Integrated Visual Specification of Structural and Temporal Properties *

Florian Klein† and Holger Giese
Software Engineering Group, University of Paderborn,
Warburger Str. 100, D-33098 Paderborn, Germany
[fklein |hg]@upb.de

October 2006

Abstract

Complex software systems, and self-adaptive systems in particular, are characterized by complex structures and behavior. For their design, appropriate notations for the specification of properties that integrate structural and temporal aspects are required.

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 two visual languages for specifying requirements, Story Decision Diagrams for structural properties and Timed Story Scenario Diagrams for scenario specifications that integrate structural and temporal aspects. Based on UML Object Diagrams, our approach is capable of specifying both detailed static properties and requirements concerning structural dynamics. Combining structure, first order and temporal logic, it is more expressive than existing visual constraint and scenario languages. Based on the formal semantics we define, it is furthermore possible to turn a specification into a powerful behavioral monitor, enabling the verification of dynamic structural properties of models at run-time or in a model checker.

*This work was developed in the course of the Special Research Initiative 614 - Self-optimizing Concepts and Structures in Mechanical Engineering - University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

†Supported by the International Graduate School of Dynamic Intelligent Systems.

Contents

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 [32]. 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 UML 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 [11], the situation is even graver. As reported in [13], 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.

The Approach. In this paper, we demonstrate that visual languages can be used for specifying structural as well as dynamic properties. First, we show how *Story Decision Diagrams* (SDD) [18] can be used to capture structural requirements. SDD are an extension of Story Patterns [27], 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) [19], 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 par-

tial 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.

Outline. The technical report is organized as follows: After reviewing and discussing the current state of the art in Subsection 1.1, we introduce our application example from the mechatronic domain in Subsection 1.2. In Section 2, we present the existing notations for graph-based modeling, their formalization, and the existing tool support. In the following two main sections, we introduce our extended notations, list the complete syntax, define the formal semantics, and provide a mapping from the property specification to operational detector behavior for each of them. Section 3 discusses the concepts for modeling structural properties. Section 4 embeds the concepts for structural modeling into our approach for modeling temporal properties. In Section 5, we focus on the application of the proposed specification techniques and discuss how the introduced concepts can be used to derive a formal model from textual requirements. Finally, the paper provides a conclusion and an outlook on planned future work.

1.1 Related Work

Visual Structural Properties. Constraint diagrams [24] 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 [25] 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 [6] 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. [27]), 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. [30]). 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) [21] or Triggered Message Sequence Charts (TMSCs) [39], 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. [33, 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.

Specification patterns for temporal properties represent an attempt to alleviate the problem that temporal logics are difficult to apply. As outlined in [13], many useful temporal properties can be constructed using a small set of elementary building blocks. This idea has been extended and applied to real-time systems in [28]. However, while applying the patterns may be intuitive, the resulting formulas themselves are no more transparent or readable than before.

In the UML 1.x, real-time properties could only be expressed using the *UML Profile for Schedulability, Performance, and Time* [31]. It allows attaching specific schedulability or quality of service characteristics to classes, but only provides rudimentary support for the detailed specification of real-time behavior. UML 2.0 introduces only marginal improvements w.r.t. Sequence Diagrams.

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.

Combined Structual and Temporal Properties. 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., [8]. Temporal extensions of the OCL that consider real-time issues have been proposed for events in OCL/RT [10] and for states in RT-OCL [15]. As temporal logic alone already causes an even more demanding mapping problem (cf. [13]), integrating the OCL and some temporal logic concepts at the textual level does not yield a sufficiently comprehensible solution.

In [17], an embedding of graph pattern into LTL formulas is proposed in order to be able to capture structural properties. This approach tackles the theoretical aspects of the proposed integration rather than the design of a practical specification

language, which would suffer from the intricate nature of the underlying LTL.

Story Diagrams [14] 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.

The only notation that takes an approach similar to ours is a recent proposal [37] for writing temporal graph queries. The approach extends Story Diagrams by annotating unary forward or past operators from LTL with additional explicitly encoded time constraints. It requires the explicit specification of an accepting automaton rather than employing the idea of scenarios. In cases where only partial orders of events or time constraints between partially ordered situations have to be specified, the encoding of the time constraints in the automaton will therefore become rather complex.

1.2 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 [9] 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 [20], modular system coordination using social structures, and the verification of safety properties that are inductive invariants of the system [4]. Here, we focus on specifying the associated structural and behavioral system requirements in a manner that is expressive, ac-

cessible 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 responsible controllers. Shuttles can perform 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 1.1 provides an overview of these elements.

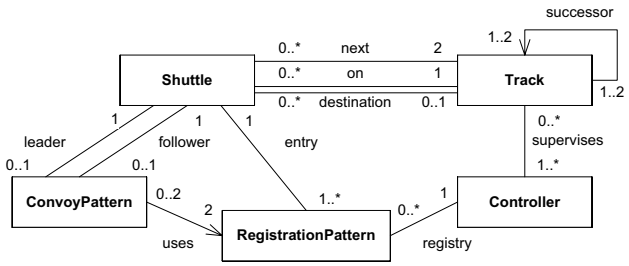


Figure 1.1: The elements of the shuttle system

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.

2 Foundations: A Graph-based Approach

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 labeled graph. The theory of graph transformation systems (cf. [36]) 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.

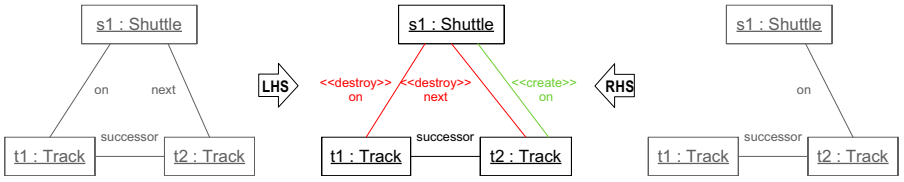
2.1 UML Diagrams and Story Patterns

Class Diagrams. We use UML Class Diagrams for structural modeling. The diagram not only defines the elements of the system and their relationships, but characterizes the set of all possible system states. Figure 1.1 above presents a basic example. Additionally, it is also possible to attach attributes and methods to classes and to define subtyping relationships.

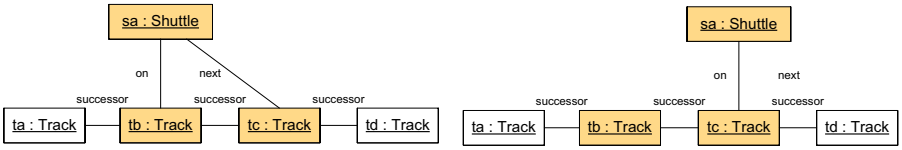
Story Patterns. Object Diagrams can be used to depict specific configurations of objects which are valid instances of a given Class Diagram. *Story Patterns* are an extended type of UML Object Diagram (cf. [27]) that allow expressing properties and transformations, especially structural changes. A Story Pattern consists of two Object Diagrams representing a pre- and a postcondition, the left hand side (LHS) and the right hand side (RHS). At runtime, the LHS is matched against the instance graph, and the free elements of the pattern are bound to specific nodes and edges. If a match is found, it is transformed in order to match the RHS by adding, modifying and deleting the appropriate nodes and edges using the Single Push Out strategy (SPO).

When specifying Story Patterns, the RHS and the LHS are integrated into a single diagram in order to obtain a more compact representation. This is achieved by using the stereotypes ≪create≫ for marking exclusive elements of the RHS that need to be created and ≪destroy≫ for denoting elements of the LHS which should be deleted as a side-effect of the rule. Figure 2.1 shows the definition (Figure 2.1a) of a Story Pattern for moving a shuttle from its current Track to the next track and an instance graph representing a small fragment of the system before (Figure 2.1b) and after (Figure 2.1c) the pattern is applied.

Negation. Furthermore, it is possible to indicate forbidden elements in a Story Pat-



a. Pattern specification (center) combining LHS (left) and RHS (right) in a single graph



b. Matching the precondition (LHS) c. Applying the postcondition (RHS)

Figure 2.1: A shuttle moving to its next track

tern by crossing them out. They can be employed to specify patterns that are only applied when no match for any *one* of their forbidden elements is found, enabling more differentiated rules. E.g., Figure 2.2 encodes that the track that a shuttle is moving to needs to be empty. However, it is not possible to express that a combination of elements should be absent, as the forbidden elements are interpreted as alternatives, i.e. the pattern application fails as soon as the first forbidden element is found.

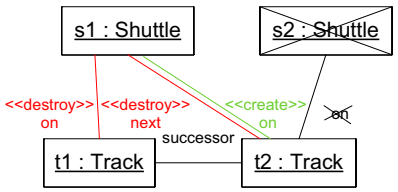
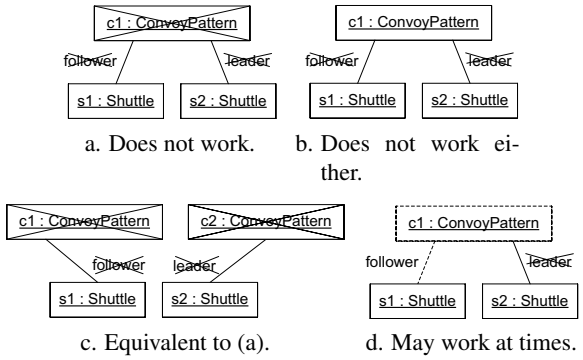


Figure 2.2: Forbidden element - movement is only allowed into vacant tracks

For the same reason, it is not possible to specify forbidden elements that are characterized by multiple associations. This poses a serious practical problem, as such a construct is needed to encode many comparatively simple properties, e.g., 'no pattern exists *between* shuttles s1 and s2'. Figure 2.3 presents several failed attempts to specify this property. The pattern in Figure 2.3a will fail as soon as any of the shuttles has any pattern (false negative). The pattern in Figure 2.3b is completely wrong, as it will not only not match if there is a pattern belonging to either of the shuttles (false negative), but also if there is no pattern in the system at all (false negative), and additionally also match if there is a pattern that is unrelated to either shuttle, even though the shuttles share a pattern (false positive). Figure 2.3c is equivalent to Figure 2.3a. Finally, Figure 2.3d is a makeshift solution using optional elements (read as 's1 may or may not have a pattern, but if so, then not with s2'). While this works as long as s1 has at most one pattern, the solution

is not robust and does not convey the intended semantics. If s1 has two patterns, the property might hold or not depending on which pattern is bound to c1. If the element in question is characterized by more than two associations, this problem intensifies.

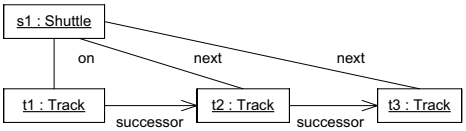


a. Does not work. b. Does not work either.

c. Equivalent to (a). d. May work at times.

Figure 2.3: Attempts to encode that s1 and s2 do not already share a pattern.

Invariant Story Patterns. When a Story Pattern contains no stereotypes, the LHS and the RHS are identical and the pattern has no side effects. Such Story Patterns describe and allow testing for system properties. E.g., the Story Pattern in Figure 2.4 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.




```
context s1: Shuttle inv
  s1.next→exists(t2 |
    s1.on.successor→exists(ta | ta = t2))
      and
        s1.next→exists(t2 |
          s1.next→exists(t3 |
            t2.successor→exists(tb | tb = t3)))
```

Figure 2.4: Story pattern: a simple positive invariant

In [4], 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 unintuitive multiple negations, i.e. a required

element of a positive invariant is translated into a forbidden element of a forbidden pattern. Combined with the described limitations concerning negation, this significantly complicated modeling.

2.2 Formal Semantics

The presented (extended) UML Diagrams provide a visual modeling language for the specificiation and presentation of systems and their associated constraints which – apart from the mentioned limitations – is expressive and accessible to human users. However, in order to provide them with the formal semantics that UML-based notations are typically lacking, we internally map our notations to a formal graph-based model which, though less suitable for presentation, can subsequently serve as the basis for theoretical analysis, formal verification, and code generation. We first introduce all required concepts and afterwards map the elements of the visual modeling language to them.

2.2.1 Graphs

As graphs are the foundation of our modeling language, we start by providing a formal definition and a set of related properties and operators.

Basic definitions. Our formalization is based on sets and functions over these sets. In our definitions, we use the following notations:

For a function $f : A \to B$, we denote by $f|_C$ the function f' with domain $A \cap C$ for which for all $x \in A \cap C$ holds $f'(x) = f(x)$.

We compose two functions $f : A \to B$ and $g : C \to A$ using the operator $f \circ g$, resulting in a function $f' : C \to B$ for which for all $x \in C$ holds $f'(x) = f(g(x))$.

Two functions $f : A \to B$ and $g : C \to D$ can be composed using the operator $f \oplus g$ if for all $x \in A \cap B$ holds $f(x) = g(x)$, resulting in a function $h : A \cup C \to B \cup D$ for which for all $x \in A$ holds $h(x) = f(x)$ and for all $x \in C$ holds $h(x) = g(x)$.

Labeled graphs. Following the conventions used in [36], we define a graph G as a directed graph that can accommodate multiple edges between two nodes.

Definition 1 *A* graph *is a tuple $G = (N_G, E_G, src_G, tgt_G)$, where N_G is a finite set of nodes, E_G is finite set of edges, $src : E_G \to N_G$ is the source function, which*

assigns a source node to each edge, andtgt : $E_G \rightarrow N_G$ *is the target function, which assigns a target node to each edge.*

We can then extend this definition to the definition of a *labeled graph* by adding labeling functions:

Definition 2 *A* labeled graph *is a pair* (G, L_G) *of a graph* $G = (N_G, E_G, src_G, tgt_G)$ *and an appropriate labeling* $L_G = (\Omega_G^N, \Omega_G^E, l_G^N, l_G^E)$ *where* Ω_G^N *is a set of node labels,* Ω_G^E *is a set of edge labels,* $l_G^N : N \rightarrow \Omega_G^N$ *is a node labeling function that assigns a label to each node, and* $l_G^E : E \rightarrow \Omega_G^E$ *is an edge labeling function that assigns a label to each edge.*

Two graphs G_1 and G_2 are *label compatible* iff the labelings of both graphs are compatible, i.e., identical for the shared elements of both graphs: $l_{G_1}^N|_{(N_{G_1} \cap N_{G_2})} = l_{G_2}^N|_{(N_{G_1} \cap N_{G_2})}$ and $l_{G_1}^E|_{(E_{G_1} \cap E_{G_2})} = l_{G_2}^E|_{(E_{G_1} \cap E_{G_2})}$.

They are *edge compatible* iff the source and target functions are identical for shared edges that are contained in both graphs: $src_{G_1}|_{(E_{G_1} \cap E_{G_2})} = src_{G_2}|_{(E_{G_1} \cap E_{G_2})}$ and $tgt_{G_1}|_{(E_{G_1} \cap E_{G_2})} = tgt_{G_2}|_{(E_{G_1} \cap E_{G_2})}$).

Two graphs that are both label and edge compatible are called *compatible*.

We use $G_\emptyset$ to denote the empty graph with $N_{G_\emptyset} = E_{G_\emptyset} = \emptyset$.

Graph Operators. For compatible graphs, we define the union, intersection and substraction of the graphs.

Given two compatible graphs G_1 and G_2, their *union* is built by combining their node and edge sets and combining the labeling, source and target functions: $G' = G_1 \cup G_2$ with $G' := (N', E', src', tgt', \Omega^{N'}, \Omega^{E'}, l^{N'}, l^{E'})$, where $N' := N_{G_1} \cup N_{G_2}$, $E' := E_{G_1} \cup E_{G_2}$, $src' := src_{G_1} \oplus src_{G_2}$, $tgt := tgt_{G_1} \oplus tgt_{G_2}$, $\Omega^{N'} := \Omega_{G_1}^N \cup \Omega_{G_2}^N$, $\Omega^{E'} := \Omega_{G_1}^E \cup \Omega_{G_2}^E$, $l^{N'} := l_{G_1}^N \oplus l_{G_2}^N$ and $l'_E := l_{G_1}^E \oplus l_{G_2}^E$. The union is commutative, $G_1 \cup G_2 = G_2 \cup G_1$ holds.

Their *intersection* of G_1 and G_2 is built by intersecting their node and edge sets and restricting the labeling, source and target functions: $G' = G_1 \cap G_2$ with $G' := (N', E', src', tgt', \Omega^{N'}, \Omega^{E'}, l^{N'}, l^{E'})$, where $N' := N_{G_1} \cap N_{G_2}$, $E' := E_{G_1} \cap E_{G_2}$, $src' := src_{G_1}|_{(E_{G_1} \cap E_{G_2})}$, $tgt := tgt_{G_1}|_{(E_{G_1} \cap E_{G_2})}$, $\Omega^{N'} := \Omega_{G_1}^N \cap \Omega_{G_2}^N$, $\Omega^{E'} := \Omega_{G_1}^E \cap \Omega_{G_2}^E$, $l^{N'} := l_{G_1}^N|_{(N_{G_1} \cap N_{G_2})}$ and $l'_E := l_{G_1}^E|_{(E_{G_1} \cap E_{G_2})}$. The intersection is commutative, $G_1 \cap G_2 = G_2 \cap G_1$ holds.

The *subtraction* of the two graphs G_1 and G_2 is similar to intersection. The node and edge sets of a graph are subtracted from the sets of the other graph,

and the functions are restricted accordingly: $G' = G_1 \setminus G_2$ with $G' := (N', E', src', tgt', \Omega^{N'}, \Omega^{E'}, l^{N'}, l^{E'})$, where $N' := N_{G_1} \setminus N_{G_2}$, $E' := \{e \in E_{G_1} \setminus E_{G_2} | src_{G_1}(e) \in N' \wedge tgt_{G_1}(e) \in N'\}$, $src' := src_{G_1}|_{E'}$, $tgt := tgt_{G_1}|_{E'}$, $\Omega^{N'} := \Omega^N_{G_1}$, $\Omega^{E'} := \Omega^E_{G_1}$, $l^{N'} := l^N_{G_1}|_{N'}$ and $l'_E := l^E_{G_1}|_{E'}$. For non-empty graphs, subtraction is not commutative, $G_1 \setminus G_2 \neq G_2 \setminus G_1$ holds. The definition of E' results in the implicit deletion of dangling edges, i.e. edges whose source or target node is undefined. Otherwise, the resulting tuple might not represent a graph, as the functions $src_{E'}$ and $tgt_{E'}$ would not necessarily be restricted to N'.

Typed graphs. We now add the notion of types to our definition of a graph. In a *type graph* $G_T = (N_T, E_T, src_T, tgt_T, \Omega^N_T, \Omega^E_T, l^N_T, l^E_T)$, nodes represent node types, edges represent edge types, and labels are used to assign type names.

A *typed graph* G is then a labeled graph whose node and edge labels are the nodes and edges of some *type graph* G_T, i.e. $\Omega^N_G = N_T$ and $\Omega^E_G = E_T$.[1] We call G *type conformant* for G_T if the labeling of G is compatible with G_T, which means that if there is an edge labeled with $e_1 \in E_T$ between nodes labeled with $n_1 \in N_T$ and $n_2 \in N_T$ in G, e_1 must be an edge connecting nodes n_1 and n_2 in G_T:

Definition 3 *The labeling of a graph* $G = (N_G, E_G, src_G, tgt_G, \Omega^N_G, \Omega^E_G, l^N_G, l^E_G)$ *is* type conformant *for the type graph* $G_T = (N_T, E_T, src_T, tgt_T, \Omega^N_T, \Omega^E_T, l^N_T, l^E_T)$ *iff* $\Omega^N_G \subseteq N_T$, $\Omega^E_G \subseteq E_T$ *and* $\forall e \in E_G : (\exists e_T \in E_T : l^E_G(e) = e_T \wedge l^N_G(src_G(e)) = src_T(e_T) \wedge l^N_G(tgt_G(e)) = tgt_T(e_T))$.

We denote the set of all type conformant labeled graphs for a type graph G_T by $\mathcal{G}[G_T]$.

In order to accomodate subtyping, we need to extend our notion of a type graph and of type conformity. An *inheritance type graph* G_T is a type graph whose edge label alphabet Ω^E_T contains a special element isa. If there is an edge labeled with isa from node n_{sub} to node n_{super}, we say that n_{sub} is a *subtype* of n_{super}. We define $subtype(n_{sub}, n_{super}) := \exists e \in E_T : l^N_T(e) = isa \wedge n_{sub} = src_T(e) \wedge n_{super} = tgt_T(e)$. The transitive closure of $subtype$ then yields the set $super(n) := \{n' | (n, n') \in subtype^+\}$, while the reflexive-transitive closure yields $types(n) := super(n) \cup n$.

We can now extend our previous definition of type conformity to include subtying:

Definition 4 *The labeling of a graph* $G = (N_G, E_G, src_G, tgt_G, \Omega^N_G, \Omega^E_G, l^N_G, l^E_G)$ *is* type conformant *for the inheritance type graph* $G_T =$

[1] Note that we do not assign type names (strings) to objects, which we then would have to (string) compare with the assigned type name of the corresponding type graph node, but directly use the nodes of the type graph themselves to label the nodes of the instance graph, which simplifies checking type conformity. The labeling function does not care whether its alphabet is letters or nodes.

$(N_T, E_T, src_T, tgt_T, \Omega_T^N, \Omega_T^E, l_T^N, l_T^E)$ *iff* $\Omega_G^N \subseteq N_T$, $\Omega_G^E \subseteq E_T \setminus isa$ *and* $\forall e \in E_G : (\exists e_T \in E_T : l_G^E(e) = e_T \wedge src_T(e_T) \in types(l_G^N(src_G(e))) \wedge tgt_T(e_T) \in types(l_G^N(tgt_G(e))))$.

As for simple type graphs not containing isa we simply have $types(n) = n$, this definition includes the previous definition.

Attributed graphs. Finally, we introduce attributed graphs. Following [22], we only allow node attributes, but no edge attributes. Attributes are represented by nodes that are the target of special edges whose source is the attributed node. To abstract from the data types of the attributes, we describe them in terms of an algebra A over a many sorted signature $\Sigma = \langle S_\Sigma, OP_\Sigma \rangle$ consisting of sets of sort symbols S_Σ and of operation symbols OP_Σ.

Definition 5 *An* attributed graph *is a pair* (G, A) *of a graph* G *and an algebra* A *over* Σ*, where for* $|A| := \biguplus_{s \in S_\Sigma} A_s$*, the disjoint union of the carrier sets of* A*, we have* $|A| \subseteq N_G$ *and* $\forall e \in E_G : src_G(e) \notin |A|$.

For an attributed graph G, we define attribute value nodes $N_G^A := |A|$ and instance nodes $N_G^I := N_G \setminus N_G^A$. We further differentiate between attributes $E_G^A := \{e \in E_G : tgt(e) \in |A|\}$ and links $E_G^I := E_G \setminus E_G^I$.

The notion of type conformance is not affected by this extension. The only additional convention is that when labeling a type graph G_T (which does not have to be an attributed graph itself), we label nodes that represent attribute types (i.e. are later used to label nodes from N_G^A) with the appropriate sort symbol $s \in S_\Sigma$.

2.2.2 Graph Patterns

In order to formalize the notion of matching and applying a pattern, we now formalize these notions based on the above definitions.

Containment. We formalize the notion of containment of a labeled graph in another labeled graph by comparing their defining functions: For two graphs SG and G we say that SG is a *subgraph* of G (written as $SG \leq G$) iff $N_{SG} \subseteq N_G$, $E_{SG} \subseteq E_G$, $src_{SG} = src_G|_{E_{SG}}$, $tgt_{SG} = tgt_G|_{E_{SG}}$, $\Omega_{SG}^N \subseteq \Omega_G^N$, $\Omega_{SG}^E \subseteq \Omega_G^E$, $l_{SG}^N = l_G^N|_{N_{SG}}$, and $l_{SG}^E = l_G^E|_{E_{SG}}$. Two graphs are equal iff $SG \leq G$ and $G \leq SG$.

Pattern Matching. As a pattern is supposed to be a generalized way of encoding a recurrent structure. When matching patterns against instance graphs, we only want

to compare the graphs w.r.t. their structure, i.e. without considering the identity of the nodes and edges. Instead of the simple subgraph relationship, we therefore need to use the more general concept of graph morphisms (cf. [36]).

Definition 6 *A* graph morphism $m : G_1 \to G_2$ *is a pair of functions* $m := \langle m^N : N_{G_1} \to N_{G_2}, m^E : E_{G_1} \to E_{G_2} \rangle$ *mapping the nodes and edges of* G_1 *to the elements of* G_2 *while preserving sources, targets and labels.* m *thus satisfies the properties* $m^N \circ tgt_{G_1} = tgt_{G_2} \circ m^E$, $m^N \circ src_{G_1} = src_{G_2} \circ m^E$, $l^N_{G_1} = l^N_{G_2} \circ m^N$ *and* $l^E_{G_1} = l^E_{G_2} \circ m^E$. *A* graph isomorphism m *is a graph morphism whose functions* m^N *and* m^E *are both bijective.*

This definition can be extended to cover attributed graphs:

Definition 7 *An* attributed graph morphism $m : (G_1, A_1) \to (G_2, A_2)$ *is a pair of a graph morphism* m_G *and a* Σ*-morphism* $m_A : A_1 \to A_2$ *mapping the elements of the carrier sets of* A_1 *to* A_2 *so that* $m_A \subseteq m^N_G$.

If there is a graph isomorphism $m : G_1 \to G_2$, we write $G_1 =_m G_2$ or $G_1 \approx G_2$ to abstract from the specific morphism m. However, as a pattern will typically be smaller than the graph against which we are matching it, the more relevant question is usually whether there is a graph isomorphism from the pattern G_1 to a subgraph SG_2 of G_2, i.e. $m : G_1 \to SG_2$ with $SG_2 \leq G_2$. If such an isomorphism exists, we write $G_1 \leq_m G_2$, respectively $G_1 \precsim G_2$ to abstract from the morphism.

In the literature on graph theory, graph homomorphisms, i.e., morphisms that are not necessarily bijective, are commonly used instead of isomorphisms. As our definition of $\precsim$ basically eliminates the surjectivity requirement from the matching process, the decisive difference is that pattern matching using isomorphisms requires injectivity while matching using homomorphisms does not. We have found that, in most cases, the principle that different pattern elements map to different instances is closer to the intuitive interpretation of a pattern. Consider a pattern encoding that two shuttles $s1$ and $s2$ are on the same track $t1$ (see Figure 3.2 in Section 3.1). For every single shuttle on a track in the system, there is a homomorphism for matching that pattern by simply mapping both shuttles from the pattern to the same shuttle in the system. The pattern then is basically flagging each shuttle as a collision with itself, which hardly reflects the intended meaning. Though this can be prevented by adding an additional constraint $s1 \neq s2$ requiring the two shuttles to be different, this is cumbersome. We therefore prefer using isomorphisms as the default matching strategy and only employ homomorphisms where explicitly indicated.

Based on subgraph isomorphisms, we define simple graph patterns as follows:

Definition 8 *A* simple graph pattern $[G]$ *consists of a graph* G*. If there is a graph* AG *and an isomorphism* m *with* $G \leq_m AG$*, we write* $AG, m \vdash [G]$ *and say that the graph* AG *fulfills the pattern.*

Negative Application Conditions (NAC) formalize the concept of forbidden elements. The basic idea is that a pattern will only match if a forbidden second pattern does not match as well. The semantics of forbidden elements are thus defined as follows:

Definition 9 *A* negative application condition *(NAC) over a graph* G *is a finite set* $\hat{\mathcal{G}}$ *of connected graphs with* $\forall \hat{G}_i \in \hat{\mathcal{G}} : G \leq \hat{G}_i$*, called constraints. A constraint* $\hat{G}_i$ *is fulfilled by a graph* AG *if* $\exists m : G \leq_m AG$ *but* $\nexists m'$ *with* $m'|_G = m$ *and* $\hat{G}_i \leq_{m'} AG$*, written* $AG, G, m' \vdash \hat{G}_i$*. A graph* AG *and the isomorphism* m *satisfy a NAC* $\hat{\mathcal{G}}$*, written* $AG, G, m \vdash \hat{\mathcal{G}}$*, if it satisfies all constraints* $\hat{G}_i \in \hat{\mathcal{G}}$*, i.e* $\forall \hat{G}_i \in \hat{\mathcal{G}} : AG, G, m \vdash \hat{G}_i$*.*

This leads to the general definition of a graph pattern and a match of such a pattern:

Definition 10 *A* graph pattern $[G, \hat{\mathcal{G}}]$ *consists of a graph* G *and a set of NACs* $\hat{\mathcal{G}}$ *of* G*.*[2] *It characterizes the set of graphs that contain the graph* G *but do not contain any extension* $\hat{G}_i$ *of* G*.*

Definition 11 *A* match m *for a graph pattern* $[L, \hat{\mathcal{L}}]$ *in some graph* G *with a subgraph* $SG \leq G$ *is a graph isomorphism* $m : L \rightarrow SG$ *with* $G, L, m \vdash \hat{\mathcal{L}}$*. We write* $G, m \vdash [L, \hat{\mathcal{L}}]$ *or* $G \vdash [L, \hat{\mathcal{L}}]$*.*

2.2.3 Graph Transformation Rules

Graph transformation rules describe modifications of a graph by means of two graph patterns, a precondition and a postcondition. We define:

Definition 12 *A* graph transformation rule $[L, \hat{\mathcal{L}}] \rightarrow_r [R]$ *consists of* r *the rule name,* $[L, \hat{\mathcal{L}}]$ *the left hand side (LHS), a graph pattern encoding the precondition, and* $[R]$ *the right hand side (RHS), a simple graph pattern encoding the postcondition, with* L*, all elements of* $\mathcal{L}$*, and* R *compatible and* $L \cap R \neq G_\emptyset$*.*

A rule is type conformant to a type graph G_T if all graphs in the rule are type conformant to G_T.

[2] A simple graph pattern can be interpreted as a graph pattern with an empty set of NACs.

When a rule r is applied to a graph G, G is called the *application graph* or *source graph*. The resulting graph G' is called the *target graph*.

In order to effect the actual graph transformation, we use the *Single Pushout Approach* (cf. [36]):

Definition 13 *The* Single Pushout Approach *defines the application of a graph transformation rule r to an application graph G as a* direct transformation *of the source graph G into a compatible target graph G'. Given the rule $[L, \hat{\mathcal{L}}] \rightarrow_r [R]$ and a match m for $[L, \hat{\mathcal{L}}]$, such a* direct transformation *is characterized by the* occurrence *o, which is a graph isomorphism $o : L \cup R \rightarrow G \cup G'$ with the following properties: $o|_L = m$, i.e. o matches the left hand side in accordance with m, $L \leq_o G \wedge R \leq_o G'$ i.e. the left hand side of r is contained in G and the right hand side of r is contained in G', and $o(L \setminus R) = G \setminus G' \wedge o(R \setminus L) = G' \setminus G$, i.e. those elements belonging to L but not to R are deleted, while those elements belonging to R but not to L are created. We write $G \models\!\Rightarrow_{r,o} G'$ to denote such a transformation or $G \models\!\Rightarrow_r G'$ to abstract from o.*

Informally, when r is applied to G, all elements (nodes and edges) that are contained in both the left and right hand side are preserved, elements that are only contained in the left hand side are deleted, and elements that are only contained in the right hand side are added, using appropriate morphisms.

If a sequence of direct graph transformations of the form $G_0 \models\!\Rightarrow_{r_0,o_0} G_1 \models\!\Rightarrow_{r_1,o_1} \ldots \models\!\Rightarrow_{r_{n-1},o_{n-1}} G_n$ exists, where $r_0, \ldots, r_{n-1}$ are rules and $o_0, \ldots, o_{n-1}$ their occurrences, so that for $0 \leq i < n$ holds $G_i \models\!\Rightarrow_{r_i} G_{i+1}$, we write $G_0 \models\!\Rightarrow^*_{(r_0,o_0);\ldots;(r_{n-1},o_{n-1})} G_n$, or shorter $G_0 \models\!\Rightarrow^*_{r_0;\ldots;r_{n-1}} G_n$ if the occurences are unambiguous or irrelevant in the given context. Even more compactly, $G_0 \models\!\Rightarrow^* G_n$ denotes that *some* transformation sequence from G_0 to G_n exists.

2.2.4 Graph Transformation Systems

GTS. Using the concepts we have introduced above, we can now define *graph transformation systems* (GTS), a type of state transition system where every state is represented by a graph and every transition is described as a graph rewrite rule:

Definition 14 *A* typed graph transformation system *(GTS) S is a tuple $(\mathcal{T}_S, \mathcal{G}^i_S, \mathcal{R}_S)$ with $\mathcal{T}_S$ a type graph, $\mathcal{G}^i_S$ the set of all type conformant initial graphs of the system, and $\mathcal{R}_S$ a finite set of type conformant graph transformation rules.*

For each system $S = (\mathcal{T}_S, \mathcal{G}^i_S, \mathcal{R}_S)$ and a graph G, the valid applications are denoted by $\models S \Rightarrow_{r,o}$, $\models S \Rightarrow_r$, or $\models S \Rightarrow^*_w$ respectively. We define the – potentially

infinite – set of all reachable states as $\mathsf{REACH}(S) := \{G \mid G \in \mathcal{G}[\mathcal{T}_S] \wedge \exists G_0 \in \mathcal{G}^i_S, w \in \mathcal{R}^*_S : G_0 \models S \Rightarrow^*_w G\}$.

Extended GTS. We can extend this definition in various ways:

Definition 15 *A GTS S can be extended into a* prioritized graph transformation system *by adding a priority function $prio_S : \mathcal{R}_S \to \mathbb{N}$ assigning a priority $prio_S(r)$ to every $r \in \mathcal{R}_S$, where priority 0 has the highest precedence.*

For a prioritized system $S = (\mathcal{T}_S, \mathcal{G}^i_S, \mathcal{R}_S, prio_S)$ and a graph G, we further restrict valid rule applications $G \models\!\Rightarrow_{r,o} G'$ to those cases where no other application $G \models\!\Rightarrow_{r',o'} G''$ with $prio_S(r') < prio_S(r)$ exists.

Definition 16 *A GTS S can be extended into a* constrained graph transformation system $(\mathcal{T}_S, \mathcal{G}^i_S, \mathcal{R}_S, \Phi_S)$ *by specifying a set of forbidden graph patterns Φ_S which must never match the system state at any time.*

For a constrained system $S = (\mathcal{T}_S, \mathcal{G}^i_S, \mathcal{R}_S, \Phi_S)$ and a graph G, we define a *violation* as a rule application $G \models\!\Rightarrow_{r,o} G'$ where $\exists \phi \in \Phi_S : G' \vdash \phi \wedge \neg G \vdash \phi$.

Definition 17 *A GTS S can be extended into a* labeled graph transformation system *with (multiple) rule labels from the label set $\mathcal{B}$ by providing a mapping $l_S : \mathcal{R}_S \to \wp(\mathcal{B})$*

Parallel composition. When two graph transformation systems are executed concurrently, the resulting system corresponds to their parallel composition. We define the *parallel composition* $S \| T$ of two graph transformation systems $S = (\mathcal{T}_S, \mathcal{G}^i_S, \mathcal{R}_S)$ and $T = (\mathcal{T}_T, \mathcal{G}^i_T, \mathcal{R}_T)$ as a GTS $U := (\mathcal{T}_U, \mathcal{G}^i_U, \mathcal{R}_U)$ with $\mathcal{T}_U := \mathcal{T}_S \cup \mathcal{T}_T$, $\mathcal{G}^i_U := \mathcal{G}^i_S \cup \mathcal{G}^i_T \cup \{G \cup G' | G \in \mathcal{G}^i_S \wedge G' \in \mathcal{G}^i_T\}$, and $\mathcal{R}_U := \mathcal{R}_S \cup \mathcal{R}_T$.

For the parallel composition $U := (\mathcal{T}_U, \mathcal{G}^i_U, \mathcal{R}_U, \Phi_U)$ of two constrained graph transformation systems $S = (\mathcal{T}_S, \mathcal{G}^i_S, \mathcal{R}_S, \Phi_S)$ and $T = (\mathcal{T}_T, \mathcal{G}^i_T, \mathcal{R}_T, \Phi_T)$, we additionally define $\Phi_U := \Phi_S \cup \Phi_T$. For prioritized graph transformation systems, we define $prio_U := prio_S \oplus prio_T$. For labeled graph transformation systems, we likewise define $l_U := l_S \oplus l_T$.

Paths. A path $\pi := G_0 \models S \Rightarrow_{r_1,o_1} G_1 \models S \Rightarrow_{r_2,o_2} G_2 \ldots$ is an alternating sequence of states and valid rule applications connecting these states. We denote the – potentially infinite – length of a path by $l(\pi)$. For $i \in [0, l(\pi))$, we refer to the state graph generated by the i-th rule appliction (i.e., G_i) as $\pi[i]$. We use π^i to denote the suffix of π starting with $\pi[i]$.

The set of all finite or infinite possible paths π starting from G is defined as $\mathsf{PATH}(S,G) := \{G_0 \models S \Rightarrow_{r_1,o_1} G_1 \models S \Rightarrow_{r_2,o_2} G_2 \ldots \mid G_0 = G\}$. $\mathsf{PATH}(S)$ denotes all paths that can be generated by S and is defined as the union of all sets $\mathsf{PATH}(S,G)$ with $G \in \mathcal{G}_S^i$. We also write $[\![S]\!]$ for $\mathsf{PATH}(S)$.

When we are considering time, we additionally use a function $T(\pi,i) : [\![S]\!] \times [0,l(\pi)] \rightarrow I\!R$ to determine the time when each particular state of a path has been reached. Depending on the notion of time that is available in the context where these concepts are applied, we may need to substitute a discrete notion for the continous notion of time.

2.2.5 Properties of Graph Transformation Systems

Using graph patterns as basic propositions, we can derive more complex graph properties. The Computational Tree Logic CTL* (cf. [11]) with its path quantifiers A (for all paths) and E (for some path) and temporal operators X (next), F (eventually), G (always), U (until), and R (release) can be used to embed these basic propositions to form an expressive notation for temporal conditions. In [16], it is shown that a sound and complete general propositional temporal calculus remains sound and complete when interpreted on graph transformation systems.

We then have the following syntax for state and path formulas:

- If ϕ is a graph pattern or the constant $true$ or $false$, then ϕ is a state formula.
- If ϕ and ψ are state formulas, then $\neg\phi$, $\phi \vee \psi$, and $\phi \wedge \psi$ are state formulas.
- If ϕ is a state formula, then ϕ is also a path formula.
- If p is a path formula, then Ep and Ap are state formulas.
- If p and p' are path formulas, then $\neg p$, $p \vee p'$, $p \wedge p'$, Xp, Fp, Gp, pUp', and pRp' are path formulas.

We write $S,G \models \phi$ iff the CTL* formula ϕ holds for the state G and $S,\pi \models \phi$ iff the CTL* formula ϕ holds for the path π. We further write $S \models \phi$ to denote that $\forall G \in \mathcal{G}_S^i$ holds $S,G \models \phi$.

The semantics of state and path formulas is then defined as follows for a GTS S,a graph G, and a trace π:

- $S,G \models \phi$ iff ϕ is a graph pattern and $G \vdash \phi$.
- $S,G \models \neg\phi$ iff $S,G \not\models \phi$.
- $S,G \models \phi \vee \psi$ iff $S,G \models \phi \vee S,G \models \phi$.
- $S,G \models \phi \wedge \psi$ iff $S,G \models \phi \wedge S,G \models \phi$.
- $S,G \models E\phi$ iff $\exists \pi \in \mathsf{PATH}(S,G) : S,\pi \models \phi$.
- $S,G \models A\phi$ iff $\forall \pi \in \mathsf{PATH}(S,G) : S,\pi \models \phi$.

- $S, \pi \models \phi$ iff $G = \pi[0] \wedge S, G \models \phi$.
- $S, \pi \models \neg\phi$ iff $S, G \not\models \phi$.
- $S, \pi \models \phi \vee \psi$ iff $S, \pi \models \phi \vee S, \pi \models \phi$.
- $S, \pi \models \phi \wedge \psi$ iff $S, \pi \models \phi \wedge S, \pi \models \phi$.
- $S, \pi \models X\phi$ iff $S, \pi^1 \models \phi$.
- $S, \pi \models F\phi$ iff $\exists k, k \geq 0 : S, \pi^k \models \phi$.
- $S, \pi \models G\phi$ iff $\forall i, i \geq 0 : S, \pi^i \models \phi$.
- $S, \pi \models \phi U \psi$ iff $\exists k, k \geq 0 : S, \pi^k \models \psi \wedge \forall j, 0 \leq j < k : S, \pi^k \models \phi$
- $S, \pi \models \phi R \psi$ iff $\forall j, j \geq 0 : (\forall i, i < j : S, \pi^i \not\models \phi) \Rightarrow S, \pi^j \models \psi$

To describe, for example, that a given graph pattern $[P, \hat{\mathcal{P}}]$ should never be matched in any reachable configuration, we can then write:

$$S \models AG(\neg[P, \hat{\mathcal{P}}]).$$

2.2.6 UML Models

We have now defined all the necessary preliminaries that will allow us to formalize the employed UML notations and Story Diagrams.

UML Class and Object Diagrams. A *Class Diagram* can be represented as a *type graph* G_T, where nodes represent classes, edges represent associations, and labels define their names. An inheritance relationship in the diagram translates to an edge labeled with isa from the node representing the subclass to the node representing the superclass.

If the diagram contains attributes, we define a signature Σ whose set of sort symbols comprises the required value types, typically $S_\Sigma := \{boolean, integer, real, string, \dots\}$, and add value type nodes labeled with $s \in S_\Sigma$. Attributes are then encoded as edges from class nodes to value type nodes, labeled with the attribute name.

Cardinalities are not incorporated into the type graph itself, but need to be translated into appropriate constraints. A maximum cardinality of $*$ or n requires no constraint. A maximum cardinality of $k \in \mathbb{N}$ can be encoded as a graph pattern where $k + 1$ copies of the constrained element are present, which will match any instance situation with $i > k$ instances. A minimum cardinality of 0 requires no constraints. A minimum cardinality $k > 0$ can be encoded by adding a pattern containing i copies of the constrained element plus one additional forbidden copy of the element, which will therefore match a configuration containing i, but not $i + 1$ copies of the element, for each $0 \leq i < k$. This is not practical for larger minimum cardinalities, but poses no problems for typical values in the range $[0..2]$.

An *Object Diagram* can be represented as a *typed graph* G that is *type conformant* for the (inheritance) type graph G_T representing the corresponding Class Diagram. Nodes represent objects and edges represent links, each labeled with the respective class or association.

If the Class Diagram defines attributes, the Object Diagram needs to be an *attributed graph*. Objects are then represented by instance nodes N_G^I, links are represented by edges from E_G^I, attribute value nodes N_G^A represent literals, and the edges E_G^A represent attribute assignments.

Story Patterns. Just as the UML Object Diagrams that they extend, Story Patterns can be formally expressed using graphs. Again, objects and attribute values become nodes, while links and attribute assignments become edges.

An *Invariant Story Pattern* without forbidden elements can thus be translated to a *simple graph pattern* consisting of the corresponding graph. In the more general case including negative (forbidden) elements, a given Story Pattern can be translated to a *graph pattern* $[G, \hat{\mathcal{G}}]$ by encoding its positive objects, links and attributes as an attributed typed graph G and building the set of NACs $\hat{\mathcal{G}}$ by adding, for each negated link l, a labeled graph $\hat{G}$ consisting of G and the negated link l with its source node $src(l)$ and target node $tgt(l)$. At least one of these nodes already is in G – if l connects two positive nodes, both source and target are in G, whereas if l connects to a negative node (as in Figure 2.2), that node only is in $\hat{G}$.

Note that the problems with respect to negation that were discussed above are due to limitations of the notation and its established semantics and not inherent in the underlying formalization. It would be possible, instead, to add a NAC $\hat{G}$ (1) for every negative link *between positive objects* and (2) for every *negative object* including *all* its links, thus making Figure 2.3a a correct specification with the intended semantics. However, independently of the chosen semantics, the basic problem caused by the decision to integrate the NACs into the positive graph to allow for more compact diagrams remains, namely that the relationships between multiple negated elements (Which ones are alternatives? Which ones need to occur together?) are subtle.

The property encoded by an Invariant Story Pattern thus holds for a configuration represented by an Object Diagram iff the attributed graph AG representing the object diagram fulfills the corresponding graph pattern $[P, \hat{\mathcal{P}}]$: $AG \vdash [P, \hat{\mathcal{P}}]$.

For *Story Patterns* with side effects, we can derive a graph transformation rule $r := [L, \hat{\mathcal{L}}] \rightarrow_r [R]$. We encode the LHS of the Story Pattern, which we obtain by disregarding all elements marked with $\ll$create$\gg$ and treating those marked with $\ll$destroy$\gg$ as regular positive elements, as the graph pattern $[L, \hat{\mathcal{L}}]$. The RHS, i.e. the unmarked and newly created elements, are encoded as the simple graph

pattern $[R]$.

Applying the Story Pattern to a configuration represented by an Object Diagram AG then corresponds to the rule application $AG \models\Rightarrow_{r,o} AG'$.

System Model. By means of the above definitions, we can now derive a representation of a complete UML model as a constrained *graph transformation system* by combining the above concepts. The underlying Class Diagram becomes the type graph $\mathcal{T}_S$, the Story Patterns with side effects become the rule set $\mathcal{R}_S$, the Invariant Story Patterns make up the constraint set Φ_S, and the set of initial graphs $\mathcal{G}_S^i$ is derived from the Object Diagrams representing initial configurations. If there are forbidden patterns encoding the cardinalities of the Class Diagram, those are also added to Φ_S.

2.3 Tool support

While tool support for standard UML models is abundant, Story Patterns represent a proprietary extension that, to our knowledge, is only supported by the Fujaba Tool Suite. We also present GROOVE, a tool for evaluating graph transformation systems that we have integrated with Fujaba.

2.3.1 Fujaba

The open source UML CASE tool Fujaba [3] offers an extensible platform for visual modeling, verification and code generation. Among others, the tool offers Class Diagrams, Object Diagrams, Story Patterns and Activity Diagrams containing Story Patterns.

Thanks to the formal semantics that have been defined for the relevant UML concepts, notably Class Diagrams, Object Diagrams, and the proprietary extensions such as Story Patterns, based on the theory of GTS, it is possible to generate a completely operational prototype or production system from the visual specification (cf. [27]). Currently, the code generation of Java and C++ source code is supported for all employed diagram types.

2.3.2 GROOVE

GROOVE [34] is a GTS model checker for prioritized constrained graph transformation systems, capable of simulating the GTS and generating state spaces.

[3] http://www.fujaba.de

For a GTS specifications, GROOVE can compute all reachable states of the transformation system, optionally bounded by the occurrence of a forbidden graph. GROOVE's visualization engine allows a manual exploration of the system, indicating the enabled rules and highlighting the occurences.

We have previously developed a Fujaba plugin for exporting Story Patterns from Fujaba into GROOVE. Apart from the technical issues, the export has to accommodate the fact that GROOVE's pattern matching engine employs graph homomorphisms instead of isomorphisms by adding an inequality constraint between each pair of objects of the same type.

3 Structural Properties

We now present our extended notation for the modeling of structural properties, its syntax, semantics, and application.

3.1 Story Decision Diagrams

Story Decision Diagrams (*SDD*) are an extension of Story Patterns that allow expressing more complex properties while retaining or surpassing the intuitiveness of the original visual notation. The extensions we introduce include quantors, implication, alternatives, negation of complex properties, and a concept for modularity.

3.1.1 Basic Principles

An SDD is a directed acyclic graph (DAG). Each node contains a *Story Decision Diagram Pattern* (*SDDP*) specifying some property, which basically corresponds to a Story Pattern without side effects or forbidden elements. The SDDPs on the same path through the SDD share the same variables; i.e., once a pattern element has been bound to an instance, it remains bound in all subsequent nodes.

When evaluating the SDD, the nodes are processed starting from the root node with an empty binding, i.e., all variables unbound. Which node is evaluated next depends on whether the current node matches or not. Each node in the SDD can essentially be seen as a local if-then-else decision based on the current binding. If a match is found, we extend the binding with the corresponding object and link assigments so that successfully matched elements are propagated to subsequent elements and follow the solid then connector; if no match is found, we leave the binding unchanged and follow the dashed else connector.

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. SDDs are thus similar to decision trees. However, like reduced binary decision diagrams (RBDD), SDDs are not trees, but allow sharing isomorphic subtrees and leaf nodes to reduce diagram size. Like in decision diagrams, consecutive conditions correspond to logical conjunction, respectively implication. Both interpretations are equivalent: The intuitive interpretation of the statement if a then b else c is $(a \Rightarrow b) \wedge (\neg a \Rightarrow c)$, using two implications. Using the definition of implication, this can be reduced to the simpler statement $(a \wedge b) \vee (\neg a \wedge c)$, using two conjunctions. Unlike standard decision diagrams, SDDs support alternatives by allowing multiple then or else connectors per node. It is then sufficient for one of the available paths to reach (1) in order to evaluate the whole branch as true.

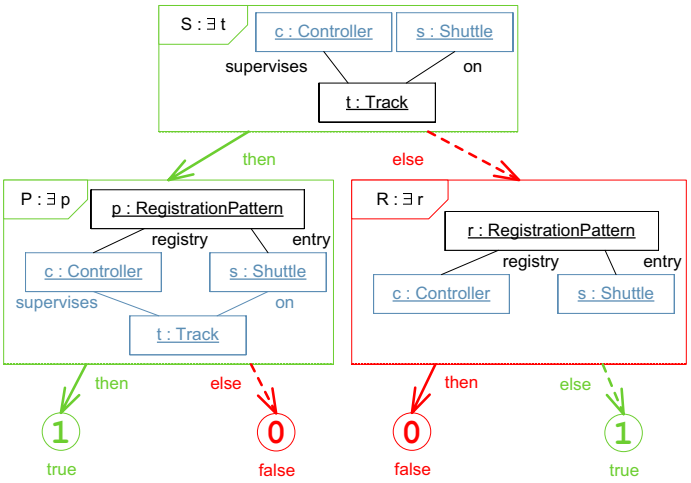


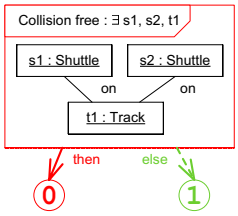
Figure 3.1: SDD Fragment illustrating basic syntax

In the SDD in Figure 3.1, the root node matches when a given controller is the supervisor of the track a given shuttle is on. Now, if the root node matches, then (left child) the controller and the shuttle have to run a registration pattern, else (right child) they must *not* be running such a pattern.

Observe that there are only positive elements in the patterns - besides their limitations, negative elements often prove problematic when interpreting Story Patterns. The negation in the right branch is expressed by modeling the forbidden situation as a positive match and switching the then and else connectors, i.e. a match leads to failure and no match leads to success. By appropriately chaining the corresponding nodes, complex negative conditions can be expressed.

In absence of negation, most leaf nodes can be omitted. If not specified otherwise, a node is interpreted as a positive requirement: matching (then) results in success, i.e. (1) not matching (else) results in failure, i.e. (0)ÀAll connector and leaf labels are optional.

Though we extensively use color to make diagrams more readable, color is *never* semantically relevant, i.e. the coloring is deduced automatically from the structure. Unbound elements in Story Patterns are black, bound elements blue (grey). (1) and connectors leading to (1) are green (light grey), (0) and connectors leading to (0) are red (dark grey). The remaining then connectors are green, the remaining else connectors are red. Node frames share the color of their then connector as a visual cue to make undesired properties stand out. Figure 3.2 illustrates this principle, marking the existence of an accident (two shuttles on the same track) as an undesired instance situation.




```
context s1 : Shuttle inv
  NOT Shuttle.allInstances()→exists(s1, s2 |
    s2.on = s1.on)
```

Figure 3.2: No two shuttles occupy the same track

3.1.2 Quantification

As a significant enhancement, we allow quantification over the free (i.e., previously unbound) variables of an SDD node. Accordingly, we differentiate between existential nodes, which require that at least one of the bindings they propagate evaluates to true, i.e., reaches a (1) leaf node, and universal nodes, which require this of every binding they propagate.

Existential nodes fall into two categories, depending on the kinds of elements contained in their SDDP:

Existentially quantified nodes contain free variables, which are bound to objects and links by the node's SDDP. When a binding reaches it, the node attempts to generate an extended binding including its free variables that is consistent with its SDDP. If such an extension or several alternative extensions of this kind exist, they are propagated down the then connector. If no such extension exists, the original binding is propagated down the else connector. If the node binds *explicitly* named variables var_i to objects or links, it is marked with $[\exists\, var_i^+]$. If the node only binds *anonymous* variables to links, it is marked with $[\exists\ \_]$.

Guard nodes do not contain free variables that could be bound and thus do not extend the bindings that reach them. They merely act as a filter that decides whether a binding should be propagated down the then or else connector, depending on whether it fulfills the node's SDDP. Guard nodes are marked with $[\bullet]$.

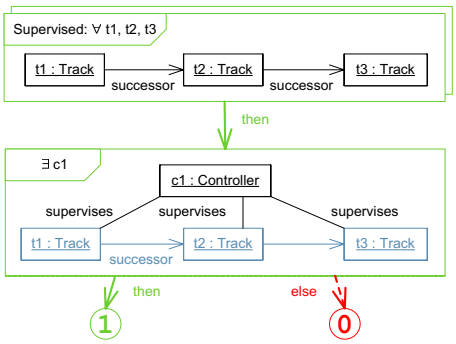
If an existential node only features a then connector, an else connector to (0) is implied. In the less common case that the node only specifies an else connector, a then connector to (1) is implied, but it is recommended to specify this explicitly to avoid confusion.

We now revisit Figure 3.1 to explain the subtleties of the way existentially quan-

tified nodes operate. The SDD in the diagram actually only is a fragment, as c and s already need to be previously bound in the root node. Using the interpretation based on implication, the diagram then corresponds to $((\exists t|S[c,s,t]) \Rightarrow \exists t'|(S[c,s,t'] \wedge \exists p|P[c,p,s,t'])) \wedge ((\nexists t|S[c,s,t]) \Rightarrow \nexists r|R[c,r,s])$, given a binding (c,s). The first subterm, where S appears twice, deserves some explanation: If there is any binding for t that completes S given (c,s), we choose the then connector. However, this does not imply that the specific t we have picked also has to complete P. As the node S is existentially quantified, it is sufficient if one of the alternatives generated by S, i.e. some t', completes P. In the second subterm covering the else case, $\nexists r$ reflects the fact that R's outgoing connectors are switched so that R represents a negative condition.

While the above is more illustrative of the rationale behind the adopted semantics, we prefer the equivalent interpretation based on conjunction for all practical purposes, as it less complex: $(\exists t|(S[c,s,t] \wedge \exists p|P[c,p,s,t])) \vee ((\nexists t|S[c,s,t]) \wedge \nexists r|R[c,r,s])$.

Universal nodes. There is only one type of universal node. A *universally quantified* node containing the free variables var_i is marked with $[\forall\ var_i^+]$. It works like an existentially quantified node, except that the extended bindings it generates are not alternatives, but *all* need to succeed. If no extended binding matching the node's SDDP exists in the first place, the expected semantics of $\forall$ quantification require that the expression evaluate to true – therefore, an else connector to (1) is always implied.




```
context Track inv
  Track.allInstances()→forAll(t1, t2, t3 |
    (t1.successor→exists(tx | tx = t2)
       and t2.successor→exists(ty | ty = t3))
    implies (Controller.allInstances()→exists(c1 |
      c1.tracks→exists(ta, tb, tc |
        ta = t1 and tb = t2 and tc = t3))))
```

Figure 3.3: Connected tracks share a controller

Figure 3.3 encodes the requirement that for any three consecutive tracks (∀), there

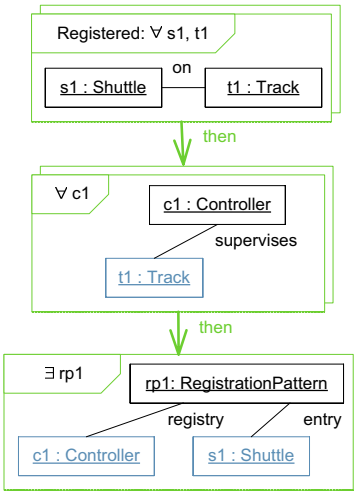


Figure 3.4: Registered with all available controllers

must be a controller (∃) supervising them all (see the included OCL listing). Figure 3.4 contains nested ∀ quantors – every shuttle needs to execute a registration pattern with every controller supervising its track. This is merely an illustrative example – as there is no interposed existential node, the two universal nodes could simply be merged.

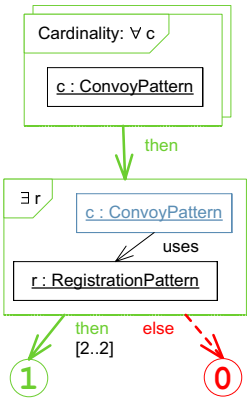


Figure 3.5: SDD encoding a cardinality from the Class Diagram

Cardinalities. It is possible to specify *cardinalities* for a node's then connector. These cardinalities concern the number of extended bindings that are generated for each individual binding that reaches a (existentially or for all) quantified node. If fewer alternatives than the minimum cardinality or more alternatives than the maximum cardinality are generated from a binding, the extended bindings are discarded and the original binding is propagated down the else connector. It is therefore not

possible to specify a cardinality for the else connector – whenever it is chosen, there is always exactly one binding, the original binding, anyway. For the same reason, it does not make sense to place cardinalities on either connector of a guard node, as there will always be one binding on the selected connector.

Figure 3.5 encodes one of the cardinalities specified in the underlying Class Diagram (see Figure 1.1), namely that each convoy pattern uses exactly two registration patterns. SDDs thus elimate the need to encode cardinalities by means of a set of graph patterns as presented in Section 2.2.6. Moreover, SDDs are capable of expressing constraints that cannot be captured by Class Diagrams, e.g., restricting the number of permitted concurrent object instances for each class or imposing conditional cardinalities.

3.1.3 Embedded Story Decision Diagrams

Formal specification languages typically allow the composition of complex properties from simpler properties. In the OCL, it is possible to reference more concrete properties in the definition of a property, whereas most visual specification techniques lack this capability. The presented approach offers similar functionality that provides a visual abstraction for arbitrarily complex structural relationships and constraints.

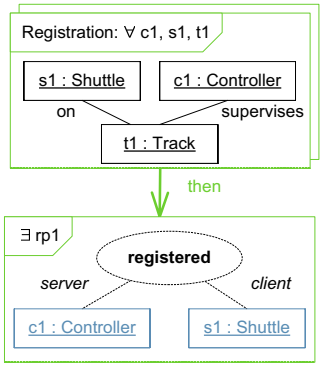
SDDs support the composition of specifications through *Embedded Story Decision Diagrams* (*ESDD*). ESDDs can be used to encode nontrivial properties that can then be referenced by several SDDs. In order to allow their reuse in different contexts, ESDDs are defined as patterns with free variables that are bound depending on the respective current context. If a node contains a reference to an ESDD, a binding only matches the node if it also fulfills the embedded pattern.

Simple ESDDs. An ESDD specification begins with a dedicated node type – a λ node $[Name : \lambda role_1, role_2, \dots]$ – that defines its name and available roles, i.e., free variables. While all other node labels are optional and mostly serve as comments, the λ node's label is mandatory, as it is later used to reference the ESDD. When the ESDD is invoked in a given context, the λ node then binds the local variables in accordance with the provided context. The ESDD is otherwise processed like a regular SDD, eventually evaluating to *true* or *false*. ESDDs introduce a local scope, which means that their bindings are not accessible from the host SDD and discarded as soon as a result has been obtained.

In the host node containing the reference to named property in question, we represent the ESDD using the UML symbol for a pattern, a dashed circle. The bound elements of the host node are assigned to the roles of the ESDD by means of dashed lines labeled with the respective role name. By default, ESDDs support optional

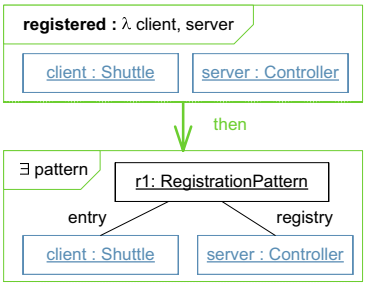
arguments: If a role is not explicitly bound by the host node, we implicitly add an existentially quantified element of the required type to the host node.

As a basic example, Figure 3.6 requires that all shuttles are correctly registered with all controllers that supervise the track they are on. Being registered simply requires the existence of a registration pattern as encoded by the ESDD in Figure 3.7.




```
context s1 : Shuttle inv
  s1.on.controllers→forAll(c1 |
    Shuttle::registered(s1,c1)
```

Figure 3.6: Shuttles must be properly registered




```
context Shuttle::registered(c1 : Controller, s1 : Shuttle)
  s1.registrationPatterns→exists(rp1 |
    rp1.controller = c1))
```

Figure 3.7: ESDD encoding the registered property

A more complex property that actually benefits from using an ESDD is encoded in Figures 3.8 and 3.9. If two shuttles are not close to each other, they must *not* form a convoy. If the leading shuttle is on the trailing shuttle's immediate next track, they *must* form a convoy. If the shuttles are two tracks apart, there is no restriction, giving them time to form or break the convoy. Figure 3.8 encodes these conditions. The ESDD Convoy in Figure 3.9 then checks whether a convoy pattern with all required dependencies exists.

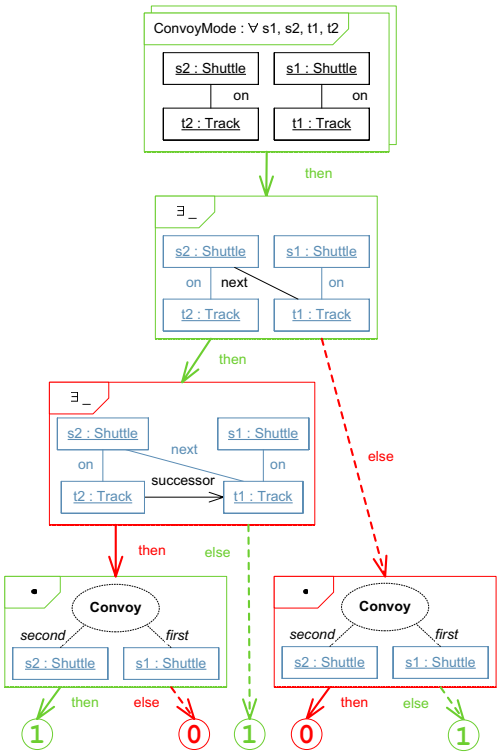


Figure 3.8: Shuttles must or must *not* form a convoy

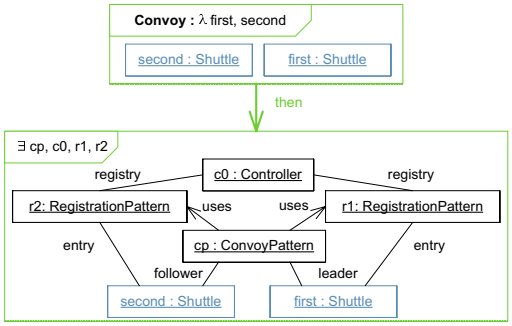


Figure 3.9: ESDD: The shuttles form a convoy

Parametrized ESDDs. It is possible to pass parameters to an ESDD in addition to the role bindings. Parameters with their types are also declared in the λ node: $[Name : \lambda\, parameter_1 : type_1, parameter_2 : type_2, \dots]$. They can be used wherever a value is required, e.g. in constraints on attributes. It is also possible to use parameters as cardinalities on then connectors.

Recursive ESDDs. As ESDDs are SDDs, it is possible to have nested ESDDs. This quite naturally leads to recursively defined patterns. Figure 3.10 recursively defines the property that track *to* is reachable from track *from*. The only restriction we place on recursive or indirectly recursive ESDD definitions is that it is not possible to specify a cardinality for the node containing the recursive invocation, as this may lead to paradox statements.

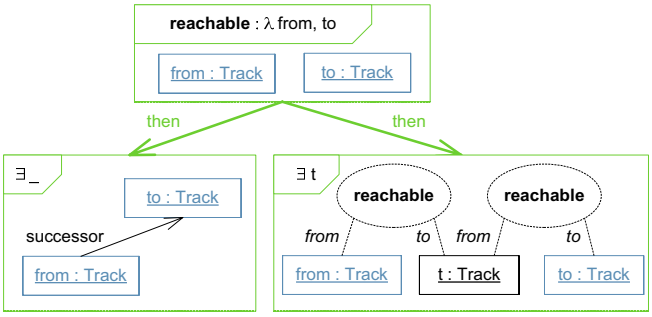


Figure 3.10: *to* is reachable from *from*

Recursion raises the question of termination. As the only context is provided by the previous application, the ESDD in Figure 3.10 could be applied infinitely, and the evaluation would not terminate if the tracks are not connected. On the other hand, we can assume that any instance graph consists of only a finite number of elements. There is therefore only a finite number of distinct initial bindings that can be passed to an ESDD's λ node. By adopting the restriction that, in any recursion, each initial binding is evaluated at most once, we can thus guarantee termination.

The ESDD in Figure 3.11 provides a more constructive definition of reachability. Additionally, it can be parametrized with the allowed minimum and maximum distance between the tracks.

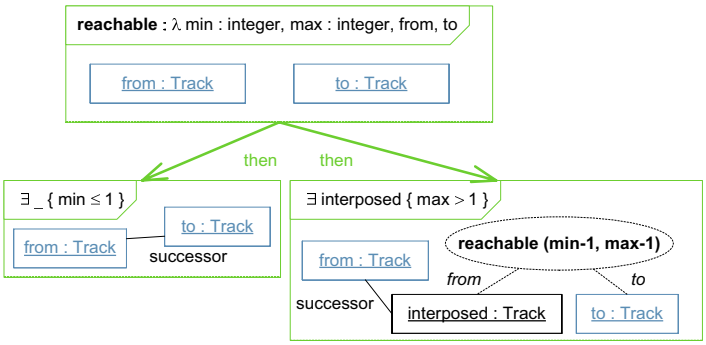


Figure 3.11: to is reachable from from

Figure 3.12 encodes the requirement that any two tracks in the system are connected. By means of the invocation parameters, we express the additional requirement that the distance between them is at most 100 links.

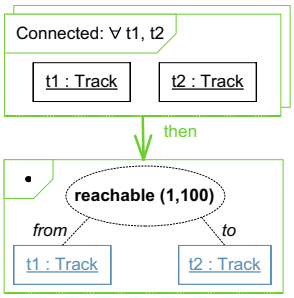


Figure 3.12: All tracks of the system are connected

In the example, the upper bound enforces a recursion depth of at most 100 and thus ensures termination. In general, however, proving termination for parametrized ESDDs is more difficult, as, potentially, there may now be infinitely many distinct initial bindings. We need to prove additional termination conditions, e.g., in the present case, that the upper bound is strictly decreasing.

Scoped nodes are a merely syntactical enhancement that is based on ESDDs. A *scoped node* contains a nested SDD, which inherits all the bindings of the host SDD, but itself only creates bindings that have local scope. Internally, the nested SDD is interpreted as an ESDD definition, whereas the scoped node is replaced with a guard node containing an ESDD invocation assigning each bound variable of the host SDD to the ESDD role of the same name.

While anything that can be expressed using scoped nodes can thus be specified using ESDDs, the mechanism provides a lightweight notation that is mostly useful for emulating parentheses. Especially if an SDD contains several unrelated ∀ quantifiers, scoped nodes can group related nodes, which also makes computation more efficient. For convenience, *scoped AND nodes* contain two nested SDDs that both need to be fulfilled, which simply provides a shorthand that is equivalent to two consecutive simple scoped nodes.

3.1.4 Transformations

The extensions we have so far presented enable the specification of more complex (pre-)conditions, i.e. they enhance the LHS of Story Patterns. As SDDPs in universal and existential nodes by definition never have side effects, such nodes can only encode an *Invariant Story Pattern*, which is sufficient for the main focus of SDDs, the modeling of constraints or system properties.

Transformation nodes. In order to provide a full replacement for Story Patterns, we introduce transformation nodes as a means of specifying transformations, i.e. the RHS of a Story Pattern. They are marked with $\rightarrow \sim\ var_i^+$; $+\, var_i^+$, listing which elements are destroyed ($\sim$) or created (+) by the node. Elements that are to be destroyed need to be bound already; the node only transforms an existing match. Which objects and links are to be created or destroyed is indicated using the same stereotypes as in Story Patterns, ≪create≫ and ≪destroy≫. As a more compact notation, SDDPs also allow using ++ for creation and $\sim$ for destruction.

Transformation nodes replace (1) leaf nodes. When a binding reaches the node, it is stored, and when the SDD evaluates to true, the transformations are applied to a randomly chosen set of bindings that fulfills the SDD. For existentially quantified properties, this simply means that once the first binding reaches a transformation node, the transformation is applied to that binding. Universally quantified transformations are only applied when a binding for each alternative has reached a transformation (or (1)) node. Note the significant difference between iteration (over existentially quantified properties), e.g., 'iterate over all tasks: if the task is completed, delete the task', and universal quantification, e.g., 'if *all* tasks are completed, delete *all* tasks'.

Figure 3.13 provides a simple example for an existentially quantified transformation, creating a pattern provided that no matching pattern already exists.

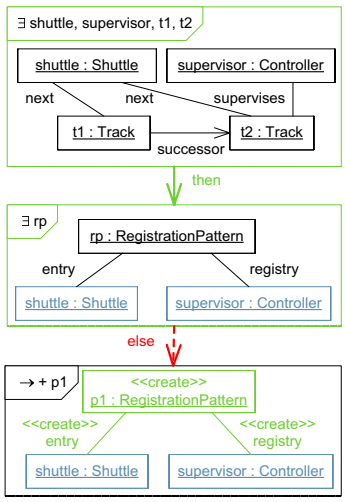


Figure 3.13: Creating a registration pattern

Conditional transformation nodes. The previous example, creating an element only if it is not already there, represents a very common idiom. Without such a guard, subsequent applications of the pattern would create arbitrarily many new instances. We therefore introduce a dedicated notation for compactly specifying

such a conditional transition: If we mark a transformation node as *conditional*, the node will only *ensure* that the postcondition holds. We indicate this with a dashed border and by writing $\rightarrow \exists\, var_i^+$. Internally, the pattern in Figure 3.14 will expand to Figure 3.13, but we have eliminated the need to explicitly specify the second node. If there are multiple created elements, we perform the check for each element individually, thus ensuring that all missing elements are created.

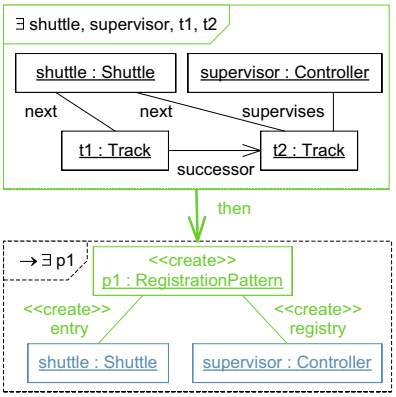


Figure 3.14: Conditional transformation: only create the pattern if it does not exist.

We use the same notation for a similar though different purpose. While SDDs are much more expressive than Story Patterns, they are also less compact. The clean separation of LHS and RHS entails that even the simplest transformation requires two nodes. We therefore allow quantified elements in conditional transformation nodes, indicated by $\exists\, var_i^+ \rightarrow \ldots$, and the deletion of previously unbound elements, indicated by $\rightarrow \nexists\, var_i^+$ (i.e. the pattern *ensures* that var_i does not exist). Figure 3.15 provides an example, deleting the second pattern if two are present. This is a purely pragmatic extension that basically provides a compatibility mode that simplifies the migration of existing Story Patterns (we are, however, inflexible on the point of forbidden elements, which are not supported).

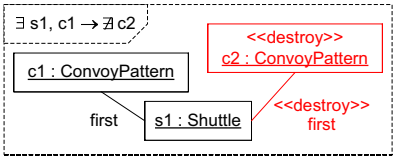


Figure 3.15: Conditional transformation: permits unbound elements.

Merge connectors. The merge connector is a syntactical feature that allows joining transformation nodes in order to avoid redundant specifications. Figure 3.16 provides an example: If a convoy pattern exists, the shuttle is moved forward. If there is no pattern, we would like to create a pattern and also move the shuttle.

Instead of specifying the shuttle movement twice, we only specify the pattern creation and reference the previously defined transformation using the merge connector. Note that the merge connector does not indicate a sequence – both transformations are applied at once.

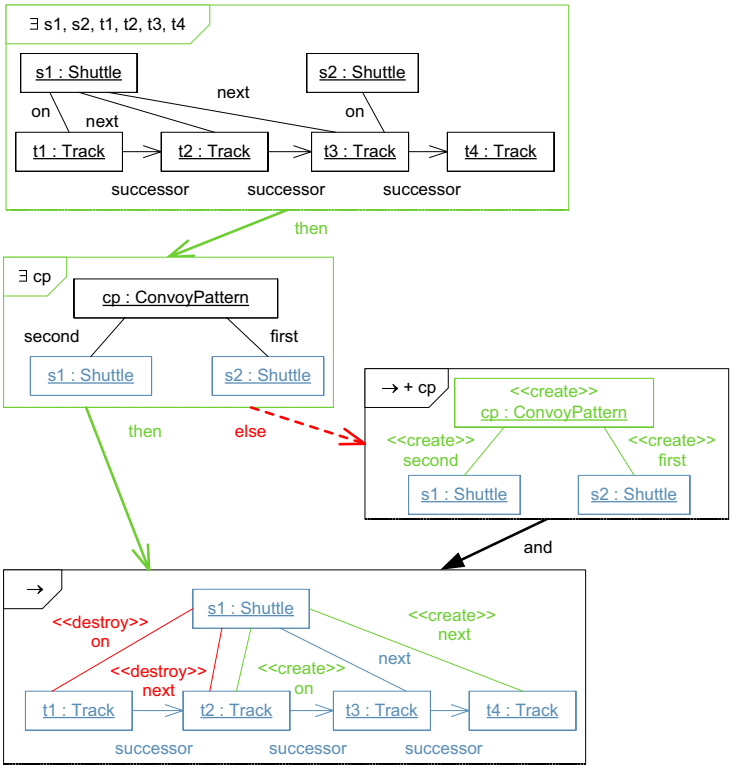


Figure 3.16: Merge connector: move the shuttle, if necessary create a pattern.

3.1.5 Annotations

Like in UML Object Diagrams or Story Patterns, it is possible to annotate SDDPs with constraints on attributes or collaboration statements.

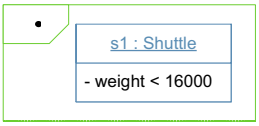


Figure 3.17: Constraint on an attribute of a single object

Constraints on the attributes of a single object can be specified within the object (see Figure 3.17. This is possible both for either quantified or bound objects. In

transformation nodes, new values can be assigned using :=. The guard expressions for an attribute can consist of literals, other attributes, ESDD parameters and calls to queries as defined by the UML (i.e., functions without side effects).

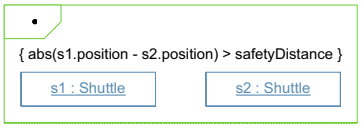
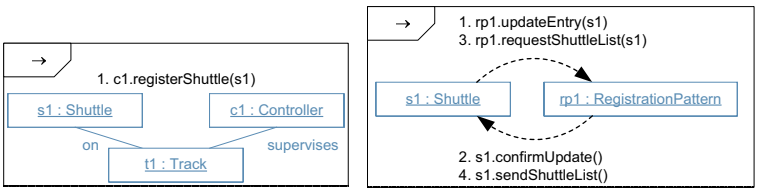


Figure 3.18: Constraints concerning multiple objects

Constraints that concern the attributes of multiple objects can be placed freely within the SDDP (see Figure 3.18). Attributes are referenced using the standard notation $object.attribute$. Expressions may contain the same elements as listed above.



a. Freely placed collaboration b. Collaborations with indicative arrows

Figure 3.19: Collaboration statements

Collaboration statements typically only appear in transformation nodes and encode a sequence of function calls. Collaboration statements may be freely placed (see Figure 3.19a), but preferably, they are placed on arrows indicating which object is calling the function (see Figure 3.19b).[4]

SDDPs are matched based on graph isomorphisms, in accordance with our formalization of Story Patterns. While we believe that isomorphisms are generally closer to the intuitive interpretation of a pattern, there may be cases where it is desirable to allow homomorphisms, i.e., different variables to refer to the same object. In Figure 3.20, the shuttle has a current position (on) and a destination. The destination has to be reachable from the current position (left branch). However, this would not match if the shuttle already is a the destination. We therefore allow that the two tracks might be identical using the special constraint $\cong$. The right branch then checks whether they actually are identical using the constraint $==$, which is equivalent to writing $\{t1 == t2\}$. Note that this example could easily be rewritten

[4] When modeling temporal properties in specific application contexts, it may be useful to employ collaboration statements in LHS nodes, e.g. to require the arrival of an incoming message in order to emulate the semantics of interface state charts. However, we define no semantics for such constraints in the general case.

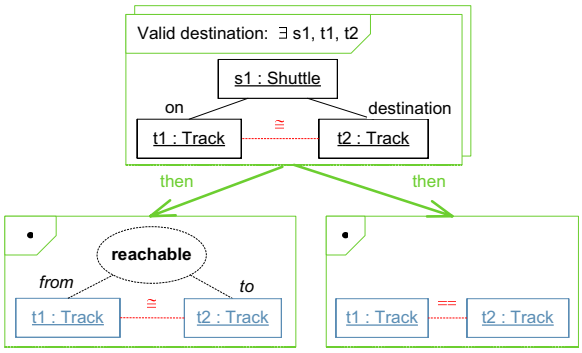


Figure 3.20: Explicitly permitted homomorphism

without homomorphisms. In general, it is always possible to use a set of alternatives using isomorphisms instead – it may just be much more verbose.

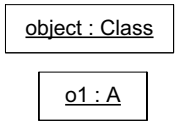
3.2 Syntax reference

In this section, we list the complete syntax of SDDs. We start with the syntax of Story Decision Diagram Patterns(SDDP), which slightly extends and modifies the existing Story Pattern respectively Story Diagram (Activity Diagram with Story Patterns) syntax, and then present the syntax of the surrounding SDDs. For each element, we present the abstract syntax (top) and a basic application example (bottom).

3.2.1 Story Decision Diagram Patterns

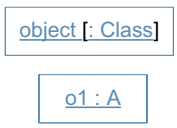
SDDPs may contain objects, links between objects, constraints and ESDDs.

Objects



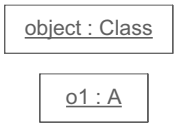
Quantified object. An unbound object that, if possible, is bound by the SDDP is drawn in black.[a] The object identifier and class name are both mandatory. A quantified object may also specify constraints concerning its attributes; the pattern will then backtrack over all candidates if the selected object does not fulfill the requirements.

[a] As black, we define 100% black or rgb(0, 0, 0).



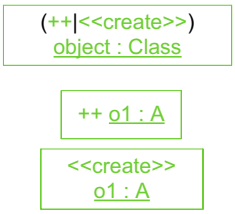
Bound object. An object that has been bound by a previous SDDP in the same SDD, i.e. was present as a quantified object with the same object identifier in a previous SDDP, is automatically drawn in slate blue.[b] In SDDs, the class name is optional, but unlike in Story Diagrams, where omitting the class name indicates that an object is previously bound, it is not omitted by default. In SDDs, it is therefore not possible to rebind an object identifier or even reassign it a different class – we believe, however, that providing this capability would only lead to confusing diagrams. On the other hand, type information is essential and should be made explicitly available if the user desires it. A bound object may introduce additional constraints (drawn in black) concerning its attributes. The constraints act as a filter; if the bound object does not meet the requirements, the SDDP does not match.

[b] As slate blue, we define rgb(90, 140, 180).



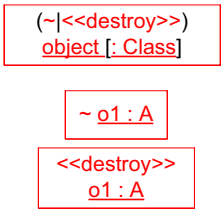
Locally quantified object. An unbound object that becomes locally bound within a scoped node is drawn in grey.[c] The object identifier and class name are both mandatory. At the end of the local scope, the binding is deleted, and the object identifier becomes unbound again.

[c] As grey, we define 60% black or rgb(95, 95, 95).



Created object. In SDDPs with side effects, newly created objects are marked with the stereotype ≪create≫ or ++ and are drawn in green.[d] The class name is mandatory. They may also specify attribute values that will be used to initialize the new object. In SDDs, SDDPs with side effects may only appear in transformation nodes.

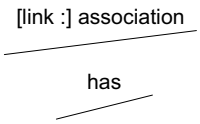
[d] As green, we define rgb(108, 206, 48).



Destroyed object. In SDDPs with side effects, objects that are deleted are marked with the stereotype ≪destroy≫ or and are drawn in red.[e] As transformation nodes do not contain quantifiers, all destroyed objects need to be previously bound and the class name is thus optional.

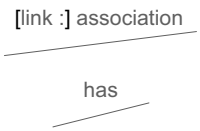
[e] As red, we define 100% red or rgb(255, 0, 0).

Links

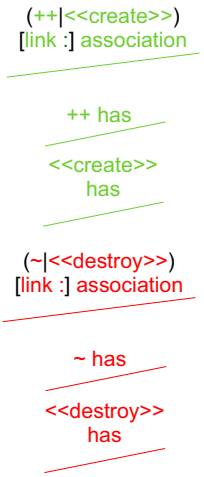


Quantified link. Previously unbound links are drawn in black. The association type is mandatory. As an extension of normal object diagram syntax, it is possible to assign a link identifier to a link, just like to an object. This is useful only in connection with quantification, mostly in case the modeler wants to universally quantify over a set of links between two unique objects.

Bound link. Previously bound links are drawn in slate blue. The association type is mandatory, the link identifier is optional even if the quantified link that introduced the binding used one.



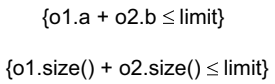
Locally quantified link. A previously unbound link that becomes locally bound within a scoped node is drawn in grey. The association type is mandatory, the link identifier is optional. At the end of the local scope, the binding is deleted, and any potential link identifier becomes unbound again.



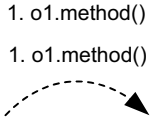
Create link. In SDDPs with side effects, newly created links are marked with the stereotype ≪create≫ or ++ and are drawn in green. The association type is mandatory, the link identifier is optional.

Destroy link. In SDDPs with side effects, links that are deleted are marked with the stereotype ≪destroy≫ or and are drawn in red. The association type is mandatory, the link identifier is optional even if the quantified link that introduced the binding used one.

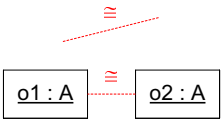
Constraints



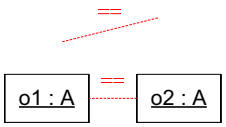
Constraint (attributes). Besides constraints concerning only a single attribute of an object, which can be specified inside the object itself, it is also possible to specify constraints that are expressions over multiple attributes of an object or different objects, literals, and queries. It is also possible to use built-in operators, e.g. arithmetic, on attributes. The restriction to UML query functions, i.e. predicates or functions without side effects, is not a technical requirement, but a precondition for reasonable matching semantics with predictable results.



Constraint (collaboration). In RHS nodes, collaboration statements enable SDDPs to call methods on bound objects, just as in UML Collaboration Diagrams. The leading number allows determining the order of the sequence of invocations in case of multiple statements. Preferably, the interacting objects are marked with a dashed arrow indicating the direction of the call. If used in existential nodes, collaboration statements serve as guards, i.e. the SDDP only matches when the corresponding method is called. While this feature is not included in the standard matching semantics, it might be useful for specific applications, e.g. translating Message Sequence Charts (MSC) into SDD-based scenarios.

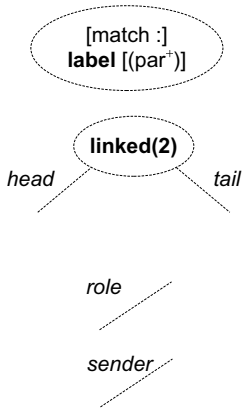


Homomorphism link. SDDPs are matched using graph isomorphism by default, i.e. different identifiers in the pattern are mapped to different objects in the instance graph. However, it is sometimes desirable to specify that two objects may or may not be the same. This kind of ambiguity is provided by non-injective graph homomorphisms. Story Patterns support this by means of textual *maybe*-annotations. SDDPs introduce a dashed red line with the label ≅ as a graphical notation. The link is automatically carried over into subsequent SDDPs in the same SDD.



Identity constraint. In connection with homomorphism links, one might later on in the SDD want to address the case where both objects are actually identical. This can, of course, simply be expressed by a regular constraint, but as the homomorphism link is there anyway, SDDPs support the == label as a special syntax for expressing this.

Embedded SDDs



ESDD. An Embedded SDD is represented by a dashed circle (the symbol for a pattern in the UML), with its name in bold face. Optionally, the ESDD may also have an instance identifier that differentiates multiple patterns in the same SDDP. Besides the role bindings that are assigned to it, an ESDD can also accept a list of parameters, usually primitive types such as numeric values, as a more economic alternative to assigning roles to value objects.

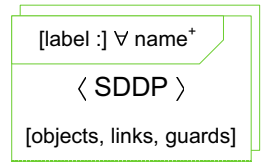
Role for ESDD. A role in an Embedded SDD is assigned to a quantified or bound object by a dashed black line, with the role name in italics.

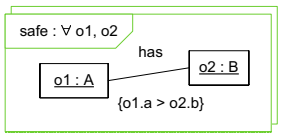
3.2.2 Story Decision Diagrams

SDDs consist of five classes of nodes and three types of connectors.

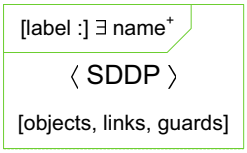
Node types

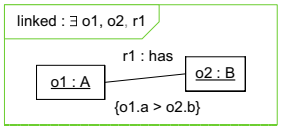
All non-leaf nodes use UML 2.0 boxes with a header field. Except for transformation nodes, which are always black, these boxes are typically green, but may automatically turn red if they are used to model forbidden properties.

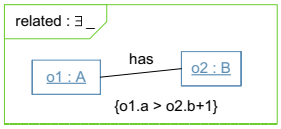




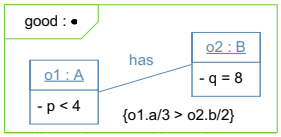
Quantified universal node. A quantified universal node contains at least one quantified object or link. It creates a set of required bindings, each of which needs to successfully match the overall SDD, i.e. reach a (1) leaf node, at some point. Operationally, the node causes the iteration over a set of bindings, AND joining the results. The quantor expression in the header can be computed automatically from the SDDP. The node label is optional. A quantified universal nodes may have (multiple) then connectors. The implied else connector always leads to a (1) leaf node, as any for all expression is true over the empty set.



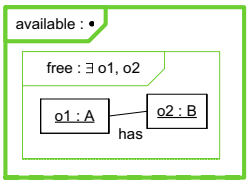




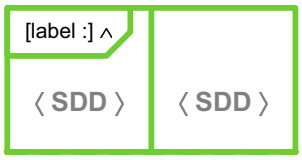
Existential node. An existential node that contains at least one quantified name element, e.g. a quantified object or a named quantified link, becomes a quantified existential node. Quantified existential nodes can introduce new possible bindings, i.e. increase the number of alternatives to be considered. Operationally, they create one possible binding at a time and, if it does not manage to reach a (1) leaf node, backtrack and try the next alternative, i.e. OR joining the results. The list of quantified elements in the header can be computed automatically from the SDDP. If the node contains no named quantified elements but only anonymous quantified links, the header contains an existential quantor followed by an underscore. The node label is only used for documentation and better readability and is thus optional. Like all existential nodes, quantified existential nodes can have (multiple) then and else connectors.

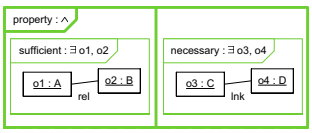


Guard existential node. An existential node that contains only bound elements and introduces additional constraints concering their attributes becomes a guard node. A guard node acts as a filter and can only reduce the number of eligible alternative bindings. In the header, it is marked by a bold dot. The node label is optional.

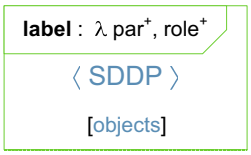


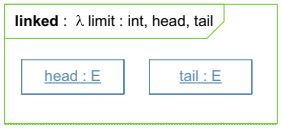
Scoped node. A scoped node is a guard existential node containing a nested SDD. It matches if the nested SDD matches, i.e. acts as a filter. The node introduces a local scope; all quantified objects and links inside the block node are thus only locally bound. However, all previous bindings remain valid inside the block node. A scoped node can always be replaced by an equivalent ESDD, but offers a more compact representation if reuse is not an objective. A scoped node is particularly useful when an SDD contains two unrelated conditions, the first of which is universally quantified. By placing the first condition inside a block, the independent second condition is not evaluated for each of the additional bindings created by the first condition. The node label is optional.



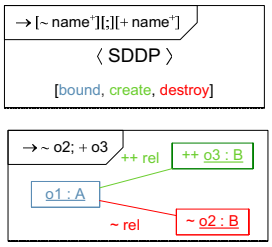


Scoped AND node. A scoped AND node basically combines to simple scoped nodes. It is equivalent to connecting two scoped nodes with a then connector. It is again mostly useful in connection with universally quantified subexpressions, as existentially quantified nodes can simply be chained without any adverse effects in order to obtain AND-semantics. The node label is optional.

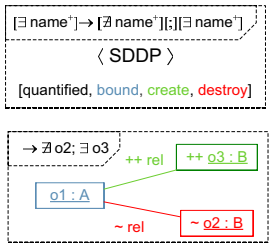




Lambda node. A lambda node is the initial node of an ESDD definition. The node label is required, as it defines the name of the ESDD by which it can later be referenced in SDDPs. The lambda node contains an SDDP which only contains bound objects (as there is no quantor creating new bindings, but only one binding that is passed in) which define the types of the ESDD's roles. The lambda expression in the header can be computed from these roles. Additionally, it may contain a list of typed parameters. The parameters, usually primitive numeric values, are useful for simple customizations and controlling recursion. A more verbose alternative to parameters is simply assigning roles to value objects.



Transformation node. Transformation nodes are marked with →. They may only contain bound, create and destroy objects and links, and destroy objects and links must be previously bound. Transformation nodes do not add or filter, but only process bindings. Transformation nodes appear in place of (1) leaf nodes. When a binding reaches the node, it is stored, and when the SDD evaluates to true, the transformations are applied to a randomly selected sufficient set of bindings.

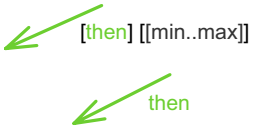


Conditional transformation node. Conditional transformation or ensure nodes expand to a set of existential and transition nodes and provide a more efficient syntax for certain idioms. If they create elements, they check whether matching elements already exist, indicated by → ∃. For backwards compatibility with Story Patterns, they also allow existentially quantified objects (∃ →) and destroying previously unbound obejcts (→ ∄) – if they can be bound.

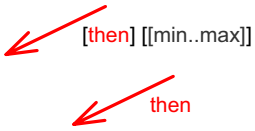
True leaf node. The (1) leaf node indicates that a binding has successfully satisfied this branch of the SDD. Whether this already means that the whole SDD matches depends on the preceding quantifiers. The label is optional.

False leaf node. The (0) leaf node indicates that a binding has failed to satisfy this particular branch of the SDD. However, there are two possibilities how the SDD might still be fulfilled: Either there is a preceding quantified existential node which can provide an alternative binding that succeeds, or there previously was a node with multiple exiting connectors (OR-branches), one of which might evaluate to true. The label is optional.

Connector types



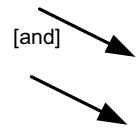
Then connector. The then connector connects a node to the condition that must follow if the node's SDDP matches at all. Optionally, it is possible to specify a cardinality, i.e. a restriction on how many different alternative bindings fulfilling the SDDP there may or must be per original binding. It is possible to specify more than one then connector; the multiple connectors then specify alternatives. The then connector is drawn as a solid green line. The label is optional, but typically included.



Then connector. There is a red variant of the then connector. It solely exists to make the modeler's intent more obvious – its semantics are identical to those of the green then connector. The then connector automatically turns red when it is connected to a (0) leaf node, or if the node's else connector is green. All node borders always share the color of the attached then connector; therefore, node borders turn red whenever the red then connector is used.

Else connector. The else connector connects a node to the condition that must follow if the node's SDDP does not match at all. This means that a specific binding arriving at a node either goes on down the then or the else connector, but never both. Specifically, those newly created bindings that fail to match the SDDP are *not* passed down the else connector. It is therefore pointless to specify a cardinality for the else connector, as the number of alternative bindings generated by the node will always be 0. The else connector is drawn as a dashed red line. The label is optional, but typically included.

Else connector. There is a green variant of the else connector. It solely exists to make the modeler's intent more obvious – its semantics are identical to those of the red else connector. The else connector automatically turns green when it is connected to a (0) leaf node, or if the node's then connector is red.



Merge connector. The merge connector is a solid black arrow with a massive tip. It is used to connect multiple transformation nodes. It exists for convenience and enables reusing transformation nodes. The condition 'IF a and not b THEN DO A, IF a and b THEN DO A AND B' can then be rendered using two transformation nodes, one specifying 'A' and one specifying 'B', and connecting 'B' to 'A' using a merge connector, instead of a transformation node specifying 'A AND B' (thus redundantly repeating 'A'). The label is optional.

3.3 Formal Semantics

Language definitions that focus on expressiveness and intuitive semantics often run into problems when it comes to defining the formal semantics, which the OCL itself illustrates. On the other hand, languages that are constructed starting from a set of formally motivated operators with precise semantics often suffer in terms of expressiveness and especially practical applicability. We therefore now show that the operational, control-flow-oriented informal semantics we have used above to introduce the specification technique can be mapped to a formal graph-based semantics that allows us to analyze and reason about the matching process. We will then be able to use SDDs for the specification of positive invariants of the system that must hold in every reachable state of the system, i.e., match every graph that is generated by the corresponding GTS. SDDs with side effects can be used to specify more complex graph transformation rules.

3.3.1 Variable Bindings

Story Decision Diagram Pattern Semantics. For the patterns in each individual node, we can build on our formalization of the semantics of Story Patterns. As SDDPs do not contain forbidden elements, each SDDP can be encoded as a simple graph pattern $[P]$, which can then be matched using standard matching semantics. Likewise, the SDDPs of transformation nodes can be translated into graph rules $[L] \rightarrow_r [R]$, which can also be applied normally – with the exception that the occurence of the LHS $o(L)$ is already determined by the preceding SDD nodes.

This is where a new aspect comes in: The SDDPs of an SDD are not independent of each other, but may contain bound objects that have already been matched by preceding nodes. When matching the pattern, we therefore have to respect these previous matchings. The straight-forward way to achieve this would be to take the graph morphism m mapping a pattern P into an instance graph G, pass it down to the subsequent pattern P', and merely extend it for the additional elements of P'. However, this would introduce the requirement that all SDDPs in an SDD are compatible, i.e., that the elements of P and P' are actually identical – otherwise, the morphisms for P could not be applied to P'. We therefore adopt a similar, but slightly more general solution.

Bindings. In order to relate the matches from different patterns in the same diagram to each other, we introduce an additional labeling $l_P^v := (V_S^N, V_S^E, v_P^N, v_P^E)$ for every graph P representing an SDDP of the SDD S. We label each node and edge with the corresponding variable from the set of node variables V_S^N and the set of edge variables V_S^E of S. V_S^N consists of all declared object identifiers and

V_S^E consists of all declared link identifiers and, as most links are anonymous, generated unique link identifiers. As we are working with attributed graphs, attributes are represented by attribute edges – so that $V_S^A \subseteq V_S^E$ – and attribute values are represented by value nodes in V_S^N. Finally, parameters (of ESDDs) are also represented by node variables $V_S^P \subseteq V_S^N$ that point to value nodes.

Based on this labeling, we can now share matched elements, attributes and parameters between patterns in the same SDD. A variable *binding* ξ for the node and edge variables of S and an attributed instance graph G is then a pair of functions $\xi = (\xi^N, \xi^E)$ with $\xi^N : V^N \to N_G \cup \bot$, $\xi^E : V^E \to E_G \cup \bot$, where N_G is the set of nodes of G, E_G is the set of edges of G, and $\bot$ is the undefined element. The binding functions are typically partial, as some variables may not be bound yet or, in case of alternative paths through the SDD, may never be bound at the same time. We write $\xi_1 \leq \xi_2 := \forall v \in \mathsf{dom}(\xi_1) : (\xi_1(v) = \xi_2(v)) \vee (\xi_1(v) = \bot)$ if ξ_2 is equal to or a more restrictive extension of ξ_1. We denote the empty binding that maps all variables to $\bot$ by τ.

Pattern matching. We use $var(P)$ to denote the pair of sets of node and edge variables that occur in P, i.e. are in the range of the labeling functions v_P^N and v_P^E. In order to match the pattern P in the instance graph G, we define $P[\xi]$ as the graph which results from substituting all nodes and edges of P with the elements assigned to the corresponding variables by ξ, i.e., we replace each $n \in N_P$ with $n_{P[\xi]} := \xi^N(v_P^N(n))$ and each $e \in E_P$ with $e_{P[\xi]} := \xi^E(v_P^E(e))$, provided that ξ is defined for all variables $v \in var(P)$. Together, the variable labeling l_P^v of P and the binding ξ define a graph morphism between P and $P[\xi]$. We call a binding ξ *valid* if $P[\xi]$ is a correct subgraph of G, i.e., $P[\xi] \leq G$.

Given a pattern P and a binding ξ, we define the set of free variables of P as $free(P, \xi) := \{v \mid v \in var(P) \wedge \xi(v) = \bot\}$. We then say that the pattern P constrained by the existing binding ξ matches a graph G, written as $P|_\xi \precsim G$, if there is a binding ξ' that extends ξ for the variables in $free(P, \xi)$ so that $P[\xi'] \leq G$.

We use $\mathcal{X}_S[G]$ respectively $\mathcal{X}_S[N_G, E_G, V_S^N, V_S^E]$ (for N_G the set of all nodes of G, E_G the set of all edges of G, and variables V_S^N and V_S^E of S) to denote the set of all possible bindings of an SDD S over a graph G.

3.3.2 Witness Sets

Diagram structure. For an SDD S, we define $\mathcal{N}_S$ as the set of its nodes. For each node $n \in \mathcal{N}_S$, we define P_n as the pattern contained by n, $\mathsf{parent}(n)$ as the set of parent nodes connected to n by outgoing connectors, with its transitive closure $\mathsf{parent}^*(n)$, and $\mathsf{then}(n)$ and $\mathsf{else}(n)$ as the set of nodes connected to n by

then respectively else connectors. Cardinalities are represented by two functions $min : \mathcal{N}_S \times \mathcal{N}_S \to \mathbb{N}$ and $max : \mathcal{N}_S \times \mathcal{N}_S \to \mathbb{N}$, where $min(n, n')$ respectively $max(n, n')$ is the minimum respectively maximum cardinality for the connector from n to n'.

λ_S denotes the unique root node of the SDD S with $\mathsf{parent}(\lambda_S) = \emptyset$. The set true_S contains all (1) and transformation nodes of S, false_S contains all (0) nodes of S.

We further define $var(n) := var(P_n)$ as the variables appearing inside n and $free(n, \xi) := free(P_n, \xi)$ as the free variables of n that are not bound, i.e. mapped to $\perp$, by ξ.

Witnesses. Only a subset of the possible bindings $\mathcal{X}_S$ *satisfies* the SDD S, i.e. is valid for a set of patterns P_n on a path to a (1) node. We can immediately discard all those bindings that are not valid for any pattern P_n, e.g. because they do not all required variables. However, even those bindings that are valid for one pattern P_n might not be valid for some other pattern $P_{n'}$ on the same path. When evaluating an SDD, we therefore need to consider a binding's context, i.e., nodes and their connections.

We define an application ζ as a pair (n, ξ) of a node n and a binding ξ. We call a valid application a *witness*. An application is valid if a path from λ_S to n exists so that ξ is valid for all nodes on the path (excluding n) but binds no additional variables:

$$\begin{aligned}\omega(n, \xi) :=& \exists (n_1, \ldots, n_k) \in \mathcal{N}_S{}^* : \\ & (n_1 = \lambda_S \wedge n_k = n \wedge \bigwedge_{i=1..k-1} (n_i \in \mathsf{parent}(n_{i+1}) \wedge P_{n_i}[\xi] \leq G) \wedge \\ & \forall v : \xi(v) \neq \perp \Rightarrow v \in \bigcup_{i=1..k-1} var(n_i)). \end{aligned} \tag{3.1}$$

The set of possible witnesses for an SDD S is then

$$\mathcal{Z}_S := \{(n, \xi) \mid n \in \mathcal{N}_S \wedge \xi \in \mathcal{X}_S \wedge \omega(n, \xi)\}. \tag{3.2}$$

We further define the truth value $eval(\zeta)$ of a witness $\zeta = (n, \xi)$ as $true$ if n is a (1) or transformation node, $false$ if n is a (0) node, and else $\perp$:

$$eval(\zeta) := \begin{cases} true & \mid n \in \mathsf{true}_S \\ false & \mid n \in \mathsf{false}_S \\ \perp & \mid otherwise. \end{cases} \tag{3.3}$$

As the truth value of a witness may thus be undefined, we use the convention that boolean operators ($\wedge$, $\vee$ and $\neg$) applied to $\perp$ also yield $\perp$ in the following. A

witness whose truth value is defined is *final*, all other witnesses are *intermediate* and represent unfinished evaluations.

Candidate sets. When informally introducing the semantics of SDDs above, we used an operational interpretation where we iteratively propagated individual bindings across the SDD. In order to define the formal semantics using set-based logic, we need to consider sets of bindings.

For a witness ζ of a universal node n, *each* extension of the contained binding that the node generates ultimately needs to satisfy the SDD S, or ζ will not satisfy the SDD. We group the new witnesses that n generates out of ζ into a *candidate set* of witnesses that need to succeed together. We define such a candidate set as $\mathcal{C} \in \wp(\mathcal{Z}_S)$. $\mathcal{C}$ only satisfies S if all witnesses $\zeta \in \mathcal{C}$ satisfy S. The truth value of $\mathcal{C}$ is thus defined as

$$eval(\mathcal{C}) := \bigwedge_{\zeta \in \mathcal{C}} eval(\zeta). \tag{3.4}$$

As for witnesses, a candidate set is *final* if its truth value is defined, i.e., it only contains *final* witnesses.

Alternative sets. An existential node or the presence of multiple then or else connectors can create multiple alternative ways to extend the binding ξ of a witness ζ, only one of which needs to satisfy S. The new witnesses that the node generates out of ζ thus form an *alternative set* of witnesses.

Result sets. If, starting with a single initial binding for the root node, we naively applied these definitions, we would end up with an nested structure of candidate and alternative sets. If the witness we process is part of some candidate set, we would generate a new candidate set that contains an alternative or candidate set in place of the witness – likewise for witnesses in alternative sets. Such a structure would greatly complicate the formalization. We therefore prefer a flattened structure with only two levels, a set of alternative candidate sets $\mathcal{A} \in \wp(\wp(\mathcal{Z}_S))$. We call such a set of candidate sets a *result set*.

We start evaluation with a single candidate set containing the initial binding. For the root node λ_S of an SDD, we define $\mathcal{A}_\lambda := \{\{(\lambda_S, \tau)\}\}$, i.e. there is one candidate set consisting of the only witness, the empty binding τ at λ_S.

Now, whenever a node generates alternatives ζ^i from a witness ζ, for each $\mathcal{C}$ containing ζ we add a new candidate set $\mathcal{C}^i$ where ζ is replaced by $\zeta^i$to the result set $\mathcal{A}$. As $\mathcal{C}$ is a set of witnesses, each of which may have alternative extensions, the number of new candidate sets $\mathcal{C}^{ijk\cdots}$ generated from $\mathcal{C}$ by a node depends on the Cartesian product of the extensions for each witness in $\mathcal{C}$. Existential nodes thus increase the *number* of candidate sets and thus the size of the result set.

When a universal node generates new interpedendent witnesses from a witness ζ, we simply create a new candidate set $\mathcal{C}'$ where ζ is replaced by the generated witnesses $\zeta^1 \dots \zeta^k$ for each $\mathcal{C}$ containing ζ. Universal nodes thus increase the *size* of the candidate sets.

As the sets are alternatives, i.e., one valid candidate set is sufficient, the truth value of a result set $\mathcal{A}$ is defined as

$$eval(\mathcal{A}) := \bigvee_{\mathcal{C} \in \mathcal{A}} eval(\mathcal{C}). \tag{3.5}$$

The set of all witnesses occuring in a result set $\mathcal{A}$ is denoted by $\mathcal{W}_\mathcal{A} := \bigcup_{\mathcal{C}_i \in \mathcal{A}} \mathcal{C}_i$. A result set is *final* if all witnesses in $\mathcal{W}_\mathcal{A}$ are *final*.

Propagation. We now formalize the computations on result sets that we have described above. For each node n, we define the *propagation function*

$$apply_n : \mathcal{G} \times \wp(\wp(\mathcal{Z}_S)) \to \wp(\wp(\mathcal{Z}_S)), \tag{3.6}$$

which basically removes obsolete candidates and adds appropriately extended versions. When computing the updated result set $\mathcal{A}' = apply_n(G, \mathcal{A})$, we initialize $\mathcal{A}' = \mathcal{A}$. For each witness $\zeta = (n_\zeta, \xi_\zeta)$ for n from $\mathcal{W}_{\mathcal{A}'}$ (i.e. $\zeta \in \mathcal{W}_{\mathcal{A}'} \wedge n_\zeta = n$), the following steps are then performed by $apply_n$:

1. The possible extensions of the binding ξ_ζ are computed. We define

$$\mathcal{X}_\zeta^{(\mathsf{t})} := \{\xi'_\zeta \mid P_n[\xi'_\zeta] \leq G \wedge \xi_\zeta \leq \xi'_\zeta \wedge \forall v : \xi'_\zeta(v) \neq \xi_\zeta(v) \Rightarrow v \in free(n, \xi_\zeta)\}, \tag{3.7}$$

   i.e. those ξ'_ζ that are valid for P_n and extend ξ_ζ with the variables introduced by P_n. If no such ξ'_ζ exists, i.e. $\mathcal{X}_\zeta^{(\mathsf{t})} := \emptyset$, we have $\mathcal{X}_\zeta^{(\mathsf{e})} := \{\xi_\zeta\}$, otherwise $\mathcal{X}_\zeta^{(\mathsf{e})} := \emptyset$, i.e.

$$\mathcal{X}_\zeta^{(\mathsf{e})} := \begin{cases} \emptyset & \mid \mathcal{X}_\zeta^{(\mathsf{t})} \neq \emptyset \\ \{\xi_\zeta\} & \mid \mathcal{X}_\zeta^{(\mathsf{t})} = \emptyset. \end{cases} \tag{3.8}$$

   Note that exactly one of the sets is thus always empty. The definition covers both quantified and guard nodes. As guard nodes do not introduce any new variables, we have the special case that $\xi'_\zeta = \xi_\zeta$ so that ξ_ζ is either placed in $\mathcal{X}_\zeta^{(\mathsf{t})}$ or $\mathcal{X}_\zeta^{(\mathsf{e})}$ depending on whether $P_n[\xi'_\zeta] \leq G$ holds.

2. The corresponding witnesses are computed, i.e. the generated bindings are propagated along all applicable connectors – which are *either* the then or the else connectors.

If cardinalities are specified, we first need to verify whether the number of generated extended bindings satisfies the constraints of at least one then connector, i.e. $\exists n' \in \mathsf{then}(n) : min(n,n') \leq \#\mathcal{X}_\zeta^{(\mathsf{t})} \leq max(n,n')$. Otherwise, we need to discard the generated bindings and send the original binding down the else connector by setting $\mathcal{X}_\zeta^{(\mathsf{t})} := \emptyset$ and, accordingly, $\mathcal{X}_\zeta^{(\mathsf{e})} := \{\xi_\zeta\}$.

We then define the set of generated witnesses as

$$\begin{aligned}\mathcal{W}_\zeta^+ :=& \{(n',\xi') \mid n' \in \mathsf{then}(n) \wedge \xi' \in \mathcal{X}_\zeta^{(\mathsf{t})} \\ & \qquad \wedge min(n,n') \leq \#\mathcal{X}_\zeta^{(\mathsf{t})} \leq max(n,n')\} \cup \\ & \{(n',\xi') \mid n' \in \mathsf{else}(n) \wedge \xi' \in \mathcal{X}_\zeta^{(\mathsf{e})}\}. \end{aligned} \tag{3.9}$$

3. The result set $\mathcal{A}'$ is updated. This implicitly removes ζ from $\mathcal{W}_{\mathcal{A}'}$ and adds the new bindings: $\mathcal{W}_{\mathcal{A}'} := \mathcal{W}_{\mathcal{A}'} \setminus \zeta \cup \mathcal{W}_\zeta^+$.

   (a) If n is universal, we define

$$\begin{aligned}\mathcal{A}'_\forall := \{\mathcal{C}' \mid \exists \mathcal{C} \in \mathcal{A}' :& (\zeta \in \mathcal{C} \wedge \mathcal{C}' = \mathcal{C} \setminus \zeta \cup \mathcal{W}_\zeta^+) \vee \\ & (\zeta \notin \mathcal{C} \wedge \mathcal{C}' = \mathcal{C})\}, \end{aligned} \tag{3.10}$$

   i.e. extending each candidate set with the new witnesses.

   (b) If n is existential, we define

$$\begin{aligned}\mathcal{A}'_\exists := \{\mathcal{C}' \mid \exists \mathcal{C} \in \mathcal{A}' :& (\exists \zeta' \in \mathcal{W}_\zeta^+ : (\zeta \in \mathcal{C} \wedge \mathcal{C}' = \mathcal{C} \setminus \zeta \cup \zeta')) \vee \\ & (\zeta \notin \mathcal{C} \wedge \mathcal{C}' = \mathcal{C})\}, \end{aligned} \tag{3.11}$$

   i.e. adding a new alternative candidate set for each new witness.

3.3.3 Story Decision Diagram Semantics

In order to evaluate an SDD S, we start with a result set $\mathcal{A}$ containing a single candidate (consisting of the initial binding) and successively apply the propagation function of each node of the SDD (using a breadth-first or preorder depth-first traversal) to it, extending and modifying the result set until it is *final*. The evaluation results in a unique final result set that serves to define the semantics of S. Note that all nodes actually need to operate on the same instance of $\mathcal{A}$ as candidate sets may contain witnesses for any node in the SDD so that simple recursion down any particular branch could only return results for individual witnesses, but typically not candidate sets.

Iteration function. In order to achieve the required evaluation order, i.e. that every node uses the output of the previous node as its input, we define the iteration function

$$iterate(\mathcal{N}, \mathcal{A}) := \begin{cases} [\![n]\!]^G_{iterate(\mathcal{N}\setminus n, \mathcal{A})} \mid n \in \mathcal{N} & \mid \mathcal{N} \neq \emptyset \\ \mathcal{A} & \mid \mathcal{N} = \emptyset, \end{cases} \tag{3.12}$$

where $[\![n]\!]^G_{\mathcal{A}}$ is the semantics of node n for graph G and set of alternative candidate sets $\mathcal{A}$ as defined below. The iteration function passes the result set $\mathcal{A}$ through every node in the set of sibling nodes $\mathcal{N}$ in turn.[5]

Semantics definition. We can now define the semantics of an SDD S. For leaf nodes, we have

$$[\![(1)]\!]^G_{\mathcal{A}} := \mathcal{A}, \tag{3.13}$$

$$[\![(0)]\!]^G_{\mathcal{A}} := \mathcal{A}, \tag{3.14}$$

i.e. they simply return the original result set.

For non-leaf nodes, we define

$$[\![n]\!]^G_{\mathcal{A}} := iterate(\mathsf{then}(n) \cup \mathsf{else}(n), apply_n(G, \mathcal{A})), \tag{3.15}$$

i.e. we first apply n's propagation function and then pass the result through all of n's children.

Finally, we define for the whole SDD:

$$[\![S]\!]^G := \{\mathcal{C} \mid \mathcal{C} \in [\![\lambda_S]\!]^G_{\{\{(\lambda_S, \tau)\}\}} \ \wedge \ eval(\mathcal{C})\}, \tag{3.16}$$

i.e. the semantics of the SDD S are defined as the satisfying final candidate sets generated by its root node λ_S, evaluated for the single candidate set consisting of the empty binding τ at λ_S. Note that all candidate sets in $[\![\lambda_S]\!]^G_{\{\{(\lambda_S,\tau)\}\}}$ are *final* so that $eval(\mathcal{C})$ is always defined.

The truth value of an SDD S is then

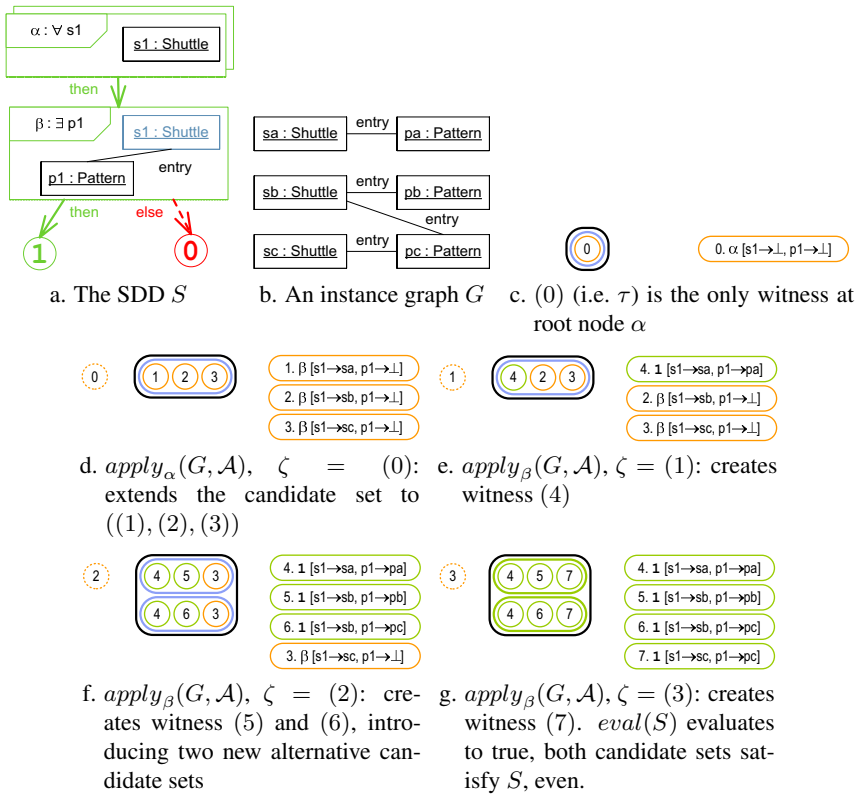
$$eval(S) := ([\![S]\!]^G \neq \emptyset). \tag{3.17}$$

Negation. We define the negation of an invariant SDD S, written as $\overline{S}$, as the SDD that is satisfied by all graphs G that do not satisfy S. $\overline{S}$ can be derived by inverting

[5] Note that $iterate$ is in fact a function in spite of the fact that n is chosen non-deterministically. As the nodes in $\mathcal{N}$ are siblings, no node in $\mathcal{N}$ will generate new witnesses for any other node in $\mathcal{N}$. For a fixed set of witnesses, we have $apply_n(G, apply_{n'}(G, \mathcal{A})) = apply_{n'}(G, apply_n(G, \mathcal{A}))$ as the invocations operate on disjunct subsets of the witness set and their effects on the result set are orthogonal.

all leaf nodes and quantifiers of S, i.e. turning all (explicitly specified and implied) (1) leaf nodes of S into (0) leaf nodes and vice versa, and turning all existential quantifiers ($\exists$) in S into universal quantifiers ($\forall$) and vice versa.

Examples. We now discuss three examples that illustrate the introduced semantics, especially the relationship between candidate sets and witnesses.



a. The SDD S

b. An instance graph G

c. (0) (i.e. τ) is the only witness at root node α

d. $apply_\alpha(G, \mathcal{A})$, $\zeta = (0)$: extends the candidate set to $((1), (2), (3))$

e. $apply_\beta(G, \mathcal{A})$, $\zeta = (1)$: creates witness (4)

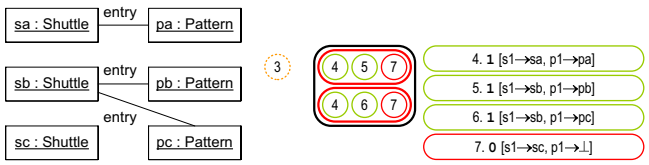
f. $apply_\beta(G, \mathcal{A})$, $\zeta = (2)$: creates witness (5) and (6), introducing two new alternative candidate sets

g. $apply_\beta(G, \mathcal{A})$, $\zeta = (3)$: creates witness (7). $eval(S)$ evaluates to true, both candidate sets satisfy S, even.

Figure 3.21: Example 1: Successful evaluation of a simple property

In Figure 3.21, we present a basic example. The SDD S in Figure 3.21a is evaluated on graph G in Figure 3.21b. Figures 3.21c–g then list the witness ζ that is currently processed by $apply$, the result set $\mathcal{A}$, and the set of witnesses $\mathcal{W}_\mathcal{A}$ for each iteration of the propagation functions. The result set $\mathcal{A}$ is marked by the outer (black) border, the candidate sets $\mathcal{C}$ in $\mathcal{A}$ are symbolised by the inner (blue) border, and the witnesses are represented by numbers in (orange) circles referencing the corresponding elements of $\mathcal{W}_\mathcal{A}$. Final witnesses and candidate sets are drawn in green or red, according to their truth value.

While the property holds for graph G, graph G' in Figure 3.22a is not a correct match (as sc is missing a pattern). Evaluation proceeds in an identical fashion to Figure 3.21, except for witness (3) in Figure 3.22b. As no pattern is found, the



a. An incorrect instance graph G'

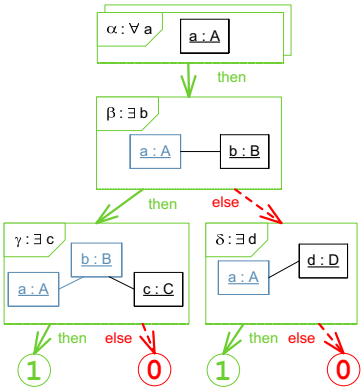
b. $apply_\beta(G', \mathcal{A})$, $\zeta = (3)$: creates witness (7). $eval(S)$ evaluates to false, no candidate set satisfies S.

Figure 3.22: Example 1': For the incorrect graph G', the last step differs

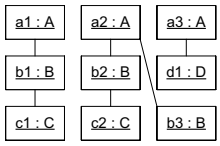
witness proceeds to the (0) node.

The second example in Figure 3.23 is more complex. Each A must have a B with a C, or a D. There are multiple (1) nodes, and as $a1$ and $a2$ have a valid B but no D, whereas $a3$ only has a valid D, the successful candidate set unites witnesses that are at different leaf nodes.

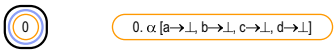
The third example is introduced in Figure 3.24 and evaluated in Figures 3.25 and 3.26. The example contains two nested universal quantors and serves to illustrate how evaluation is nonetheless based on a flattened data structure. In this example, we not only list the current result set and the currently selected witness, but also which obsolete candidate sets are eleminated in each step.



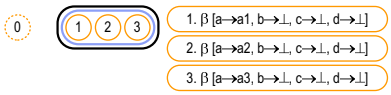
a. The SDD S



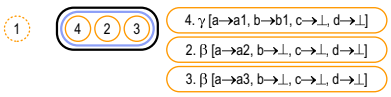
b. An instance graph G



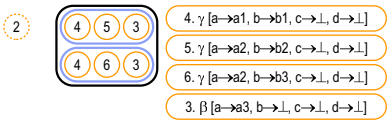
c. (0) (τ) at the root node α



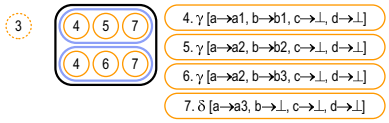
d. $apply_{\alpha}(G, \mathcal{A}), \zeta = (0)$



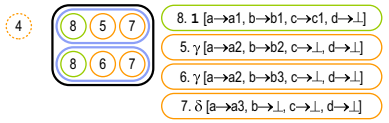
e. $apply_{\beta}(G, \mathcal{A}), \zeta = (1)$



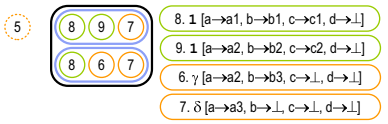
f. $apply_{\beta}(G, \mathcal{A}), \zeta = (2)$



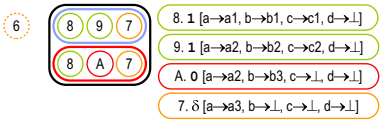
g. $apply_{\beta}(G, \mathcal{A}), \zeta = (3)$



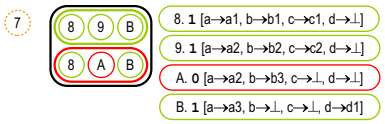
h. $apply_{\gamma}(G, \mathcal{A}), \zeta = (4)$



i. $apply_{\gamma}(G, \mathcal{A}), \zeta = (5)$

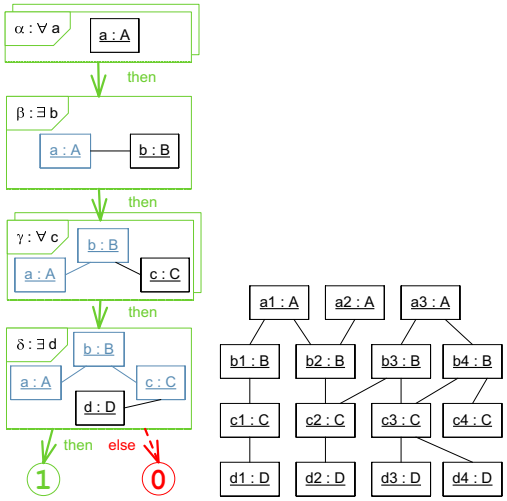


j. $apply_{\gamma}(G, \mathcal{A}), \zeta = (6)$



k. $apply_{\delta}(G, \mathcal{A}), \zeta = (7)$. $eval(S)$ evaluates to true as one of the candidate sets is valid. Note that the successful witnesses are at different (1) nodes.

Figure 3.23: Example 2: Successful evaluation of a more complex property



a. The SDD S b. An instance graph G

Figure 3.24: Example 3: Nested universally quantified nodes

a. Witnesses (0)-(8) b. Witnesses (9)-(F) c. Final witness set $\mathcal{W}_{\mathcal{A}}$

Figure 3.25: Example 3: Intermediate and final witnesses

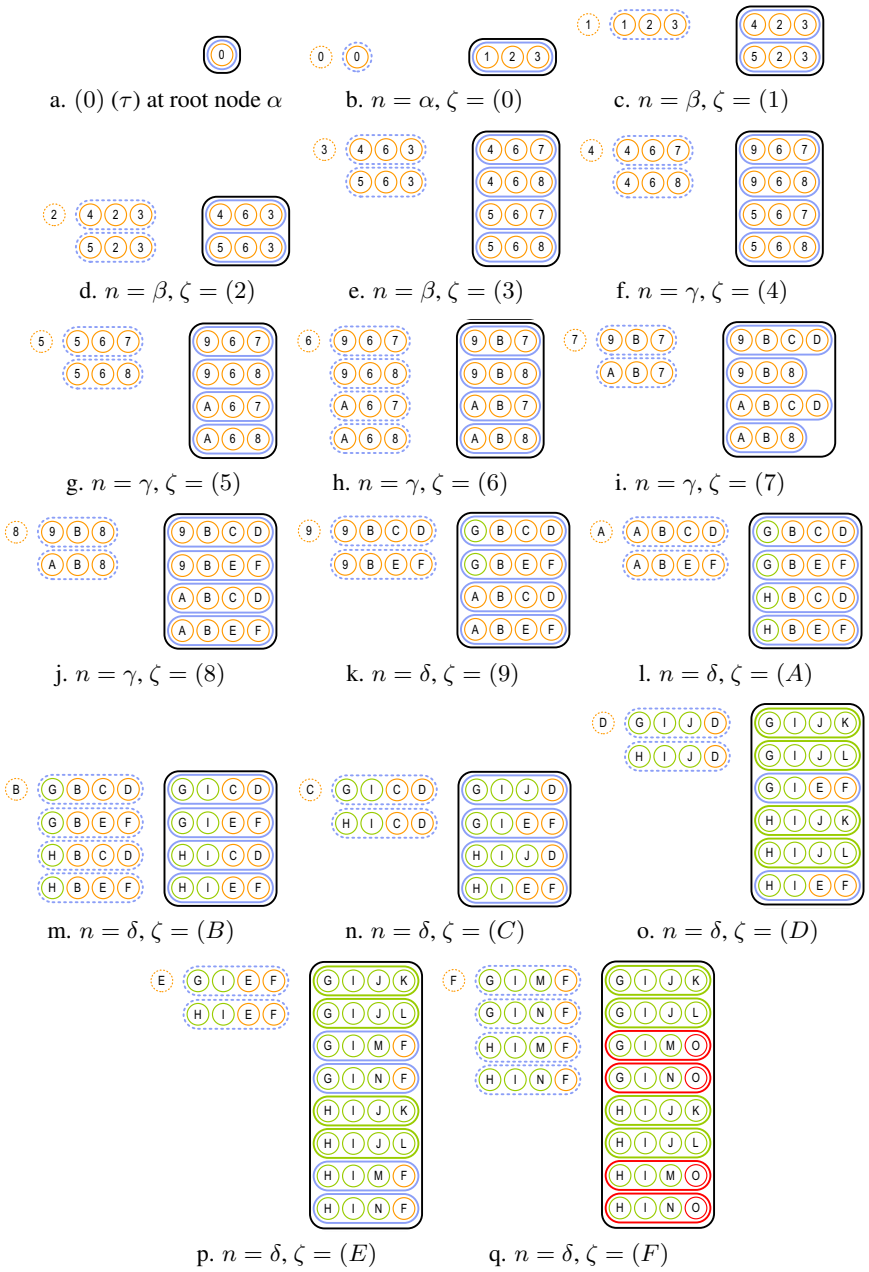


Figure 3.26: Example 3: Result sets. Evaluation succeeds

3.3.4 Embedded Story Decision Diagram Semantics

In order to define the semantics of ESDDs, we need to extend the semantics of SDDs in three places: We need to define the way how an ESDD's λ node bind roles to instances, we need to deal with ESDD invocations in the host nodes containing them, and we need to formalize the semantics of recursive ESDDs.

λ nodes. Differently from SDDs, ESDDs typically do not use τ as their initial binding, but define roles in their λ node which are bound externally. The roles are defined as the elements of the pattern R_λ. For each host node n containing a reference to an ESDD F, we define a partial graph isomorphism m_F from P_n to R_λ, mapping elements of the host SDDP to roles of F in accordance with the dashed role connectors in the diagram.

The mapping function $\ell_{F_n} : \mathcal{X}_S \to \mathcal{X}_F$ then performs the actual rebinding, binding F's variables in accordance with the binding in the host node. For a binding ξ_P and variable labelings $l_P^v := (V_S^N, V_S^E, v_P^N, v_P^E)$ for P_n and $l_R^v := (V_F^N, V_F^E, v_R^N, v_R^E)$ for R_λ, we define

$$\ell_{F_n}(\xi_P) := (\xi_F^N, \xi_F^E) | \xi_F^N \circ v_R^N \circ m_F^N = \xi_P^N \circ v_P^N \wedge \xi_F^E \circ v_R^E \circ m_F^E = \xi_P^E \circ v_P^E, \quad (3.18)$$

i.e. each element of R_λ is bound to the same instance as the element of P that is matched onto it by m_F.

Candidate set evolution. For candidate sets, we define the *evolved from* relation $\mathcal{C} \sqsubseteq \mathcal{C}'$ which indicates that $\mathcal{C}'$ has evolved out of $\mathcal{C}$. We have

$$\mathcal{C} \sqsubseteq \mathcal{C}' := \forall (n, \xi) \in \mathcal{C} : (\exists (n', \xi') \in \mathcal{C}' : \xi \leq \xi' \wedge n \in \mathsf{parent}^*(n')), \quad (3.19)$$

i.e. for each witness in $\mathcal{C}'$, there needs to be a witness in $\mathcal{C}$ that is less or equally restrictive at a possible parent node.

We extend this notation to result sets so that for a candidate set $\mathcal{C}$ and a result set $\mathcal{A}'$, we have

$$\mathcal{C} \sqsubseteq \mathcal{A}' := \exists \mathcal{C}' \in \mathcal{A}' : \mathcal{C} \sqsubseteq \mathcal{C}'. \quad (3.20)$$

ESDD invocations do not generate new bindings but merely act as an extended form of guard, declaring a binding to be either valid or invalid. Accordingly, they are processed in step (1) of the evaluation of the propagation function.

Let $\mathcal{F}_n$ be the set of ESDDs invoked in the SDDP of node n. When computing $\mathcal{X}_\zeta^{(\mathsf{t})}$, we extend Equation 3.7 and additionally require that each $\xi'_\zeta \in \mathcal{X}_\zeta^{(\mathsf{t})}$ fulfills every ESDD $F \in \mathcal{F}_n$, i.e., there needs to be a candidate set in the result set generated by

the ESDD that has evolved from the rebound binding $\{\ell_{F_n}(\xi'_\zeta)\}$. For brevity, we use $F(\zeta)$ to denote the witness $(\lambda_F, \ell_{F_n}(\xi))$ with $\zeta = (n, \xi)$. We then have:

$$\mathcal{X}_\zeta^{(\mathsf{t})} := \{\xi'_\zeta \mid P_n[\xi'_\zeta] \leq G \ \wedge\ \forall F \in \mathcal{F}_n : \{F((n, \xi'_\zeta))\} \sqsubseteq [\![F]\!]^G \wedge \xi_\zeta \leq \xi'_\zeta \ \wedge\ \forall v : \xi'_\zeta(v) \neq \xi_\zeta(v) \Rightarrow v \in free(n, \xi_\zeta)\}. \tag{3.21}$$

Non-recursive ESDD Semantics. Non-recursive ESDDs, i.e. ESDD definitions not containing direct or indirect references to themselves, can efficiently be computed like regular SDDs. As the semantics of the ESDD F with λ node λ_F for a graph G and an initial binding ξ_P, we can then define

$$[\![F]\!]^G_{\xi_P} := [\![\lambda_F]\!]^G_{\{\{(\lambda_F, \ell_F(\xi_P))\}\}} \tag{3.22}$$

and use the generated result set in place of $[\![F]\!]^G$ in Equation 3.21.

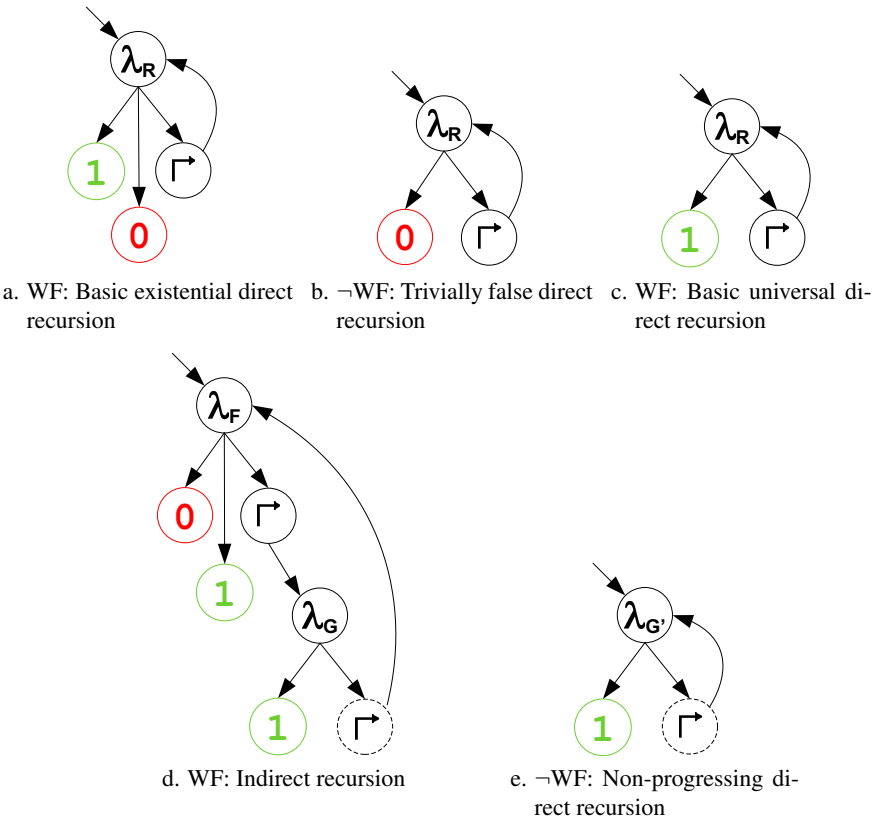
Well-formedness of recursive ESDDs. We require recursive ESDDs to be *well-formed*. A well-formed ESDD does not contain vacuous cycles, i.e. there always needs to exist a potential path to a termination condition from any node.

Whether a set of ESDDs is well-formed can be analyzed using the *invocation graph* of the ESDDs. The invocation graph is a reduced representation of the diagrams' structure that focuses on the aspects that are relevant for recursion. We are only interested in λ nodes, leaf nodes, nodes containing ESDD invocations, and the paths connecting them. Starting from the λ node, we connect it to all leaf nodes that are reachable without invoking another ESDD first. For each invocation, we add an invocation node and connect it with the λ node of the invoked ESDD. We also need to note whether the invocation arguments are all unmodified roles and parameters of the preceding λ node.

An ESDD is then well-formed if (a) it is *fulfillable*, i.e. a (1) leaf node is reachable from its λ node in the invocation graph, and (b) it is *progressing*, i.e. on every cycle through the invocation graph, there is at least one quantor affecting an element that is used as an argument for an ESDD invocation, or a parameter is modified from the previous invocation.

Figure 3.27a presents the invocation graph for our definition of reachable in Figure 3.11. From the λ node, we can reach the (1) node on the left branch if the termination condition holds. If the termination condition does not hold, we will have to evaluate the recursive invocation as the branches represent alternatives, hence the connection back to λ_R. However, the invoking node is quantified, and if we fail to bind some intermediate track, the right branch will evaluate to (0) without recursion.

Now imagine the same definition without the left branch. The recursion could still terminate once it reaches tracks without successors that could be bound to



a. WF: Basic existential direct recursion

b. ¬WF: Trivially false direct recursion

c. WF: Basic universal direct recursion

d. WF: Indirect recursion

e. ¬WF: Non-progressing direct recursion

Figure 3.27: Invocation graphs for recursive ESDDs

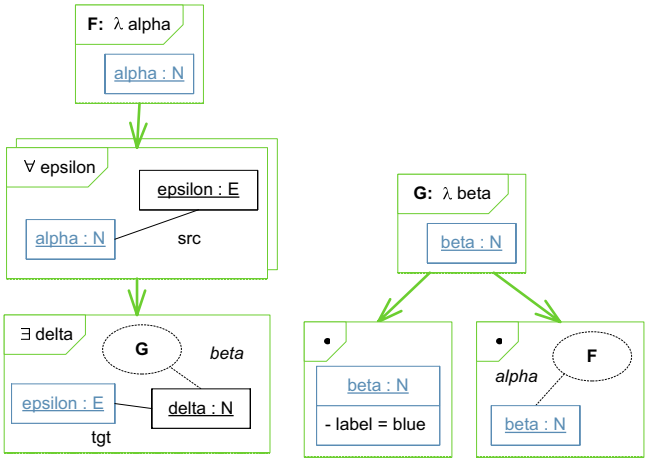
intermediate, but the property could never be fulfilled and would thus be trivially false. Figure 3.27b presents the corresponding invocation graph.

If the invoking node was universally quantified instead, a track without successors would fulfill the right branch, which results in the minimal invocation graph in Figure 3.27c.

In Figure 3.27d, we present the invocation graph for the example in Figure 3.28 below. F invokes G, and G invokes F. F can terminate in two ways: If there is no epsilon, evaluation reaches (1) while if there is no delta, it reaches (0). G can reach (1) without further recursion if beta has the required label.

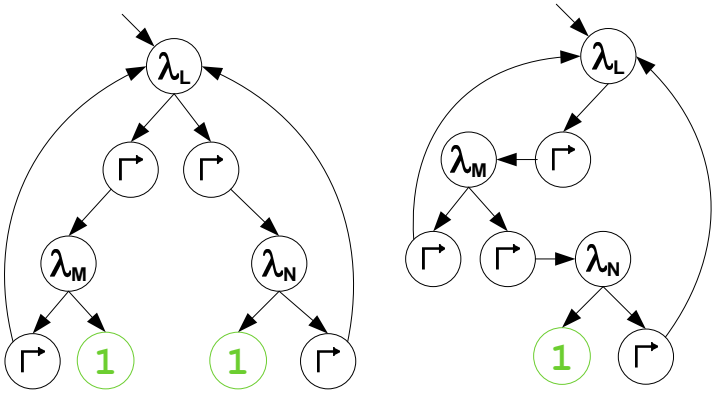
G does not contain any quantifiers, but passes beta straight on to F, which is indicated by the dashed border around the invoking node. If G invoked itself instead of F, there would be a non-progressing cycle as shown in Figure 3.27e.

Finally, we present synthetic examples where an ESDD L depends on both M and N, which in turn reference L. If the two invocations are alternatives, the invocation



a. F invokes G b. G invokes F

Figure 3.28: Example for indirect recursion



a. WF: Disjunctive recursions b. WF: Conjunctive recursions

Figure 3.29: Invocation graphs for recursive ESDDs

graph in Figure 3.29a results. As all nodes form a single cycle, reaching one of the (1) nodes is sufficient. If the the invocation of N follows on the then branch of M, the invocation of N is added in place of the (1) node in the subgraph representing M as in Figure 3.29b, thus ensuring that, for the same binding, first M and then N is fulfilled.

Recursive ESDD Semantics. In order to define the semantics of a well-formed recursively defined ESDD F, we need to compute a fixed point of F. If $\mathcal{F}_I$ is a set of interdependent ESDDs F_i that are connected in a single invocation graph, we need to compute their fixed points together.

The semantics $[\![F_i]\!]^G$ of an ESDD F_i should correspond to a result set containing all valid final candidate sets that can evolve from any initial binding that could be passed to F_i for a given graph G. In order to compute this result set, we extend F_i with an auxiliary existential node α_{F_i} quantifying all roles of the ESDD F_i, which is added before the λ node λ_{F_i} and thus becomes the new root node. The existential node will generate all possible combinations of bindings for the roles and pass them on to the λ node.

The unconstrained semantics of non-recursive ESDDs can then be computed directly as

$$[\![F_i]\!]^G := [\![\alpha_{F_i}]\!]^G_{\{\{(\alpha_{F_i},\tau)\}\}}. \tag{3.23}$$

However, this will not work for recursive ESDDs, as $[\![F_i]\!]^G$ is required in order to evaluate the propagation function $apply$ (see Equation 3.21).

We therefore introduce the fixed point operator $\mp$, which successively computes the semantics using approximations $[\![f_i^{(j)}]\!]^G$ of $[\![F_i]\!]^G$. Instead of relying on the – undefined – semantics $[\![F_i]\!]^G$, $\mp$ substitutes $[\![f_i^{(j)}]\!]^G$ for $[\![F_i]\!]^G$ when computing the extended bindings. Furthermore, as $[\![f_i^{(j)}]\!]^G$ is only an approximation of the final semantics, we cannot just check whether there is a candidate set in $[\![f_i^{(j)}]\!]^G$ that has evolved from a given role binding, but have to differentiate between *unsuccessful* and *undefined* invocations. We therefore do not use Equation 3.21, but the original Equation 3.7

$$\begin{aligned}\mathcal{X}'_\zeta := \{\xi'_\zeta \mid P_n[\xi'_\zeta] \leq G \wedge \\ \xi_\zeta \leq \xi'_\zeta \ \wedge\ \forall v : \xi'_\zeta(v) \neq \xi_\zeta(v) \Rightarrow v \in free(n, \xi_\zeta)\},\end{aligned} \tag{3.24}$$

and evaluate the constraints represented by ESDDs in a separate step. We compute the valid extended bindings for which all invocations are successful as

$$\begin{aligned}\mathcal{X}^{(\mathsf{t})}_\zeta := \{\xi^{(\mathsf{t})}_\zeta \mid \xi^{(\mathsf{t})}_\zeta \in \mathcal{X}'_\zeta \ \wedge\ \forall F \in \mathcal{F}_n : \\ (\exists \mathcal{C} \in [\![F]\!]^G : eval(\mathcal{C}) \ \wedge\ \{F((n, \xi^{(\mathsf{t})}_\zeta))\} \sqsubseteq \mathcal{C})\}\end{aligned} \tag{3.25}$$

and the indeterminate bindings that are not valid, but not definitely invalid because there is no invocation that is definitely unsuccessful as

$$\begin{aligned}\mathcal{X}^{(\perp)}_\zeta := \{\xi^{(\perp)}_\zeta \mid \xi^{(\perp)}_\zeta \in \mathcal{X}'_\zeta \setminus \mathcal{X}^{(\mathsf{t})}_\zeta \ \wedge\ \nexists F \in \mathcal{F}_n : \\ (\forall \mathcal{C} \in [\![F]\!]^G \mid \{F((n, \xi'_\zeta))\} \sqsubseteq \mathcal{C} : \neg eval(\mathcal{C}))\}.\end{aligned} \tag{3.26}$$

Consequently, we only follow the else branch if there are no valid or indeterminate bindings and have

$$\mathcal{X}^{(\mathsf{e})}_\zeta := \begin{cases} \emptyset & \mid (\mathcal{X}^{(\mathsf{t})}_\zeta \cup \mathcal{X}^{(\perp)}_\zeta) \neq \emptyset \\ \{\xi_\zeta\} & \mid (\mathcal{X}^{(\mathsf{t})}_\zeta \cup \mathcal{X}^{(\perp)}_\zeta) = \emptyset. \end{cases} \tag{3.27}$$

If one of the ESDDs in $\mathcal{F}_n$ is recursively defined, we ignore the cardinalities and use a modified version of Equation 3.9, defining the set of generated witnesses as

$$\begin{aligned}\mathcal{W}_\zeta^+ :=& \{(n', \xi') \mid \xi' \in \mathcal{X}_\zeta^{(\mathsf{t})} \wedge n' \in \mathsf{then}(n)\} \cup \\ & \{(n', \xi') \mid \xi' \in \mathcal{X}_\zeta^{(\mathsf{e})} \wedge n' \in \mathsf{else}(n)\} \cup \\ & \{(\bot, \xi') \mid \xi' \in \mathcal{X}_\zeta^{(\bot)}\}. \end{aligned} \tag{3.28}$$

By adding the permanently intermediate witnesses $(\bot, \xi')$, we prevent premature negative results — they basically indicate that ξ' might or might not turn out to be a valid binding.

Starting with the initial result sets $[\![f_i^{(0)}]\!]^G$, we then apply $\mathbb{\dashv}$ for all ESDDs F_i, in turn, to compute

$$[\![f^{(j+1)}]\!]^G := \mathbb{\dashv}([\![f_i^{(j)}]\!]^G), \tag{3.29}$$

where the actual fixed point operator is defined as

$$\mathbb{\dashv}([\![f_i^{(j)}]\!]^G) := [\![\alpha_{F_i}]\!]^G_{\{\{(\alpha_{F_i}, \tau)\}\}} \mid \forall F_i \in \mathcal{F}_I : [\![F_i]\!]^G := [\![f_i^{(j)}]\!]^G. \tag{3.30}$$

$\mathbb{\dashv}$ is applied until $\mathbb{\dashv}([\![f_i^{(j)}]\!]^G) = [\![f_i^{(j)}]\!]^G$ for all of the involved ESDDs F_i. We have then computed a fixed point $[\![f_i]\!]^G$ which allows us to define the semantics of the ESDDs F_i as

$$[\![F_i]\!]^G := \{\mathcal{C} \mid \mathcal{C} \in [\![f_i]\!]^G \wedge \mathit{eval}(\mathcal{C})\}. \tag{3.31}$$

We define two versions of $\mathbb{\dashv}$, the least fixed point operator $\mathbb{\dashv}_\mu$ and the greatest fixed point operator $\mathbb{\dashv}_\nu$. The standard semantics of SDDs are defined by means of $\mathbb{\dashv}_\mu$, i.e. using least fixed points.

The least fixed point operator $\mathbb{\dashv}_\mu$ starts with empty initial result sets:

$$\forall F_i \in \mathcal{F}_I : [\![f_i^{(0)}]\!]^G := \emptyset. \tag{3.32}$$

The result set is then successively extended with additional valid candidate sets. $[\![f_i^{(1)}]\!]^G$ contains those candidate sets that succeed without recursive invocations, and $[\![f_i^{(j)}]\!]^G$ contains those candidate sets that succeed with a recursion depth of at most $j - 1$.

All involved sets (especially the result sets $[\![f_i^{(j)}]\!]^G$) are finite. The intermediate witnesses $(\bot, \xi')$ make sure that $[\![F_i]\!]^G$ grows monotonically, i.e. a candidate set that has been added to $[\![F_i]\!]^G$ is never eliminated in subsequent iterations. $\mathbb{\dashv}_\mu$ can thus only be applied to $[\![f_i^{(j)}]\!]^G$ finitely often before a fixed point is reached.

The greatest fixed point operator $\mathbb{\dashv}_\nu$ starts by assuming that all ESDD invocations are successful. This can be realized by using the set of all possible candidate sets

as the initial result set:

$$\forall F_i \in \mathcal{F}_I : [\![f_i^{(0)}]\!]^G := \{\mathcal{C} \mid \mathcal{C} \in \wp(\mathcal{Z}_{F_i})\}. \tag{3.33}$$

Successive applications will then eliminate those candidate sets that contain invalid witnesses.

As the fixed point operator $\mathbb{F}_\nu$ only changes $[\![F_i]\!]^G$ by eliminating, never adding, candidate sets, it can again only be applied to $[\![f_i^{(j)}]\!]^G$ finitely often before a fixed point is reached. We can therefore guarantee that the fixed points exist and that their computation terminates for both operators.

The effective difference between the two operators lies in their treatment of cyclic dependencies between recursive invocations. $\mathbb{F}_\mu$ evaluates sets of mutually dependent invocations to $false$, while $\mathbb{F}_\nu$ evaluates them to $true$. An example for such a cycle would be generated by reachable, applied to a circle of tracks that is not connected to the destination tracks. In this case, $\mathbb{F}_\mu$ provides the intuitively correct result ($false$). When the recursion is existentially quantified, cycles only occur if G is not acyclic and the evaluation cannot reach a termination condition at all. When the recursion is universally quantified, cycles may occur whenever G is not acyclic. The universally quantified property 'a node is valid if it is certified or if all its neighbors are valid' of a two-dimensional grid leads to cyclic dependencies between uncertified nodes. If no node is certified, $\mathbb{F}_\mu$ computes that no node is valid, while $\mathbb{F}_\nu$ yields that all nodes are valid. As we have so far encountered no actual practical examples that required greatest fixed point semantics, there currently is no way to specify that $\mathbb{F}_\nu$ should be used in place of $\mathbb{F}_\mu$ in the syntax.

For recursively defined parametrized ESDDs, which can accept and manipulate arbitrary parameters, we can guarantee the existence of a fixed point and termination based on the above definitions if we restrict the domains of the parameters to a finite set represented by value nodes in G and treat the parameters as roles that are bound to the corresponding value node. While the restriction to a finite domain holds on any physical machine, the size of the potential result set would prohibit explicitly computing the fixed point. Unsurprisingly, parametrized ESDDs are thus in the same situation as regular recursive function definitions over infinite domains and subject to conventional recursion theory. In particular, we require that F be *monotonic* as a necessary condition for the existence of a fixed point, following the argument in [23].

3.3.5 Transformation Semantics

The semantics of transformation nodes are very closely related to standard Story Pattern semantics as they do not contain any quantification or other advanced features, but merely apply are graph rule to a single binding.

Selection. In the presence of transformation nodes, we randomly pick a *final* candidate set $\mathcal{C} \in [\![S]\!]^G$ after the SDD S has been successfully matched. For each witness $\zeta = (n, \xi) \in \mathcal{C}$, we then have $n \in \mathsf{true}_S$, i.e. the witness is either at a (1) leaf node or at a transformation node.

Application. If n is a transformation node, we interpret its SDDP as a graph transformation rule $[L] \rightarrow_r [R]$, where bound and destroyed elements make up the LHS and bound and created elements make up the RHS as defined in Section 2.2.6.

The rule is then applied using the standard semantics defined in Definition 13, using the graph morphism from P_n to $P_n[\xi]$ as determined by the binding ξ as the match m. However, in order to avoid problems with destroyed elements that are part of several bindings in the selected candidate set, we split the rule application into two parts: We first create the elements in $o(R \setminus L)$ for all witnesses in $\mathcal{C}$ that are at transformation nodes in a first pass, and then delete the elements in $o(L \setminus R)$ in a second pass.

Conditional transformations are actually quite simple at the formal level, as they simply expand into an existential node and a regular transformation node. Note that, however, this expansion may be somewhat more complex than it may seem at first glance. If the conditional transformation is supposed to *ensure* the presence of three elements, one of which is missing, we do not want the transformation to create three elements, but reuse the two existing elements. We therefore need to verify the presence of each element individually and only create the missing ones. If the elements are unconnected, we can simply perform the individual conditional transformations in a *scoped AND node*, but if they are connected, we need to expand the different cases as alternatives, which may result in a large number of nodes.

3.3.6 Expressiveness

First-order predicate logic formulas φ with p ranging over a finite set of predicates $\mathcal{P}$, sets X, and elements $x \in X$, are defined as

$$\varphi ::= p(x) \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \exists x \in X : \varphi \mid \forall x \in X : \varphi. \tag{3.34}$$

Predicate logic for graphs. Based on the definitions in Sections 2.2.1, we can encode a typed graph by means of predicates $\langle typename \rangle(n)$, $\langle typename \rangle(n, e, n')$, where n and n' are graph nodes, e is a graph edge, and $\langle typename \rangle$ represents some typename from the type graph. For a given graph G, these predicates can be derived based on the labeling, source and target functions.

A predicate logic for graphs over a given graph G can then be derived by using the predicates encoding G as $\mathcal{P}$ and the nodes and edges of G as the domain of the predicates and the quantors. Based on Section 3.3.1, the set $\mathcal{X}_{\mathcal{P}}[G]$ of possible bindings becomes the set X, and the elements x are bindings ξ.

SDDs and first-order predicate logic. We can now compare the expressiveness of first-order predicate logic for graphs and SDDs.

Theorem 3.1 *Story Decision Diagrams over a given graph G are at least as expressive as first-order predicate logic for graphs over G.*

Proof. We prove the theorem by showing that for every first-order predicate logic formula over G, there is an equivalent SDD:

- $p(x)$: If p is a predicate encoding a node and x accordingly is a binding ξ for a node variable, $p(x)$ can be encoded as $P|_\xi \precsim G$ where P is a graph pattern containing a single node with type p. If p is a predicate encoding an edge and x accordingly is a binding ξ for an edge and two node variables, $p(x)$ can be encoded as $P|_\xi \precsim G$ where P is a graph pattern containing two nodes connected by an edge with type p. The formula can thus be expressed as a guard node containing P as its SDDP.
- $\neg\varphi$: If φ is encoded by S, $\neg\varphi$ is encoded by S's negation $\overline{S}$.
- $\varphi \wedge \varphi$: If two terms φ_1 and φ_2 are encoded by S_1 and S_2, $\varphi_1 \wedge \varphi_2$ is encoded by a scoped and node containing S_1 and S_2, or by two scoped nodes (S_1) then (S_2) then ... (or two regular nodes if the terms have no common variables).
- $\varphi \vee \varphi$: If two terms φ_1 and φ_2 are encoded by S_1 and S_2, $\varphi_1 \vee \varphi_2$ can be encoded using two scoped nodes as (S_1) then ... else ((S_2) then ...), or using two alternative then connectors issuing from a scoped node ((1)) then ((S_1) then ...) $\vee$ then ((S_2) then ...).
- $\exists x \in X : \varphi$: If φ is encoded by S, $\exists x \in X : \varphi$ can be encoded using an existential node n containing only the type constraints for x as n then S.
- $\forall x \in X : \varphi$: If φ is encoded by S, $\forall x \in X : \varphi$ can be encoded using a universal node n containing only the type constraints for x as n then S. □

These encodings show that SDDs, though obviously less compact on paper, also are as succinct as first-order predicate logic.

SDDs and the predicate μ-calculus. The predicate μ-calculus extends predicate logic with variables V, the least fixed point operator μ, and the greatest fixed point operator ν:

$$\varphi ::= p(V) \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid V \mid \mu V(\varphi) \mid \nu V(\varphi). \tag{3.35}$$

As SDDs provide recursion by means of ESDDs, formulas of the predicate μ-calculus for graphs can be written as SDDs:

Theorem 3.2 *Story Decision Diagrams over a given graph G are at least as expressive as the predicate μ-calculus for graphs over G.*

Proof. As we have already shown that any expression of first-order predicate logic can be written as an SDD, we merely need to focus on variables and the μ-operator:

- V : If V is a variable, any expression containing V can be written as an ESDD with role V.
- $\mu V(\varphi)$: If φ is a term containing V and φ is encoded by the ESDD F defining role V, $\mu V(\varphi)$ is equivalent to $[\![F]\!]^G$ using the least fixed point operator $\mathbb{F}_\mu$.
- $\nu V(\varphi)$: If φ is a term containing V and φ is encoded by the ESDD F defining role V, $\nu V(\varphi)$ is equivalent to $[\![F]\!]^G$ using the greatest fixed point operator $\mathbb{F}_\nu$. □

3.4 Property Detectors

SDDs are not merely a tool for communicating and reasoning about structural, but can be verified at run time by running a *property detector* monitoring the specified properties in parallel with the system.

There are two fundamentely different ways to implement such a property detector open to us. The first possibility is to start from the formal graph-based semantics, translate the property detector into a GTS and execute it inside a graph model checker. The second, more efficient possibility is to start from the operational, control-flow-based semantics and ultimately implement the detector as a Java or C++ program.

When translating SDDs into either format, we have two further options: If we need to determine all valid matches of the SDD, we can directly implement the formal semantics based on sets of candidate sets. On the other hand, if we are only interested in determining whether an instance situation satisfies an SDD or not, we can build on the informal operational semantics that will, in most cases, result in a much more efficient evaluation because we will focus on computing a single candidate set for the SDD, typically disregarding a large number of equally valid alternatives.

3.4.1 GROOVE

Implementing the desired property detector using GTS rules is more complex than the equivalent code generation, but at the same time quite quite intriguing, as it allows us to actually model check a system w.r.t. properties specified as SDDs. At this point, however, our main goal was to demonstrate the feasibility of evaluating an SDD using GROOVE.

For our experiments, we therefore used the operational semantics in order to reduce complexity and keep the evaluation efficient. The exhaustive search employed in the definition of the formal semantics is not only much less efficient for complex graphs, but also more complex. As tool support for SDDs is not yet available, we had to manually encode the SDDs as graph rules using the GROOVE editor, adding a further argument for the simpler alternative. We thus use a depth-first search that stops as soon as a sufficient match has been identified.

Approach

A single SDD is transformed into many small graph rules (e.g., the *supervised* SDD resulted in 23 rules) following a repetitive pattern that can 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. Markers thus play the role of witnesses in the formal semantics. 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 the matching bindings. 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 free variables bound by itself; the complete binding is thus defined by the path from the marker to the root element. Figure 3.30 shows a snapshot of the matching process for the property from Figure 2.4 – 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 Figure 3.31 for the shuttle on the right.

Rule priorities are very important for the correct execution.

1. Cleanup has the highest priority. It is triggered by a result marker at the root element.
2. 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.

3. 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 enforce 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. Figure 3.31). (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.

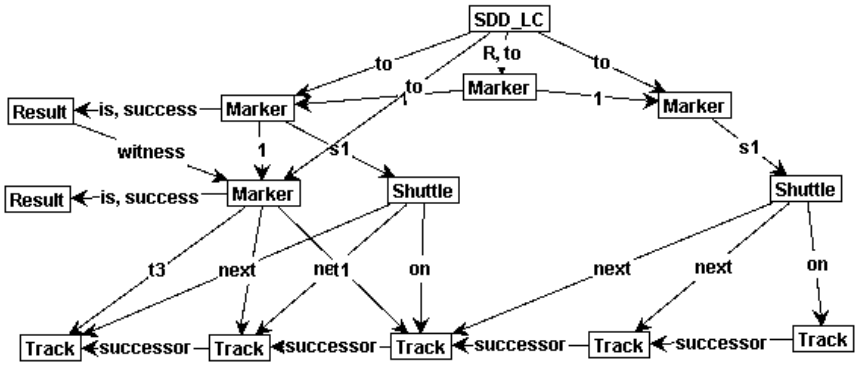


Figure 3.30: *Located*: Markers while processing

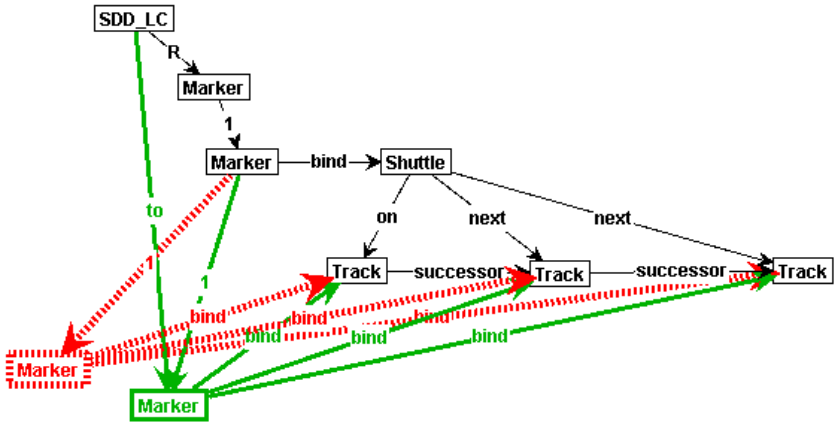


Figure 3.31: *Located*: Graph rule encoding the implication

4. Nodes containing ESDDs have a fourth rule with the highest priority (3) that triggers the ESDD 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.

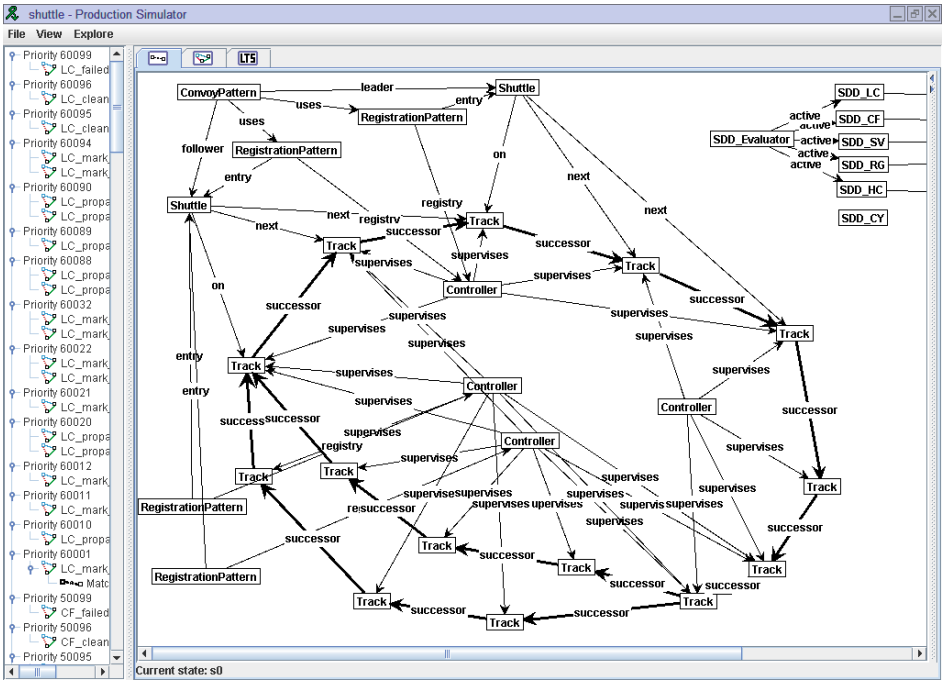


Figure 3.32: The test scenario in GROOVE

When all rules were translated, we created a set of correct and incorrect instance graphs (see Figure 3.32) 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.

Detailed Example

As an example, we present the patterns that are used to encode the requirement that for any three consecutive Tracks ($\forall$), there must be a Controller ($\exists$) supervising them all (Figure 3.3). We discuss what is happening in detail, presenting each rule. The letters at the beginning of each section indicate in which situation the described rule is applied, i.e. where the control flow currently is, to facilitate navigation. The letters (S) and (F) indicate that the active node has been marked with success or failure.

The SDD node SDD-SV represents the SDD that is to be evaluated. A trigger node is recreated by a lowest priority rule after all SDDs have finished evaluating.

SDD. The first rule mark SDD $\rightarrow$ R (Figure 3.33) deletes the trigger and creates a

Figure 3.33: mark SDD → R:1

marker, the root marker (R). A to link is created between every marker and SDD-SV, which is used for cleaning up the markers later. This rule is structurally identical for all SDDs. Its relative priority is 1, but no other rule is enabled at this time.

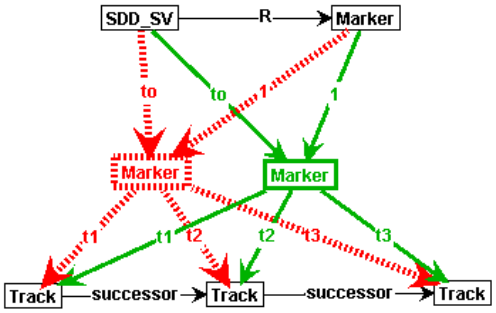


Figure 3.34: mark R → R1:12

R. Now, one of the few truly specific rules is applied: mark R → R1 (Figure 3.34). It encodes the SDDP from the universally quantified root node of the SDD. It is triggered by the R marker. If the required pattern (here the tracks t1, t2 and t3) is found and no 1 marker exists that already binds the instances in question, a 1 marker is created and connected to the bound objects with links bearing the names of the variables. Unsurprisingly, only the rules of the type mark R* → R*1, i.e. adding a 1 marker indicating a successful match, are truly structurally differing from SDD to SDD. Its priority is 12 - the first digit indicates the depth, i.e. the distance of the node from the root element, the second digit is the local priority, which is 2 for a positive marking rule, as explained above.

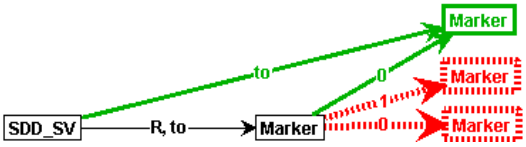


Figure 3.35: mark R → R0:11

R. Rule mark R → R0 (Figure 3.35) encodes the failure to match of the SDD's root node. It is triggered by the R marker as well. It only matches if there is no 1 marker (indicating a match) and no 0 marker (rule has already been applied). Its priority

is 11 (group priority 10, local priorty 1 for a negative marking rule), i.e. it only matches after mark R → R1 has been processed. This rule is structurally identical for all SDDs.

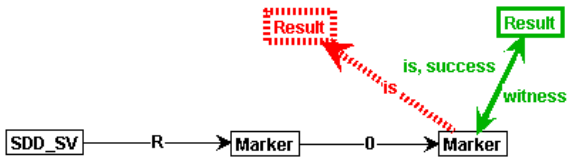


Figure 3.36: mark R0 → R0(S):22

R0. Let us first consider the simpler case that the first node does not match, i.e. there are no three connected tracks in the system. Remember that – as the candidate set is thus empty, the node is universally quantified, and any universally quantified statement is true for the empty set – there is an implicit else edge from every universally quantified node to the true leaf node. Rule mark R0 → R0(S) (Figure 3.36) encodes this. It is triggered by the R-0 marker path, i.e. the control flow is at the leaf node. The rule creates a result node which has an is relation which is used for cleanup (similar to the markers' to relation) and a success relation for indicating that we have reached a true leaf node. It also points to the marker that is a witness for the success, i.e. has the successful bindings – which is rather pointless in this case, as nothing was bound. The result marker is then attached to the R0 marker representing the leaf node. The priority is 22, 20 for the depth, 2 as the local priority. This rule is structurally identical for all SDDs.

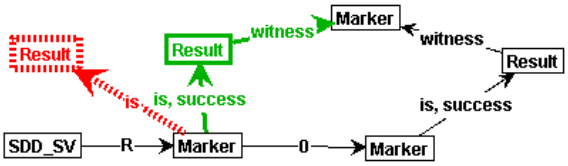


Figure 3.37: propagate R0(S) → R(S):10

R0(S). Now, we need to propagate this positive result. Rule propagate R0(S) → R(S) (Figure 3.37) copies the result node down from the leaf node at R-0 to R, the universally quantified root node, provided that the target node is not already marked with a result value. Its priority is 10, 10 for the depth, 0 as the local priority for propagation rules. This rule is structurally identical for all SDDs.

R(S). Rule propagate R(S) → SDD(S) (Figure 3.38) then copies the result node down to SDD-SV. This rule has a very high priority of $p_{max} - depth$ with $depth = 0$. In our manually generated rules, we used $p_{max} = 990$, which limits SDDs to a generous depth of 99 nodes – a limitation which can easily be lifted by dynamically

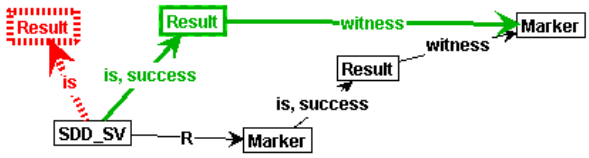


Figure 3.38: propagate R(S) → SDD(S):990

using a sufficiently high priority in the automated implementation. This rule is structurally identical for all SDDs.

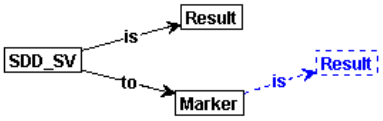


Figure 3.39: clean results:996

SDD(S). The result value has now reached the node representing the SDD; we can thus look at the result and study the witness if we are interested in analyzing the responsible valid or invalid binding. Afterwards, cleanup starts. Rule clean results (Figure 3.39) deletes all result nodes except the one directly connected to the SDD node by iterating over the markers. Its priority is $p_{max} + 6$, i.e. 996. This rule is structurally identical for all SDDs.

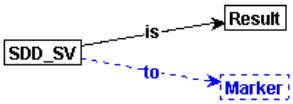


Figure 3.40: clean markers:995

SDD(S). We can now also remove the markers. Rule clean markers (Figure 3.40) deletes all markers using the to link. Its priority is $p_{max} + 5$, i.e. 995, because we need the markers to delete the result nodes. This rule is structurally identical for all SDDs.

Figure 3.41: mark success:994

SDD(S). Finally, the mark success rule (Figure 3.41) simply deletes the last result node if it indicates success. As the invariant held, we are no longer interested. The priority is $p_{max} + 4$, i.e. 994. This rule is structurally identical for all SDDs.

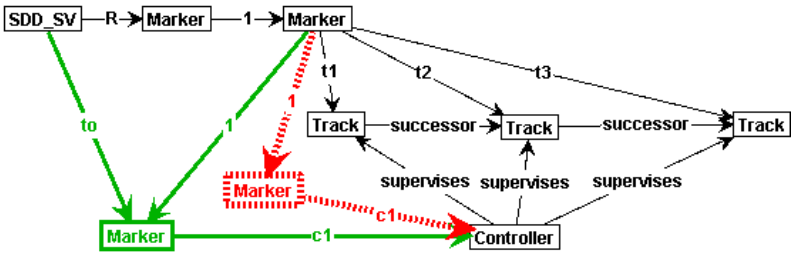


Figure 3.42: mark R1 → R11:22

R1. Let us now return to the more interesting case where mark R → R1 matches. The control flow is now at R1, i.e. the second SDD node. Rule mark R1 → R11 Figure 3.42 is the SDD-specific rule that tries to match the SDDP from the SDD's existentially quantified second node, i.e. it matches if there is a common controller supervising the three tracks. The controller (c1) is bound to the new marker; the complete binding is thus the combination of R1, the tracks, and R11, the controller. The rule's priority is 22, 20 for depth 2 and 2 as the typical local priority for a positive marking rule.

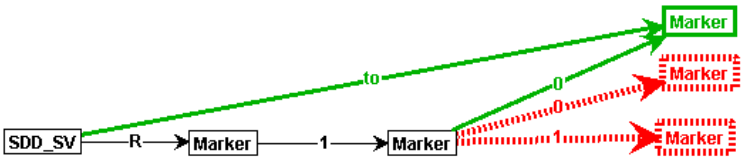


Figure 3.43: mark R1 → R10:21

R1. Again, if the SDDP does not match, i.e. mark R1 → R11 is not applied, the negative rule mark R1 → R10 (Figure 3.43) matches. The priority is 21, 20 for the depth, 1 as the typical local priority for a negative marking rule.

Figure 3.44: mark R10 → R10(F):32

R10. Let us again first look at the else-case. mark R10 → R10(F) (Figure 3.44) encodes that the path R-1-0 leads to a false leaf node and thus marks the marker with a result node just like rule mark R0 → R0(S) above, only this time with a failure instead of a success link. As a positve marking rule, its priority 2 at a depth 3, i.e. 32.

Figure 3.45: propagate R10(F) → R1(F):20

R10(F). The failure is then propagated down by rule propagate R10(F) → R1(F) (Figure 3.45). Its priority is 20.

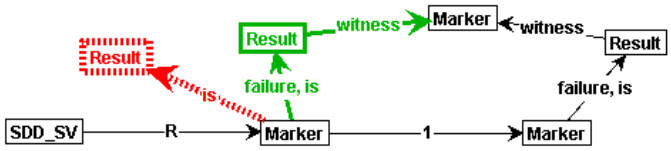


Figure 3.46: propagate R1(F) → R(F):989

R1(F). We now have a set of tracks for which no controller was found. High priority rule propagate R1(F) → R(F) (Figure 3.46) immediately propagates this down, marking the universally quantified root node as failed. Its priority is again $p_{max} - depth$, i.e. 989.

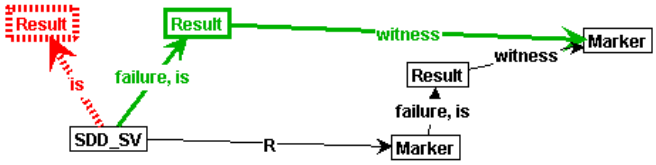


Figure 3.47: propagate R(F) → SDD(F):990

R(F). What ensues is similar to what we have already seen. propagate R(F) → SDD(F) (Figure 3.47) copies the failure result down to the SDD-SV node. Its priority is again $p_{max} - depth$, i.e. 990. Then, clean results at 996 and clean markers at 995 are applied.

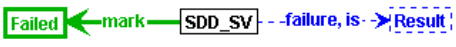


Figure 3.48: mark failure:994

SDD(F). Like mark success, mark failure (Figure 3.48) deletes the last result node, but replaces it with a failed marker. The priority is $p_{max} + 4$, i.e. 994.

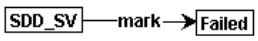


Figure 3.49: violation:999

SDD failed. violation, the highest priority rule of the SDD at a relative priority of 999, i.e. $p_{max} + 9$, is simply an instance graph where the SDD-SV is marked with the failed marker (Figure 3.49). It is specified as a forbidden state, i.e. the model checker will check for it and thus, implicitly, for a violation of the invariant specified by the SDD.

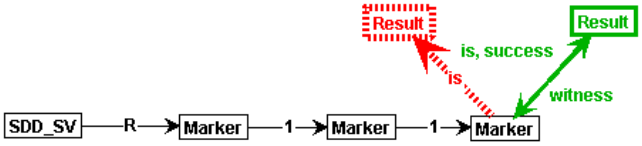


Figure 3.50: mark R11 → R11(S):32

R11. We now jump back one last time. Both SDDPs have just matched and we have reached the true leaf node at R-1-1. Rule mark R11 → R11(S) (Figure 3.50) thus marks the R11 marker with a success result node. The priority is 32 (depth 3, positive marking rule).

Figure 3.51: propagate R11(S) → R1(S):988

Rule propagate R11(S) → R1(S) (Figure 3.51) then propagates this success down with a high priority of $p_{max} - depth$, i.e. 988.

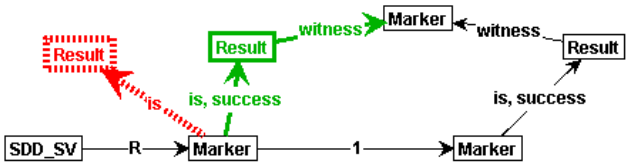


Figure 3.52: propagate R1(S) → R(S):10

Now things get more interesting again! Rule propagate R1(S) → R(S) (Figure 3.52) is ready to propagate the success down to the root node. However, its priority 10 is comparatively low, most notably lower than the priority of rule mark R → R1,

which is 12. This means that the success will not be propagated down as long as there are unmarked, i.e. unchecked, groups of tracks. The whole recursion will be repeated until a controller has been found for all matching tracks – that is, unless this check fails for some track group and propagate R1(F) → R(F) with priority 989 fires first. Only once all required bindings have returned a success result will this rule be applied.

Figure 3.53: propagate R11(F) → R1(F):20

Figure 3.54: propagate R10(S) → R1(S):988

Figure 3.55: propagate R0(F) → R(F):989

Finally, for completeness and symmetry, we also specify rule propagate R11(F) → R1(F) (Figure 3.53) at a priority of 20, rule propagate R10(S) → R1(S) (Figure 3.54) at a priority of 988 and rule propagate R0(F) → R(F) (Figure 3.55) at a priority of 989. However, they are never applied, as they try to propagate failure from true and success from false leaf nodes, which can never occur there.

Algorithm

Looking at the complete list of rules (including the above superfluous rules), the underlying structure becomes visible:

```
999. violation
```

```
996. clean results
995. clean markers
994. mark success
994. mark failure
990. propagate R(S) -> SDD(S)
990. propagate R(F) -> SDD(F)
989. propagate R1(F) -> R(F)
989. propagate R0(F) -> R(F)
988. propagate R11(S) -> R1(S)
988. propagate R10(S) -> R1(S)
032. mark R11 -> R11(S)
032. mark R10 -> R10(F)
022. mark R1 -> R11
022. mark R0 -> R0(S)
021. mark R1 -> R10
020. propagate R11(F) -> R1(F)
020. propagate R10(F) -> R1(F)
012. mark R -> R1
011. mark R -> R0
010. propagate R1(S) -> R(S)
010. propagate R0(S) -> R(S)
001. mark SDD -> R
```

- The initialization rule mark SDD → R at priority 1 is structurally identical for all SDDs.
- The termination rules with priorities $\geq p_{max}$, i.e. 990, are also identical for all SDDs, except for the exact name of the SDD node (SDD-SV). They clean up the auxiliary nodes (result and markers) and mark the SDD as failed if the invariant is violated.
- For each leaf node at depth d and path R*, where * represents a sequence of $d-1$ 1s or 0s encoding the node's position, there is one marker rule: mark R* → R*(S) for true leaf nodes, mark R* → R*(F) for false leaf nodes. Their priority is $10d+2$.
- For each existential or universal node at depth d, there are two rules: mark R* → R*1 at priority $10d+2$ and mark R* → R*0 at priority $10d+1$. All rules follow the same schema, with the former including the specific elements that encode the SDDP in question, and the latter just varying dependening on the node's position.
- For each existential node at depth d, there are four propagation rules for retrieving the results from its child nodes: rules propagate R*1(S) → R*(S) and propagate R*0(S) → R*(S) at priority $p_{max}-d$ for propagating success,

and rules propagate R*1(F) → R*(F) and propagate R*0(F) → R*(F) at priority $10d$ for propagating failure. All these rules follow the exact same schema. As one valid alternative binding is sufficient for an existentially quantified node, success is propagated with high priority, while failure is only propagated if backtracking has failed and there are no more alternatives to try.

- For each universal node at depth d, there are four propagation rules for retrieving the results from its child nodes: rules propagate R*1(F) → R*(F) and propagate R*0(F) → R*(F) at priority $p_{max} - d$ for propagating failure, and rules propagate R*1(S) → R*(S) and propagate R*0(S) → R*(S) at priority $10d$ for propagating success. These rules are identical to the rules used for existentially quantified nodes, only the priorities are inverted. As a universally quantified node requires a valid binding for each required binding, failure is propagated with high priority, while success is only propagated if there are no more required bindings to validate.

ESDDs

Encoding an ESDD is basically no different from encoding any other SDD. The only notable difference is that there is no rule mark SDD → R, and no need for mark success/failure rules. Also, as the root node is a lambda node, it always matches and already provides some bindings. Rule mark R → R1 (Figure 3.56) illustrates this for the *Convoy* property.

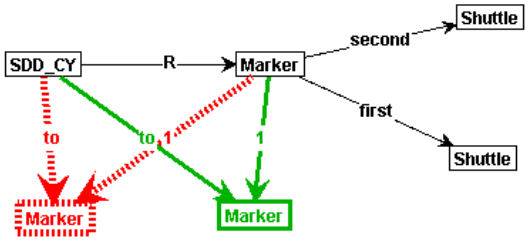


Figure 3.56: mark R → R1:12

Invoking the ESDD requires one additional rule in the invoking SDD. Also, the base priority of the ESDD needs to be higher than the base priority of all SDDs that include it. We provide an example from the *ConvoyMode* SDD:

The rule mark R10 → R10L (Figure 3.57) activates the ESDD. Its local priority is higher than for positive matching rules; the priority is therefore $10d + 3$, i.e. 33. If no 1 or 0 marker is present, and if there is no pattern link to the ESDD node, the rule is applied. It creates a root marker (R) for the ESDD, thus activating its evaluation, and the pattern and lambda links. The root marker is also bound to the two previously bound shuttles s1 and s2, which become first and second.

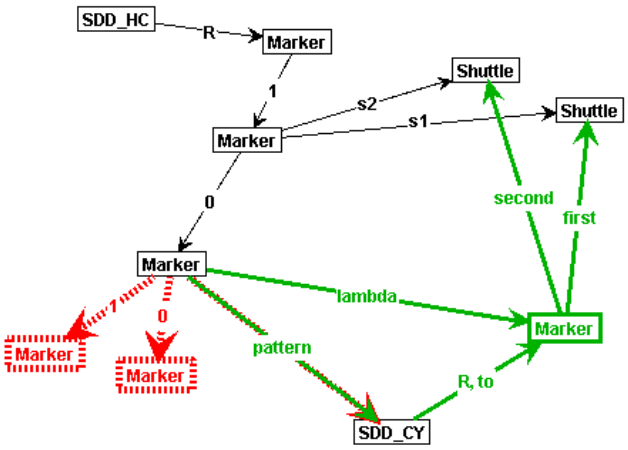


Figure 3.57: mark R10 → R10L:33

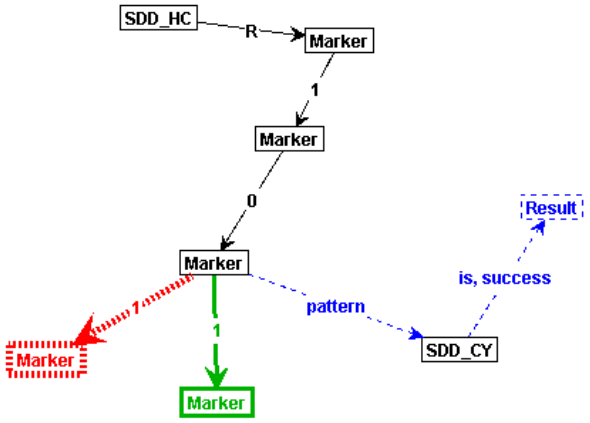


Figure 3.58: mark R10 → R101:32

Rule mark R10 → R101 (Figure 3.58) has priority 32 and is thus evaluated after the ESDD. It adds the additional constraint that the pattern needs to have the result success, and deletes the auxiliary constructs. In this case, the rule only checks for the success of the ESDD, but it would be possible for the rule to create its own additional bindings as well, as any other positive matching rule.

Rule mark R10 → R100 (Figure 3.59) is applied if mark R10 → R101 did not succeed. The pattern link referencing the ESDD is removed, so is the result node. The result of the ESDD is not necessarily negative, as the positive matching rule might have failed to create some other bindings unrelated to the (successful) ESDD (though in this example, the result *would* always be negative as there are no such bindings).

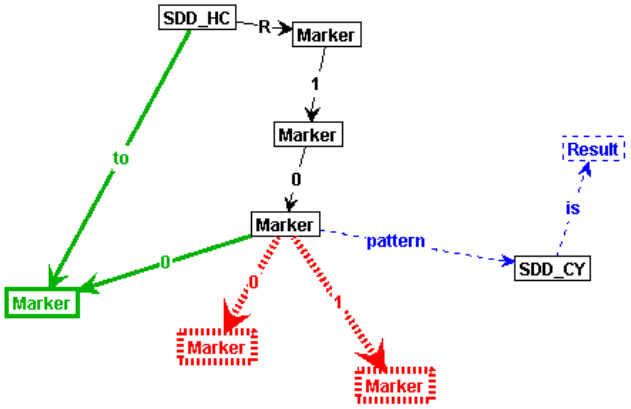


Figure 3.59: mark R10 → R100:31

Multiple alternative paths

Alternative paths, i.e. multiple then or else edges for the same node, require special attention in some cases. When all alternative nodes are existential, it is sufficient to simply add the rules representing the different nodes with the same priority – as it is sufficient that one of them succeeds, the usual mechanism (propagating the first success) is still working.

If one or all of the alternatives are universally quantified, however, the usual mechanism of propagating the first failure does not work, however. Therefore, we present a slight extension of the basic approach, using a universally quantified root node with two universally quantified then nodes as the example:

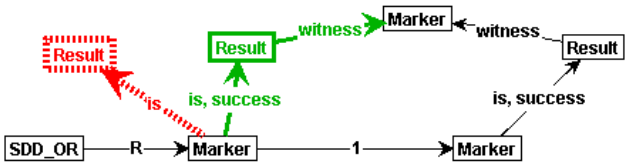


Figure 3.60: propagate R1(S) → R(S):989

Rule propagate R1(S) → R(S) (Figure 3.60) is unchanged.

Rule mark R1 → R11a (Figure 3.61) matches the SDDP of alternative a (some condition about 'all shuttles s2') normally, but only creates and reacts to markers with label 1 and a.

Rule mark R1 → R11b (Figure 3.62) does the same for alternative b (some condition about 'all predecessor tracks of t2') normally, but only creates and reacts to markers with label 1 and b.

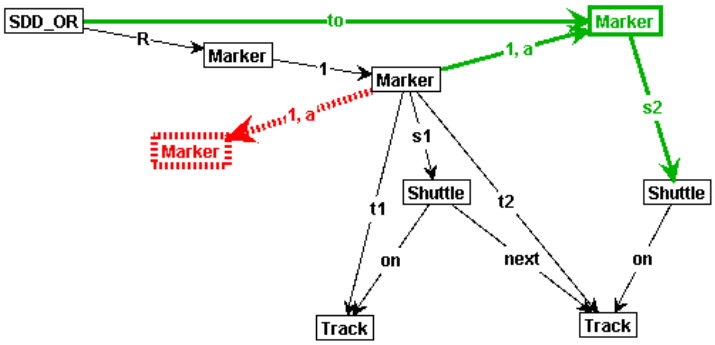


Figure 3.61: mark R1 → R11a:22

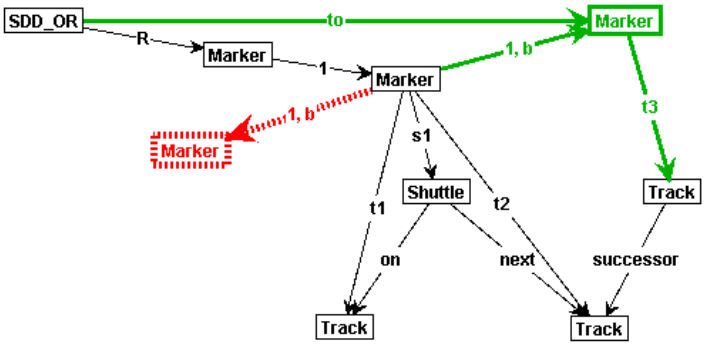


Figure 3.62: mark R1 → R11b:22

Figure 3.63: propagate R11a(S) → R1(S):20

Low priority rule propagate R11a(S) → R1(S) (Figure 3.63) propagates down success for alternative a.

Figure 3.64: propagate R11b(S) → R1(S):20

Low priority rule propagate R11b(S) → R1(S) (Figure 3.64) propagates down success for alternative b.

Figure 3.65: propagate R11a(F) → R1(F):988

High priority rule propagate R11a(F) → R1(F) (Figure 3.65) propagates down failure for alternative a.

Figure 3.66: propagate R11b(F) → R1(F):988

High priority rule propagate R11b(F) → R1(F) (Figure 3.66) propagates down failure for alternative b.

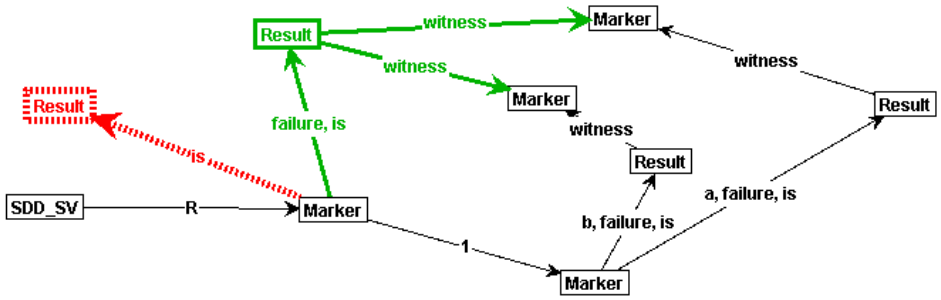


Figure 3.67: propagate R1(F) → R(F):989

Finally, the most important rule: propagate R1(F) → R(F) (Figure 3.67) only marks a binding from the root node as failed if both path a and path b have failed.

3.4.2 Code Generation

We now look at possibilities to derive a program that acts as a property detector from an SDD.

One possibility for translating an SDD into code would be to create a Story Diagram (i.e. an Activity Diagram with Story Patterns), add a foreach loop for every quantified variable, add the Story Patterns from the SDD, and connect the activities according to the connectors. The loops then iterate over all possible bindings. However, additional auxiliary variables would be required to log the success of each iteration, with existential nodes returning at the first success, universal nodes aborting at the first failure. All the actual graph matching is provided by the existing code generation for Story Patterns, even though the increased expressive power, e.g. w.r.t. negation, would make it necessary to split some of the SDDPs into several connected patterns.

We prefer the direct translation into source code (C++ and Java) as the cleaner solution. Again, we present manually implemented prototypes. In the future, the detectors will automatically be generated using Fujaba.

Basic structure. Using the above example, we present the resulting program for checking the SDD. For run-time monitoring, it is sufficient to check whether the pattern matches at all, not where. The monitoring version below thus signals success as soon as it encounters the first valid alternative binding:

Listing 1: Supervised (Java), runtime monitor

```
1  // SDD root
2  boolean isSupervised()
3  {
4    // Evaluate SDD
5    boolean result = isSupervisedR();
6    // Does the pattern match?
7    return result;
8  }
9
10 // Universally quantified node
11 boolean isSupervisedR()
12 {
13   // Default is true, AND join the results for all required bindings
14   boolean result = true;
15   // Bind variables
16   Iterator<Track> it1 = Track.iterator();
17   while(it1.hasNext())
18   {
19     Track t1 = it1.next();
20     Iterator<Track> it2 = t1.getSuccessors().iterator();
21     while(it2.hasNext())
22     {
23       Track t2 = it2.next();
24       if(t1 != t2)
25       {
26         Iterator<Track> it3 = t2.getSuccessors().iterator();
27         while(it3.hasNext())
28         {
29           Track t3 = it3.next();
30           if(t1 != t3 && t2 != t3)
31           {
32             // Evaluate THEN branch
33             result &= isSupervisedR1(t1,t2,t3);
```

```
34            }
35          }
36        }
37      }
38    }
39    return result;
40  }
41
42  // Existentially quantified node
43  boolean isSupervisedR1(Track t1, Track t2, Track t3)
44  {
45    // Track whether the node's pattern has matched
46    boolean matched = false;
47    // Bind variables
48    Iterator<Controller> it1 = t1.getControllers().iterator();
49    while(it1.hasNext())
50    {
51      Controller c1 = it1.next();
52      if(t2.getControllers.contains(c1))
53      {
54        if(t3.getControllers.contains(c1))
55        {
56          // THEN case
57          matched = true;
58          // Evaluate THEN branch
59          if(SDD.LeafTrue)
60            return true;
61        }
62      }
63    }
64    // ELSE case
65    if(!matched)
66      // Evaluate ELSE branch
67      return SDD.LeafFalse;
68    // THEN case, evaluation of THEN branch negative
69    return false;
70  }
```

In order to monitor the invariants specified by the SDDs, the generated code can then be run in parallel or lock-step with the monitored program. Note that we abstract from the navigability problem by assuming that there is a way to statically access all instances of a type, e.g. for obtaining a binding for t1.

ESDDs. The handling of ESDDs is much less complicated than in GROOVE, as programming languages already provide a comparable feature in the form of functions. We thus encapsulate the ESDD in a method that we query as required.

Listing 2: Alternatives (Java fragment), runtime monitor

```
1  // Universally quantified node
2  boolean isPropertyR()
3  {
4    ...
5    // Bind variables
6    Shuttle s1 = ...;
7    Shuttle s2 = ...;
8    Registry r1 = ...;
9    ...
```

```
10    // ESDD
11    if(isESDDProperty(r1, s1, s2)
12    {
13      // THEN CASE
14      ...
15    }
16    ...
17  }
18
19  bool isESDDProperty(Registry r1, Shuttle s1, Shuttle s2)
20  {
21    // Result
22    bool result = false;
23    // Evaluate
24    ...
25    return result;
26  }
```

Multiple alternative paths. Implementing multiple alternative paths is trivial in code for the monitoring version:

Listing 3: Alternatives (Java fragment), runtime monitor

```
1  // Universally quantified node
2  boolean isPropertyR()
3  {
4    // Default is true, AND join the results for all required bindings
5    boolean result = true;
6    // Bind variables
7    ...
8      // THEN case
9      result &= (isPropertyR1a(v1, v2, v3) || isPropertyR1b(v1, v2, v3));
10   ...
11   return result;
12 }
```

Advanced applications. The above code represents a compact solution that is suitable where SDDs are used to specify system behavior in place of Story Patterns or to simply monitor properties. For advanced applications like formal verification or debugging, it may be desirable to explicitly keep track of successful or unsuccessful bindings. We may even want to compute the entire result set.

In order to be able to efficiently generate such verifiers, we provide a more sophisticated framework, consisting of classes that implement key concepts of the formalization, i.e. the SDD structure and bindings, witnesses and witness sets etc., applying to all SDDs. As these classes closely reflect what was presented in Section 3.3 above, presenting the details of such an implementation would be mostly redundant.

Implementations of the propagation functions ($apply_n$) and the evaluation procedure ($iterate$ etc.) are provided using the strategy pattern. We provide three strategies – computing the first valid candidate set, computing all valid candidate sets,

and computing all valid candidate sets with guaranteed termination for recursive ESDDs.

In order to generate a specific SDD, we then only have to specialize the AbstractSDD and AbstractSDDNode classes, connect them in the correct order and overload the matching function of each node so that it correctly reflects the node's SDDP.

4 Temporal Properties

With SDDs, we have introduced a notation for expressing complex static properties. It is tempting to apply the same approach to temporal properties and use it to describe the structural evolution of a system.

The temporal behavior of a system can be described as a sequence of states. When we model the system as a *graph transformation system* (see Definition 14), each of these states corresponds to a graph. Between states, the identity of nodes and edges is preserved.

When evaluating temporal properties, it is no longer sufficient to focus on the question whether certain structural properties hold for a single state – we need to consider the duration and temporal ordering of the individual incidences of the properties.

4.1 Timed Story Scenario Diagrams

The idea behind *Timed Story Scenario Diagrams* (*TSSD*) is to use the ordering of incidences of structural properties in order to specify temporal properties as sets of valid orderings. The diagrams are thus directed acyclic graphs consisting of nodes, each containing an SDD defining a structural property, and edges, constraining the ordering of incidences.

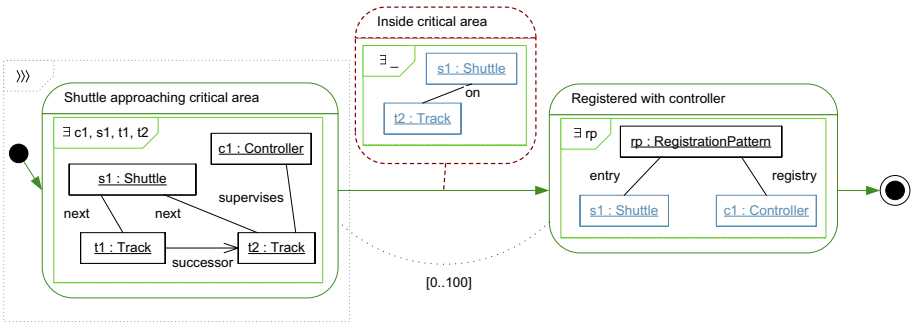


Figure 4.1: TSSD - A Shuttle registers with a Registry

Figure 4.1 is a basic example presenting the key elements of a TSSD. When a Shuttle is approaching a Controller's supervised area, they have between 0 and 100 time units to instantiate 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.

We now systematically introduce the underlying concepts.

4.1.1 Basic Principles

Situations. Each node of a TSSD defines a *situation*. While a situation characterizes a set of states, calling it a state would be misleading. As we discuss in Section 4.1.3 below, a TSSD is not a state chart, as multiple situations of the same TSSD can be incident, i.e. active, at the same time.

A situation may have a label, which can then be used to represent the situation. As the SDDs themselves may already be quite large, it may sometimes be preferable to define the situations separately and then draw the actual TSSD using such situation references – especially if the TSSD itself is complex or the same situation appears multiple times in it.

All SDDs that appear on the same path through a TSSD are connected so that bindings are shared between subsequent situations. If a variable is bound by a situation, it cannot be rebound later. If bindings were not retained, it would be difficult to specify simple properties such as 'If a shuttle accepts a task, it needs to complete it.' because any shuttle fulfilling any task would then complete the scenario.

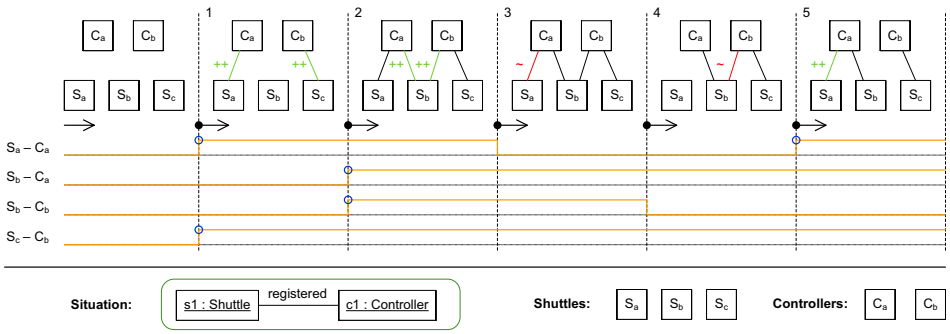


Figure 4.2: The relationship between a situation and its observations

When matching a situation, its SDD generates a result set. Each valid candidate set in the result set is called an *observation* of the situation. However, as the SDD encodes a structural property, whose incidence is typically not limited to a single point in time but spans an interval, the situation could generate infinitely many observations for the same candidate set. An observation is thus made only at the specific time when a structural property is first present after being absent. For a given situation encoding a property p and a given candidate set $\mathcal{C}$, an observation is thus generated at every time t where $p(\mathcal{C})$ at t and there is a time u with $u < t$ so that $\neg p(\mathcal{C})$ at all times v with $u \leq v < t$. Figure 4.2 illustrates this for a situation encoding that a specific shuttle is registered with a specific controller. As the truth value of this property changes over time for the different pairs, observations

(marked by small circles) are only generated where the truth value changes from $false$ to $true$. For the pair (S_a, C_a), two observations are generated, one a time 1 and one at time 5.

Temporal connectors. The observations for a TSSD are then placed in relation to each other using temporal connectors between situations specifying their temporal ordering.

- The *eventually* connector ($A \longrightarrow B$) denotes that an observation for situation A is made before an observation for situation B. Note that this includes the case that the observations for A and B occur simultaneously, i.e. $t(o_A) \leq t(o_B)$. Figure 4.3 shows an example of such a connector: A task that is started needs to be completed at some future time.

Figure 4.3: Eventually connector: $A \wedge \mathbf{F}B$

- The *until* connector ($A \longrightarrow\!\!\!\!\blacktriangleright\!\blacktriangleright B$) denotes that an observation for situation A is made and that the encoded structural property remains valid until a compatible observation for situation B is made. If no appropriate B is matched before the structural property ceases to be valid, the observation for A is discarded. Figure 4.4 shows an example of such a connector: The convoy pattern needs to remain active until the convoy is dissolved.

Figure 4.4: Until connector: $A \wedge A\mathbf{U}B$

- The *immediately* connector ($A \longrightarrow\!\!\triangleright\ B$) denotes that an observation for situation B is made at the same time as the corresponding observation for situation A, i.e. in the same state of the system. If no such observation of B exists, the observation for A is also discarded. Figure 4.5 provides an example: When an additional task is accepted, the schedule still needs to be consistent.

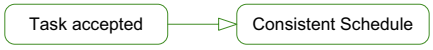


Figure 4.5: Immediately connector: $A \wedge B$

Technically, only the eventually connector is actually fundamental. Both the immediately connector, which can be emulated using an eventually connector and a time constraint of 0, and the until connector, which can be emulated using an eventually connector and an appropriate transition guard, are redundant. They are included for convenience, to facilitate the application of optimized evaluation strategies, and to reduce clutter in diagrams.

Traces. As situations generate sets of observations and as bindings are retained across situations, the indicated temporal ordering only makes sense when applied to compatible pairs of observations, i.e. if the candidate set of the more recent observation actually evolved from the candidate set of the earlier observation. For example, Figure 4.3 constrains the life-cycle of a single task, whereas the behavior of separate tasks is completely independent. Moreover, this same argument applies to multiple observations based on the same candidate set (such as the pair (S_a, C_a) in Figure 4.2) as well – a subsequent observation should not be invalidated just because the structure matched by the antecedent reappears. $A \longrightarrow\!\!\blacktriangleright B$ therefore does not imply that all compatible A need to occur before B, but rather that a compatible A exists before B. Such a sequence of correctly ordered compatible observations is called a *trace*. As there may be multiple antecedent observations with identical candidate sets, a single observation can extend multiple traces. As a candidate set may later be extended in multiple ways, each trace may furthermore be extended by several concurrent observations, resulting in a set of alternative traces.

Pseudostates. The evaluation semantics of a TSSD are determined by the graph structure set up by the connectors and a set of pseudostates, which specify where evaluation should start and terminate and provide a way to encode logical operators.

Evaluation always starts at the initial node • which always matches exactly once at the earliest time possible. The descriptive, sequential character of TSSDs implies the assumption that time is bounded in the past so that this point in time is uniquely identified.

The termination node ⦿ marks the end of a branch of a TSSD, i.e. sequence of connected situations. A ⦿ node always matches as late as possible, i.e. the current state during runtime monitoring or the last state of a finite system execution path π when analyzing a completed run. When the path π is infinite, the ⦿ node technically never matches at all.

A trace is completed once it has reached a termination node, i.e. when it could be extended with an observation for the ⦿ node. A system execution path π then fulfills a TSSD if a completed trace to a ⦿ node exists within a prefix of π.

The basic example in Figure 4.6 specifies that sometime during the system execution, a shuttle selects a task and completes it. In conjunction with an $\longrightarrow\!\!\!\!\blacktriangleright$

Figure 4.6: Some task is eventually completed

connector, a $\odot$ node can express that a property, e.g. safety, should hold globally, as illustrated in Figure 4.7.

Figure 4.7: The system is globally safe

Branches. While TSSDs need to be acyclic, each situation may have multiple successors and predecessors. If the TSSD forks, both branches progress independently and in parallel. Observations are only partially ordered.

By using multiple $\odot$ nodes on independent branches, disjunction (logical $\vee$) can be expressed, as a completed trace at any one of the $\odot$ nodes is sufficient. Figure 4.8 provides an example: Once a shuttle has acquired a task, it must either complete or delegate it.

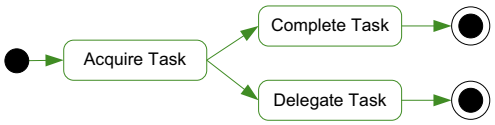


Figure 4.8: TSSD containing disjunction

If a situation has multiple ingoing temporal ordering edges, observations for all situations directly preceding it need to exist. Multiple incoming connectors thus correspond to conjunction (logical $\wedge$). Figure 4.9 provides an example: A shuttle needs to notify the source storage facility and reserve a route before it can pick up the cargo.

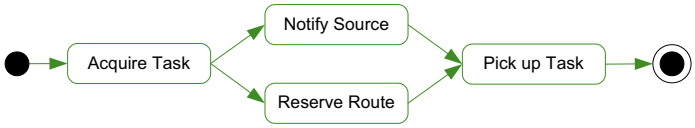


Figure 4.9: TSSD containing conjunction

In order to keep the notation based on a limited number of concepts, there are no specific $\wedge$ or $\vee$ nodes (as exist e.g. in EPCs). This means that it is not trivially possible to $\vee$-join two branches.[6] Using subscenarios (see Section 4.1.4), the otherwise required duplicate specification of the postfix can, however, be avoided.

A branch that does not end in a $\odot$ node is optional and has no effect on the satisfaction of a TSSD as it can never generate a completed trace. Figure 4.10 provides an example: The shuttle may stop for maintenance while executing a task.

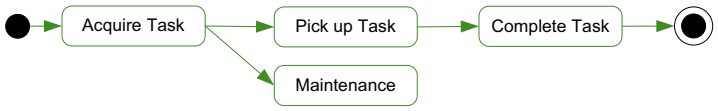


Figure 4.10: TSSD with an optional branch

More relevantly, it is possible in situations with multiple ingoing connectors to have a branch that does not lead back to an initial node in order to make statements about the past. While the $eventually$ connector then serves as the $past$ operator, $until$ can be used to emulate $since$ as time is assumed to be bounded in the past. In the example in Figure 4.11, the shuttle must have been licensed for passenger transport sometime before picking up a passenger. Using references to the past is mostly useful on the conceptual level in order to denote a property as a necessary precondition to, but not an integral part of the specified sequence. Semantically, connecting all branches to the initial node would yield the same result as time is bounded in the past.

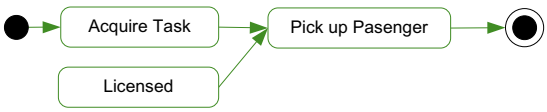


Figure 4.11: TSSD containing a reference to past events

Forbidden scenarios. As a way of expressing logical $\neg$ and negate whole scenarios, it is possible to turn branches of a TSSD or the entire diagram into forbidden scenarios. In the style of SDD connectors, required situations and connectors are drawn with solid green lines, while forbidden situations and connectors use dashed red lines. In order to avoid visual confusion with the contained SDDs, TSSDs use darker shades of green and red.

[6] It could be argued that $\vee$-joins are contra-intuitive anyway. If a customer has a complaint, and can then send it by post or email, what if he does both? Both traces continue to be live, and if there are no appropriate procedures in place, the complaint may actually be processed twice. While using xor resolves this specific ambiguity, its use can easily introduce subtle semantic issues of its own, as witnessed by EPCs [26].

Forbidden scenarios are defined by means of *inhibitors*. Normally, a connector is disabled and becomes enabled when it is reached by an appropriate trace. Inhibitors are enabled and become disabled if a trace reaches them. Inhibitors mark the end of a forbidden scenario and thus are the connectors leading from forbidden to required elements. This can either occur where a forbidden scenario is joined with a required one or at the end of a branch in the ⦿ node, which is considered a required element. Normally, a situation is incident if all inbound connectors are enabled, i.e. a compatible trace for each predecessor exists, and if its structural property can be matched. In the presence of an inhibitor, the subsequent required situation is only enabled if *no* trace completing the forbidden branch exists. The semantics of all other situations and connectors in a forbidden scenario is unchanged.

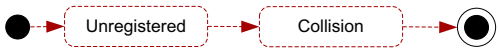


Figure 4.12: A forbidden scenario

The TSSD in Figure 4.12 will thus immediately be fulfilled until a shuttle is not properly registered and then collides with another shuttle. As soon as this sequence of events occurs, the TSSD is no longer satisfied.

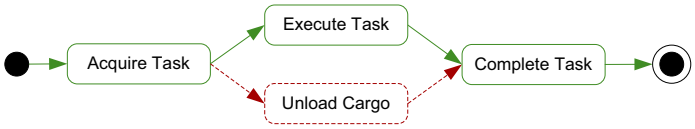


Figure 4.13: Forbidden during execution

Figure 4.13 encodes that a shuttle may not unload the cargo before the task is completed, i.e. the shuttle has arrived at the proper destination. Complete Task will only match if Unload Cargo has not been observed before.

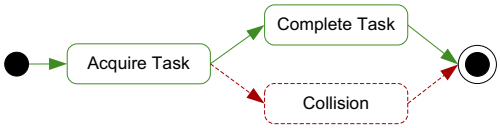


Figure 4.14: Forbidden during and after execution

The TSSD in Figure 4.14 is similar to the previous example, but as the join only takes place at the ⦿ node which will match as late as possible, any collision, even after the task is completed, will invalidate the scenario.

As multiple ⦿ nodes represent alternatives, the example in Figure 4.15 is satisfied if the shuttle on standby either receives a task and completes it, or never receives a task at all.

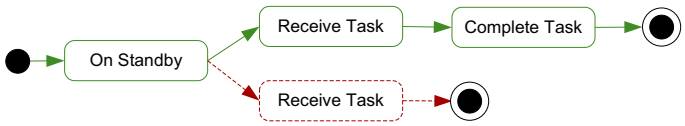


Figure 4.15: Forbidden scenario as an alternative

Parallel composition. If there are multiple initial nodes in a single diagram, evaluation starts at all initial nodes simultaneously. The TSSD will then only be satisfied if a ⊙ node is reached by a trace from every initial node. In particular, this mechanism can be used to create the parallel composition of multiple TSSDs. If the diagram graph is not connected because the branches have no situations in common, a dotted constraint edge is drawn between the initial nodes in order to indicate that the branches actually constitute a single diagram.

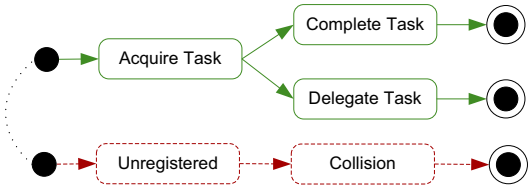


Figure 4.16: Parallel composition of scenarios

A particularly useful application of this construct is the specification of invariants. In Figure 4.16, parallel composition is used to specify that there must be no collision during task execution. The same effect could be achieved by inserting the forbidden scenario between the initial node and both ⊙ nodes of the constrained scenario, which would, however, result in a more complex diagram.

4.1.2 Constraints

The valid scenarios recognized by a TSSD can be further restricted by specifying guards that constrain the observations that are admissible between situations and introducing time bounds. The notations we use for specifying constraints are specifically inspired by the Visual Timed Event Scenario approach [1].

Constraint Edges. Constraints can appear directly on the temporal connectors defining the ordering of situations or on dedicated *constraint edges* that may connect any two situations regardless of their relative position in the diagram. Constraint edges are drawn as curved, dotted connectors between situations. They have no direction.

Guards. Forbidden scenarios provide a generic notation for prohibiting certain observation sequences between any two situations of a required scenario. However, as subsequent situations may be observed in the same system state, a forbidden scenario forbidding C between A and B requires $A \wedge (\neg C \,\mathbf{U}\, (B \wedge \neg C))$, i.e. B will not match if C is observed at the same time. While it would be possible to prohibit C only strictly before B is observed by placing a non-zero time constraint on the inhibitor leading from C to B, this seemed unnecessarily complex, particularly if the forbidden scenario consists of a single situation. In order to directly support the common idiom that a situation is forbidden *between* two situations, we introduce *guards* as a more lightweight notation. By annotating a connector from A to B with a situation C, written as $A \xrightarrow{\neg C} B$, we forbid compatible observations for C between two compatible observations for A and B, i.e. require $A \wedge (\neg C \,\mathbf{U}\, B)$. Additionally, it is possible to constrain the interval between two observations on concurrent branches of the diagram using a constraint edge $A \cdots \neg C \cdots B$, which cannot be expressed using a single forbidden scenario, which would introduce an implied temporal ordering.

When specifying a guard in a diagram, we link the forbidden situation to the connector with a line. As the situation is forbidden, it is drawn in the style of forbidden scenarios, i.e. dark red and dashed, albeit with a slightly bolder border. See Figure 4.17 for an example: While inside the critical section, the shuttle must not be disconnected from the controller.

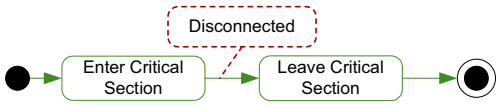


Figure 4.17: A forbidden guard

In Figure 4.18, we use a constraint edge in order to specify a constraint spanning multiple situations.

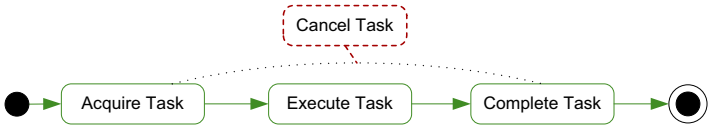


Figure 4.18: A forbidden guard spanning multiple situations

As a natural extension, we also allow specifying required situations $A \xrightarrow{C} B$ that define invariants that always need to hold between the two situations in order to eliminate the need for unintuitive double negations. Their border is drawn in the style for required elements, i.e. as a dark green solid line. See Figure 4.19 for an example: The shuttle needs to be registered with the controller while inside the critical section. As the example suggests, it is easy to convert the two types of

guards into each other by negating the contained SDD.

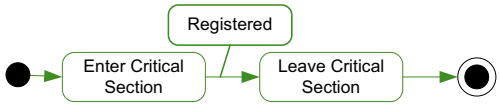
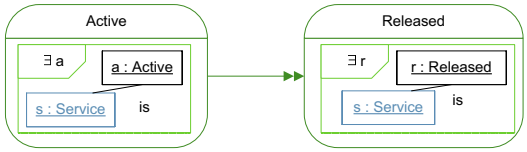
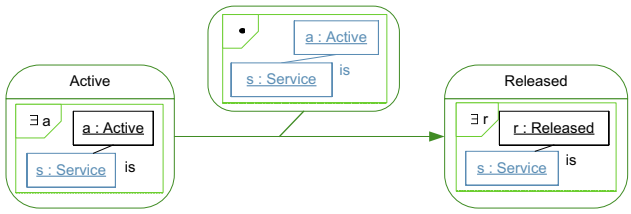


Figure 4.19: A required situation

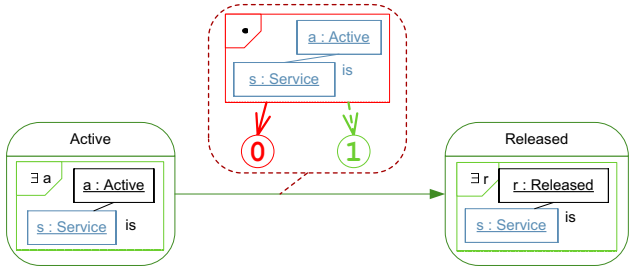
Note that $A \xrightarrow{A} B$ is different from $A \longrightarrow\!\!\blacktriangleright\!\!\blacktriangleright B$ – the former just requires that at any time between the two observations, some instance of A can be observed, while the latter requires that a specific observation of A remains valid. The example $A \xrightarrow{A} B$ brings up a technical point: As variables may not be rebound and the same variables may thus not be quantified twice in the same branch, the free variables of the contained SDD of a situation that is used as a guard are implicitly renamed to make their names unique. $A \xrightarrow{A} B$ therefore actually corresponds to $A \xrightarrow{A'} B$, where A' is the situation that is derived from A by renaming A's free variables.



a. Specification using an $\longrightarrow\!\!\blacktriangleright\!\!\blacktriangleright$ connector



b. Specification using a required guard



c. Specification using a forbidden guard

Figure 4.20: The relationship between *until*, guards and forbidden scenarios

In Figure 4.20, we specify the property that a service needs to remain active until it is released using an $\longrightarrow\!\!\blacktriangleright\!\!\blacktriangleright$ connector, which can be rewritten using a required

guard. The required guard is, in turn, just a syntactic feature that is expanded into a forbidden guard containing the negated property, which provides an efficient way of verifying the property.

Strict situations. There are four commonly used idioms in connection with guards that are supported by a logical extension of the syntax.

Figure 4.21: Strictly next situation

The standard situation semantics only ensure that the same observation cannot be made earlier. However, it is frequently desirable to require that a situation should not have matched at all before it is observed. In the example in Figure 4.21, we would like to require that the task that is selected is actually the *first* one to be selected, i.e. that no other task has previously been selected by the same shuttle (this also excludes that the same task is selected again later). Completion of the scenario can thus only be achieved by completing the first selected task – the shuttle may select and complete other tasks, but this is not recognized by the scenario. This effect can be achieved by placing a situation on a connector leading to itself as a forbidden guard. As a more compact notation, it is possible to 'bend' the forbidden guard on top of the situation itself, which is then drawn with an additional slightly bolder dashed dark red border. The situation is then marked as *strictly next*.

Figure 4.22: Strictly previous situation

The same concept can be applied in the other direction as well: If a situation is placed on one of its outgoing connections as forbidden guard, the scenario will only accept the *last* observation that is made for the situation. In the example in Figure 4.22, requests are superseded by subsequent ones. Such a situation is marked as *strictly previous*. A situation may have multiple guard constraint connectors and be marked strictly previous and strictly next at the same time.

Figure 4.23: Strict situation

In LSCs, the message types that appear in a scenario are forbidden where they are not explicitly allowed. This behavior, which is useful to enfore strict orderings between situations, can be emulated by placing a forbidden guard for every situation on every connector within the scenario that is not coming from or leading

to a pseudostate. As explicitly specifying this would result in an excessive number of guards, however, it is possible to express this property by simply placing an additional slightly bolder dashed dark red border around the situation without connecting the border to any connectors, which basically says 'required here, forbidden elsewhere'. Such a situation is called *strict*. In the example in Figure 4.23, the resource may only be used, exactly once, between acquiring and releasing it (for each acquisition).

Figure 4.24: Globally strict situation

Strict situations do not constrain the connectors from and to pseudostates in order to allow the scenario to match repeatedly during the same run of the system, provided that the instances do not overlap. If the intention is to actually express that the scenario appears only once, the scope of a strict situation can be extended to the connectors either coming from inital nodes, leading to ⦿ nodes, or both, by placing the corresponding symbol on the node border. The situation then becomes *globally strict*. Figure 4.24 encodes the requirement that the system is initialized exactly once.

An important point to note for any strict situation is that situations are structural properties, not events, and may depend on bindings from previous observations. The guards can therefore only be violated on a connector where all the situation's bound variables are already bound, as these will otherwise be bound to $\perp$ and never match. Implicitly quantifying these variables does not yield the expected behavior, as the guards would then no longer differentiate between traces. In the above example, only the first one among all the users who acquire resources would be allowed to use a resource, which is hardly the intended behavior.

Time constraints. While the temporal connectors constrain the temporal ordering of observations, they place no restriction on the elapsed time, making it impossible to prove that a finite trace will not eventually fulfill the scenario. While we can say that a required scenario has not occured yet, there is no specific point where we can stop waiting for eventual completion. In particular, we lack a means of requiring a practically relevant notion of progress as any finite period of inactivity would be acceptable.

However, by means of *time constraints*, we can specify an interval defining the permitted delay between two observations for two situations A and B within the same trace or related traces. The interval is defined by an lower bound l and an upper bound u and may be either open or closed at both ends. A time constraint can either be placed directly on a temporal connector ($A \xrightarrow{[l...u]} B$) or on a dedicated

constraint edge ($A \cdots [l \ldots u] \cdots B$). Note that in connection with a constraint edge, the time constraint does not imply an ordering, i.e. that A has to precede B or vice versa – the situations do not even need to be on the same branch of the TSSD.

Figure 4.25: Basic time constraint

Figure 4.25 shows a simple time constraint bounding an eventually connector.

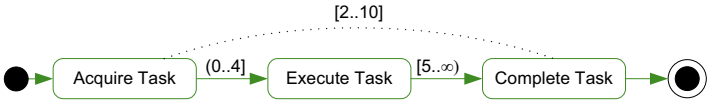


Figure 4.26: Multiple time constraint edges

Figure 4.26 presents an example with multiple constraints. As all constraints need to hold, the more restrictive bounds dominate the less restrictive ones. Time bounds need to be consistent, i.e. not mutually exclusive and thus contradictory.

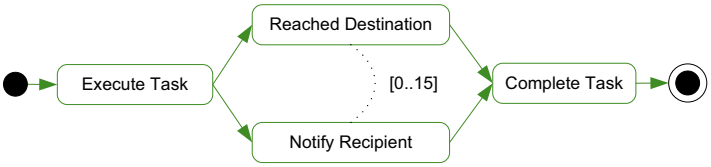


Figure 4.27: Constraint across branches

In the example in Figure 4.27, the maximum delay between reaching the destination and notifying the recipient is constraint, even though the two situations do not have to occur in any particular order.

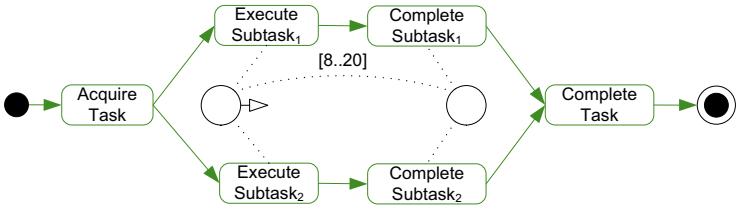


Figure 4.28: Constraint on the first/last observation in a set

There are two dedicated pseudostates, the *first of* and the *last of* node. The former matches when the first of the attached situations is observerd, while the latter matches when the last of the attached situations is observed. Using these nodes, it

is possible to specify a constraint on the time that elapses between the first observation for one set of situations and the last observation for another set of situations on multiple branches indicating a partial order between observations as illustrated in Figure 4.28: The subtasks may not be executed independently of each other, but need to respect a time constraint for the delay between the begin of the first subtask and the end of the last subtask.

Homomorphism. If the same situation appears multiple times in the same TSSD (by reference), its quantified variables need to be renamed internally in order to avoid binding the same variable twice. As graph isomorphisms are used for matching and different variables are thus not bound to the same graph element, the second situation will not match the same subgraph as the first. While homomorphism for the affected variables can be permitted at the SDD level, this defeats the purpose of using situation references. We therefore allow placing a *homomorphism constraint* at the TSSD level as a shortcut for the corresponding expansion at the level of the contained SDDs.

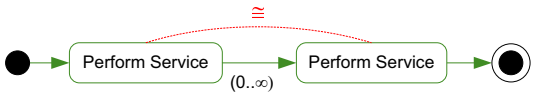


Figure 4.29: Allowing homomorphism across multiple instances of a situation

Figure 4.29 provides an example: Two different services may be performed, but the same service may also be performed twice (note the time constraint that prevents the second situation from simply matching the same observation as the first).

4.1.3 Quantification

TSSDs provide quantification on several different levels, both with respect to structure and time. As observations are generated by SDDs, a situation can be observed as structurally equivalent but distinct instances of the same pattern. This is quite 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, e.g., a scenario encoding a – rather easy – undergraduate program, requiring that students sign up for, attend, and eventually complete at least one course. Students may then sign up for any number of courses and still fulfill the scenario by completing just one of them. This means that we have to relate the right observations to each other, i.e. keep track of which student is completing which course, and that he or she had actually signed up for it.

The example also illustrates why TSSDs are unlike statecharts. When trying to recognize all conformant sequences, we need to keep checking for new students. But even for a single student who is already attending one course, we still need to check whether he or she has signed up for another one, or we might miss the one that is actually completed. This means that a situation, once enabled, will keep generating additional observations, which means that a TSSD, which is representing a *set* of (potential) scenarios, can 'be' in many states at once.

Situation level quantification. As we have seen, universal quantifiers in SDDs have made it necessary to introduce candidate sets. While it is much more common to only use existential quantifiers in TSSDs (or at least use universal quantification for local properties inside of ESDDs or scoped nodes only), it is possible to use universal quantification. When evaluating the TSSD, we then have to propagate candidate sets as well. All subsequent situations have to match based on the generated candidate set, i.e. for every single witness, just like the child nodes of universally quantified nodes in an SDD. Consider the property 'If all tasks are approved (at the same time), then eventually all (these) tasks have to be completed (at the same time)', which could be written by universally quantifying over all approved tasks in the first situation.

Scenario level quantification. A more typical requirement would be 'Every *individual* task that is approved eventually needs to be completed'. Here, the involved SDDs are existentially quantified, describing a single task and its states. However, we want this scenario to hold every single time an approved task is observed.

Triggers. This is achieved by means of *trigger* blocks. Whenever the sequence within the trigger block has been observed in its entirety,[7] the corresponding trace becomes a *root trace*. The TSSD is then only fulfilled if an extension of each root trace successfully completes the triggered scenario. On the other hand, if the trigger is never completed, there is no root trace and the TSSD places no constraints on the system behavior.

Triggers perform a function similar to precharts in Live Sequence Charts [21]. Adopting the corresponding terminology, we distinguish between *universal* TSSDs, which possess a trigger and need to be fulfilled every time it matches, and *existential* TSSDs, which do not have a trigger and need to match just once during the execution of the system. Existential TSSDs can be seen as a special case of triggered TSSDs – they are implicitly triggered by their initial node, which matches immediately, but only once.

In Figure 4.1, we wanted the scenario to be triggered and successfully completed

[7] In order to allow branches inside the trigger, the trigger is completed as soon as a situation inside the trigger block that does not have a successor inside the trigger block is observed.

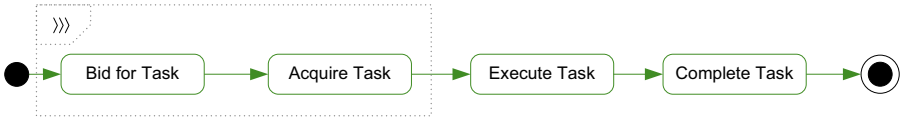


Figure 4.30: A complex trigger block

for all cases when a matching shuttle-controller-pair is detected. Here, a trigger consisting of a single situation is sufficient. However, arbitrarily long initial sections of a TSSD can be placed inside a trigger. Figure 4.30 provides an example with a non-trivial trigger block containing a sequence of two situations. A root trace is only created once the task is acquired.

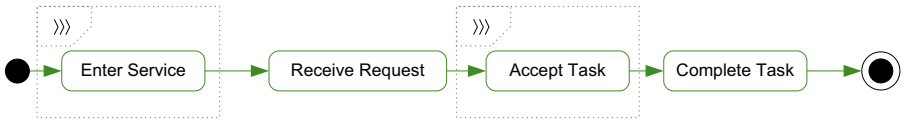


Figure 4.31: Multiple trigger blocks

It is possible to have multiple triggers in the same TSSD. In Figure 4.31, each shuttle entering service needs to complete every task it accepts. Note that the scenario fails if a shuttle does not accept any task at all because the first trigger has already created a root trace.

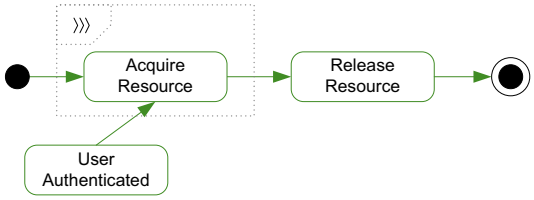


Figure 4.32: Antecedent triggered scenario

As a powerful feature, triggers can require the presence of antecedent observations. In Figure 4.32, we require that a user acquiring a resource eventually releases it *and* was previously authenticated.

Likewise, the scenario that is defined in Figure 4.33 requires that a route was reserved sometime between being planned and actually being used (but only if it is actually used).

The ability to express past and intervening triggered scenarios requires a slight extension of the syntax. Without the trigger block, Acquire Resource would never match without a compatible observation for User Authenticated, nor would Use Route ever match without Reserve Route. A violation would thus not be recog-

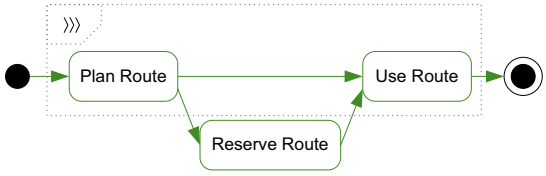


Figure 4.33: Intervening triggered scenario

nized. We therefore define that when evaluating whether a trigger block is completed, only those previous situations that are directly connected to an initial node are considered as preconditions, not those without predecessors or only with predecessors from the trigger block itself. When defining the formal semantics, we present a way to rewrite past and intervening triggered scenarios in a way that makes this explicit.

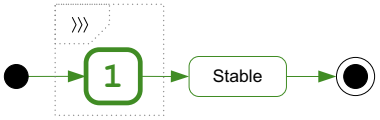


Figure 4.34: Globally triggered scenario

A global trigger can be used to express properties that need to hold in every state (i.e. of the form $\mathbf{AG}\,\varphi$) such as fairness (i.e. $\mathbf{AG}(\mathbf{AF}P)$). The example in Figure 4.34 expresses the requirement that the system always reaches a stable state *again*, which is expressed by means of a trigger block containing a node marked with 1, which represents a *trivial situation* that is trivially *true* and matches every time compatible traces for all its predecessors are present. The trivially *true* situation thus generates a new root trace in every system state.

Sequence labels. We allow attaching labels to a sequence in a TSSD by connecting the first and last element of the sequence with a special blue dotted arrow. This can be used to structure the diagram and may be useful for monitoring, e.g. for listing all currently triggered instances of the sequence. In Figure 4.35, the scenario is structured into a selection phase and an execution phase.

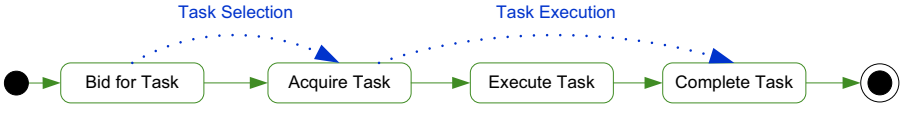
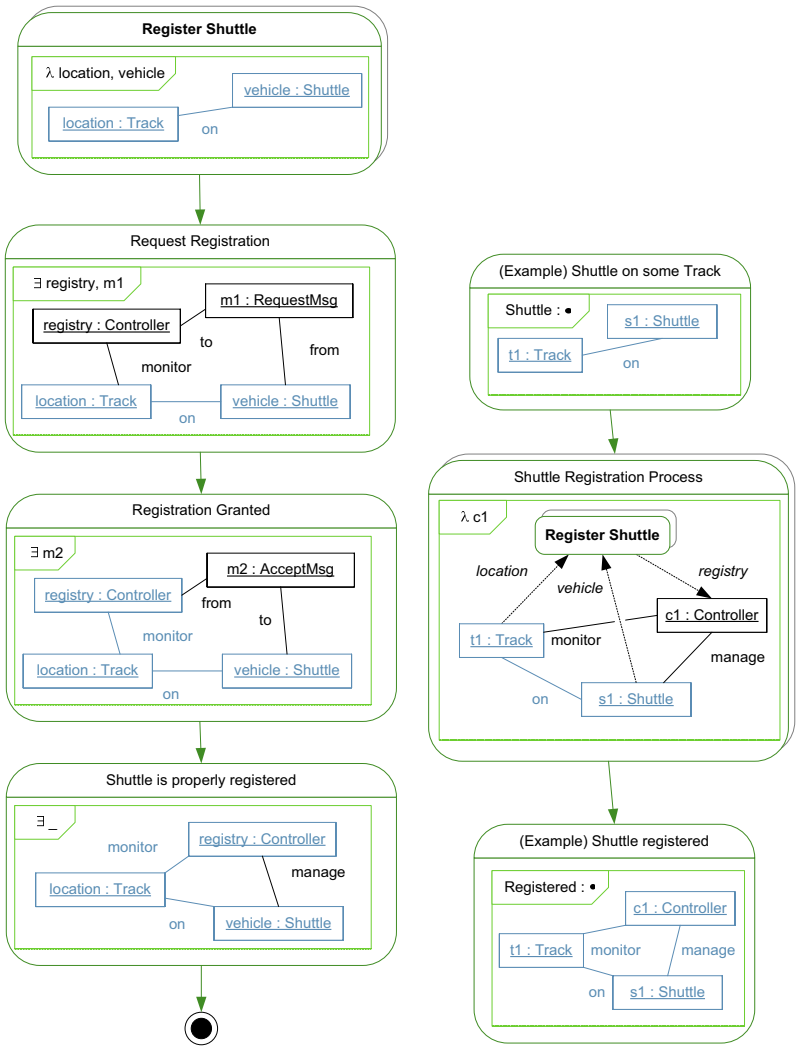


Figure 4.35: Sequence labels on a simple scenario

4.1.4 Subscenarios

Modularity is again of paramount importance for practical scalability. We therefore provide the ability to invoke a previously defined *subscenario* as part of a TSSD. Subscenarios perform a similar function in TSSDs as ESDDs do in SDDs.



a. Definition of the subscenario b. Invocation of the subscenario

Figure 4.36: Subscenario

Definition. A subscenario definition begins with a special λ situation which, just like an ESDD's λ node, binds roles and parameters. Other than that, a subscenario is just a regular TSSD. As in parametrized ESDDs, the parameters in subscenarios may appear anywhere where constants would be allow, e.g. as part of guards. A subscenario may also reference other subscenarios. As a TSSD needs to be acyclic,

recursive subscenarios are the only way to specify loops. A subscenario can also be used to encode an $\vee$-join by encapsulating the alternative branches in a subscenario. The subscenario in Figure 4.36a describes how a shuttle registers with a controller.

Invocation. Invocation works just like ESDD invocation, with one notable difference. In the context of scenarios, we will often need to access the bindings that are created by the subscenario in subsequent situations of the invoking scenario. If we want an ESDD to 'bind' a variable, we can simply assign all possible bindings to it in the host node, letting the ESDD confirm those bindings that match it. Using this approach for subscenarios would not only be inefficient, but potentially impossible, as the bound element may not have existed yet at the time of invocation.

Therefore, the invocation itself takes place inside a λ-node that allows exporting arbitrary bindings (e.g. registry $\rightarrow$ c1) back from the subscenario. In Figure 4.36b, we see how the shuttle is not registered beforehand, the subscenario is invoked, and the shuttle is registered afterwards.

Scenario situations are the equivalent of scoped nodes in SDDs. They are situations, drawn with a bold border, that may contain a sequence of situations and pseudostates. Again, they mostly serve as parentheses and can be defined using the existing mechanism for modularity, i.e. subscenarios. The most typical example for the use of scenario situations is the explicit $\vee$-join (see Figure 4.37), which saves the need to specify the following suffix twice (or at least use two identical subscenario invocations).

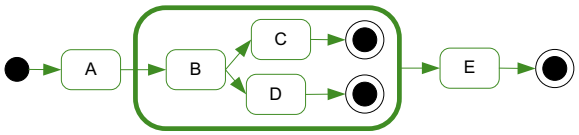


Figure 4.37: Explicit $\vee$-join

The embedded scenario is evaluated in its given context in the surrounding scenario. A possible initial node cannot match earlier than any situation preceding the scenario situation in the surrounding scenario, and a $\odot$ node may and need not match later than any subsequent situation. Nonetheless, they still match as early respectively late as possible within the given constraints. A 'global' property inside a scenario situation (see Figure 4.38) thus constrains exactly the interval between the surrounding situations (e.g. A and C (and is thus equivalent to a required guard). Choosing a different semantics, i.e. interpreting the $\odot$ node inside the scenario situation as the end of the surrounding scenario, would break the monotonicity of the scenario interpretation, i.e. a valid observation for C could later be invalidated because B ceases to be valid.

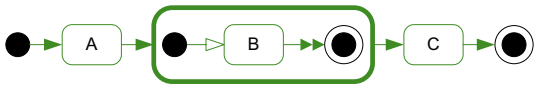


Figure 4.38: A globally required property inside a scenario situation is limited to the surrounding interval

Loops. For convenience, TSSDs provide a dedicated syntactical construct for specifying loops based on scenario situations. Internally, these loops can be represented as recursively defined subscenarios.

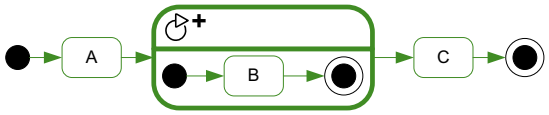


Figure 4.39: A loop that needs to match at least once

A loop is marked with $\circlearrowleft$ or $\circlearrowleft^+$ as in Figure 4.39. It is required to be observed only once, but, as connectors allow generating additional extensions of the same trace as long as they are enabled, also greedily matches as many iterations of the loop as possible.

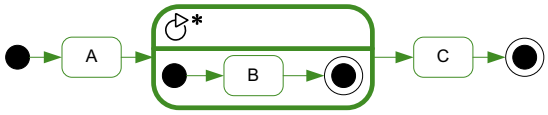


Figure 4.40: A loop that is matched zero or more times

An optional loop is marked with $\circlearrowleft^*$ as in Figure 4.40. It is equivalent to a regular loop and an additional connector bypassing the loop, i.e. does not need to be observed at all, but may be observed any number of times.

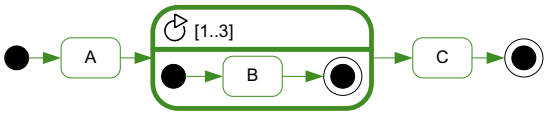


Figure 4.41: A loop that is matched a bounded number of times (1 to 3)

Finally, a bounded loop is marked with $\circlearrowleft\ [l..u]$ as in Figure 4.41. It needs to be observed at least l and at most u times. $\circlearrowleft^+$ is thus equivalent to $\circlearrowleft\ [1..\infty]$, while $\circlearrowleft^*$ is equivalent to $\circlearrowleft\ [0..\infty]$. Internally, the bounded loop can simply be unrolled or, more compactly, be represented by a parametrized recursive subscenario that is decreasing the bounds in each iteration.

Figure 4.42 illustrates two idioms that are relevant in connection with loops: An

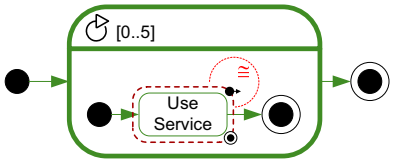


Figure 4.42: The user uses services, no more than 5 times altogether

upper bound is really only meaningful if the TSSD contains guards forbidding 'unobserved' iterations, e.g. by making one situation in the loop globally strict. Secondly, if the user does not have to use five different services, but may use the same service several times, we need to include a self-referential homomorphism constraint that is expanded when the loop is unrolled.

4.2 Syntax Reference

4.2.1 Situations

Situation. Defines a situation, i.e. a set of states, by means of an SDD. The label is optional. As most elements, situations can be required or forbidden. Required elements are drawn with dark green[a] solid lines, forbidden elements are drawn with dark red[b] dashed lines.

[a] As dark green, we define rgb(64, 140, 36).
[b] As dark red, we define rgb(164, 0, 0).

Situation reference. References a previously defined situation in order to save space. The label is required.

Scenario situation. Defines a scope that may contain other states and situations.

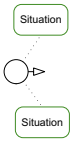
Scenario situation reference. References a previously defined scenario situation in order to save space. The label is required.

Trivial situation. Defines a situation that trivially matches whenever its preconditions are fulfilled, respectively never matches.

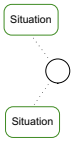
4.2.2 Pseudostates

Initial node. Marks the starting point(s) of the evaluation.

Termination node. Marks the end of a trace.



First of node. Matches when the first of the attached situations is observed.



Last of node. Matches when the last of the attached situations is observed.

4.2.3 Temporal connectors

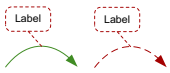
Eventually connector. Points to a situation that must follow eventually or finally.

Immediately connector. Points to a situation that must follow immediately, i.e in the same system state. Exists for convenience, equivalent to a $[0..0]$ time constraint.

Until connector. The source situation must match until the target situation matches.

4.2.4 Constraints

Constraint edge. Allows specifying constraints between arbitrary situations of a diagram.



Forbidden situation. The forbidden situation must not match between the source and target situation of the attached connector. Drawn using dark red dashed lines.

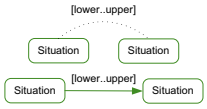
Required situation. The required situation must match between the source and target situation of the attached connector. Drawn using dark green solid lines.

Strictly next situation. The situation only accepts the first matching observation (per distinct trace).

Strictly previous situation. The situation only accepts the last matching observation (per distinct trace).

Strict situation. The situation is part of a branch, but an identical observation must not appear elsewhere between any two situations of the scenario. As all bindings that are reused by the strict situation need to be previously bound, the restriction only operates per distinct trace.

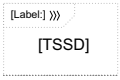
Globally strict situation. While strict situations only constrain connectors between situations, placing the corresponding symbol on the border of the strict situation indicates that the situation is equally forbidden on connectors either from initial nodes, to termination nodes, or both.



Time bound. Constrains the time that may pass between observations of the connected situations. The situations do not have to be on the same branch.

Homomorphism constraint. Allows several situation references in the same TSSD to match the exact same instances by adding the appropriate elements and homomorphism constraints in the corresponding SDDs. The self reference is only meaningful in loop definitions. Drawn in red, as the constraint refers to the SDD level.

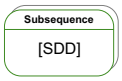
4.2.5 Quantification



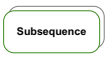
Trigger. The dashed grey box marked with >>> designates a set of situations as the trigger of a universal TSSD. The trigger ∀-quantifies over the indicated variables.

Sequence label. Used to label a set of situations as a logical unit.

4.2.6 Subsequences



Subscenario situation. Represents the intial situation of a subscenario definition, corresponds to an ESDD's λ node. Likewise accepts roles and parameters.



Subscenario reference. Used to invoke a subscenario in the invoking λ situation of a scenario.

Role rebinding. Used to rebind roles to variables in the invoking λ situation of a scenario.

Parametrized reference. Used to invoke a subscenario with parameters as defined by the λ situation of the subscenario.

Loop. The subscenario in the qualified scenario situation is matched at least once. Internally translated into a recursive subscenario definition.

Optional loop. The subscenario in the qualified scenario situation is matched zero or more times. Internally translated into a recursive subscenario definition.

Bounded loop. The subscenario in the qualified scenario situation is matched at least l and at most u times. Internally translated into a parametrized recursive subscenario definition or unrolled explicitly.

4.3 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 [11] are hard to understand and even harder to write for any non-trivial property (cf. [13]).

As the temporal properties we specify constrain the structural evolution of a system, we are not concerned with individual states, but sequences of states. As before, we represent each state by a graph – the system can thus be represented by a graph transformation system (GTS) as defined by Definition 14. In Section 2.2.5, we have discussed how propositions in the form of state and path formula can be specified for such systems.

We will first define the semantics of TSSDs, based on the formalization of GTS and of SDDs, and then discuss how they relate to LTL and extensions of LTL with time constraints.

4.3.1 Definitions

System. The system the TSSD is monitoring or verifying is given as a typed GTS Y. While for SDDs, we checked whether a particular state satisfied the specified structural property, we are now interested in the question whether a particular path π (as defined in Section 2.2.4) that has been generated by Y (or is currently being generated by Y) satisfies the TSSD. The states of the path π are $\pi[i]$ and occur at time $T(\pi, i)$. Additionally, we define

$$T^{-1}(\pi, t) := i \mid T(\pi, i) \leq t < T(\pi, i+1) \tag{4.1}$$

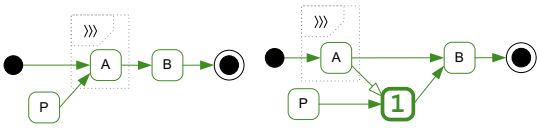
as the inverse of T that returns the index of the current state for a time t. We write $\pi^{-1}[t]$ as a shortcut for $\pi[T^{-1}(\pi, t)]$.

Diagram structure. A TSSD D consists of a set of situations and pseudostates $\mathcal{U}_D$. A *situation* U is characterized by its SDD S_U. We have $var(D) := \bigcup_{U \in \mathcal{U}_D} var(S_U)$ as the free variables of the TSSD. For each situation $U \in \mathcal{U}_D$, we define $\mathsf{pred}_F(U)$, $\mathsf{pred}_I(U)$, and $\mathsf{pred}_U(U)$ as the predecessor situations of U connected to it by **F**, **I**, and **U** connectors. $\mathsf{pred}(U) := \mathsf{pred}_F(U) \cup \mathsf{pred}_I(U) \cup \mathsf{pred}_U(U)$ is then the set of all direct predecessors of U. $\mathsf{prefix}(U)$ is the transitive closure over pred, i.e. all direct and indirect predecessors. Likewise, we define $\mathsf{succ}(U)$ with its subsets $\mathsf{succ}_F(U)$, $\mathsf{succ}_I(U)$, and $\mathsf{succ}_U(U)$ and the transitive closure $\mathsf{suffix}(U)$ for the successors of U based on $\mathsf{pred}(U)$.

The set init_D contains the initial pseudostates λ_D of the TSSD. The set term_D contains all ⦿ nodes of D.

Trigger Blocks. In order to deal with trigger blocks, we define $\mathsf{triggers}_D$ as the set of situations that complete a trigger block of the TSSD, i.e. are inside a trigger block but have no successors inside the trigger block. If D is universal, those are the situations U inside a trigger block where all successors $U' \in \mathsf{succ}(U)$ are not inside the trigger block. If D is existential, we have $\mathsf{triggers}_D := \mathsf{init}_D$.

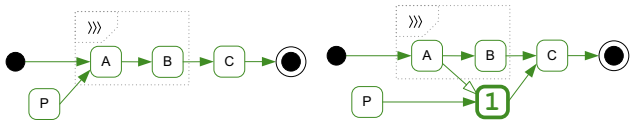
In order to be able to evaluate past and intervening triggered scenarios without having to change the standard evaluation semantics (situations can only be observed when all predecessors have been observed), such scenarios are internally encoded using additional trivially *true* situations. The trivially *true* situation preceded or followed by an immediately connector represents a neutral element that can be added between any two situations without changing the semantics. This is used to move the references to past or intervening events outside of the trigger.



a. Syntax b. Internal Encoding

Figure 4.43: Past triggered scenario (end of trigger block)

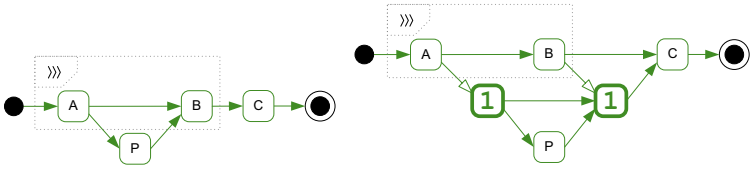
Figure 4.43 illustrates how the past triggered scenario (Figure 4.43a) is rewritten (Figure 4.43bso that A can match without P, but the scenario will not complete unless, immediately, a P that was observed before A is found.



a. Syntax b. Internal Encoding

Figure 4.44: Past triggered scenario (inside trigger block)

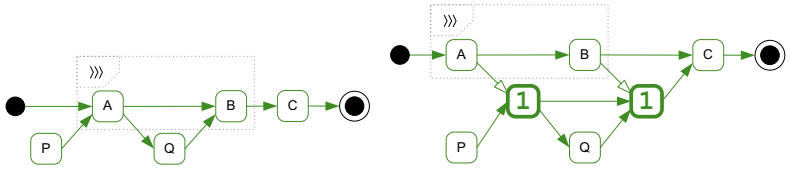
Figure 4.44 illustrates that this also works if the past triggered scenario is connected to an element of the trigger block other than the last.



a. Syntax b. Internal Encoding

Figure 4.45: Intervening triggered scenario

In Figure 4.45, an intervening triggered scenario is encoded. The first trivially *true* node is redundant, but is useful because it provides a simple procedure for encoding any triggered scenario (a sequence of trivially *true* situations, one for every situation inside the trigger block that has outside connections), even if past and intermediate scenarios occur simultaneously as in Figure 4.46.



a. Syntax b. Internal Encoding

Figure 4.46: Intervening and past triggered scenario

SDD adaptation. The SDDs in a TSSD are not independent of each other, but extend the candidate sets generated by their predecessors. Given an SDD S and its predecessor S', we define

$$\lambda_S[\mathcal{C}_{S'}] := \{(\lambda_S, \xi) \mid \exists (n, \xi) \in \mathcal{C}_{S'} \wedge eval(\mathcal{C}_{S'})\} \tag{4.2}$$

which takes the bindings of a valid final candidate set $\mathcal{C}_{S'}$ for S' and creates a corresponding candidate set at the initial node λ_S of S.

$$\lambda_S[\mathcal{A}_{S'}] := \{\lambda_S[\mathcal{C}_{S'}] \mid \mathcal{C}_{S'} \in \mathcal{A}_{S'}\} \tag{4.3}$$

performs this for a whole result set. We accordingly extend $\mathsf{parent}(n)$ so that the λ node of S_U has all the (1) (and transformation) nodes of the SDDs of all situations in $\mathsf{pred}(U)$ as parents in order to make the *evolved from* relation $\mathcal{C} \sqsubseteq \mathcal{C}'$ applicable accross situations.

Situation references allow reusing the same situation definition and, in particular, the same contained SDD. In order to make sure that all quantified variable names are unique, we define a relabeling function ℓ_U which, when applied to the SDD S_U, relabels all variables in $free(S_U)$ with globally unique variable names. ℓ_U is then also applied to the SDDs of all situations in $\mathsf{suffix}(U)$ so that variable names on the same branch are consistent.

While this allows multiple references to the same situation definition to appear on the same branch, the graph isomorphisms used for matching ensure that the subsequent instances will never generate observations of an identical structure, even if the structure becomes invalid and valid again and the first situation generates a new observation. If a different behavior is desired, this has to be made explicit by using homomorphism constraints. Internally, this will add the original variables to the SDD definitions of the subsequent situations and connect them to their renamed counterparts with homomorphism links.

Forbidden Scenarios. Basically, there is no such thing as a forbidden scenario at the semantical level, only *inhibitors*. Inhibitors are connectors that keep the situation they point to from matching when they are enabled, whereas regular connectors need to be enabled in order to allow their target to match. For every situation U, we identify the inhibitors by means of the set $\mathsf{inhibit}(U) \subseteq \mathsf{pred}(U)$, consisting of the situations U' connected to U by inhibitors. Only the last connector in a forbidden scenario is an inhibitor; all other situations and connectors of the forbidden scenario are just normal elements of $\mathcal{U}_D$. The TSSD in Figure 4.47a is thus internally represented as Figure 4.47b. Only for convenience, situations from which all paths to a ⦿ node lead across an inhibitor, and the connectors leading to them, are displayed as forbidden elements to make the undesirable parts of the scenario stand out.[8]

Given a situation U, $U_I \in \mathsf{inhibit}(U)$, and U_T as the last node in $\mathsf{prefix}(U_I)$ that is not part of the forbidden scenario, or λ_D, we add U_T to $\mathsf{pred}_F(U)$, i.e. a connector as shown in Figure 4.47c.[9]

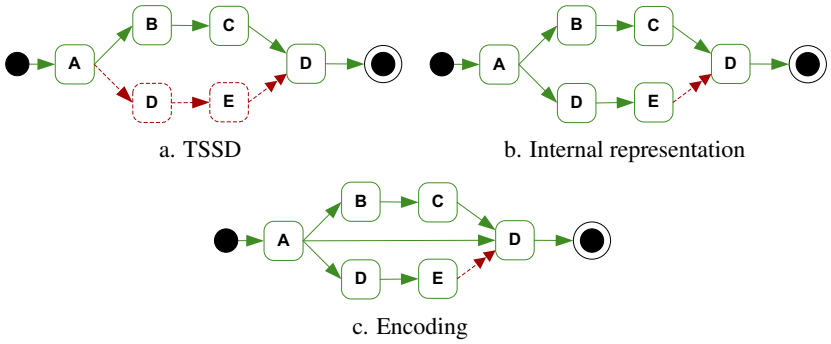


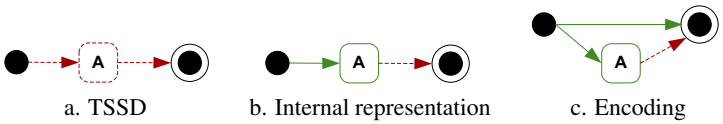
Figure 4.47: Encoding a forbidden scenario

This expansion eliminates the need for a special treatment of TSSDs that only consist of forbidden elements. While the intended semantics are not obvious from Figure 4.48a respectively 4.48b, the expanded version in Figure 4.48c directly ensures that the TSSD succeeds unless an A is found.

SDD restriction. In order to verify the requirement imposed on a situation U with SDD S_U by a **U** connector, we need to derive an SDD S'_U that verifies for a valid candidate set $\mathcal{C}$ generated by S_U whether it continues to be valid in the present state. S'_U can be obtained by eliminating the quantifiers for the free variables in

[8] In the – rather theoretical – case that we explicitly want to specify a forbidden scenario within a forbidden scenario, the tool will automatically turn the coloring off so that the actual inhibitors can be identified.

[9] As an optimization to accelerate matching, we can also choose to add a trivially *true* situation U_1 as padding between U_I and U as part of $\mathsf{pred}_I(U)$ and transfer all guards and constraints between U_I and U to the connector between U_I and U_1, as this will speed up evaluation of Equation 4.18.



a. TSSD b. Internal representation c. Encoding

Figure 4.48: Encoding of a forbidden property

$free(S_U)$ from S_U.

We introduce $S|_{\mathcal{V}}$ as notation for the SDD S' that removes the quantifiers for all variables $V \in \mathcal{V}$ from S but is otherwise identical to S. We define

$$S|_{\mathcal{V}} := S' \mid \mathcal{N}_{S'} = \mathcal{N}_S \wedge free(S') = free(S) \setminus \mathcal{V} \tag{4.4}$$

$$S|_{S'} := S|_{\mathcal{V}} \mid \mathcal{V} = free(S') \tag{4.5}$$

$$S|_{\mathcal{C}} := S|_{\mathcal{V}} \mid \mathcal{V} = \{V \mid \exists \xi \mid (n, \xi) \in \mathcal{C} : \xi(V) \neq \bot\}. \tag{4.6}$$

We can then define $S'_U := S_U|_{S_U}$.

Guards. The specified guards and strictness conditions are stored in a function $guard : \mathcal{U}_D \times \mathcal{U}_D \rightarrow \wp(\mathcal{U}_D)$, mapping pairs of situations to a set of situations that are forbidden between them. A required situation U with SDD S_U is treated as a forbidden situation with guard $\overline{S_U}$, the negation of the SDD, so that all guards represent forbidden situations. No guard for a pair of situations may thus match for any state between the observations for the pair (but may match in conjunction with the second observation).

U connectors are reduced to **F** connectors with an additional guard ensuring that the generated candidate set has not ceased to satisfy the structural constraint. We require that

$$\forall U' \in \mathsf{pred}_U(U) : \exists U_G \in guard(U', U) : S_{U_G} := \overline{S_{U'}}|_{S_{U'}}). \tag{4.7}$$

Time constraints. The time constraints are encoded by a function

$$delay : \wp(\mathcal{U}_D) \times \wp(\mathcal{U}_D) \rightarrow I, \tag{4.8}$$

where I is the set of all intervals $[l, u]$, $(l, u]$, $[l, u)$, and (l, u) with $l \in \mathbb{R}$ and $u \in \mathbb{R} \cup \infty$. $delay$ assigns an interval constraining the permitted delay between the earliest element of the first and latest element of the second set. The function is total: Unless defined otherwise, $delay$ returns $[0, \infty)$. For simple time constraints, the two sets simply contain just a single element. We further require that

$$\forall U' \in \mathsf{pred}_I(U) : delay(\{U'\}, \{U\}) = [0, 0], \tag{4.9}$$

thus reducing the **I** connectors to **F** connectors with time constraints.

Observations. Provided a TSSD D and a path π, we define an *observation* o as a tuple $(U, \mathcal{C}, t)$ of a situation U, a candidate set $\mathcal{C}$, and a time t. Note that this implies an extension of the codomain of the binding functions ξ contained in $\mathcal{C}$ from the nodes and edges of a single graph G to the nodes and edges of all type conformant graphs $\mathcal{G}[\mathcal{T}_Y]$ for Y's type graph $\mathcal{T}_Y$. We use $\mathcal{O}_D(\pi)$ to denote the set of all possible observations for D and π with

$$\mathcal{O}_D(\pi) := \{o = (U, \mathcal{C}, t) \mid \mathcal{C} \in [\![S_U]\!]^{\pi^{-1}[t]}\}. \tag{4.10}$$

Traces. A *trace* $\rho \in \mathcal{O}_D(\pi)^*$ is a valid sequence of observations. The question whether or not a sequence of observations is valid is central to the semantics of TSSDs and discussed in the next subsection. Note the difference between a path and a trace: as the scenario defined by a TSSD can occur multiple times during the same run of the system, i.e. in the same path, there can be many traces within a single path.

In an analogous manner to our definitions for a path π, we define $\rho[i]$ as the i^{th} observation, $T(\rho, i) := t \,|\, \exists U, \mathcal{C} : (U, \mathcal{C}, t) = \rho[i]$ as the time of the i^{th} observation, $\rho^{-1}[t]$ as the last observation before time t, and $l(\rho)$ as the length of ρ. The set of all observations on a trace ρ is denoted by $\mathcal{O}_\rho$. We write $\rho[U] := (o = (U_o, \mathcal{C}_o, t_o) \in \mathcal{O}_\rho \mid U_o = U)$ for the unique observation for U in ρ.

For observations on a trace ρ, we define

$$\begin{aligned} \mathsf{pred}(o) := \{o' \mid o = (U, \mathcal{C}, t) \in \mathcal{O}_\rho \wedge \\ o' = (U', \mathcal{C}', t') \in \mathcal{O}_\rho \wedge U' \in \mathsf{pred}(U)\} \end{aligned} \tag{4.11}$$

and, analogously, $\mathsf{prefix}(o)$, $\mathsf{succ}(o)$, and $\mathsf{suffix}(o)$.

For two traces ρ and ρ', we write $\rho \sqsubseteq \rho'$ if ρ is a prefix of ρ'.

Trace trees. The set of traces generated by a TSSD D for path π is stored in a *trace tree* $\mathcal{R}$. Each tree node represents an observation, every path from the root node represents a trace. The tree is simply built from the set by reusing common prefixes; all set operations are thus defined for the tree. The set of all observations in $\mathcal{R}$ is denoted by $\mathcal{O}_\mathcal{R}$. If there are multiple initial nodes due to a parallel composition, the trace tree becomes a trace forest.

The trace tree is a tree rather than a DAG because when the two branches of a TSSD reunite in an $\wedge$-join, we combine the traces for the branches into a new single trace. For that purpose, we define the notion of *observation compatible* traces. We define

$$\begin{aligned} \rho_1 \Uparrow \rho_2 := (\exists \rho' : \rho' \sqsubseteq \rho_1 \wedge \rho' \sqsubseteq \rho_2 \wedge \forall (U_1, \mathcal{C}_1, t_1) \in \mathcal{O}_{\rho_1} \setminus \mathcal{O}_{\rho'}, \\ (U_2, \mathcal{C}_2, t_2) \in \mathcal{O}_{\rho_2} \setminus \mathcal{O}_{\rho'} : U_1 \neq U_2), \end{aligned} \tag{4.12}$$

i.e. the two traces have a common prefix and afterwards contain no competing observations for the same situation. We then define the combination of two compatible traces

$$\rho_1 \cup \rho_2 := \rho' \mid \mathcal{O}_{\rho'} = \mathcal{O}_{\rho_1} \cup \mathcal{O}_{\rho_2} \wedge \\ \forall i \mid 0 < i < l(\rho') : T(\rho', i-1) \leq T(\rho', i). \tag{4.13}$$

Comparing SDDs and TSSDs, an observation can be likened to a binding, whereas a trace corresponds to a witness. There is a small difference, however — a witness is a binding that has *arrived* at a node, whereas a trace ends at the last situation that has already been *matched*. Trace trees play the role of result sets.

4.3.2 Situation Semantics

We can now proceed to define the semantics $[\![U]\!]^\pi_t$ of a situation U at time t as the trace tree generated by U until t, containing the *valid* traces. We can exploit the fact that a TSSD is a directed acyclic graph to recursively derive this trace tree. A situation's semantics depend on the semantics of all previous situations that – both structurally and temporally – came before it, i.e. $\forall t', U' : t' < t, U' \in \mathsf{prefix}(U)$.

As we use a continuous notion of time, there would be infinitely many points in time t' for which we would have to compute the semantics. However, as the states generated by a GTS are discrete and there is only a finite number, namely i, of states that have occured before $\pi[i]$, we can restrict our attention to a finite number of observation points in time. We therefore define $[\![U]\!]^\pi_t := [\![U]\!]^\pi_{T^{-1}(\pi,t)}$, i.e. the semantics for the last state of π reached before t.

We thus have to define the semantics of $[\![U]\!]^\pi_i$. They are computed using a four step process: (1) We need to identify the sets of traces that satisfy all preconditions for U (structural recursion), (2) we need to match U for the candidate sets generated by those traces and compute a result set, (3) we need verify which elements of the result set represent original observations for state $\pi[i]$ (temporal recursion), and (4) we need to generate the appropriate extended traces using the new observations. We now present each of these steps in detail.

Computing valid prefixes. Determining which sets of traces satisfy all preconditions is probably the most complex step, as most syntactical features (connectors, guards, time constraints, forbidden scenarios) need to be treated at this point.

In the following, we treat D, U, π, i, and $t = T(\pi, i)$ as given. In order to be able to match U, there first of all needs to be a set of compatible traces containing a valid trace for every situation $U' \in \mathsf{pred}(U) \setminus \mathsf{inhibit}(U)$. We combine each such set into a new combined trace. The unfiltered set (I) of these combined traces is

then

$$\mathcal{R}^{U,i}_{(I)} := \{\rho \mid (\rho = \bigcup_{\rho_j \in \mathcal{R}'} \rho_j) \wedge \mathcal{R}' \in \{\{\rho_1, \ldots, \rho_m\} \mid (\forall\, 1 \leq j,k \leq m : \rho_j \Uparrow \rho_k) \\ \wedge \forall U' \in \mathsf{pred}(U) \setminus \mathsf{inhibit}(U) : (\exists j : \rho_j \in [\![U']\!]^{\pi}_i)\}\}. \tag{4.14}$$

We first validate the time constraints of the TSSD. We consider the observations in the traces ρ, plus the observation $(U, \emptyset, t)$ serving as a placeholder for any new observation we might make for U at time t to ensure that time constraints for the current situations are also considered. We then require that the maximal time difference between any two subsets of this observation set observe the bounds set by the $delay$ function. The time-filtered set (II) is then

$$\mathcal{R}^{U,i}_{(II)} := \{\rho \mid \rho \in \mathcal{R}^{U,i}_{(I)} \wedge \forall \mathcal{O}_1, \mathcal{O}_2 \in \wp(\mathcal{O}_\rho \cup (U, \emptyset, t)) : \Delta t \in delay(\mathcal{U}_{\mathcal{O}_1}, \mathcal{U}_{\mathcal{O}_2}) \mid \\ \Delta t := |max(\{t \mid (U, \mathcal{C}, t) \in \mathcal{O}_2\}) - min(\{t \mid (U, \mathcal{C}, t) \in \mathcal{O}_1\})|\}. \tag{4.15}$$

We then check whether the traces respect all guards. First of all, we need to check whether any guards involving U and any other U_o in the trace have matched before $\pi[i]$:

$$\mathcal{R}^{U,i}_{(IIIa)} := \{\rho \mid \rho \in \mathcal{R}^{U,i}_{(II)} \wedge \\ \forall (U_o, \mathcal{C}_o, t_o) \in \mathcal{O}_\rho : (\forall U_G \in guard(U_o, U) : \\ (\forall j \mid T^{-1}(\pi, t_o) \leq j < i : [\![S_{U_G}]\!]^{\pi[j]}_{\lambda_{U_G}[\mathcal{C}_o]} = \emptyset))\}. \tag{4.16}$$

The guards for any two observations in the trace that are on the same branch have already been verified earlier, before the second observation was generated. However, we need to check the guards for observations that originate from separate branches that are joined at U, because these have not previously been verified. The guard-filtered set (III) is then

$$\mathcal{R}^{U,i}_{(III)} := \{\rho \mid \rho \in \mathcal{R}^{U,i}_{(IIIa)} \wedge \forall (U_1, \mathcal{C}_1, t_1), (U_2, \mathcal{C}_2, t_2) \in \mathcal{O}_\rho \mid \\ (\mathsf{suffix}(U_1) \cap \mathsf{suffix}(U_2) \cap \mathsf{prefix}(U) = \emptyset) : \\ (\forall U_G \in guard(U_1, U_2) : (\forall j \mid T^{-1}(\pi, min(t_1, t_2)) \leq j \\ < T^{-1}(\pi, max(t_1, t_2)) : [\![S_{U_G}]\!]^{\pi[j]}_{\lambda_{U_G}[\mathcal{C}_o]} = \emptyset))\}. \tag{4.17}$$

Finally, we have check for fulfilled forbidden scenarios that could inhibit new observations. If $\mathsf{inhibit}(U) \neq \emptyset$, we define $\hat{U} := U$ but set $\mathsf{inhibit}(\hat{U}) := \emptyset$ and compute $\mathcal{R}^{\hat{U},i}_{(III)}$ using Equations 4.14–4.17. As we are not excluding the inhibitors this time, the traces in $\mathcal{R}^{\hat{U},i}_{(III)}$ also need to extend the forbidden scenarios. If a

trace is valid for $\hat{U}$, it therefore contains a forbidden trace. The corresponding trace in $\mathcal{R}^{U,i}_{(III)}$, which is the trace for $\hat{U}$ without the observations for the forbidden scenarios, is then not valid for U. The inhibition-filtered set (IV) is then

$$\mathcal{R}^{U,i}_{(IV)} := \{\rho \mid \rho \in \mathcal{R}^{U,i}_{(III)} \wedge \nexists \hat{\rho} \in \mathcal{R}^{\hat{U},i}_{(III)} : \mathcal{O}_\rho \subseteq \mathcal{O}_{\hat{\rho}}\}. \tag{4.18}$$

We then have $\mathcal{R}^{U,i}_{\lambda} := \mathcal{R}^{U,i}_{(IV)}$ as the set of valid prefixes, i.e. traces for which an observation for U and the i^{th} state at time t might exist that is a valid extension of the trace.

Generating candidate sets. We now need to compute the observations for U in the i^{th} state. Each valid prefix $\rho \in \mathcal{R}^{U,i}_{\lambda}$ defines a candidate set that an observation for U could extend. If U has only one predecessor, this candidate set is just the candidate set of the latest observation in ρ. However, if U is at an $\wedge$-join, there are multiple observations with multiple candidate sets that we need to combine. As we have already ensured that the traces for the different branches are compatible, we already know that these candidate sets do not contain conflicting bindings. If the situations in the branches contain only existential quantifiers, there is also just one candidate set as there is just one way to combine the different extensions of each originial witness into a new witness (if $\{(a_1), (a_2)\}$ was extended into $\{(a_1, b_3), (a_2, b_4)\}$ and $\{(a_1, c_8), (a_2, c_9)\}$, the only combination is $\{\{(a_1, b_3, c_8), (a_2, b_4, c_9)\}\}$). If there are universal quantifiers, the result is a set of several alternatives that merely need to contain each required witness at least once in some combination (if $\{(a_1)\}$ was extended into $\{(a_1, d_1), (a_1, d_2), (a_1, d_3)\}$ and $\{(a_1, e_1), (a_1, e_2)\}$, one possibility would be $\{(a_1, d_1, e_2), (a_1, d_2, e_1), (a_1, d_3, e_1)\}$). We define the set of these combinations as

$$\begin{aligned} combine^+(\rho, U) := \{\mathcal{C} \mid \forall U_i \in \mathsf{pred}(U) : ((U_i, \mathcal{C}_i, t_i) = \rho[U_i] \wedge \\ (\forall (n, \xi) \in \mathcal{C} : \exists (n_i, \xi_i) \in \mathcal{C}_i : \xi_i \leq \xi) \wedge \\ (\forall (n_i, \xi_i) \in \mathcal{C}_i : \exists (n, \xi) \in \mathcal{C} : \xi_i \leq \xi))\}. \end{aligned} \tag{4.19}$$

This definition includes some unnecessarily restrictive candidate sets (e.g. $\{(a_1, d_1, e_2), (a_1, d_2, e_1), (a_1, d_3, e_1), (a_1, d_3, e_2\}$ containing a fourth required witness where three would be sufficient). We thus remove those candidate sets for which there already is a less restrictive equivalent and define

$$\begin{aligned} combine(\rho, U) := \{\mathcal{C} \mid \mathcal{C} \in combine^+(\rho, U) \wedge \\ \nexists \mathcal{C}' \in combine^+(\rho, U) : \mathcal{C}' \subset \mathcal{C}. \end{aligned} \tag{4.20}$$

We can now define the set of valid input candidate sets as

$$\mathcal{A}^{U,i}_{\lambda} := \bigcup_{\rho \in \mathcal{R}^{U,i}_{\lambda}} combine(\rho, U). \tag{4.21}$$

We can then evaluate the SDD S_U and have

$$\mathcal{A}_i^U := \{\mathcal{C} \mid \mathcal{C} \in [\![S_U]\!]^{\pi[i]}_{\mathcal{A}_\lambda^{U,i}}\} \tag{4.22}$$

as the result set containing those candidate sets that match U in $\pi[i]$.

Generating observations. Classic events (like messages) occur at a particular time. Situations, however, may have a duration that spans multiple states, which means that the same instance of a situation could be observed multiple times. If we kept generating observations, this would defeat the purpose of time constraints, in most cases. We therefore use the convention that we generate only one observation per distinct match and do so at the earliest possible time. Two matches are either distinct if they are characterized by different candidate sets or if they first occurred at different times — i.e., if a situation (e.g., constraining some attribute) matches, does not match, and matches again, we generate a second observation. We therefore require that a candidate set matches U in $\pi[i]$, but did not match U in $\pi[i-1]$:

$$\mathcal{A}_i^{U+} := \mathcal{A}_i^U \setminus \mathcal{A}_{i-1}^U. \tag{4.23}$$

In state $\pi[0]$, there can be no previous matches; we therefore define $\mathcal{A}_{-1}^U := \emptyset$.

The generated observations are then

$$\mathcal{O}_i^{U+} := \{(U, \mathcal{C}, t) \mid \mathcal{C} \in \mathcal{A}_i^{U+}\}. \tag{4.24}$$

Generating traces. We finally need to extend the prefix traces in $\mathcal{R}_\lambda^{U,i}$ with the appropriate new observations. As we treated all prefix traces together in the previous step (which was required to be able to properly compare $\mathcal{A}_i^U$ and $\mathcal{A}_{i-1}^U$), we now need to pick those observations for each prefix trace which actually evolved from it. We thus have

$$\begin{aligned}\mathcal{R}_i^{U+} := \{\rho.o \mid \rho \in \mathcal{R}_\lambda^{U,i} \ \wedge\ o = (U, \mathcal{C}^+, t) \in \mathcal{O}_i^{U+} \wedge \\ \exists \mathcal{C} \in \mathit{combine}(\rho, U) : \mathcal{C} \sqsubseteq \mathcal{C}^+\}\end{aligned} \tag{4.25}$$

as the new traces generated by U in $\pi[i]$. As the semantics of the situation, we can now define the trace tree of all traces generated by U until $T(\pi, i)$, which is

$$[\![U]\!]_i^\pi := [\![U]\!]_{i-1}^\pi \cup \mathcal{R}_i^{U+}. \tag{4.26}$$

We define $[\![U]\!]_{-1}^\pi := \emptyset$ so that $[\![U]\!]_0^\pi$ is properly computed.

Special situations. Initial nodes match once and immediately and thus do not require complex computations. As the semantics of an initial node λ_D of D, we define

$$[\![\lambda_D]\!]_i^\pi := ((\lambda_D, \{\{((1), \tau)\}\}, 0)) \tag{4.27}$$

for any i.

For a ⦿ node Ω, the result set $\mathcal{A}_i^{\Omega}$ computed in step (2) simply contains all candidate sets $\mathcal{A}_\lambda^{\Omega,i}$ generated in step (1). ⦿ nodes are special because they match as late as possible, which means that they maintain no history and discard previously generated traces as no longer pertinent. This entails two significant changes: The set of generated candidate sets is not filtered for ⦿ nodes so that

$$\mathcal{A}_i^{\Omega+} := \mathcal{A}_i^{\Omega}, \tag{4.28}$$

and the semantics of a ⦿ node are defined as the freshly generated traces *only*:

$$[\![\Omega]\!]_i^{\pi} := \mathcal{R}_i^{\Omega+}. \tag{4.29}$$

Unlike the semantics $[\![U]\!]_i^{\pi}$ of regular situations, the semantics $[\![\Omega]\!]_i^{\pi}$ of a ⦿ node can thus shrink, e.g. because a forbidden scenario is completed or a guard is violated.

The trivially true situation is similar to ⦿ nodes in that the result set contains all candidate sets in $\mathcal{A}_\lambda^{\mathbf{1},i}$ and that the result set is not filtered against the result set for the previous state, i.e. the trivially true matches in every state where its preconditions are fulfilled. For the trivially false situation, $[\![\mathbf{0}]\!]_i^{\pi}$ is always empty.

4.3.3 Scenario Semantics

TSSD semantics. In SDDs, witnesses from the same candidate set can end up at different leaf nodes. In TSSDs, there is no analogon to a candidate set, and the validity of each trace can be decided independently.

As the semantics of a TSSD D, we can therefore simply define the union of the trace trees for all situations of D, i.e.

$$[\![D]\!]_i^{\pi} := \bigcup_{U \in \mathcal{U}_D} [\![U]\!]_i^{\pi}. \tag{4.30}$$

Completeness and uniqueness. The semantics of a TSSD is defined for arbitrary finite prefixes of any given path π. As the diagram only contains a finite number of nodes and thus the suffix of any initial node is finite ($\mathsf{suffix}(\lambda_D) \subset \mathcal{U}_D$), and as the number of states in the considered prefix of π is, by definition, finite, the recursive definition of the semantics always has a natural end point, both in the structural (at some λ_D) and temporal (at state $i = 0$) domain.

The semantics is also unambiguous and unique as there are no non-deterministic choices in the definition. All possible alternative observations are explicitly considered and represented by separate traces.

Root traces. Whenever we extend a trace with an observation for a situation in $\mathsf{triggers}_D$, i.e. a situation that completes a trigger block of D, we add the extended trace to the set of *root traces* $\mathcal{R}_i^{rt[D]}$. We can thus define the set of root traces as

$$\mathcal{R}_i^{rt[D]} := \bigcup_{U \in \mathsf{triggers}_D} [\![U]\!]_i^\pi.$$

TSSD satisfaction. We can now finally define *satisfaction* of a TSSD. A TSSD D is satisfied by a path π at time t, i.e., we have $eval(D, \pi, t) = true$ if

$$\forall \rho^{rt} \in \mathcal{R}_i^{rt[D]} : (\exists \rho_s \in [\![D]\!]_i^\pi : (\rho^{rt} \sqsubseteq \rho_s \wedge \rho_s^{-1}[t] = (U_s, \mathcal{C}_s, t_s) \wedge U_s \in \mathsf{term}_D)),$$

i.e. for each trace in the set of root traces, there needs to be an extension reaching a ⦿ node. This definition covers both existential and universal TSSDs. For existential TSSDs, there will only be one root trace – the root of the trace tree, generated by the initial pseudostate. For universal TSSDs, there will be a root trace for every time a trigger block was completed. Additionally, a checkpoint can double the number of root traces by turning one extension of each original root trace into an extended root trace.

In our evaluation, we are pessimistic: When the satisfaction condition is not fulfilled, we set $eval(D, \pi, t) = false$, not $\bot$, even if there are traces that still might be completed to satisfaction at a later time.

Negation. The negation $\overline{D}$ of a TSSD D can be computed by inverting (1) all triggers, (2) all inhibitors, and (3) conjunction and disjunction.

In order to deal with triggers (1), two preliminary expansion steps are necessary: We have to add a new ⦿ node, connect it directly to an initial node, and add an inhibitor from each situation completing a trigger to pointing it. This makes the semantics that the scenario is satisfied if no trigger is ever completed explicit. We also need to add all trivial trigger blocks, which consist of (a) trigger blocks containing only a single situation for which all $U' \in \mathsf{succ}(U)$ are ⦿ nodes and (b) trigger blocks containing all elements of a forbidden scenario except those that actually have outgoing inhibitors, to the diagram. We can then simply invert the triggers by placing each consecutive sequence that is not inside a trigger block into a trigger block and deleting the existing triggers.

Inverting the inhibitors (2) is effected by simply turning all regular connectors leading to ⦿ nodes into inhibitors and all inhibitors into regular connectors. Note that this has to be performed on the internal, expanded representation.

To achieve (3), all ⦿ nodes of the diagram are joined into a single node. All $\wedge$ join points are then split up by duplicating the suffix, creating new alternatives.

This algorithm for negation does not produce the minimal TSSD for expressing the negated property. The negated scenario contains gratuituous trivial triggers, ⦿ nodes connected to an intial state by an inhibitor which can never match, and tautological statements, which can be removed. Figure 4.49 iterates through an example.

Negation can also trivially be expressed by placing the whole scenario into a single forbidden scenario node.

Example. We now present a small example that illustrates the idea of traces, trace trees, and root traces. Figure 4.50 specifies the property that any process that is *ready*, i.e. not *waiting* for any external resources, must eventually be *running* until it is *terminated*. To keep the candidate sets simple, there are no quantors in the SDDs — the bindings generated by the trigger block never change, while the state is encoded as an attribute.

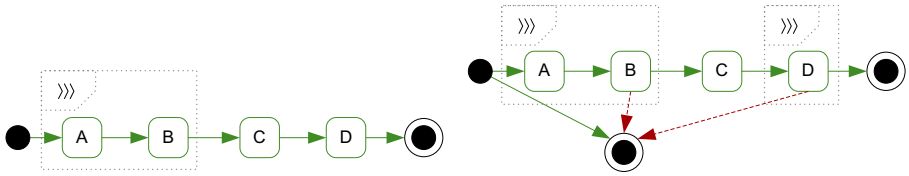
Figure 4.51 shows the states of a path representing a run of the system that we would like to analyze. As is apparent in state $\pi[5]$, both processes enventually terminate. Figure 4.52 presents the trace trees $\mathcal{R}_t$ generated by D when evaluated on π. The nodes represent observations, a trace is a path from the tree root to an observation. The root traces are marked by a bold border around the triggering observation. Their border is drawn dotted and orange while their satisfaction is undecided, solid green once they are satisfied, and dashed red once they have failed. In the tree at $t = 30$, we also use a dashed red border to indicate that this particular trace will never be extended again because the UNTIL requirement was violated as p_a had stopped running. Eventually, an extension of each root trace reaches the ⦿ node so that the TSSD is satisfied.

4.3.4 Subscenario Semantics

Even though they play a similar role, *subscenarios* are conceptually simpler than ESDDs. The three main differences are that trace trees are propagated in a linear fashion *through* the subscenario, that TSSDs already have a mechanism for moving candidate sets between TSSDs, and that termination is not a practical issue.

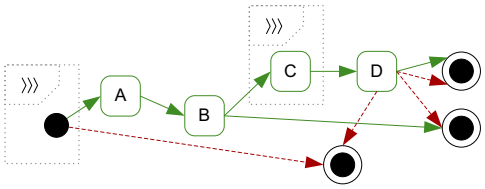
Invoking a subscenario. As a subscenario may export generated bindings, its invocation in the host scenario D occurs in a specific λ situation U_D.[10] A subscenario B can basically be seen as a macro that adds its situations between U_D and its predecessors $\mathsf{pred}(U_D)$. The subscenario's λ situation U_B than extends the witnesses in the candidate sets $\mathcal{A}_\lambda^{U_B,i}$ arriving at U_B with bindings for the roles of B. These

[10] Note that this has nothing to do with D's intial pseudostate λ_D; U_D is a λ-*type* situation.

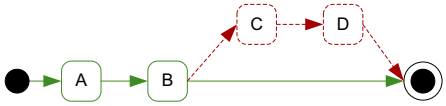


a. D, the original TSSD

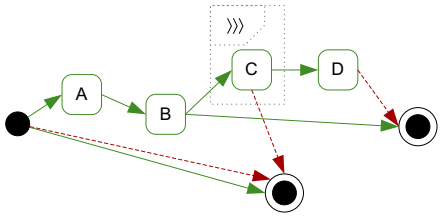
b. D', the equivalent expansion of D



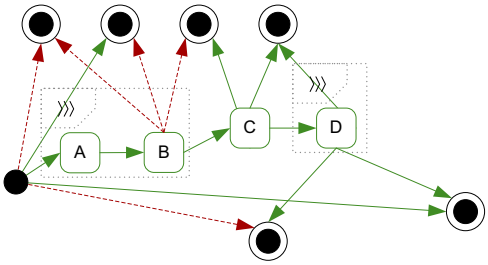
c. $\overline{D'}$, the negation of D'



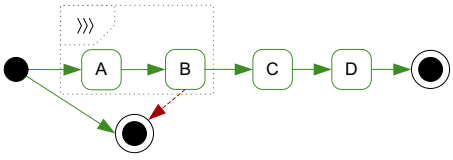
d. $\overline{D}$, the equivalent reduction of $\overline{D'}$, negation of D



e. $\overline{D}'$, the equivalent expansion of $\overline{D}$



f. $\overline{\overline{D'}}$, the negation of $\overline{D}'$



g. $\overline{\overline{D}}$, reduction of $\overline{\overline{D'}}$, negation of $\overline{D}$, equivalent to D

Figure 4.49: Double negation of a universal TSSD

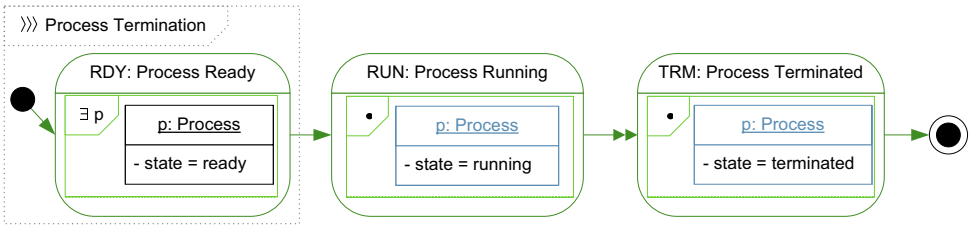
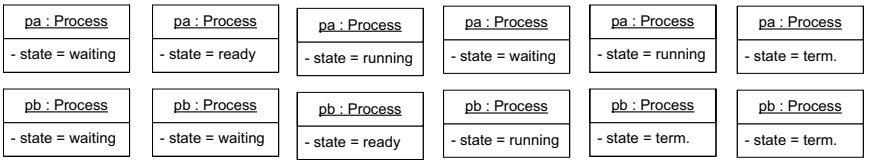


Figure 4.50: Example 1: The TSSD D



a. π at $t = 0$ b. π at $t = 10$ c. π at $t = 20$ d. π at $t = 30$ e. π at $t = 40$ f. π at $t = 50$

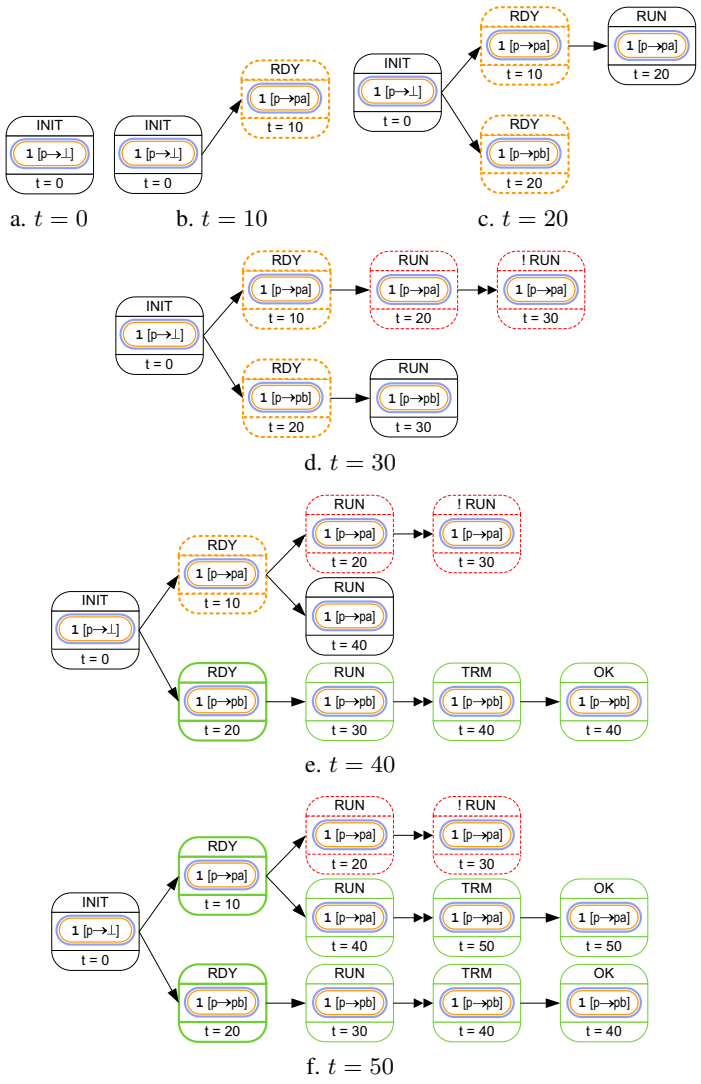
Figure 4.51: Example 1: The path π

bindings are generated from the existing bindings in accordance with the specification of U_D using a rebinding function ℓ_B (from $var(D)$ to $var(B)$), which works exactly like the rebinding function of the λ node of an ESDD. Likewise, subscenario parameters are added to the bindings and can be used in constraints, e.g., as attribute guards in SDDs or as time constraints.

Returning from a subscenario. The extended candidate sets then progress through B normally. When they complete the subscenario and reach U_D, another rebinding function ℓ_D (from $var(B)$ to $var(D)$) is used to extend the witnesses in the candidate sets in $\mathcal{A}_\lambda^{U_D,i}$ with the new bindings that are exported from the subscenario. ℓ_D also erases the bindings for all variables $v \in var(B)$ from the candidate sets, i.e. resets them to $\perp$.

Subscenario instances. When a subscenario is inserted into a TSSD, its situations and roles are qualified with a unique identifier so that multiple instances of the same subscenario are distinct. Qualifying the situations is necessary to ensure that there are no multiple observations for the same situation in the same trace, and to avoid attempts to rebind variables. Qualifying the roles allows a subscenario B to contain recursive references to itself. Recursive definitions can, of course, not be expanded statically, but need to be expanded on the fly as required.

Loops, which represent the most frequent application of recursive subscenarios, can be expressed as tail recursions. The ⦿ node of an iteration is merged with the first situation of the next iteration, while the initial node of a subsequent iteration



a. $t = 0$ b. $t = 10$ c. $t = 20$

d. $t = 30$

e. $t = 40$

f. $t = 50$

Figure 4.52: Example 1: The trace trees generated by D over π

is merged with the last situation of the previous iteration. As a consequence, the guards on both corresponding connectors need to hold between the two situations. Bounded loops can be realized by explicitly unrolling them (for sufficiently small bounds) or by means of parameters that are decreased during each iteration and used as guards in the appropriate situations.

Completeness and uniqueness. On any finite prefix of a given path π, the semantics of a TSSD containing invocations of non-recursive subscenarios are uniquely defined as the number of contained situations is guaranteed to be finite. The semantics of a TSSD containing an invocation of a recursively defined subscenario are equally uniquely defined if the subscenario encodes any kind of progress, i.e., if the definition requires a state change (e.g. because two of its situations are mutually exclusive) or includes a time constraint with a non-zero lower bound, as the subscenario can only be observed on a finite prefix of π a finite number of times. In order to ensure that the semantics definition still holds for a subscenario definition that contains no such constraints and could thus endlessly match the same state (e.g. by defining only a single non-λ situation), we have to automatically insert such a non-zero time constraint between iterations that will prevent the recursive definition of the semantics from evaluating the same state more than once. This does not reduce the expressiveness of the notation, as the specified property is actually structural if no progress is required and can therefore be specified using a single situation containing an invocation of a recursive ESDD, for which we guarantee termination.

While such a semantics could be defined, we do not define a semantics for infinite paths on which a recursive subscenario loops forever as the corresponding situations are actually observed over and over infinitely often.[11] As a trace is currently only considered valid if it has reached a ⦿ node, only finitely many iterations of any subscenario are possible in a valid trace. As a consequence, a well-formed subscenario needs to contain at least one branch without a recursive invocation to ensure that termination is at all possible.

4.3.5 Expressiveness

In Section 2.2.5, we have discussed how temporal properties of graph transformation systems can be specified based on CTL*. The discussion in [16] proves that it is possible to define a sound propositional calculus whose elementary propositions are based on graph patterns. However, in order to discuss the expressiveness of TSSDs, we additionally need a calculus that includes a concept of time and, in particular, intervals.

[11] Note, however, that it is nonetheless possible to specify that some finite scenario should occur infinitely often using appropriate triggers.

Linear Temporal Logic (LTL). Linear Temporal Logic (LTL) is a subset of CTL* that is restricted to a single path quantor, an implied initial $\mathbf{A}$. Given a GTS Y, a set of graph patterns $\mathcal{P}$ and a set of possible bindings $\mathcal{X}_{\mathcal{P}}[\mathcal{G}[\mathcal{T}_Y]]$, with pattern $P \in \mathcal{P}$, binding $\xi \in \mathcal{X}_{\mathcal{P}}[\mathcal{G}[\mathcal{T}_Y]]$, and $G \in \mathcal{G}[\mathcal{T}_Y]$, we define LTL for GTS as follows:

- A graph predicate $P|_\xi \precsim G$ and the constants $true$ and $false$ are valid LTL formulas,
- if φ is a valid LTL formula, so is $\neg\varphi$,
- for two valid LTL formulas φ and φ', $\varphi \wedge \varphi'$ is a valid LTL formula,
- for a valid LTL formula φ, $\mathbf{X}\ \varphi$ is a valid LTL formula,
- for two valid LTL formulas φ and φ', $\varphi\ \mathbf{U}\ \varphi'$ is a valid LTL formula.
- For convenience, the derived operators $\mathbf{F}$, $\mathbf{G}$, and $\mathbf{R}$ are also provided so that for two valid LTL formulas φ and φ', $\mathbf{F}\ \varphi$, $\mathbf{G}\ \varphi$, and $\varphi\ \mathbf{R}\ \varphi'$ are valid LTL formulas.

The semantics of LTL for GTS are then defined as:

- $Y, G \models \varphi$ iff φ is $true$.
- $Y, G \models \varphi$ iff φ is a graph predicate and $P|_\xi \precsim G$.
- $Y, G \models \neg\varphi$ iff $Y, G \not\models \varphi$.
- $Y, G \models \varphi \vee \psi$ iff $Y, G \models \varphi \vee Y, G \models \varphi$.
- $Y, G \models \varphi \wedge \psi$ iff $Y, G \models \varphi \wedge Y, G \models \varphi$.
- $Y, \pi \models \varphi$ iff $G = \pi[0] \wedge Y, G \models \varphi$.
- $Y, \pi \models \neg\varphi$ iff $Y, G \not\models \varphi$.
- $Y, \pi \models \varphi \vee \psi$ iff $Y, \pi \models \varphi \vee Y, \pi \models \varphi$.
- $Y, \pi \models \varphi \wedge \psi$ iff $Y, \pi \models \varphi \wedge Y, \pi \models \varphi$.
- $Y, \pi \models \mathbf{X}\varphi$ iff $Y, \pi^1 \models \varphi$.
- $Y, \pi \models \varphi\mathbf{U}\psi$ iff $\exists k \mid k \geq 0 : Y, \pi^k \models \psi \wedge \forall j \mid 0 \leq j < k : Y, \pi^k \models \varphi$
- $Y, \pi \models \mathbf{F}\varphi$ iff $Y, \pi \models true\mathbf{U}\varphi$.
- $Y, \pi \models \mathbf{G}\varphi$ iff $\neg Y, \pi \models \mathbf{F}\neg\varphi$.
- $Y, \pi \models \varphi\mathbf{R}\psi$ iff $\neg Y, \pi \models \neg\varphi\mathbf{U}\neg\psi$.

TSSDs and LTL. We can now compare the expressiveness of LTL for GTS and TSSDs. [12]

Theorem 4.1 *Timed Story Scenario Diagrams over a given path π are at least as expressive as Linear Temporal Logic for GTS over π.*

[12] As various theorems proving that any LTL formula can be expressed by first-order logic exist, we could conjecture based on Section 3.3.6 that any LTL formula could already be expressed by a single SDD. However, this is not possible because an SDD is limited to a single graph G as its argument, i.e. all nodes are implicitly evaluated on the same graph. To overcome this, we could, for each state graph of a path, add an attribute indicating the corresponding state to each node and join the state graphs into a single graph. However, such an abuse of notation would forfeit all claims to intuitiveness.

Proof. We again prove this constructively by showing that for any LTL formula, there exists an equivalent TSSD. This construction is mostly trivial, as there is an equivalent for each of the fundamental concepts of LTL in TSSDs, with the exception of the $\mathbf{X}$ operator. As TSSDs are designed for use with a dense time model, time constraints are typically of the much greater practical relevance then a concept of a *next* state.

- $P|_\xi \precsim G$: As any graph predicate can be encoded by means of an SDD, $P|_\xi \precsim G$, $true$, or $false$ can be encoded as a situation containing the corresponding SDD.
- $\neg\varphi$: If φ is encoded by D, $\neg\varphi$ is encoded by $\overline{D}$.
- $\varphi \wedge \varphi'$: If φ_1 and φ_2 are encoded by D_1 and D_2, $\varphi_1 \wedge \varphi_2$ can be written using two scenario situations containing D_1 and D_2 that are connected to the same $\circledcirc$ node.
- $\mathbf{X}\,\varphi$: As the temporal operators are expressed as connectors in TSSDs, there are no unary operators. However, if φ is encoded by D, $\mathbf{X}\varphi$ can be expressed by connecting a *strictly previous* trivially *true* situation that serves as the first operand to D using an *eventually* connector (which yields the classic encoding of $\mathbf{X}$ as $false\ \mathbf{U}\ \varphi$).
- $\varphi\ \mathbf{U}\ \varphi'$: If φ_1 and φ_2 are encoded by D_1 and D_2, $\varphi_1\ \mathbf{U}\ \varphi_2$ can be written using two scenario situations containing D_1 and D_2 that are connected by an *until* connector.
- $\mathbf{F}\,\varphi$, $\mathbf{G}\,\varphi$, and $\varphi\,\mathbf{R}\,\varphi'$: Using the above defintions, the derived operators can be defined in the same way as for LTL. $\mathbf{F}\,\varphi$ does not need to be derived, as it is supported directly by means of the *eventually* connector. $\mathbf{G}\,\varphi$ can also be written as $\longrightarrow\!\triangleright\ \varphi \longrightarrow\!\!\blacktriangleright\!\!\blacktriangleright\ \circledcirc$. □

Metric Temporal Logic (MTL). An extension of LTL that allows time constraints in the form of intervals for temoral operators is MTL (cf. [29],[2]). All temporal operators in MTL are defined in terms of the $\mathbf{U}$ operator. The $\mathbf{X}$ operator is subsumed by the $\mathbf{F}$ operator as the concept of a next state is not meaningful on dense time domains. MTL is very expressive and, e.g., allows encoding the halting problem. Satisfiability of MTL formula is undecidable.

TSSDs and MTL. Any valid MTL formula can be written as an equivalent TSSD:

Theorem 4.2 *Timed Story Scenario Diagrams over a given path π are at least as expressive as Metric Temporal Logic for GTS over π.*

Proof. As any LTL formula can be written as a TSSD, we only need to show that the extensions introduced by MTL, namely time constraints on operators, can

be expressed using TSSDs. As TSSDs directly support time constraints on the temporal connectors that encode the temporal operators, this is trivial. □

Time Point Temporal Logic (TPTL). Another extension of LTL with time constraints is TPTL (cf. [3]). TPTL introduces the concept of clocks which, in a given state, can be defined (and set to 0) or compared with the a given interval. TPTL is strictly more expressive than MTL (cf. [7]).

TSSDs and TPTL. Any valid TPTL formula can be written as an equivalent TSSD:

Theorem 4.3 *Timed Story Scenario Diagrams over a given path π are at least as expressive as Time Point Temporal Logic for GTS over π.*

Proof. Again, we only need to show that the extensions introduced by TPTL can be expressed using TSSDs. TPTL can only compare times for temporally ordered states, which corresponds to situations on the same branch in a TSSD. Defining a clock in a state can then be represented by attaching one end of a constraint edge to the corresponding situation, while a reference to the clock in a subsequent situation is encoded by attaching the other end of the constraint edge to it. The interval for comparison is than placed on the constraint edge as a time constraint. □

Conclusion. We have shown that TSSDs are very expressive in the temporal domain. They also meet the criteria for a temporal logic for real-time system specification proposed in [5]: They are based on first-order logic, prohibit quantification on time variables, have a metric for time, use the interval as the fundamental time entity, support a time model that is based on relative time (though absolute time is also available as the time relative to the initial node), and provide a limited number of basic operators that can be composed into reusable specialized building blocks by means of subscenarios.

Finally, with their integrated support for structural properties and their quantificaton, TSSDs go beyond the scope of other existing temporal logics in this respect.

4.4 Property Detectors

As for structural properties, we would like to derive a *property detector* for monitoring the specified temporal properties. Again, we have the option of using a model checker or generating source code.

4.4.1 GROOVE

While implementing runtime monitors supporting the full TSSD semantics is possible using GROOVE, the size of the resulting GTS is significant and prohibitive for a manual prototype implementation. In particular, as GROOVE neither supports time nor attributes which could have been used to encode time directly, auxiliary constructs would have been needed to cover these aspects.

Runtime monitor prototype. In principle, verifying temporal properties is remarkably similar to verifying structural properties with ESDDs at the operational level. Based on our prototype SDD detector, we can implement an approximation of the intended semantics. It is only an approximation because we do not generate the entire result set for each situation, but only one candidate set. We also exclude time constraints.

Even though we do not deal with time explicitly, we have to introduce a marker representing a time step or state that inhibits all SDD and TSSD rules in order to give the system the opportunity to actually evolve between subsequent matching attempts of the evaluation rules. For each state of the system, we start evaluation at the initial pseudostate of the TSSD and progress down the diagram. As we only match one alternative, the prototype only works properly for existential TSSDs. The only root trace marker is thus placed at the initial node. Whenever a valid candidate set has been propagated down to a situation's SDD root marker by the corresponding SDD detector, an observation is generated. New root markers are then created for all enabled subsequent situations (connected by edges labeled with **F**, **I**, and **U** representing connectors) and the candidate set is copied to the new root markers. The detector signals satisfaction as soon as a trace reaches a termination node; the result is then propagated back down the **F**, **I**, or **U** edges to the root trace marker, and all root markers are deleted.

Figure 4.53 shows the evaluation for the scenario in Figure 4.1 at the time when the evaluation has just reached the success node.

Verification. Apart from the technical limitations of the prototype, the underlying approach is only suited for monitoring the conformance of a single execution path of the system, but not for model checking an entire GTS with respect to a specification. As GROOVE was designed to perform CTL model checking (cf. [35]) with a focus on reachability, there is no inherent support for the verification of path formulas as encoded by TSSDs. In [38], a different approach is chosen: GTS are mapped to the input format of a standard LTL model checker, which might provide a viable option for verifying diagrams that are restricted to the subset of the TSSD syntax that can be mapped to LTL.

Graph model checking is faced with the general problem that the reachable state

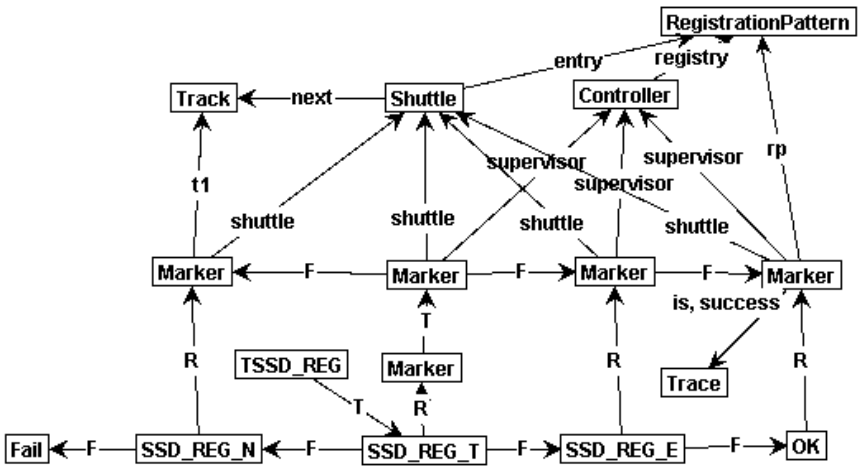


Figure 4.53: *Register Shuttle*: Scenario matched

space of a GTS is typically infinite as new elements can be added to the system by transitions. While there are approaches for verifying certain invariants of such infinite systems (e.g. [4]), there is no general solution. In any case, the inclusion of structural aspects will always greatly increase the size of the state space and thus limit the size of problems that can be treated efficiently with current technology.

The support for dense time domains also increases the size of the state space. Instead of a single path consisting of a countable number of discrete states as for runtime monitoring, verification needs to account for infinitely many possible different timings for the same sequence of states.

On the other hand, the ability to specify real time constraints also helps to limit the size of the problem. When a TSSD is evaluated on an infinite path, an enabled situation may keep generating inifinitely many new alternative observations. Meanwhile, a subsequent situation connected by means of an eventually connector may not yet been observed – and may never be observed at all. A similar problem occurs when the termination condition of a recursive subscenario is never observed, as the subscenario continues to generate new observations for each iteration. Only if the scenario is bounded, i.e. there is a time bound constraining the eventually connector respectively the subscenario, the question becomes decidable using finite resources.

4.4.2 Code Generation

In order to generate proper code for TSSD runtime monitoring, we need to build on the extended version of the SDD detector that actually generates result sets. We

have therefore not yet implemented a TSSD detector in code. When building such a detector, we can closely build on the formal semantics, but need to introduce some optimizations in order to make actual runtime monitoring feasible.

Evaluation frequency. The first question is when to evaluate the TSSD. In practice, we can only observe the system in intervals Δt, which can lead to many unnecessary computations if Δt is too small, and to missing intermediate states if Δt is too large. The optimal sampling rate will depend on the available computational resources and the specific application. However, we can only ever approximate the formal semantics, which do not face such problems.

The ideal solution would be if some type of property change mechanism was available that could trigger the evaluation every time the system state changes. We would then be able to exactly reproduce the formal semantics, evaluating the TSSD exactly once per distinct system state.

Evaluation order. The semantics of a situation were defined recursively. In an implementation, we would, for each step, start evaluation at the initial pseudostate and progress down the TSSD graph in a depth first traversal. There would be one shared trace tree that is extended by all situations in turn. The situation matches $\mathcal{A}_i^U$ would be stored, used to compute the new matches in the next step, and then overwritten by the then current matches, i.e. we would keep a match history that is one step deep.

Dead traces. In the formal semantics, we always consider all previous traces as potential prefixes. This is obviously inefficient in practice, as there are many traces that cannot possibly be extended ever again. If in a trace ρ ending in observation $\rho[U]$ at time t, there is some observation $\rho[U']$ and some situation $U'' \in \mathsf{suffix}(U)$ so that the upper bound of $delay(\{U'\}, \{U''\}$ is less than the time elapsed between $\rho[U']$ and t, any extension of this trace leading towards U'' is already dead as it can never be completed. If there is no alternative to passing through U'', the whole trace is dead. Likewise, once a guard or an until-requirement has been violated, a trace can never be extended again. A trace reaching a strict situation also dies for all situations between the strict situation U and the last branching point in $\mathsf{prefix}(U)$, as no alternative observation of the strict situation is permitted.

We therefore mark traces as dead as soon as we have determined that they can never be extended again. As there may be alternative possible extensions, we need to mark the traces as dead in the next required situation, not its current situation. Only if a trace at situation U is dead in all situations in $\mathsf{succ}(U)$, we can mark it as dead in U.

Dead trace subtrees. If a trace ρ is dead and all its extensions $\rho' \sqsupseteq \rho$ are dead

as well, the whole trace subtree is dead. If the subtree for a root trace is dead, the TSSD can never be successfully completed, and we can stop the evaluation and signal failure.

Time constraints. It is not feasible to check the time constraints on any two subsets of a trace's observations. For one, it is not necessary to check constraints for linear sequences of observations that are all in the past, as those constraints already need to hold. Furthermore, we will restrict evaluation to those constraints that are actually defined.

Branches. In the formal semantics, we merge concurrent branches first before determining whether they constitute a valid prefix. In practice, we can greatly reduce the number of potential prefixes by checking whether each branch by itself is already dead. In the presence of a first of / last of time constraint, we can run a preliminary check to see whether the branch by itself already violates the specified upper bound (though not the lower bound).

Guards. Checking guards should be performed last, as it is computationally more expensive than validating time constraints, which already may eliminate a large number of traces. As we mark a trace as dead as soon as a guard is violated, we can assume that the guards for every live trace hold for all previous states and thus only have to evaluate them for the current state, plus all guards spanning two branches that are merged in the current situation.

Root traces. Once an extension of a root trace has reached a ⦿ pseudostate and there are no guards or forbidden scenarios that might still invalidate the trace, the whole subtree can be marked as dead, as additional observations will no longer affect the satisfaction condition for the trace.

Outlook. Using these optimizations, evaluation of TSSDs will still be computationally expensive (considering that matching each situation by itself can already require extensive computations, even though most contained SDDs will be rather simple), but should be feasible in practice — if not in real-time for production systems, then at least for verification and run-time monitoring of simulated prototypes.

5 Specification of Structural and Temporal Properties

After defining the syntax and semantics of our visual languages for the description of structural and temporal properties, we now focus on their usefulness for encoding informally specified requirements. While we have already provided several examples when introducing the syntax, we discuss the systematic process of deriving such specifications from informal textual requirements in the following section.

5.1 Specification Pattern System

The Property Specification Pattern System presented in [12] and extended in [13] was proposed to address the problem of making formal specification techniques and thus formal verification accessible to practitioners, as even experts face problems encoding moderately complex real-life properties using temporal logics such as LTL. The intention behind the Specification Pattern System is to allow users to construct more complex properties from basic, assuredly correct building blocks by providing generic specification patterns encoding certain elementary properties (existence, absence, universality, bounded existence, precedence (chains), and response (chains)), each specialized for a set of different scopes (globally, before R, after Q, between Q and R, after Q until R).

In the following, we demonstrate how the patterns of the Specification Pattern System can be encoded using Timed Story Scenario Diagrams. A convenient quality of TSSDs is that they allow us to define the scopes and the properties separately as orthogonal concepts and then simply plug the appropriate property into the desired scope.

Scopes. In Table 5.1, we define the scopes as TSSDs. The original textual specification of the patterns is somewhat ambiguous – φ exists before R could be interpreted in two ways: φ needs to exist before (possibly) R is observed (putting the emphasis on *exists* φ), or φ needs to exist whenever R is observed afterwards (emphasizing *before* R). The latter is the interpretation that is encoded by the provided LTL pattern. The scopes *before*, *after*, *between*, and *until* are thus encoded using trigger blocks where φ is the triggered scenario. As the table shows, all definitions except the definition of *until* are very compact. The last case requires an additional $\odot$ node because TSSDs provide no direct encoding of for the operator $\tilde{\mathbf{U}}$ (weak until) so that the property that R may occur or not needs to be encoded explicitly. This omission is intentional as we believe that, in the context of a scenario notation, it is more intuitive to explicitly specify that the scenario might be successfully completed in an earlier situation using the standard syntax for completion ($\odot$) instead of introducing a dedicated syntax for a $\tilde{\mathbf{U}}$ connector.

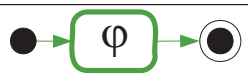
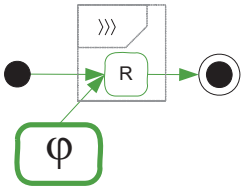
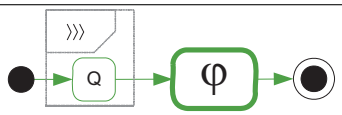
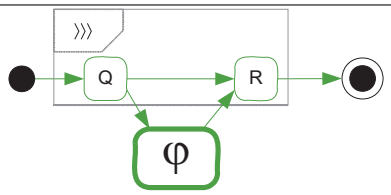
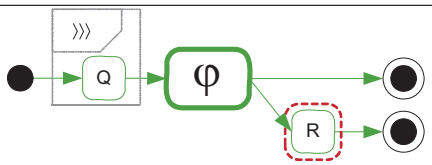
Scopes	
globally	
before R	
after Q	
between Q and R	
after Q until R	

Table 5.1: The scopes encoded as TSSDs (for a property φ)

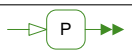
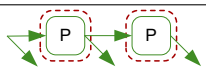
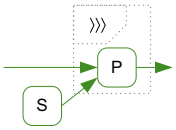
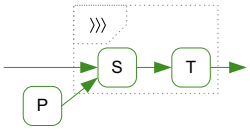
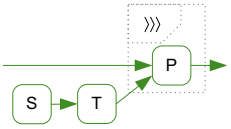
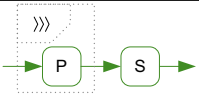
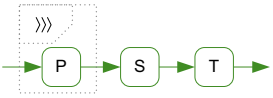
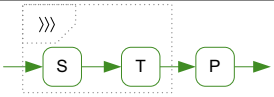
Properties		
Existence	exists P	
Absence	no P	
Universality	always P	
Bounded Existence	exist at most 2 P	
Precedence	S precedes P	
Precendence Chain $1 \rightarrow 2$	P precedes S, T	
Precedence Chain $2 \rightarrow 1$	S, T precedes P	
Response	S responds to P	
Response Chain $1 \rightarrow 2$	S, T responds to P	
Response Chain $2 \rightarrow 1$	P responds to S, T	

Table 5.2: The properties φ encoded as TSSDs)

Properties. In Table 5.2, we define the ten different properties listed by the Specification Pattern System. Inbound connectors link to possible preconditions, outbound connectors encode success and lead to possible postconditions. *Existence*, *absence*, and *universality* are trivially encoded using the standard syntax for required and forbidden scenarios. *Bounded existence* is encoded by enumerating the acceptable sequences, i.e. 0, 1, or 2 occurences. As the number of occurences is relevant, all situations are strict so that no additional occurences are permitted between the observations of a trace. Again, the weak progress (no occurence of P is also acceptable) is encoded by additional outbound connectors. When it comes to encoding response and precedence chains, the notation excels – quite unsurprisingly, as this is the use case for which it was designed. Triggers are designed for expressing response (and its dual, precedence), while sequences such as S, T are *the* basic concept in TSSDs.

Derivation. These property definitions can now simply be substituted for φ by completing them with an intial node as their precondition and ⦿ nodes as their postcondition(s). The following tables list the trivial form of each combined pattern, whose size and complexity is already at an acceptable level, in the center column in order to prove that this very systematic and mechanistic approach already yields useable results. In the right column, we also provide a simplified version that can be derived using two simple transformations that basically correspond to the elimination of redundant parentheses in mathematical expressions: A scenario situation with a single ⦿ node can be eliminated by connecting each situation inside the scope whose predecessor is the scope's intial node to each of the scope's predecessor nodes, and by connecting each situation inside the scope whose successor is the scope's ⦿ node to each of the scope's successor nodes. Secondly, if both the surrounding scenario and the scenario situation contain trigger blocks, these blocks are merged.

Existence. Table 5.3 contains the encodings for *existence*.

Absence. Table 5.4 contains the encodings for *absence*.

Universality. Table 5.5 contains the encodings for *universality*. The scenario situation in the $before$ case cannot be eliminated (unless the past triggered scenario is rewritten using the less elegant internal encoding) as the immediately connector is required to indicate that P should hold from the very beginning, but there is no predecessor of the scenario situation to connect it to (connecting it to the scenario's initial node would turn P from a triggered past scenario into a precondition of R).

Bounded existence. Table 5.6 contains the encodings for *bounded existence*. As we actually want to constrain the total number of occurences of P, the strict situations of the property need to be globally strict so that the interval before the first

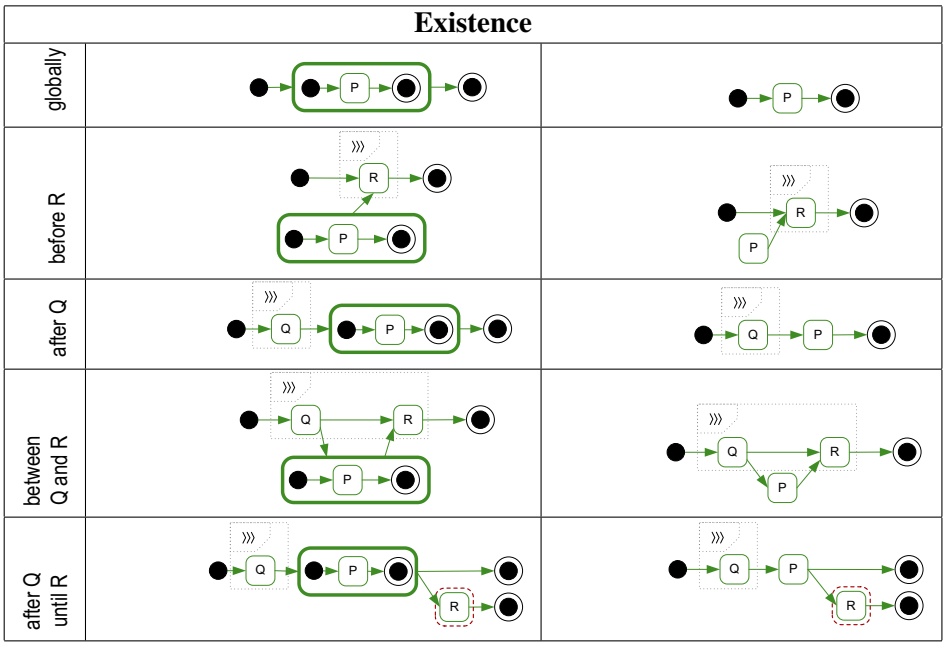


Table 5.3: Existence, trivial and simplified patterns

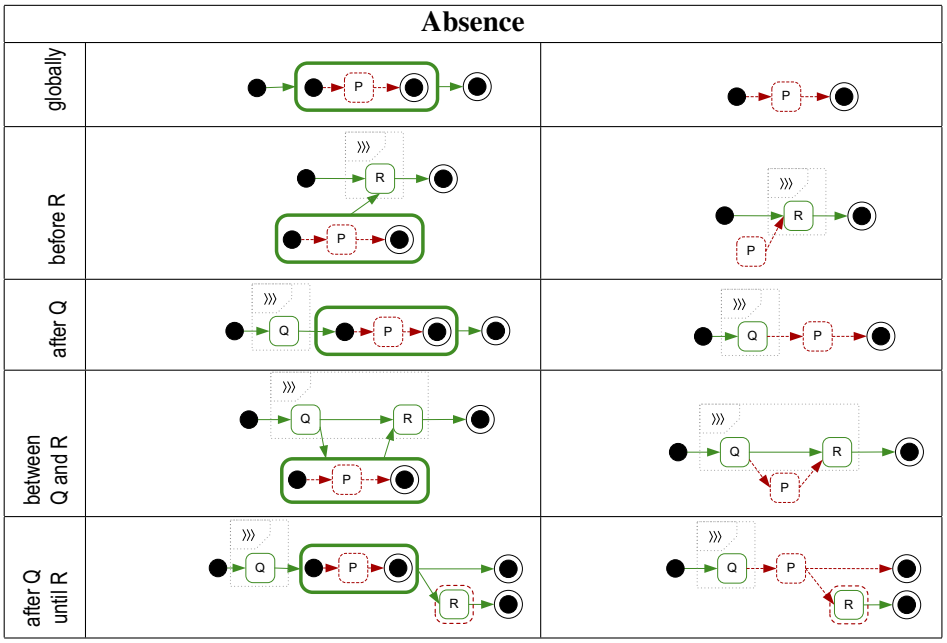


Table 5.4: Absence, trivial and simplified patterns

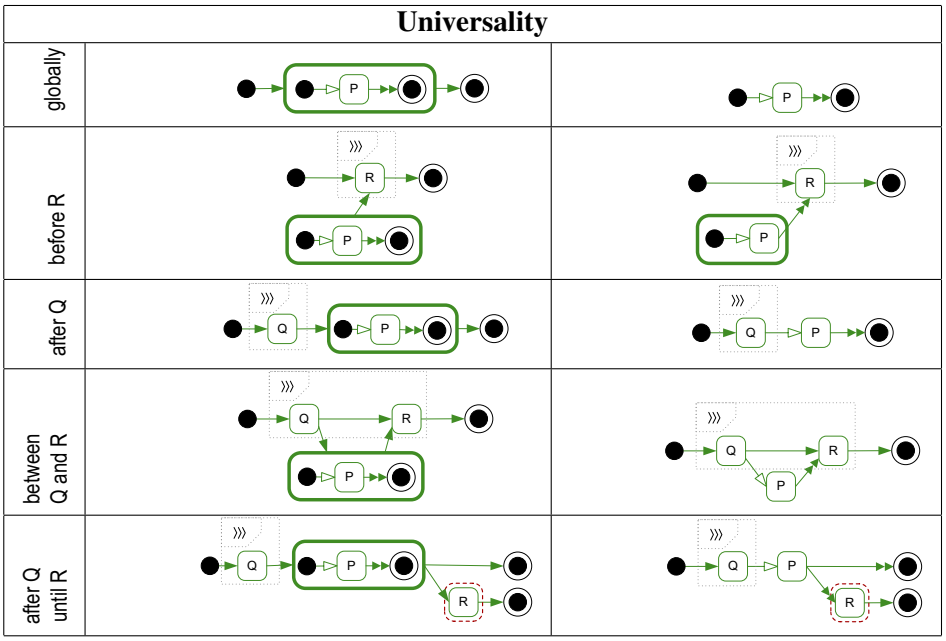


Table 5.5: Universality, trivial and simplified patterns

and after the last occurence is also constrained. Eliminating the scenario situations is not trivial in this case as they contain alternative ⦿ nodes. For the subsequent scenario, this corresponds to an $\vee$-join, which is not directly supported by the syntax. The only trivial way to eliminate the scenario situation is thus to replicate the suffix. There is an alternative, however. While we generally believe that writing positive scenarios (what should be) comes more naturally than writing forbidden scenarios (what should *not* be) for most properties, it is rather straight-forward to encode 'at most two instances' as 'not three or more instances'. Using this approach (which again employs a very direct encoding of what should not be, namely three times P in a row), we can obtain more compact encodings. Another option that does not eliminate the scenario situation but may greatly reduce the size of the diagram, in particular for bounds larger than 2, is to place the bounded property inside a bounded loop with constraints $[0..2]$. Table 5.7 lists the corresponding encodings.

Precedence. Table 5.8 contains the encodings for *precedence*. Table 5.9 contains the encodings for *precedence (1,2)*, i.e. a sequence S, T preceded by P. Table 5.10 contains the encodings for *precedence (2,1)*, i.e. P preceded by a sequence S,T.

Response. Table 5.11 contains the encodings for *response*, i.e. S responds to P. Table 5.12 contains the encodings for *response (1,2)*, i.e. the sequence S, T responds to P. Table 5.13 contains the encodings for *response (2,1)*, i.e. P responds to the sequence S,T.

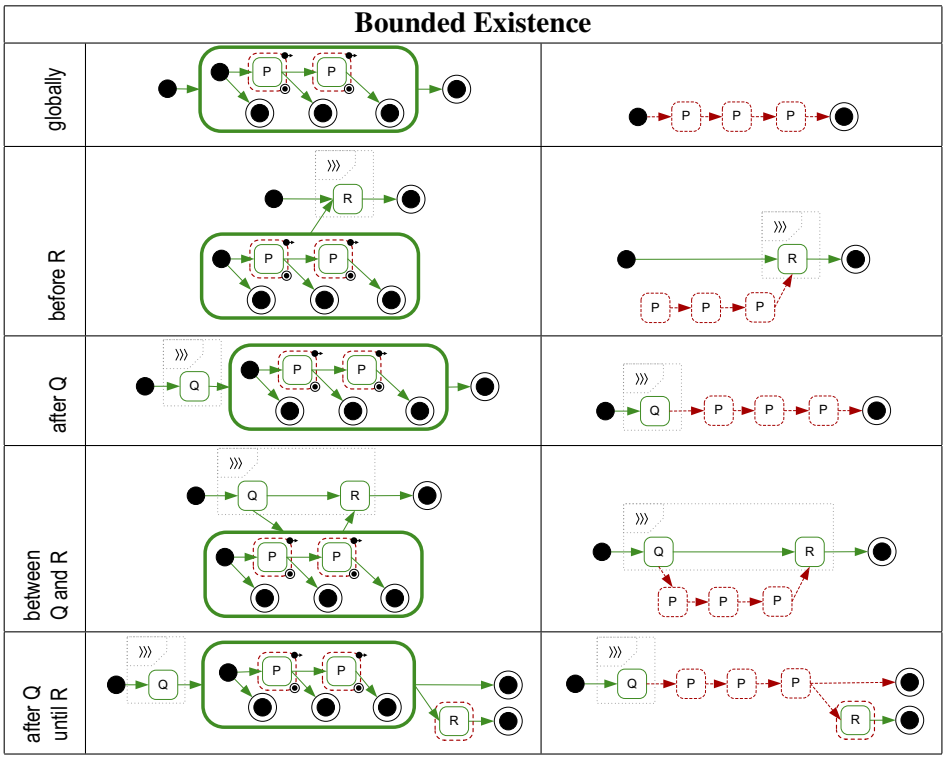


Table 5.6: Bounded Existence, trivial and simplified patterns

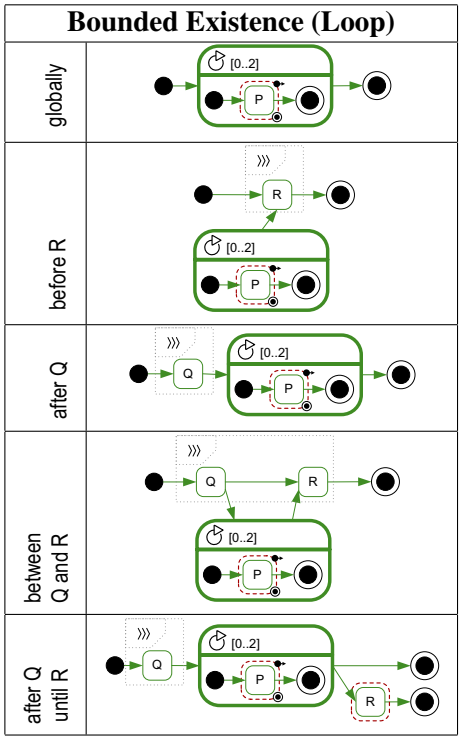


Table 5.7: Bounded Existence expressed using a loop

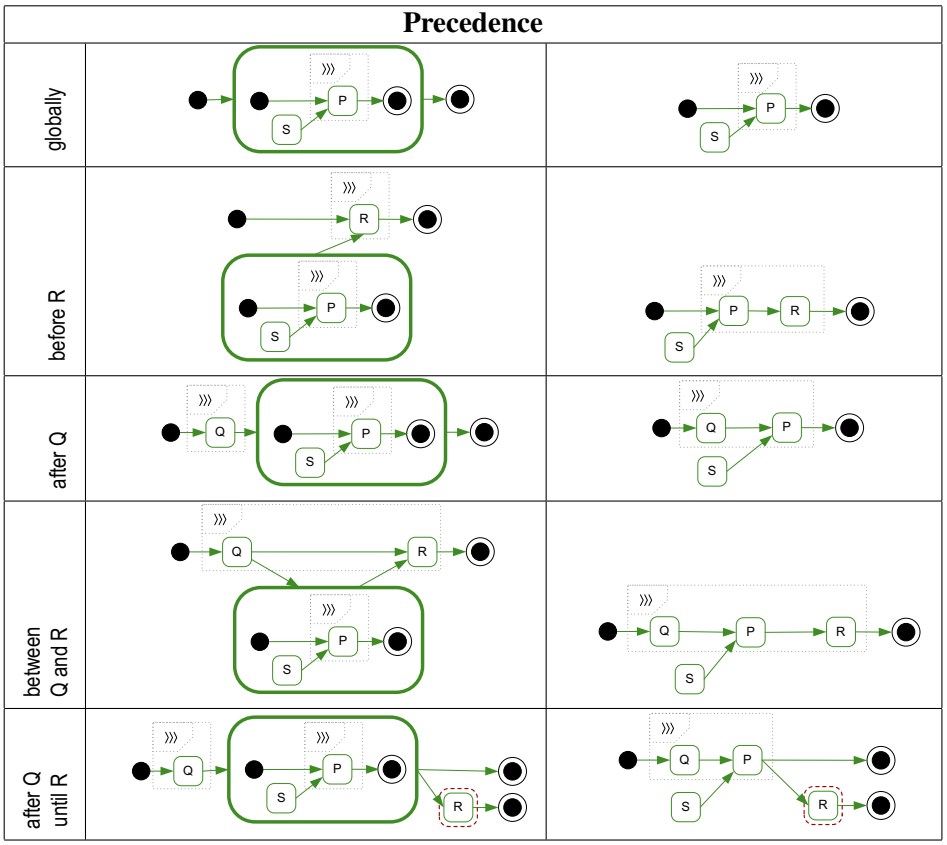


Table 5.8: Precedence, trivial and simplified patterns

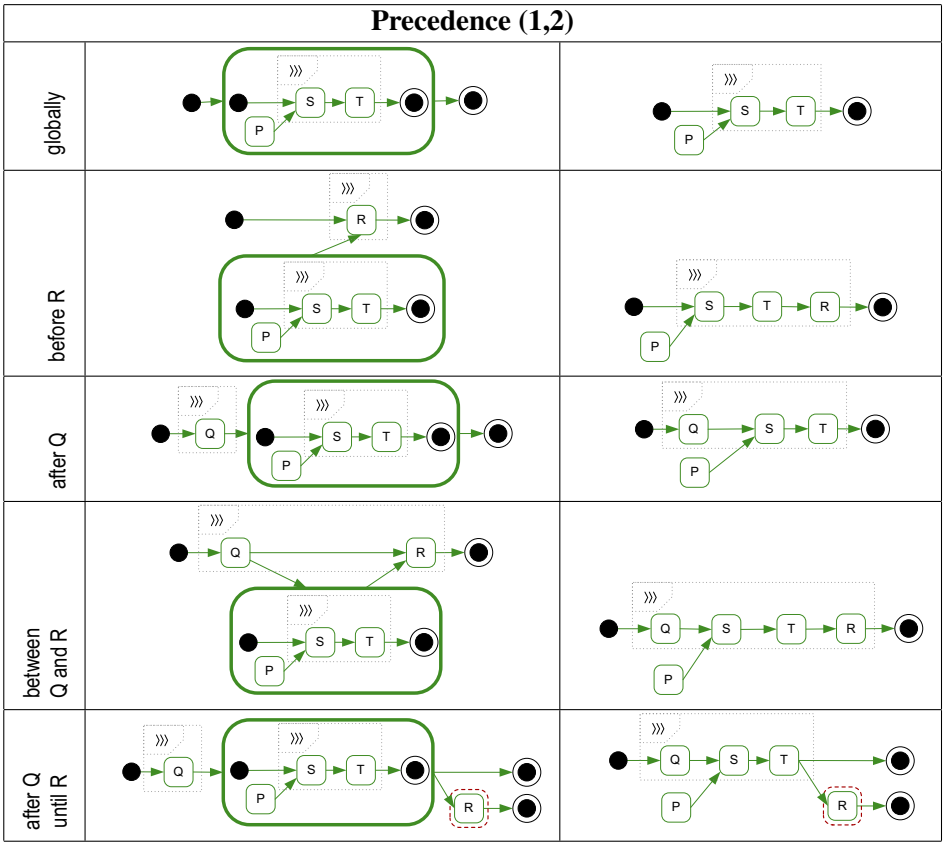


Table 5.9: Precedence (1,2), trivial and simplified patterns

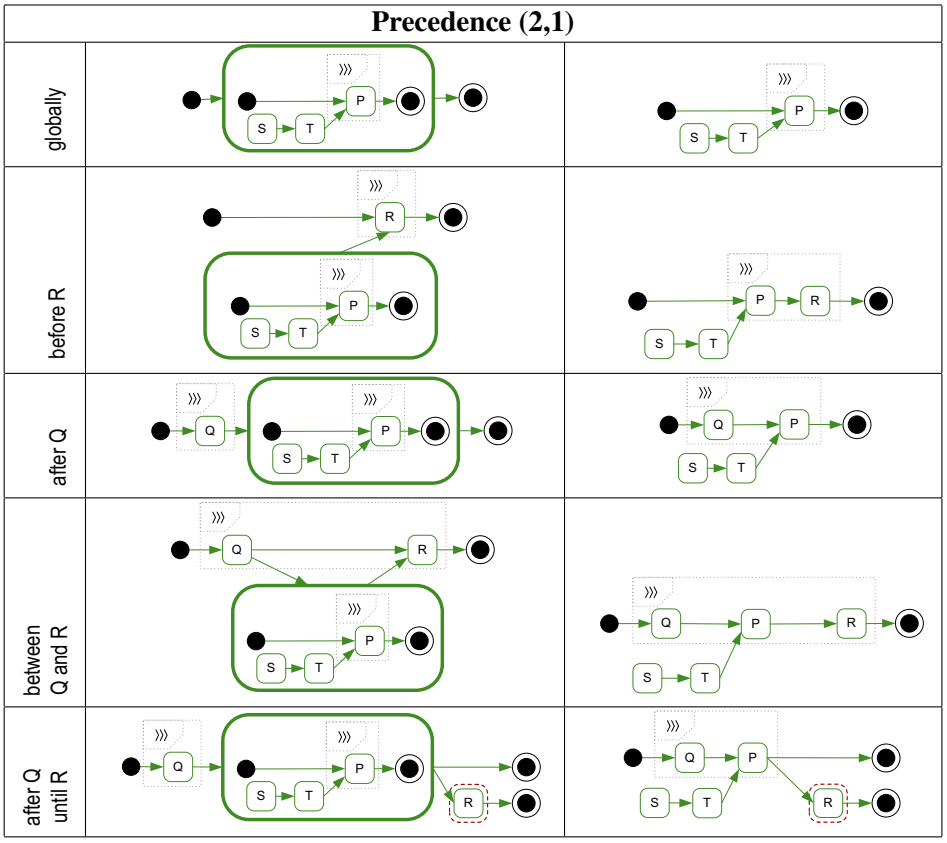


Table 5.10: Precedence (2,1), trivial and simplified patterns

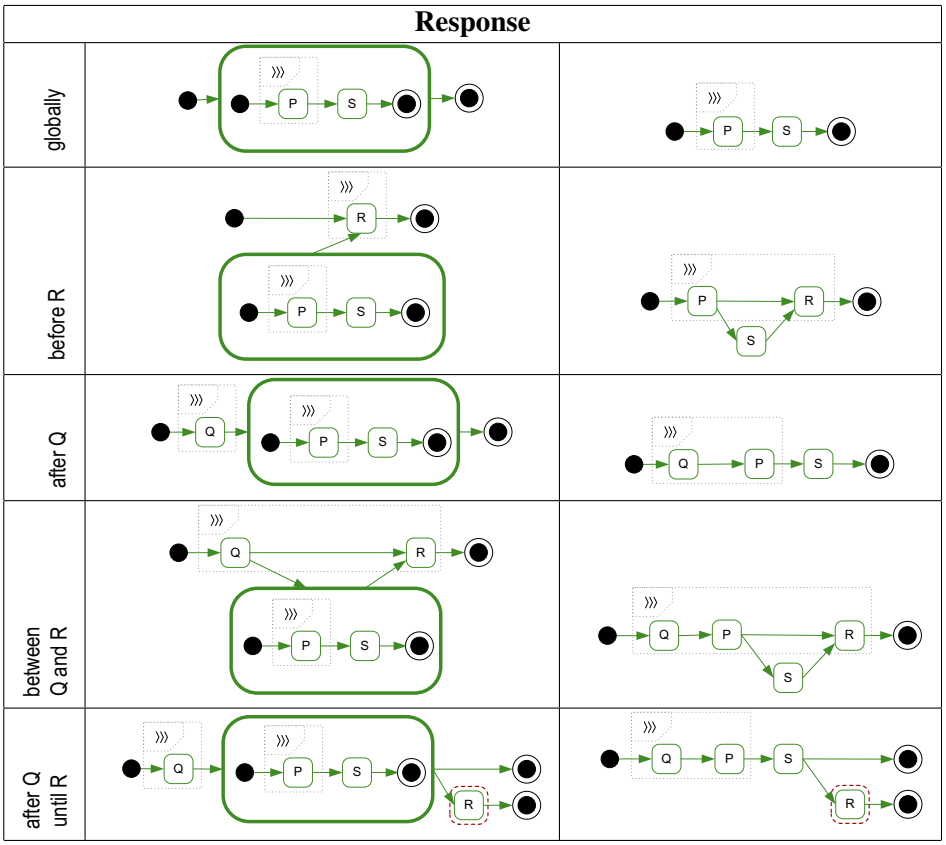


Table 5.11: Response, trivial and simplified patterns

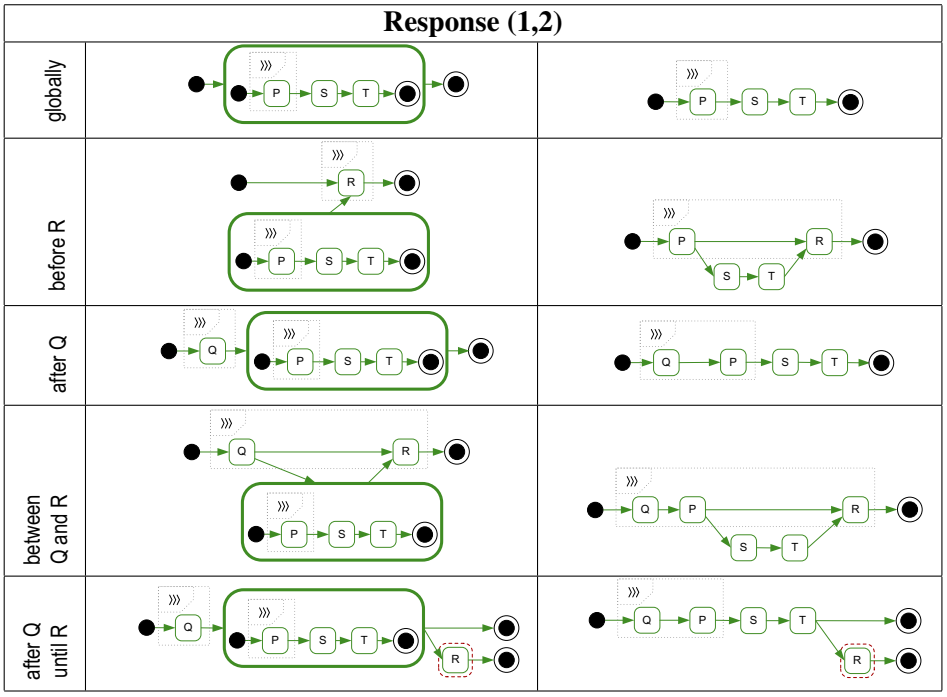


Table 5.12: Response (1,2), trivial and simplified patterns

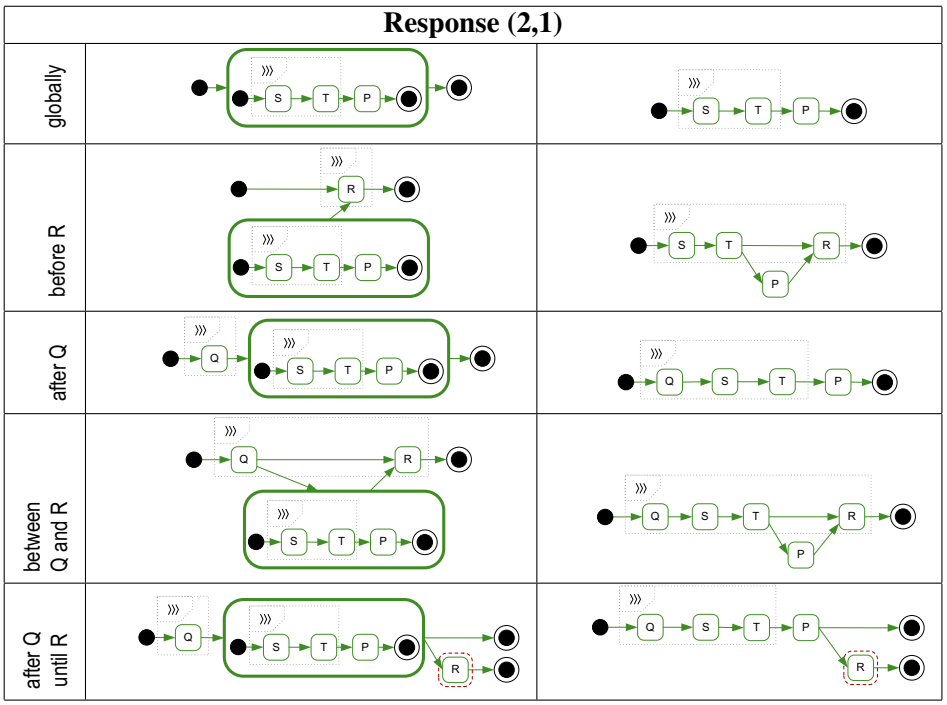


Table 5.13: Response (2,1), trivial and simplified patterns

Conclusion. Note how for response and precedence, the simplified forms are quite natural expressions of the original requirements. Disregarding the Specification Pattern System's distinction between scopes and properties, e.g. P responds to S, T after Q actually translates to 'after the sequence Q, S, T, there needs to follow P', which is exactly what the TSSD says. In general, the resulting diagrams are compact and can be interpreted in a straight-forward manner without the context of the original specification pattern. TSSDs thus avoid the problem faced by the LTL that a correct formula may be derived using the appropriate patterns, but is still very hard to parse for any reader that does not know how it was originally derived.

While TSSDs provide a very suitable way of encoding the patterns of the Specification Pattern System, we believe that TSSDs would not greatly benefit from using the Specification Pattern System in order to derive them. This is not due to any flaw or lack of usefullness in the pattern system itself, but to the fact that the intuitions it provides to designers are already directly integrated into the TSSD language.

5.2 Deriving Properties from Textual Requirements

We now discuss how structural and temporal property specifications can be derived from informal textual requirements in a systematic manner. As our case study, we use an elevator system. The application is in part inspired by an example property given in [13], but extends the system from a single elevator to a large building with an arbitrary number of floors and elevators. The following requirements are provided for the system:

1. **Safety**: Whenever an elevator is not at a floor, its doors may not be open.
2. **Responsive**: Every request for an elevator is assigned to exactly one elevator by the central dispatcher.
3. **Progress**: An elevator may not stay between floors for more than 30 seconds.
4. **Progress**: If requests have been assigned to an elevator, it may not be idle for more than 22 seconds.
5. **Purposeful**: An elevator may only move towards some assigned request.
6. **Fairness**: Concurrent requests must be fulfilled within 300 seconds of each other.
7. **Fairness**: When a request for a specific floor has been assigned to an elevator, it may only arrive at this floor at most twice before opening its doors.

Using standard OOA techniques, we extract the class diagram in Figure 5.1 from the requirements.

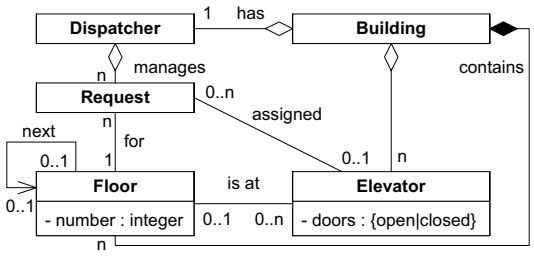


Figure 5.1: Elevator class diagram

(1) As the safety property (1) is a structural requirement, we encode it as an SDD. In order to derive the SDD structure, we decompose the textual requirement into the semantically relevant blocks, which we delimit with vertical lines in the following. The first requirement then becomes | For every elevator | not at a floor | the doors must not be open |, which can be directly translated into SDD nodes | $\forall$ elevator | $\exists$

at a floor | • door = open |. After switching the connectors where negation is required, this results in the SDD in Figure 5.2.

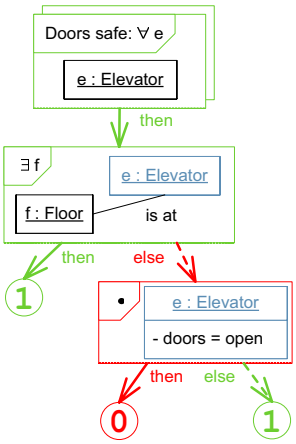
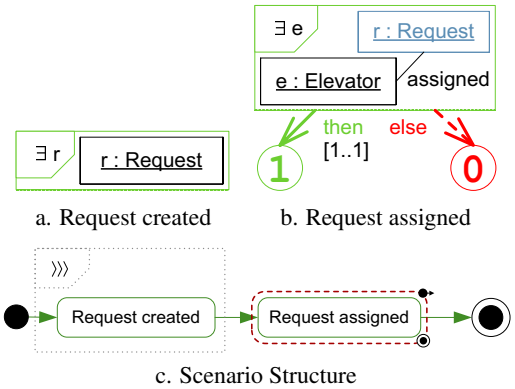


Figure 5.2: Property (1) encoded as an SDD

(2) Property (2) is not purely structural, unless requests are created with an assignment. We therefore interpret it as | Every time | a request is (created) | it then | afterwards | is assigned to one elevator | exactly |. We first encode the two structural terms ∃ request (Figure 5.3a), and ∃ elevator assigned to request with a cardinality of [1..1] (Figure 5.3b).| Every time | ... | then | becomes a trigger block around ∃ request, while | afterwards | becomes an eventually connector and | exactly | makes the situation globally strict (Figure 5.3c), resulting in the TSSD in Figure 5.4.



a. Request created b. Request assigned

c. Scenario Structure

Figure 5.3: Deriving Property (2)

(3) Property (3) is encoded using the same schema (Figures 5.5a and 5.5b), but additionally introduces a time constraint [0..30] between the two situations (Figure

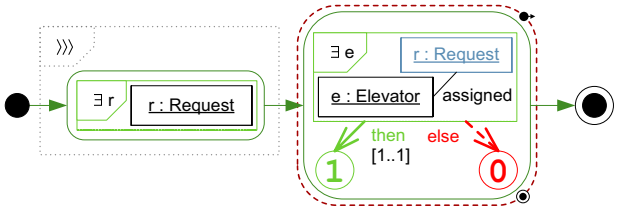
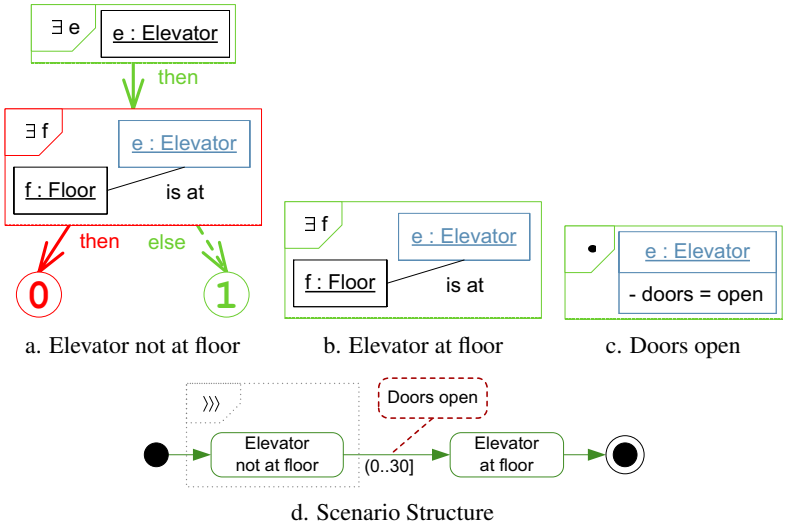


Figure 5.4: Property (2) encoded as a TSSD

5.5d). Combined with a guard (Figure 5.5c) enforcing requirement (1), this yields Figure 5.6.



a. Elevator not at floor b. Elevator at floor c. Doors open

d. Scenario Structure

Figure 5.5: Deriving Property (3)

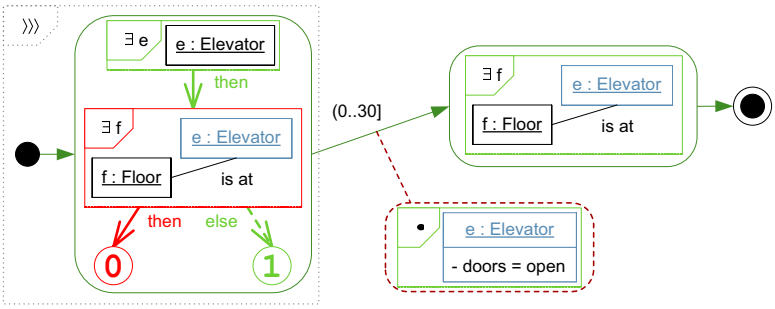


Figure 5.6: Property (3) encoded as a TSSD

(4) For property (4), we interpret idleness as | Every time | an elevator with assigned requests | is at a floor | then | it needs to move within [0..22] |. Whether an elevator is

requested (Figure 5.8a) is determined by an ESDD (Figure 5.7). The trigger block includes the first two situations (Figure 5.8d), i.e. that there are requests and that the elevator is at a floor (Figure 5.8c). It has moved if it is not at a floor anymore (Figure 5.8b). Combined, this yields the TSSD in Figure 5.9.

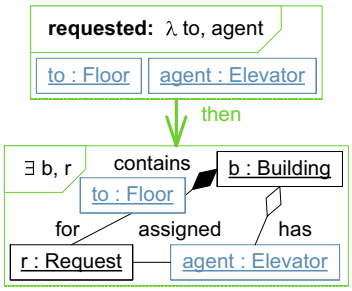
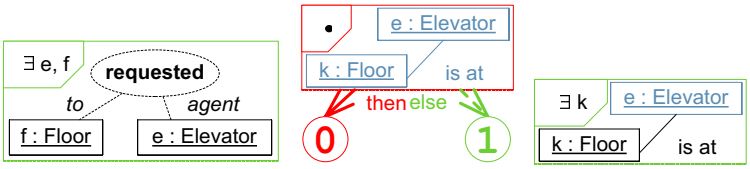
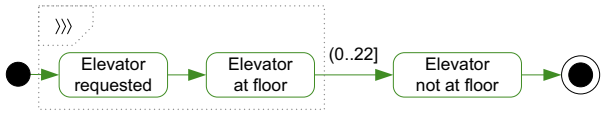


Figure 5.7: Was the elevator requested on the floor?



a. Elevator requested b. Elevator not at floor c. Elevator at floor



d. Scenario Structure

Figure 5.8: Deriving Property (4)

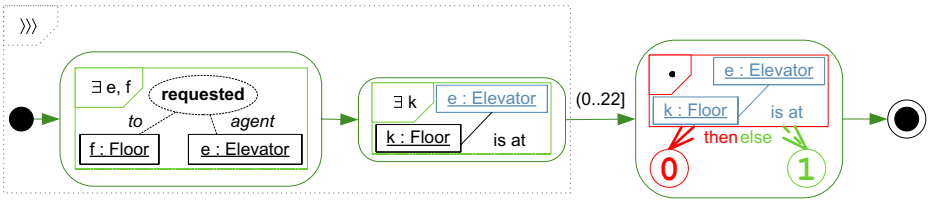
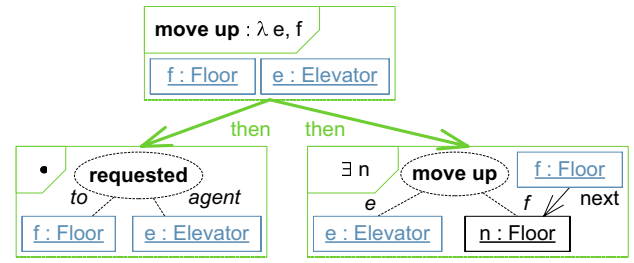


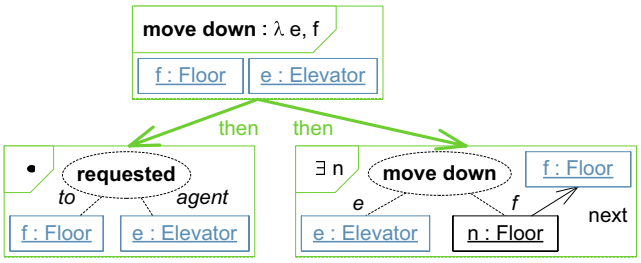
Figure 5.9: Property (4) encoded as a TSSD

(5) The difficulty in encoding property (5) is in detecting the direction of the movement from a sequence of states in the trigger, which is achieved by the sequence elevator at floor (Figure 5.11a), eventually elevator at next floor for the up-direction (Figure 5.11b) or down direction (Figure 5.11d). These two branches make up the trigger of the scenario (Figure 5.11f). When the trigger is completed, it immediately needs to be acceptable for the elevator to move in the given direction justified

by a request. The two situations in Figure 5.11c and 5.11e only reference the ESDD definitions (Figure 5.10a and 5.10b), which recursively determine whether a request exists by traversing the floors in the indicated direction until they find a request or fail. Together, this results in the TSSD in Figure 5.12.

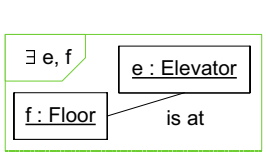


a. Upwards

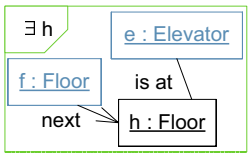


b. Downwards

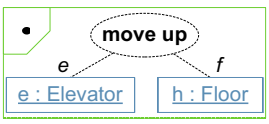
Figure 5.10: ESDD definitions: is there a request in the indicated direction



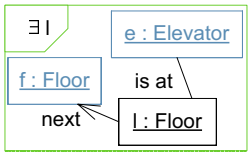
a. Elevator at floor



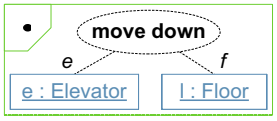
b. Elevator at higher floor



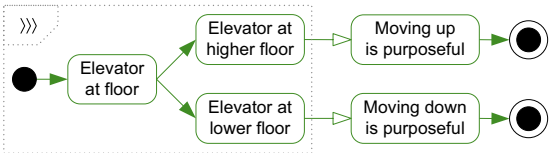
c. Purposeful to move up



d. Elevator at lower floor



e. Purposeful to move down



f. Scenario Structure

Figure 5.11: Deriving Property (5)

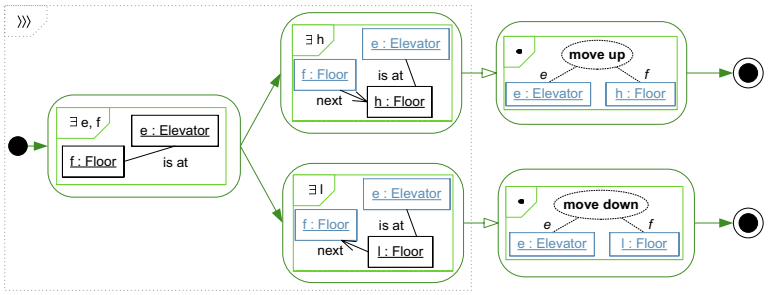
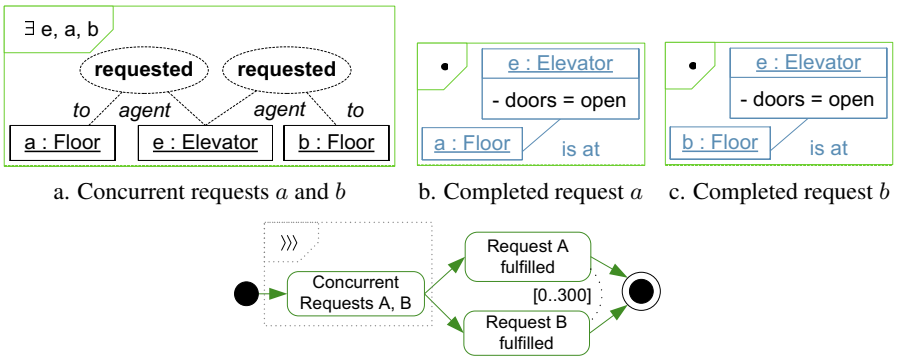


Figure 5.12: Property (5) encoded as a TSSD

(6) Property (6) becomes |Every time|two concurrent requests exist (Figure 5.13a)|then|each|is eventually|completed (Figure 5.13b and 5.13c)|within 300 seconds of the other|. After the trigger block, the |each| introduces two $\vee$-branches. The |within| time constraint results in a constraint edge across the two branches (Figure 5.13d). Combined, this yields the TSSD in Figure 5.14.



a. Concurrent requests a and b b. Completed request a c. Completed request b

d. Scenario Structure

Figure 5.13: Deriving Property (6)

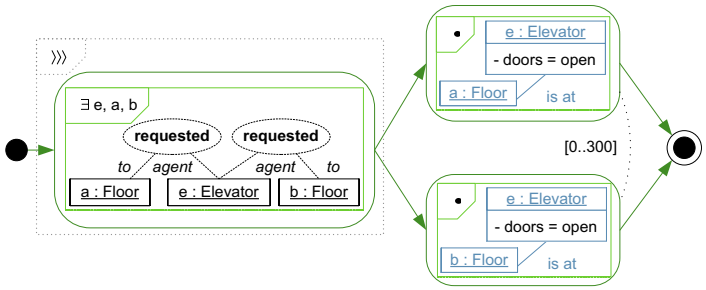
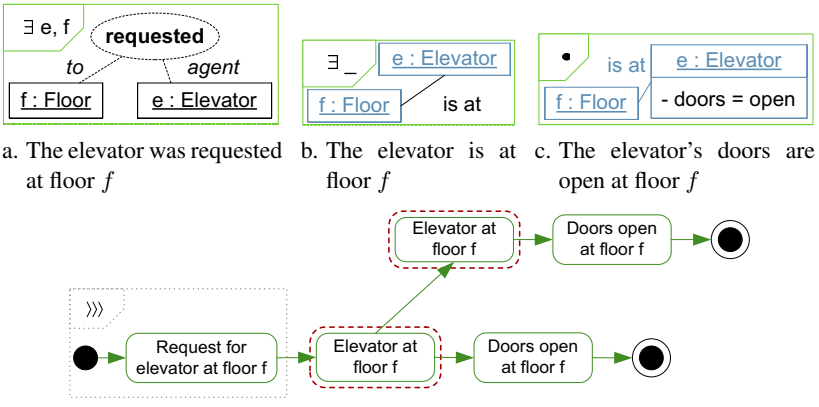


Figure 5.14: Property (6) encoded as a TSSD

(7) Property (7) is the famous example that results in a rather unwieldy LTL formula (cf. [13]). The exact meaning of that formula can be expressed using the

bounded-existence-between pattern from Table 5.7. However, we believe that a slightly stronger interpretation of the requirement better reflects what is expected of an elevator, namely that it eventually opens its door where it was requested (i.e. as a strong instead of a weak until). We therefore encode the requirement as | Every time | an elevator is requested for a floor (Figure 5.15a) | then | it eventually | is at the floor (Figure 5.15b) | for the first time | and eventually | opens its doors (Figure 5.15c) | or | eventually | is at the floor | for the second time | and eventually | opens its doors |. It is the explicit requirement | for the first/second time | that turns being at the floor into a strict situation (Figure 5.15d). The case that | is at the floor | never matches is omitted here as it is a necessary precondition for opening the doors at the floor. The corresponding SDDs clearly indicate this property, thus illustrating the fact that the ability to integrate the modeling of structural properties into a scenario definition helps to make the dependencies between the different properties explicit. The first structural property is again encoded by the ESDD in Figure 5.7. Property (7) is then encoded by the TSSD in Figure 5.16.



a. The elevator was requested at floor f

b. The elevator is at floor f

c. The elevator's doors are open at floor f

d. Scenario Structure

Figure 5.15: Deriving Property (7)

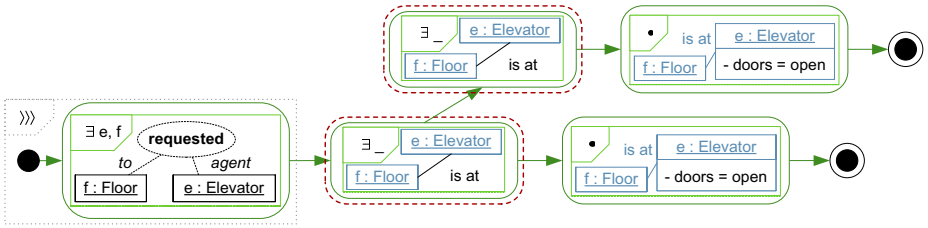


Figure 5.16: Property (7) encoded as a TSSD

6 Conclusion and Future Work

We have presented a visual approach for the specification of temporal and structural properties. We have shown how UML object diagrams as a widely accepted type of visual diagram can be extended for the description of complex structural conditions. The resulting Story Decision Diagrams have then further been employed in the context of 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 prototypical operational realization as property detectors that detect satisfaction of the specified properties based on the GTS model checker GROOVE show that the approach even works, in principle, under the restricted conditions of a model checking engine.

As future work, we will develop full tool support for the approach. We are currently working on plugins for Fujaba4Eclipse that allow using SDDs as an alternative to Story Patterns and specifying TSSDs. Based on these specifications, we will then allow the export of SDDs and TSSDs into GROOVE and the generation of optimized run-time monitors in Java and C++. Additionally, we plan to look into the extension of our symbolic invariant checking approach [4], as the underlying BDDs should provide support for many of the powerful features of SDDs.

References

[1] A. Alfonso, V. Braberman, N. Kicillof, and A. Olivero. Visual Timed Event Scenarios. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 168–177, Washington, DC, USA, 2004. IEEE Computer Society.

[2] R. Alur and T. A. Henzinger. Real-time logics: Complexity andexpressiveness. *Information and Computation*, 104(1):35–77, 1993.

[3] R. Alur and T. A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1):181–204, 1994.

[4] B. Becker, D. Beyer, H. Giese, F. Klein, and D. Schilling. Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation. In *Proc. of the* 28^{th} *International Conference on Software Engineering (ICSE), Shanghai, China*. ACM Press, 2006.

[5] P. Bellini, R. Mattolini, and P. Nesi. Temporal logics for real-time system specification. *ACM Comput. Surv.*, 32(1):12–42, 2000.

[6] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. A visualization of OCL using collaborations. *Lecture Notes in Computer Science*, 2185:257–271, 2001.

[7] P. Bouyer, F. Chevalier, and N. Markey. On the expressiveness of tptl and mtl. Technical Report Research report LSV-2005-05, Lab. Spécification et Vérification, May 2005.

[8] J. Bradfield, J. Kuester Filipe, and P. Stevens. Enriching OCL Using Observational mu-Calculus. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering (FASE 2002), Grenoble, France*, volume 2306 of *LNCS*. Springer, April 2002.

[9] D. Bradley, D. Seward, D. Dawson, and S. Burge. *Mechatronics*. Stanley Thornes, 2000.

[10] M. Cengarle and A. Knapp. Towards OCL/RT. In L.-H. Eriksson and P. Lindsay, editors, *Formal Methods – Getting IT Right, International Symposium of Formal Methods Europe, Copenhagen, Denmark*, volume 2391 of *LNCS*, pages 389–408. Springer, 2002.

[11] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Jan. 2000.

[12] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property Specification Patterns for Finite-state Verification. In *2nd Workshop on Formal Methods in Software Practice*. ACM Press, March 1998.

[13] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 411–420, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[14] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language. In G. Engels and G. Rozenberg, editors, *Proc. of the* 6^{th} *International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, LNCS 1764, pages 296–309. Springer Verlag, November 1998.

[15] S. Flake and W. Mueller. An OCL Extension for Real-Time Constraints. In *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, volume 2263 of *LNCS*, pages 150–171. Springer, February 2002.

[16] F. Gadducci, R. Heckel, and M. Koch. A fully abstract model for graph-interpreted temporal logic. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *TAGT*, volume 1764 of *Lecture Notes in Computer Science*, pages 310–322. Springer, 1998.

[17] F. Gadducci, R. Heckel, and M. Koch. A fully abstract model for graph-interpreted temporal logic. In *Proc. of the Theory and Application of Graph Transformations*, volume 1764 of *Lecture Notes in Computer Science*, pages 310–322, 2000.

[18] H. Giese and F. Klein. Beyond Story Patterns: Story Decision Diagrams. In H. Giese and B. Westfechtel, editors, *Proc. of the 4th International Fujaba Days 2006, Bayreuth, Germany*, volume tr-ri-06-275 of *Technical Report*. University of Paderborn, September 2006.

[19] H. Giese and F. Klein. Visual Specification of Structural and Temporal Properties. In H. Giese and B. Westfechtel, editors, *Proc. of the 4th International Fujaba Days 2006, Bayreuth, Germany*, volume tr-ri-06-275 of *Technical Report*. University of Paderborn, September 2006.

[20] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. Flake. Towards the Compositional Verification of Real-Time UML Designs. In *Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland*, pages 38–47. ACM Press, September 2003.

[21] D. Harel and R. Marelly. Playing with Time: On the Specification and Execution of Time-Enriched LSCs. In *Proc. 10th IEEE/ACM Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2002)*, Fort Worth, Texas, USA, 2002. (invited paper).

[22] R. Heckel, J. M. Küster, and G. Taentzer. Confluence of Typed Attributed Graph Transformation Systems. In *Graph Transformation: First Interna-*

tional Conference, ICGT 2002, Barcelona, Spain, October 7-12, 2002, volume 2505 of *Lecture Notes in Computer Science*, pages 161–176. Springer Verlag, 2002.

[23] R. Kahle and T. Studer. Formalizing non-termination of recursive programs. *JLAP*, 49(1-2):1–14, 2001.

[24] S. Kent and J. Howse. Mixing visual and textual constraint languages. In R. France and B. Rumpe, editors, *UML'99, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 384–398. Springer, 1999.

[25] S. Kent and J. Howse. Constraint trees. In T. Clark and J. Warmer, editors, *Object Modeling with the OCL*, pages 228–249. Springer, 2002.

[26] E. Kindler. On the semantics of EPCs: Resolving the vicious circle. *Data and Knowledge Engineering*, 56(1):23–40, January 2006.

[27] H. Köhler, U. Nickel, J. Niere, and A. Zündorf. Integrating UML Diagrams for Production Control Systems. In *Proc. of the* 22^{nd} *International Conference on Software Engineering (ICSE), Limerick, Irland*, pages 241–251. ACM Press, 2000.

[28] S. Konrad and B. H. C. Cheng. Real-time specification patterns. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 372–381, New York, NY, USA, 2005. ACM Press.

[29] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

[30] X. Li and J. Lilius. Timing Analysis of UML Sequence Diagrams. In R. France and B. Rumpe, editors, *UML'99 - The Second International Conference on The Unified Modeling Language Fort Collins, Colorado, USA*, volume 1723 of *Lecture Notes in Computer Science*, October 1999.

[31] Object Management Group. UML Profile for Schedulability, Performance, and Time Specification. OMG Document ptc/02-03-02, September 2002. URL: http://cgi.omg.org/docs/ptc/02-03-02.pdf.

[32] Object Management Group. UML 2.0 Object Constraint Language (OCL) Specification, 2003. http://www.omg.org/docs/ptc/03-10-14.pdf.

[33] Object Management Group. *UML 2.0 Superstructure Specification*, 2003. Document ptc/03-08-02.

[34] A. Rensink. Towards model checking graph grammars. In M. Leuschel, S. Gruner, and S. L. Presti, editors, *Workshop on Automated Verification of Critical Systems (AVoCS)*, Technical Report DSSE–TR–2003–2, pages 150–160. University of Southampton, 2003.

[35] A. Rensink, Á. Schmidt, and D. Varró. Model Checking Graph Transformations: A Comparison of Two Approaches. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *International Conference on Graph Transformations (ICGT)*, volume 3256 of *Lecture Notes in Computer Science*, pages 226–241. Springer-Verlag Heidelberg, 2004.

[36] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation : Foundations*. World Scientific Pub Co, February 1997. Volume 1.

[37] T. Rötschke and A. Schürr. Temporal Graph Queries to Support Software Evolution. In *Graph Transformation: 5th International Conference, ICGT 2006, Rio Grande do Norte, Brazil, September 17-23, 2006*, pages 1–15, 2006.

[38] Á. Schmidt and D. Varró. CheckVML: A Tool for Model Checking Visual Modeling Languages. In *Proceedings of the 6th International Conference on the Unified Modeling Language, UML2003*, volume 2863 of *Lecture Notes in Computer Science*, pages 92–95. Springer Verlag, 2003.

[39] B. Sengupta and R. Cleaveland. Triggered Message Sequence Charts. In W. G. Griswold, editor, *Proceedings of the Tenth ACM SIGSOFT Symposium on the Foundations of Softare Engineering (FSE-10)*, Charleston, South Carolina, USA, November 2002. ACM Press.

Index

A Alternative ESDD Semantics Definition

In this appendix, we present a different though equivalent approach to defining the semantics of recursive ESDDs. The approach differs from the definition in Section 3.3.4 in the way the fixed point is computed: The ESDD is evaluated only once and the fixed point is computed on the derived result set, as opposed to computing a new result set in every step. While the alternative definition is more complex and may consume more memory during computation, it should take significantly less time to compute, which prompted its inclusion in this appendix.

We compute the fixed point in two steps: First, we compute the *potential* result set of F using function $fapply$, which is an extension of the regular $apply$ function. In a second step, we then use the fixed point operator $\overline{+}$ to derive the *actual* result set of F, defining its semantics.

In order to compute the potential result set, an existential node α_F quantifying all roles of the ESDD F is added before the λ node λ_F. The existential node will then generate all possible combinations of bindings for the roles and pass them on as input to the λ node.

The modified propagation functions $fapply_n$ will not try to evaluate recursive ESDD invocations, but basically considers both possible results of the invocation at the same time. If for ξ_ζ, the two extended bindings ξ'_a and ξ'_b have been generated in the existential node n containing an invocation of the recursive ESDD F, we would send two candidate sets containing $\{(n', \xi'_a), F(\xi'_a)\}$ and $\{(n', \xi'_b), F(\xi'_b)\}$ down the then connector and a candidate set $\{(n'', \xi_\zeta), \overline{F(\xi'_a)}, \overline{F(\xi'_b)}\}$ down the else connector, where $\overline{\zeta}$ is a forbidden witness, i.e., $eval(\overline{\zeta}) := \neg eval(\zeta)$. The additional witnesses ensure that each of the candidate sets will only satisfy the diagram if the ESDD invocation has the appropriate result.

Recursive and non-recursive ESDDs can be distinguished by means of their invocation graphs. We accordingly divide $\mathcal{F}_n$ into the non-recursive ESDDs $\mathcal{F}_n^{NR}$ and the recursive ESDDs $\mathcal{F}_n^R$ with $\mathcal{F}_n := \mathcal{F}_n^{NR} \uplus \mathcal{F}_n^R$.

For each node n where $\mathcal{F}_n^R = \emptyset$, $fapply_n$ is identical to $apply_n$. For nodes where $\mathcal{F}_n^R \neq \emptyset$, the modified propagation function $fapply_n$ performs the following steps on each selected witness ζ:

1. The possible extensions of the binding ξ_ζ are computed as above, but considering only the non-recursive ESDD invocations:

$$\mathcal{X}_\zeta^1 := \{\xi'_\zeta \mid P_n[\xi'_\zeta] \leq G \ \wedge \ \forall F \in \mathcal{F}_n^{NR} : \{F((n,\xi'_\zeta))\} \sqsubseteq [\![F]\!]^G \wedge \\ \xi_\zeta \leq \xi'_\zeta \ \wedge \ \forall v : \xi'_\zeta(v) \neq \xi_\zeta(v) \Rightarrow v \in free(n,\xi_\zeta)\}. \quad \text{(A.1)}$$

Differently from above, $\mathcal{X}_\zeta^{(\mathsf{e})}$ is never empty:

$$\mathcal{X}_\zeta^{(\mathsf{e})} := \{\xi_\zeta\}. \quad \text{(A.2)}$$

2. We now compute two sets of corresponding witnesses, one for the new bindings travelling down the then connector and one for the original binding, travelling down the else connector. Cardinalities are not allowed on nodes containing recursive invocations as they can lead to paradoxical statements and thus do not need to be considered. We then have:

$$\mathcal{W}_\zeta^{(\mathsf{t})} := \{(n',\xi') \mid n' \in \mathsf{then}(n) \wedge \xi' \in \mathcal{X}_\zeta^{(\mathsf{t})} \quad \text{(A.3)}$$

$$\mathcal{W}_\zeta^{(\mathsf{e})} := \{(n',\xi') \mid n' \in \mathsf{else}(n) \wedge \xi' \in \mathcal{X}_\zeta^{(\mathsf{e})}. \quad \text{(A.4)}$$

3. The result set $\mathcal{A}'$ is updated, adding additional required and forbidden witnesses representing the result of the recursive ESDD invocation. Again, we write $F(\zeta)$ and $\overline{F(\zeta)}$ as abbreviations for required and forbidden witnesses at F's λ node. Note that $\mathcal{F}_n^R$ will typically only have one element, greatly simplifying many of the following expressions.

   The following definitions are declarative, as opposed to the constructive definitions in Equations 3.10 and 3.11. We therefore additionally require the generated result sets to be minimal, i.e.

$$\forall \mathcal{C} \in \mathcal{A}_\square : (\nexists \mathcal{C}' \in \mathcal{A}_\square : \mathcal{C} \subseteq \mathcal{C}'). \quad \text{(A.5)}$$

   For the then branch of existential nodes, we generate a set of candidate sets, each containing one of the generated extensions and all its required witnesses representing successful recursive invocations:

$$\mathcal{A}_\exists^{(\mathsf{t})} := \{\mathcal{C}' \mid \exists \mathcal{C} \in \mathcal{A}' : \zeta \in \mathcal{C} \ \wedge \ (\forall \zeta^\eta \in \mathcal{C} \setminus \zeta : \zeta^\eta \in \mathcal{C}') \wedge \\ (\exists \zeta^{(\mathsf{t})} \in \mathcal{W}_\zeta^{(\mathsf{t})} : (\zeta^{(\mathsf{t})} \in \mathcal{C}' \wedge \forall F \in \mathcal{F}_n^R : F(\zeta^{(\mathsf{t})}) \in \mathcal{C}'))\}. \quad \text{(A.6)}$$

   For the then branch of universal nodes, we also generate a set of alternative candidate sets, as each generated extension must either be in the extended

candidate set along with all its required invocation witnesses or may 'excuse itself' by means of a forbidden witness representing a failed invocation. However, to justify following the then branch, there has to be at least one valid candidate. We have:

$$\begin{aligned}\mathcal{A}_{\forall}^{(\mathsf{t})} := \{\mathcal{C}' \mid \exists \mathcal{C} \in \mathcal{A}' : \zeta \in \mathcal{C} \ \wedge \ (\forall \zeta^{\eta} \in \mathcal{C} \setminus \zeta : \zeta^{\eta} \in \mathcal{C}') \wedge \\ (\exists \zeta^{(\mathsf{t})} \in \mathcal{W}_{\zeta}^{(\mathsf{t})} : (\zeta^{(\mathsf{t})} \in \mathcal{C}')) \wedge \\ (\forall \zeta^{(\mathsf{t})} \in \mathcal{W}_{\zeta}^{(\mathsf{t})} : (\zeta^{(\mathsf{t})} \in \mathcal{C}' \wedge \forall F \in \mathcal{F}_n^R : F(\zeta^{(\mathsf{t})}) \in \mathcal{C}') \vee \\ (\zeta^{(\mathsf{t})} \notin \mathcal{C}' \wedge \exists F \in \mathcal{F}_n^R : \overline{F(\zeta^{(\mathsf{t})})} \in \mathcal{C}'))\}. \end{aligned} \tag{A.7}$$

For the else branch, we extend a candidate set with the single witness from $\mathcal{W}_{\zeta}^{(\mathsf{e})}$ and a forbidden invocation witness for every extended binding in $\mathcal{W}_{\zeta}^{(\mathsf{t})}$:

$$\begin{aligned}\mathcal{A}^{(\mathsf{e})} := \{\mathcal{C}' \mid \exists \mathcal{C} \in \mathcal{A}' : \zeta \in \mathcal{C} \ \wedge \ (\forall \zeta^{\eta} \in \mathcal{C} \setminus \zeta : \zeta^{\eta} \in \mathcal{C}') \ \wedge \ (\exists \zeta^{(\mathsf{e})} \in \mathcal{W}_{\zeta}^{(\mathsf{e})} : \\ (\zeta^{(\mathsf{e})} \in \mathcal{C}' \wedge \forall \zeta^{(\mathsf{t})} \in \mathcal{W}_{\zeta}^{(\mathsf{t})} : \exists F \in \mathcal{F}_n^R : \overline{F(\zeta^{(\mathsf{t})})} \in \mathcal{C}'))\}. \end{aligned} \tag{A.8}$$

(A.9)

Finally, those candidate sets not containing ζ are left unchanged:

$$\mathcal{A}^{\eta} := \{\mathcal{C}' \mid \exists \mathcal{C} \in \mathcal{A}' : (\zeta \notin \mathcal{C} \wedge \mathcal{C}' = \mathcal{C})\}. \tag{A.10}$$

If n is existential, we then have

$$\mathcal{A}'_{\forall} := \mathcal{A}_{\forall}^{(\mathsf{t})} \cup \mathcal{A}^{(\mathsf{e})} \cup \mathcal{A}^{\eta}. \tag{A.11}$$

If n is universal, we have

$$\mathcal{A}'_{\exists} := \mathcal{A}_{\exists}^{(\mathsf{t})} \cup \mathcal{A}^{(\mathsf{e})} \cup \mathcal{A}^{\eta}. \tag{A.12}$$

The result set for the auxiliary existential node α_F is then computed in the usual fashion using Equations 3.13 – 3.16. We define $[\![f^{(0)}]\!]^G := [\![\alpha_F]\!]^G_{\{\{(f_F,\tau)\}\}}$ as the first appoximation of the semantics $[\![F]\!]^G$ of F. The computed result set contains candidate sets that are final except for invocation witnesses $F_i(\zeta)$ or $\overline{F_i(\zeta)}$, whose truth value is $\bot$.

For all ESDDs F_i that are part of the same invocation graph, we then apply the fixed point operator $\mathbb{\mp}$ to $[\![f_i^{(0)}]\!]^G$. We keep applying it, in turn, to compute

$$[\![f^{(j+1)}]\!]^G := \mathbb{\mp}([\![f_i^{(j)}]\!]^G) \tag{A.13}$$

until we have $\mathbb{\mp}([\![f_i^{(j)}]\!]^G) = [\![f_i^{(j)}]\!]^G$ for all of the involved ESDDs F_i. We have then computed a fixed point $[\![f_i]\!]^G$ which allows us to define the ESDD semantics as

$$[\![F_i]\!]^G := \{\mathcal{C} \mid \mathcal{C} \in [\![f_i]\!]^G \wedge eval(\mathcal{C})\}. \tag{A.14}$$

We define two versions of $\mathbb{\mp}$, the least fixed point operator $\mathbb{\mp}_\mu$ and the greatest fixed point operator $\mathbb{\mp}_\nu$. The standard semantics of SDDs are defined by means of $\mathbb{\mp}_\mu$, i.e. using least fixed points.

The least fixed point operator $\mathbb{\mp}_\mu$ works by eliminating those invocation witnesses which evolve into valid final candidate sets from the candidate sets in $[\![f_i^{(j)}]\!]^G$. As more and more undefined invocation witnesses disappear, the number of valid final candidate sets making up $[\![F_i]\!]^G$ increases until no more such invocation witnesses are left. We define:

$$\mathbb{\mp}_\mu([\![f_i^{(j)}]\!]^G) := \begin{cases} \bigcup\limits_{\mathcal{C}\in[\![f_i^{(j)}]\!]^G} \mathcal{C}\backslash F(\zeta) & \left| \begin{array}{l} \exists F(\zeta) \in \mathcal{W}_{[\![f_i^{(j)}]\!]^G} : \\ \exists \mathcal{C} \in [\![f_i^{(j)}]\!]^G : \{F(\zeta)\} \sqsubseteq \mathcal{C} \wedge eval(\mathcal{C}) \end{array} \right. \\ \bigcup\limits_{\mathcal{C}\in[\![f_i^{(j)}]\!]^G} \mathcal{C} | F(\zeta) \notin \mathcal{C} & \left| \begin{array}{l} \exists F(\zeta) \in \mathcal{W}_{[\![f_i^{(j)}]\!]^G} : \\ \forall \mathcal{C} \in [\![f_i^{(j)}]\!]^G | \{F(\zeta)\} \sqsubseteq \mathcal{C} : \neg eval(\mathcal{C}) \end{array} \right. \\ [\![f_i^{(j)}]\!]^G & \mid otherwise. \end{cases} \tag{A.15}$$

The greatest fixed point operator $\mathbb{\mp}_\nu$, on the other hand, initially assumes that all invocation witnesses evaluate to $true$ and then successively eliminates those candidate sets that are definitely invalid from $[\![f_i^{(j)}]\!]^G$.

To keep the definition of $\mathbb{\mp}_\nu$ compact, we define the truth value of an invocation witness in $\mathcal{W}_{[\![f_i^{(j)}]\!]^G}$ as

$$eval(F(\zeta)) := \begin{cases} false & \mid \nexists \mathcal{C} \in [\![f_i^{(j)}]\!]^G : \{F(\zeta)\} \sqsubseteq \mathcal{C} \\ true & \mid otherwise, \end{cases} \tag{A.16}$$

i.e. an invocation witness is only $false$ if there are no candidate sets left that evolved from it. $\mathbb{F}_\nu$ then eliminates all candidate sets that contain at least one invalid witness:

$$\mathbb{F}_\nu([\![f_i^{(j)}]\!]^G) := \begin{cases} [\![f_i^{(j)}]\!]^G \setminus \mathcal{C} & | \exists \mathcal{C} \in [\![f_i^{(j)}]\!]^G : \exists \zeta \in \mathcal{C} : \neg eval(\zeta) \\ [\![f_i^{(j)}]\!]^G & | otherwise. \end{cases} \tag{A.17}$$

As all involved sets (especially the result sets $[\![f_i^{(j)}]\!]^G$) are finite and the fixed point operators only change the result sets by eliminating, never adding, witnesses and candidate sets, they can only be applied to $[\![f_i^{(j)}]\!]^G$ finitely often before a fixed point is reached. We can therefore guarantee that the fixed points exist and that their computation terminates.

B Enhanced Story Patterns (eSP)

While Story Decision Diagrams are expressive and can be interpreted based on a small number of basic concepts, they are not particularly compact. In the following, we define *enhanced Story Patterns* (eSP) which trade in expressiveness for compactness while still remedying the main issues plaguing Story Patterns. Internally, eSPs can be translated to equivalent SDDs which provide them with their formal semantics.

eSPs are visually closer to normal Story Patterns and should be immediately understandable to anyone familiar with Story Patterns. As additional syntax, they merely introduce several types of UML 2.0 boxes. While they are able to express negation of substructures, universal quantification, and alternatives, they lack the ability to compose expressions into more complex expressions.

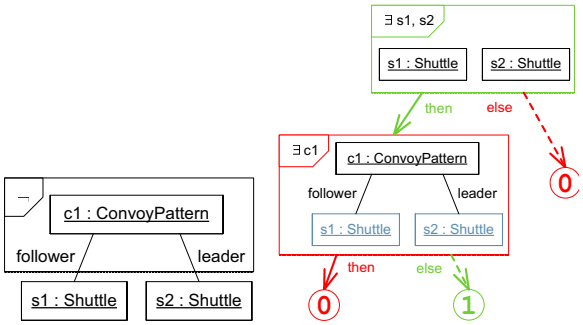
Negation. eSPs allow the negation of complex structures. Figure B.1 provides an example. Negation boxes should completely supersede crossed out elements as the means of expressing negation. Among other advantages, this will finally make negative edge labels readable.

Implication. eSPs provide implication or conditional requirements. This allows expressing that a set of elements should be present whenever another set of elements is present. Figures B.2 and B.3 provide two examples. The intended requirement that the conditional elements be present *whenever* the trigger is present implies universal quantification – while it would be possible to introduce a box expressing existential quantification, this would add no additional expressiveness.

Alternatives. eSPs provide a way to express alternative requirements, which is perhaps the least intuitive of the extensions. Of all the sets of elements inside the $\vee$ boxes, at least one needs to be present. Figure B.4 provides an example. It would also be feasible to define an XOR box.

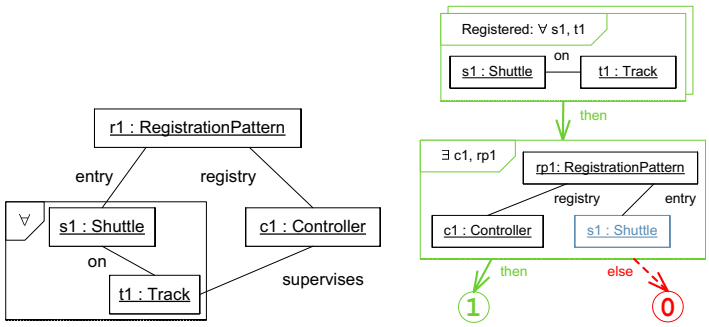
ESDDs. eSPs allow invoking ESDDs. The ESDDs then have to be defined elsewhere, using standard SDD syntax. Figure B.5 provides an example.

Transformations. eSPs allow using standard Story Pattern syntax for specifying transformations. Figure B.6 provides an example.



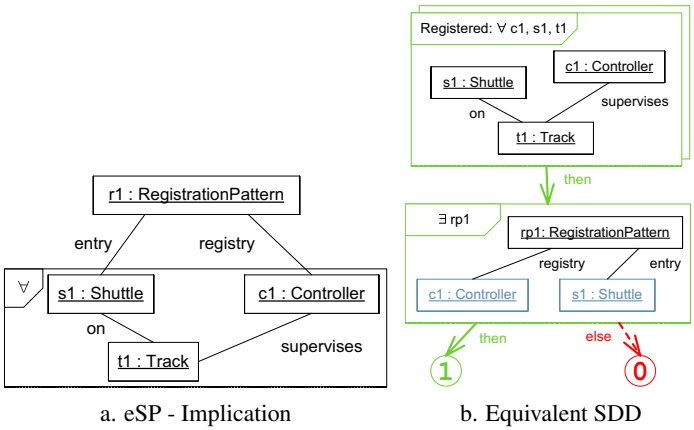
a. eSP - Negation b. Equivalent SDD

Figure B.1: No pattern exists between the shuttles



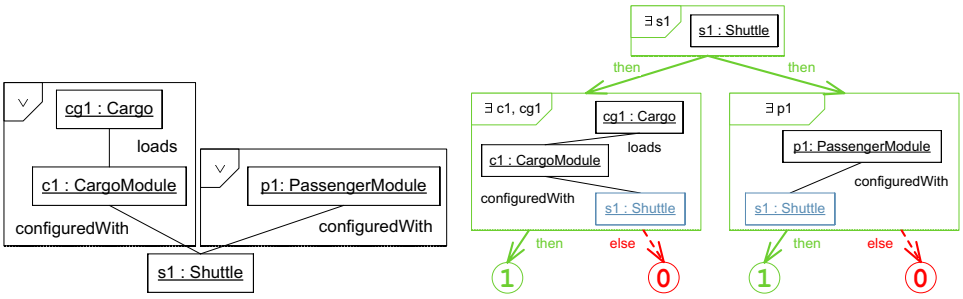
a. eSP - Implication b. Equivalent SDD

Figure B.2: A pattern exists for each shuttle



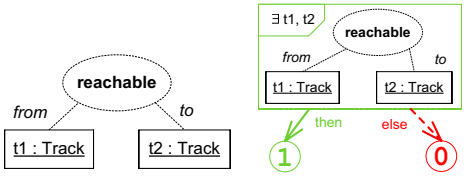
a. eSP - Implication b. Equivalent SDD

Figure B.3: A pattern exists between the shuttle and each controller



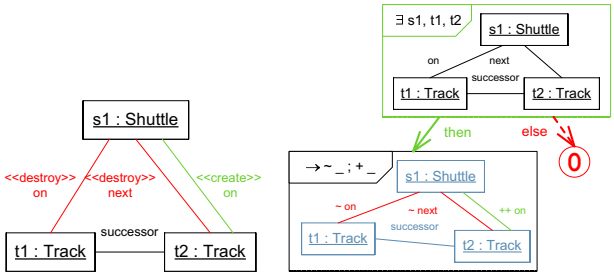
a. eSP - Disjunction b. Equivalent SDD

Figure B.4: A shuttle is either a cargo or passenger shuttle



a. eSP - Invoking an ESDD b. Equivalent SDD

Figure B.5: Connected tracks



a. eSP - Transformation b. Equivalent SDD

Figure B.6: Moving a shuttle

C Syntax quick reference

Figure C.1 provides a compact overview of the most important elements of the TSSD syntax.

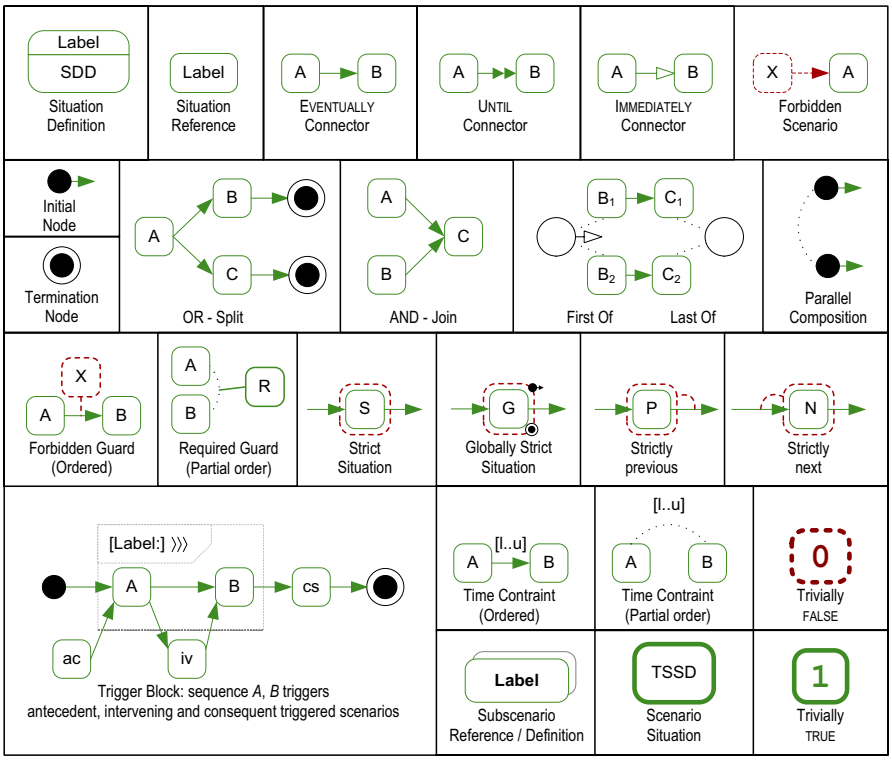


Figure C.1: TSSD Syntax quick reference