the approach even works under the restricted conditions of a model checking engine. When specifying invariants using Fujaba in our previous work, we had to missappropriate Story Diagrams for that purpose. We intend to extend Fujaba with a systematic way to explicitly attach (visually specified) constraints to models and model elements. We are currently working on an implementation of the new notations as a plug-in for FujabaEclipse. We also plan to automate the export of SDDs and TSSDs into GROOVE and provide code generation for constraint monitors in Java and C++, extending the existing code generation facilities.

8. REFERENCES


