Joint Structural and Temporal Property Specification Using Timed Story Scenario Diagrams*

Florian Klein and Holger Giese

Software Engineering Group, University of Paderborn,
Warburger Str. 100, D-33098 Paderborn, Germany
fklein@upb.de, hg@upb.de

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. We present Timed Story Scenario Diagrams (TSSD), a visual notation for scenario specifications that takes structural system properties into account and provides an integrated way of discussing system state evolution. We present the key features of the notation and demonstrate how the patterns of the Specification Pattern System [1,2] can be encoded using TSSDs. We also discuss how TSSDs can be derived from textual specifications in a straight-forward manner, using a case study.

Keywords: Property Specification, Temporal Logic, Visual Specification Language.

1 Introduction

As part of the trend towards more intelligent, efficient, and flexible software-intensive systems (cf. self-adaptive systems [3,4]), dynamic software architectures which permit structural adaptation at run-time are beginning to displace static architectures and models. While this allows building more flexible systems, designing and validating adaptable systems poses new challenges to software engineers. In order to express requirements and commitments concerning the evolution of the structure over time, appropriate integrated notations for the specification of properties covering structural and temporal aspects are required as these are closely intertwined.

The need for formal specifications expressed using logics or automata is a major obstacle for the adoption of formal verification techniques by practitioners. We do not only need techniques for the description of structural and temporal properties which are sufficiently expressive and provide the essential theoretical concepts, but which are appropriate for use by normal designers, requirements engineers, or even informed stakeholders who can handle UML class diagrams (domain experts or engineers from outside the software domain) rather than experts on logic.

* 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.

M.B. Dwyer and A. Lopes (Eds.): FASE 2007, LNCS 4422, pp. 185–199, 2007.
© Springer-Verlag Berlin Heidelberg 2007

For specifying **structural properties**, the UML only provides a textual specification language, the OCL [5]. 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 is the reason why OCL with its textual appearance is rarely used in practice, as the UML's popularity is in large part due to its visual nature and the accessibility of its structural modeling concepts (see also [6]).

Several approaches try to overcome this problem. Constraint diagrams [7] visualize constraints as restrictions on sets using Euler circles, spiders and arrows and constraint trees [8]. VisualOCL [9] focuses on mapping OCL syntax to a visual format as closely as possible, thus facilitating the parsing of structural constraints. However, the resulting visually complex diagrams have only little relation with the original UML specification so that a similar gap results. In contrast, Story Patterns (cf. [10]) extend UML Object Diagrams and thus avoid this gap. However, they have deficits when it comes to quantification and negation.

For **temporal properties**, temporal logics such as LTL or CTL [11] represent the standard. However, as reported in [2], even experts have serious problems handling the intricate nature of these logics. Even in projects with 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. Specification patterns for temporal properties represent an attempt to alleviate this problem. As outlined in [2], 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 [12]. However, while applying the patterns is intuitive, the resulting formulas themselves are no more transparent or readable than before.

Scenarios in form of UML sequence diagrams [13], Triggered Message Sequence Charts (TMSCs) [14], or Live Sequence Charts (LSC) [15] have been proposed as a more accessible means for the description of temporal properties. Visual Timed event Scenarios (VTS) [16] are an alternative which focuses 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.

The existing approaches which combine **structural and temporal properties** are mostly 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), the specification of required behavior by means of temporal restrictions among actions and events is enabled, e.g., [17]. Temporal extensions of the OCL that consider real-time issues have been proposed for events in OCL/RT [18] and for states in RT-OCL [19]. As temporal logic alone already causes an even more demanding mental mapping problem (cf. [2]), integrating the OCL and some temporal logic concepts at the textual level does not yield a sufficiently comprehensible solution.

In [20], an embedding of graph patterns 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.

The only notation that takes an approach similar to ours is a recent proposal [21] for writing temporal graph queries. The approach extends Story Diagrams [22] – an extension of UML Activity Diagrams with Story Patterns – 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.

We can conclude from our analysis of the state of the art that no existing approach fully supports the joint specification of structural and temporal properties in the desired scenario-based manner. The notations either lack support for one of the aspects or seem unsuitable for the intended audience, as they require handling the combination of two notations that already seem forbiddingly complex own their own.

In this paper, we demonstrate that the outlined requirements for jointly specifying structural and temporal properties in a comprehensible manner can be met by a visual language. First, we outline how *Story Decision Diagrams* (SDD) [23] can be used to capture structural requirements. SDD are an extension of Story Patterns [10], combining the intuitive concept of matching structural patterns with decision diagrams, which foster a consecutive if-then-else decomposition of complex properties into comprehensible smaller ones. We then introduce *Timed Story Scenario Diagrams* (TSSD), a new notation inspired by the Visual Timed Event Scenario approach [16], as a way of capturing temporal properties. They provide conditional timed scenarios describing the partial order of specific structural configurations.

The paper is structured as follows: After providing a short introduction to SDDs as a means of capturing structural properties in Section 3, we introduce TSSDs, which employ SDDs as basic building blocks for capturing joint structural and temporal properties, in Section 3. We then demonstrate the capabilities of TSSDs by showing that the Property Specification Pattern System proposed in [1,2] can be easily described in a compositional manner in Section 4 and outline how to systematically derive TSSDs from given textual description by a systematic stepwise transformation process in Section 5. Finally, the conclusions of the paper and outlook on future work is presented.

2 Specifying Structural Properties

The fundamental abstraction underlying our approach is the idea of interpreting instance situations of an object-oriented system as graphs. We map each object/value to a node and each attribute/association to an edge of a labeled graph. The theory of graph transformation systems (cf. [24]) 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.

The system structure is modeled using UML Class Diagrams characterizing all possible system states. Figure 1 provides the example that is used in the case study below, a networked system of elevators.

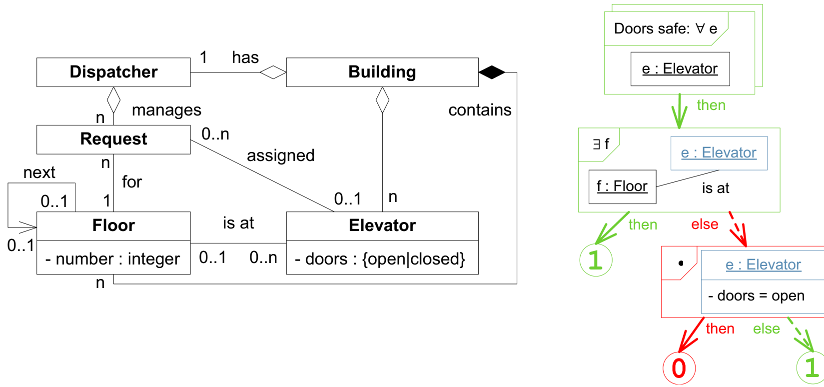


Fig. 1. Elevator class diagram — SDD illustrating basic syntax

Story Patterns are an extended type of UML Object Diagram (cf. [10]) that allow expressing properties as specific object configurations (in accordance with a given Class Diagram). They provide a notation for forbidding individual elements, but no negation of subgraphs, disjunction, implication, or modularity, which limits their use for the encoding of more complex properties.

Story Decision Diagrams (*SDD*) are an extension that remedies the shortcomings of Story Patterns while retaining an accessible visual notation. An SDD is a directed acyclic graph (DAG). Each node contains a *Story Decision Diagram Pattern* (*SDDP*) specifying an elementary structural property, basically a Story Pattern without forbidden elements. Pattern elements bound by one node remain bound in subsequent nodes.

During evaluation, the nodes are processed starting from the root node with an empty binding. 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 and follow the solid then connector, if no match is found, we leave the binding unchanged and follow the dashed else connector. When a binding reaches a (1) or (0) *leaf node*, it evaluates to *true* or *false*, respectively. SDDs are thus similar to decision trees, but allow sharing isomorphic subtrees and leaf nodes to reduce diagram size. Like in decision diagrams, consecutive conditions correspond to logical conjunction or, equivalently, implication, i.e. if a then b else c corresponds to $(a \Rightarrow b) \wedge (\neg a \Rightarrow c)$. SDDs allow multiple then or else connectors per node as a way of expressing alternatives.

As all pattern elements are positive, negation is expressed by 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, the leaf nodes are implied and can be omitted.[1]

SDDs allow **quantification** over the free variables of each node. Accordingly, we differentiate between existential nodes, which require that at least one of the bindings they propagate reaches a (1) leaf node, and universal nodes, which require this of every propagated binding. If an existential 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

[1] Note that color is used to make diagrams more readable, but redundant at the semantic level.

to links, it is marked with [∃ _]. If the node contains no free variables at all, it becomes a *guard* node – marked with [•] – that sends the original binding down the appropriate connector depending on the specified constraints on attributes. A universal node containing the free variables var_i is marked with [∀ var_i^+]. If no matching binding exists, the expected semantics of ∀ quantification require that the expression evaluate to true, i.e. an else connector to (1) is implied. Finally, it is possible to specify *cardinalities* for a node's then connector that constrain the number of extensions that may be generated for the same original binding. If too few or many bindings are found, the original binding is propagated down the else connector. The SDD in Figure 1 illustrates the basic concepts, requiring that every elevator is at a floor or else the doors are not open.

Most visual specification techniques lack the capability to compose complex properties by referencing other properties. SDDs support the composition of specifications by means of **Embedded Story Decision Diagrams** (ESDD). ESDDs are defined as patterns with free variables that are bound depending on the respective current context. The ESDD definition begins with a λ node that defines the pattern's name and rebinds variables of the host node to the local roles. If a node contains a reference to an ESDD, represented by the UML pattern symbol, a binding only matches the node if it also fulfills the embedded pattern. ESDDs are evaluated like normal SDDs, but introduce a local scope. In Figure 2, an ESDD is defined, expecting a floor and an elevator as its parameters.

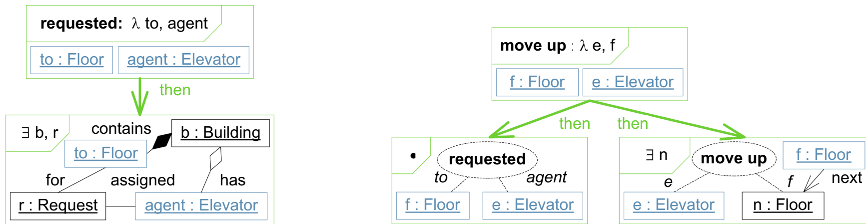


Fig. 2. Simple ESDD definition — Recursive ESDD definition

It is possible to define ESDDs recursively (see Figure 2), providing a way to encode reachability and other transitive properties. The formal semantics of SDDs is defined in [25], where we define the semantics of recursive ESDDs using least fixed points and demonstrate that ESDD evaluation terminates on arbitrary finite graph structures.

3 Specifying Temporal Properties

The temporal behavior of a system can be described as a sequence of states. As we model the system as a *graph transformation system*, each of these states corresponds to a graph. Between states, the identity of nodes and edges is preserved. 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 the specification of a structural property, and edges, constraining the ordering of incidences.

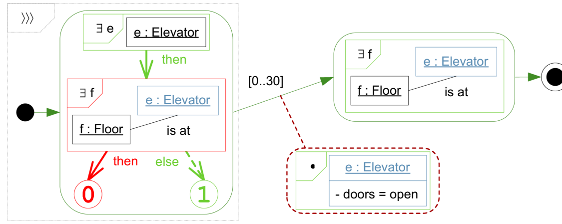


Fig. 3. Basic Example of the TSSD syntax

While TSSDs were designed with SDDs in mind, the structural properties could be encoded using any sufficiently expressive formalism.

Figure 3 is a basic example presenting the key elements of a TSSD. Whenever an elevator is not at a floor, it has to reach a floor within 30 time units. Meanwhile, the doors of the elevator must not be open, which is indicated by the (forbidden) guard on the transition. Using the overview in Figure 4, we now systematically introduce the elements of the TSSD syntax. The formal semantics of TSSDs is defined in [25].

Each node of a TSSD defines a **situation**. While a situation characterizes a set of states, calling it a state would be misleading, as multiple situations of the same TSSD can be incident, i.e. active, at the same time. A labeled situation can be referenced in other places in order to reduce diagram size and avoid redundancies. Bindings are shared between subsequent situations on the same path through the TSSD so that variables cannot be rebound in later situations. If bindings were not retained, it would be difficult to specify properties such as ’If an elevator is assigned a request, it needs to complete it.’ because any elevator completing any request would complete the scenario.

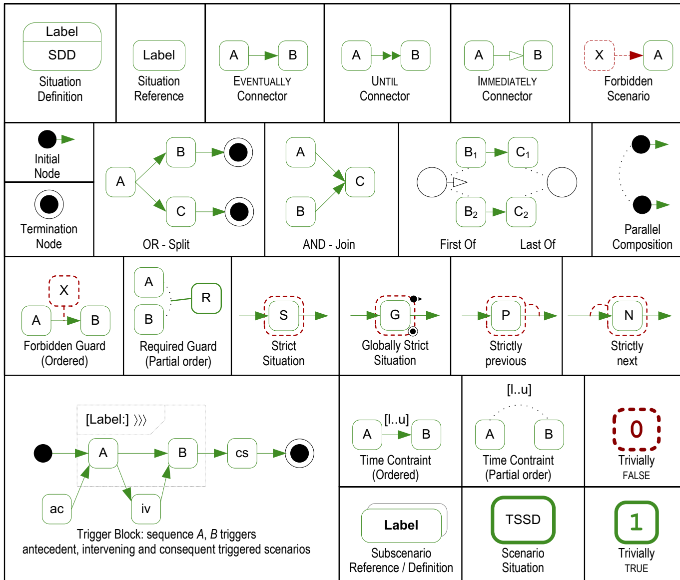


Fig. 4. Overview of the TSSD Syntax

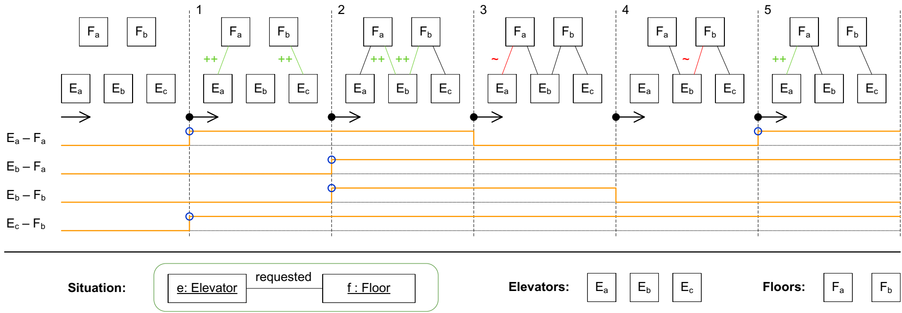


Fig. 5. The relationship between a situation and its observations

When matching a situation, its SDD generates a result set consisting of the alternative candidates, i.e. the bindings that satisfied the SDD. As SDDs may contain $\forall$ quantification and can then only be satisfied by *sets* of valid bindings, the result set actually contains candidate sets, even though these typically only contain a single binding. 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. Figure 5 illustrates this for a situation encoding that a specific elevator is requested at a specific floor. 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 (E_a, F_a), two observations are generated, one a time 1 and one at time 5.

The observations for a scenario 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 B is eventually made after an observation for situation A (or simultaneously). The *until connector* ($A \longrightarrow\!\!\!\!\blacktriangleright\!\!\blacktriangleright B$) denotes that an observation for situation A is made and that the specific observation remains valid until a compatible observation for situation B is made. Otherwise, the connector ceases to be enabled. 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. The connector is thus only enabled in a single state.

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 3 constrains the movement of a single elevator, independently of any other elevator. This same argument applies to multiple observations based on the same candidate set (such as the pair (E_a, F_a) in Figure 5) as well – a subsequent observation should not be invalidated just because the structure matched by the antecedent reappears. $A \longrightarrow B$ 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 bindings, a single observation can extend multiple traces. As a binding 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 control the scope of the scenario and encode logical operators. The initial node • always matches exactly once, as soon as possible. The descriptive, sequential character of TSSDs implies the assumption that time is bounded in the past. The termination node ⦿ marks the end of a branch of a TSSD and always matches as late as possible, i.e. the current state during runtime monitoring or the last state of a finite system. In conjunction with an ⟶⯮ connector, a ⦿ node can thus express that a property, e.g. safety, should hold globally. A trace starts at an initial node and is completed once it has reached a termination node. A system execution path π then fulfills a TSSD if a completed trace to a ⦿ node exists within a prefix of π.

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. Disjunction can be expressed using multiple ⦿ nodes on independent branches, as one successfully completed trace is sufficient. 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. Predecessor branches that do not begin in an initial node can be used to make statements concerning 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. If there are multiple initial nodes in a single diagram, we require a satisfying trace from every initial node, which can be used to create the parallel composition of multiple TSSDs.

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 (dark) green lines, while forbidden situations and connectors use dashed (dark) red lines. Forbidden scenarios are defined by means of *inhibiting connectors*. Normally, a connector is disabled and becomes enabled when it is reached by an appropriate trace. Inhibiting connectors are enabled and become disabled if a trace reaches them. Inhibiting connectors mark the end of a forbidden scenario and thus are the connectors leading from forbidden to required elements. In the presence of an inhibiting connector, the subsequent required situation is thus only enabled if *no* trace completing the forbidden branch exists. The semantics of all other situations and connectors in a forbidden scenario is unchanged. Forbidden scenarios may branch and join in any situation of the diagram. If they join in a ⦿ node, they must never occur. As multiple ⦿ nodes represent alternatives, a required and a forbidden scenario leading to different ⦿ nodes still represent alternatives.

Additional **guards and time constraints** can appear directly on the temporal connectors defining the ordering of situations or on dedicated *constraint edges* connecting any two situations regardless of their relative position in the diagram. Constraint edges have no direction. Guards are situations that are forbidden *between* two situations. They are drawn with a bolder dashed border and connected to the connector or constraint edge in question. For convenience, the notation provides support for specifying required guards as well. We also directly support some commonly used idioms

in connection with guards: While the observation semantics ensure that the same observation could not have been made earlier, we may want to require that the situation should not yet have matched at all. While this can be achieved using a dedicated guard, we allow 'bending' the forbidden guard onto the situation itself, which then becomes *strictly next*. Likewise, a *strictly previous* situation needs to be the *last* possible observation. Finally, a *strict* situation without any connectors is forbidden between any two situations unless explicitly required, or, if it is globally strict, also between situations and pseudostates.

Time constraints allow setting a lower bound l and an upper bound u for the permitted delay between two observations for two situations A and B within the same trace or related traces. A time constraint can either be placed directly on a temporal connector ($A \xrightarrow{[l...u]} B$) or on a dedicated constraint edge ($A \cdots [l \ldots u] \cdots B$). In case of multiple constraints, the more restrictive bounds dominate the less restrictive ones. The dedicated pseudostates *first of* and *last of* allow specifying (time) constraints between the earliest and the latest observation for two sets of situations.

TSSDs provide **quantification** on different levels. 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. E.g., if an elevator needs to complete any one accepted request, it is not sufficient to only match the first accepted request. The TSSD keeps matching freshly accepted requests, as it might otherwise miss the one that is actually completed. This is the reason why a TSSD, which represents a *set* of (potential) scenarios, can 'be' in many states at once. Candidate sets provide another level of quantification, e.g. 'If all requests are approved (at the same time), then eventually all (these) requests have to be completed (at the same time)', which could be written by universally quantifying over all approved requests in the first situation.

The characteristic response and precedence relationships in scenarios are encoded by means of **trigger** blocks. Whenever the sequence within the trigger block has been observed in its entirety, the corresponding trace becomes a *root trace*. The TSSD is then only fulfilled if every 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. Borrowing from Live Sequence Charts [15], we distinguish between *universal* TSSDs, which possess a trigger and need to be fulfilled every time it matches, and *existential* TSSDs, which are implicitly triggered by their initial node and need to match once during the execution of the system. It is possible to have multiple triggers in the same TSSD. An important feature is the ability to trigger antecedent and intervening triggered scenarios. In order to support this, only those previous situations that are directly connected to an initial node are considered as preconditions when evaluating whether a trigger block is completed. A trigger containing the trivially *true* situation matches in every state and allows encoding properties such as fairness ('always occurs eventually *again*').

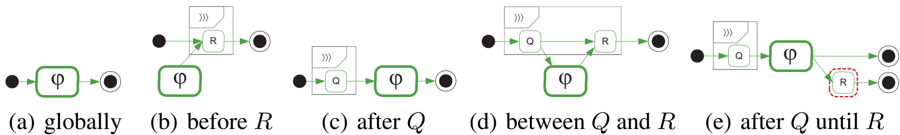
Subscenarios provide a way to define and reference whole scenarios and thus provide a concept for modularity. A subscenario definition begins with a special λ situation for rebinding roles and parameters. A subscenario invocation works in a similar fashion

to ESDD invocations, however, as we may need to reference bindings that have been created by the subscenario, the invocation itself needs to take place inside a second λ-node that allows exporting the generated bindings. *Scenario situations* are situations that contain another TSSD and can be seen as in-place subscenario definitions. Most notably, they can be used as 'parentheses' for encoding $\vee$-joins. A subscenario is evaluated in the given context of the surrounding scenario, i.e. its initial and ⦿ node cannot match earlier/later than the subscenario's predecessor/successor situations.

The expressiveness of TSSDs is discussed in [25], where we show that any LTL and TPTL (LTL with clocks) formula can be encoded using TSSDs.

4 Specification Pattern System

The Property Specification Pattern System (cf. [1],[2]) proposes to address the problem of making formal specification techniques and verification accessible to practitioners. The idea is to allow users to construct 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). We now 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.

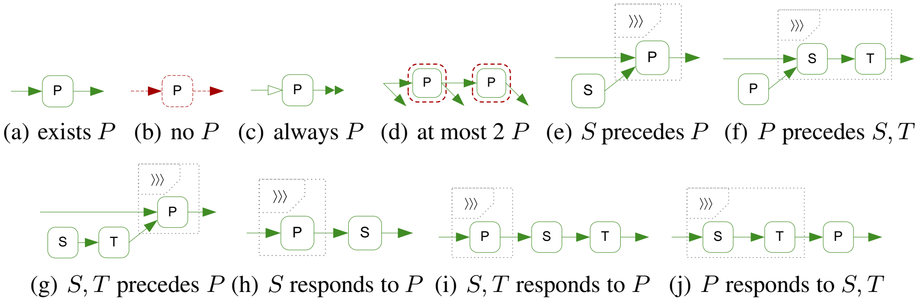


(a) globally (b) before R (c) after Q (d) between Q and R (e) after Q until R

Fig. 6. The scopes encoded as TSSDs (for a property φ)

In Figure 6, we define the scopes as TSSDs. The scopes *before*, *after*, *between*, and *until* are 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 ⦿ 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 (⦿) instead of introducing some additional, less obvious syntax for a $\tilde{\mathbf{U}}$ connector.

In Figure 7, we define the ten different properties. 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

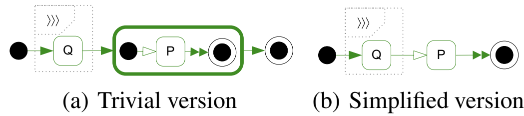


(a) exists P (b) no P (c) always P (d) at most 2 P (e) S precedes P (f) P precedes S, T

(g) S, T precedes P (h) S responds to P (i) S, T responds to P (j) P responds to S, T

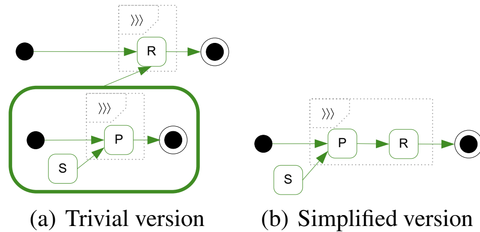
Fig. 7. The properties φ encoded as TSSDs)

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.



(a) Trivial version (b) Simplified version

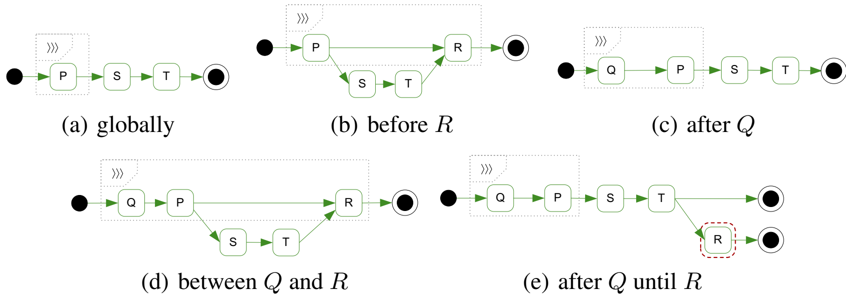
Fig. 8. Always P after Q



(a) Trivial version (b) Simplified version

Fig. 9. Always S precedes P before R

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). While the trivial form of each combined pattern obtained using this mechanistic approach already yields useable results, simplified versions 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 (see Figure 8). Secondly, if both the surrounding scenario and the scenario situation contain trigger blocks, these blocks are merged (see Figure 9).



(a) globally (b) before R (c) after Q

(d) between Q and R (e) after Q until R

Fig. 10. Response (1,2), simplified versions

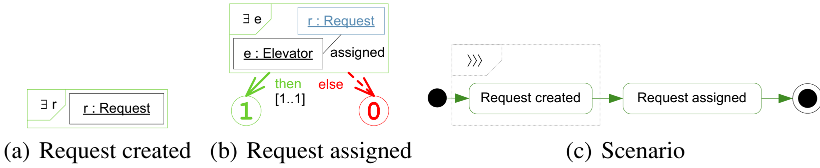
Figure 10 lists all simplified variants of the (1,2) response chain pattern. Note how the simplified forms are quite natural expressions of the original requirements. Disregarding the Specification Pattern System's distinction between scopes and properties, e.g. S, T responds to P after Q actually translates to 'after the sequence Q, P, the sequence S, T needs to follow', which is exactly what the TSSD expresses.

5 Deriving Specifications 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 [2], 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 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 1 from the requirements. As the safety property (1) is a structural requirement, we encode it as an SDD. As suggested by the vertical lines, we decompose the textual requirement into the semantically relevant blocks | every elevator | not at a floor | 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 1.

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 |. We first encode the two structural terms $\exists$ request (Figure 11(a)), and $\exists$ elevator assigned to request with a cardinality of $[1..1]$ (Figure 11(b)). | Every time | ... | then | becomes a trigger block around $\exists$ request, while



(a) Request created (b) Request assigned (c) Scenario

Fig. 11. Deriving a TSSD

| afterwards | becomes an eventually connector (Figure 11(c)), resulting in the TSSD in Figure 12(a).

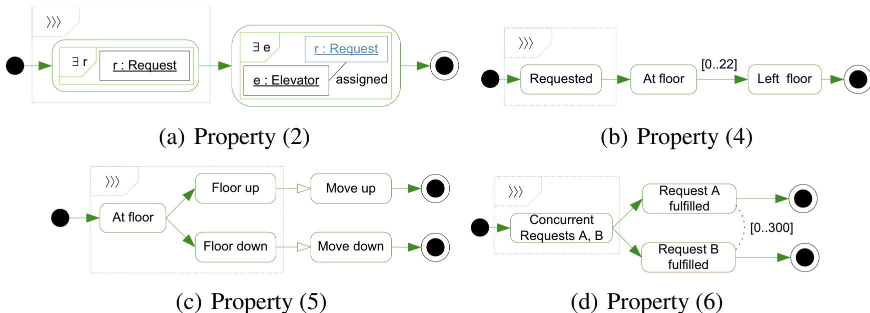
Property (3) is encoded using the same schema, but additionally introduces a time constraint [0..30] between the two situations. Combined with a guard enforcing requirement (1), this yields Figure 3.

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] |. The trigger block includes the first two situations, while there are two alternative triggered scenarios, resulting in the TSSD in Figure 12(b).

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, eventually elevator at next floor for the up-direction. The triggered scenario is then simply encoded by the recursive ESDD in Figure 2 that traverses floors, upwards, until it finds a request or fails. The result is seen in Figure 12(c).

Property (6) becomes | Every time | two concurrent requests exist | then | each | is eventually | completed | within 300 seconds of the other |. After the trigger block, the | each | introduces two ∨-branches. The | within | time constraint results in a constraint edge across the two branches, yielding the TSSD in Figure 12(d).

Property (7) is the famous example that results in a rather unwieldy LTL formula (cf. [2]). It can be expressed using the bounded-existence-between pattern above. 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 when it is 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 | then | it eventually | is at the floor | for the first time | and eventually opens its doors | or eventually | is at the floor | for the second time | and eventually opens its doors |. It is the explicit



(a) Property (2) (b) Property (4)

(c) Property (5) (d) Property (6)

Fig. 12. Requirements encoded as TSSDs

requirement | for the first/second time | that turns being at the floor into a strict situation. 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 first structural property is encoded by the ESDD in Figure 2. Property (7) is then encoded by the TSSD in Figure 13.

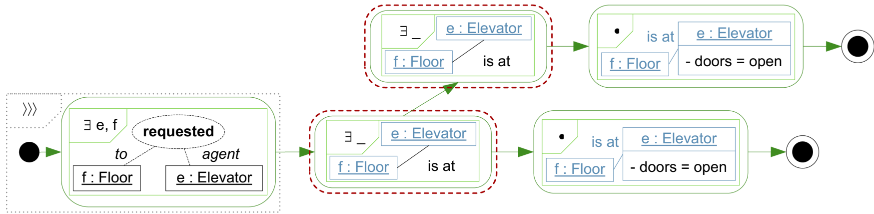


Fig. 13. The elevator may only arrive twice before opening the door

6 Conclusion and Future Work

The combination of TSSDs with SDDs allows the joint specification of structural and temporal properties that is required in the context of complex software-intensive systems, in particular self-adaptive systems. We have shown that all Property Specification Patterns proposed in [1,2] can be easily encoded and derived in a compositional manner using TSSDs (see [25] for the full catalogue). We have demonstrated that the mapping from textual property descriptions to SDD/TSSD specifications is fairly direct as the notations support many common intuitions (implication, precedence etc.).

We are currently developing tool support for the specification, monitoring, and verification of TSSDs, which will allow us to further evaluate them in larger case studies with requirements engineers and domain experts.

References

1. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property Specification Patterns for Finite-state Verification. In: 2nd Workshop on Formal Methods in Software Practice, ACM Press (1998)
2. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: ICSE '99: Proceedings of the 21st international conference on Software engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1999) 411–420
3. Musliner, D., Goldman, R., Pelican, M., Krebsbach, K.: Self-adaptive software for hard real-time environments. IEEE Intelligent Systems **14** (1999)
4. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An Architecture-Based Approach to Self-Adaptive Software. IEEE Intelligent Systems **14** (1999) 54–62
5. Object Management Group: UML 2.0 Object Constraint Language (OCL) Specification (2003) http://www.omg.org/docs/ptc/03-10-14.pdf.
6. Diskin, Z., Kadish, B., Piessens, F., Johnson, M.: Universal arrow foundations for visual modeling. In: Diagrams '00: Proceedings of the First International Conference on Theory and Application of Diagrams, London, UK, Springer-Verlag (2000) 345–360
7. Kent, S., Howse, J.: Mixing visual and textual constraint languages. In France, R., Rumpe, B., eds.: UML'99, Fort Collins, CO, USA, October 28-30. 1999, Proceedings. Volume 1723 of LNCS., Springer (1999) 384–398

8. Kent, S., Howse, J.: Constraint trees. In Clark, T., Warmer, J., eds.: Object Modeling with the OCL. Springer (2002) 228–249
9. Bottoni, P., Koch, M., Parisi-Presicce, F., Taentzer, G.: A visualization of OCL using collaborations. Lecture Notes in Computer Science **2185** (2001) 257–271
10. Köhler, H., Nickel, U., Niere, J., Zündorf, A.: Integrating UML Diagrams for Production Control Systems. In: Proc. of the 22nd International Conference on Software Engineering (ICSE), Limerick, Irland, ACM Press (2000) 241–251
11. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (2000)
12. Konrad, S., Cheng, B.H.C.: Real-time specification patterns. In: ICSE '05: Proceedings of the 27th international conference on Software engineering, New York, NY, USA, ACM Press (2005) 372–381
13. Object Management Group: UML 2.0 Superstructure Specification. (2003) Document ptc/03-08-02.
14. Sengupta, B., Cleaveland, R.: Triggered Message Sequence Charts. In Griswold, W.G., ed.: Proceedings of the Tenth ACM SIGSOFT Symposium on the Foundations of Softare Engineering (FSE-10), Charleston, South Carolina, USA, ACM Press (2002)
15. Harel, D., Marelly, R.: 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).
16. Alfonso, A., Braberman, V., Kicillof, N., Olivero, A.: Visual Timed Event Scenarios. In: ICSE '04: Proceedings of the 26th International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2004) 168–177
17. Bradfield, J., Kuester Filipe, J., Stevens, P.: Enriching OCL Using Observational mu-Calculus. In Kutsche, R.D., Weber, H., eds.: Fundamental Approaches to Software Engineering (FASE 2002), Grenoble, France. Volume 2306 of LNCS., Springer (2002)
18. Cengarle, M., Knapp, A.: Towards OCL/RT. In Eriksson, L.H., Lindsay, P., eds.: Formal Methods – Getting IT Right, International Symposium of Formal Methods Europe, Copenhagen, Denmark. Volume 2391 of LNCS., Springer (2002) 389–408
19. Flake, S., Mueller, W.: An OCL Extension for Real-Time Constraints. In: Object Modeling with the OCL: The Rationale behind the Object Constraint Language. Volume 2263 of LNCS. Springer (2002) 150–171
20. Gadducci, F., Heckel, R., Koch, M.: 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. (2000) 310–322
21. Rötschke, T., Schürr, A.: Temporal Graph Queries to Support Software Evolution. In: Graph Transformation: 5th International Conference, ICGT 2006, Rio Grande do Norte, Brazil, September 17-23, 2006. (2006) 1–15
22. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story diagrams: A new graph rewrite language based on the unified modeling language. In Engels, G., Rozenberg, G., eds.: Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany. LNCS 1764, Springer Verlag (1998) 296–309
23. Giese, H., Klein, F.: Beyond Story Patterns: Story Decision Diagrams. In Giese, H., Westfechtel, B., eds.: Proc. of the 4th International Fujaba Days 2006, Bayreuth, Germany. Volume tr-ri-06-275 of Technical Report., University of Paderborn (2006)
24. Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformation : Foundations. World Scientific Pub Co (1997) Volume 1.
25. Klein, F., Giese, H.: Integrated Visual Specification of Structural and Temporal Properties. Technical Report tr-ri-06-277, Computer Science Department, University of Paderborn (2006)