Policy Enforcement and Verification with Timed Modeling Spider Diagrams

Paolo Bottoni
Dipartimento di Informatica - Università di Roma Sapienza
Email: bottoni@di.uniroma1.it

Andrew Fish
School of Computing, Engineering and Mathematics,
University of Brighton, UK
Email: andrew.fish@brighton.ac.uk

***Abstract*—Timed Modelling Spider Diagrams (TMSDs) are a visual language which supports the modeling of object-oriented systems with time constraints. They are used to define policies in which TMSDs specify admissible evolutions of the state of some instance. We define a process for deriving a rewriting system from a policy specification, so that the generated system defines a language of sequences of basic TMSDs satisfying the policy. Moreover, by identifying the different ways in which the constraints set by the policy can be violated, we can produce special rules whose application results in erroneous sequences. The resulting transformation systems can be used both to simulate possible behaviours when reasoning on the definition of policies and to test policy verification algorithms.**

I. Introduction

Temporal policies are a means of specifying constraints on the evolution of systems, to which any model of the dynamics of that system, or any operational realisation of the system, must adhere, independently of the way in which this dynamics is defined. The following types of problems ensue: (1) if the dynamics is completely dictated by the progress of time, suitable mechanisms must be devised to enforce system evolution as time-outs are reached; (2) if the system can evolve based on external events, checks must be dynamically executed to ensure that system reactions are consistent with the temporal constraints on the possibility of performing some transitions; (3) if systems present hybrid characteristics, i.e. both reactivity to external events and time-based evolutions, the consistency of the resulting dynamics must be checked.

We approach the problem of checking the consistency of dynamics, based on a recent extension of Spider Diagrams, allowing the specification of temporal policies expressing constraints on the intervals at which elements of some given type can be in some given state [1]. Spider Diagrams (SDs) are a well known notation for visual modeling and reasoning based on Euler Diagrams (EDs). In an SD a *spider* is a tree with nodes in zones determined by a set of curves; it is used to indicate the presence of an element in one of the corresponding subsets. In a series of papers [1], [2], [3], we have extended SDs in two directions. First, we have introduced the possibility of annotating SDs with temporal information, to specify temporal constraints on the intervals at which a set or a subset can exist or is allowed to be empty, or an element can be in a given subset, or exist at all. These Temporal SDs (TSDs) can also be used to describe actual configurations of a system, i.e. which elements are in which subset at which interval of time. The usual notion of derivation between two SDs can then be extended to check that a configuration d_1 derives from a configuration d_2, while satisfying the temporal constraints established on d_2. Second, we have provided an object-oriented interpretation of a restricted form of SDs, defining Modeling SDs (MSDs) at two levels. At a specification level, spiders represent types and curves represent states. If a spider "is in" a state, then the state is an admissible one for instances of that type. At an instance level, a spider represents an instance: if it inhabits a zone this means that the instance is in the state represented by the zone. By putting together the two extensions we obtain Timed Modeling SDs (TMSDs) where we can express constraints on the intervals at which an instance of a type can be in a given state, or take snapshots of a configuration of instances and check their conformance with the constraints expressed at the type level. We have used TMSDs to define *policies*, expressing admissible evolutions of the state of the instances, and introduced the notion of *story*, as an actual evolution of some instance, conforming to the policy.

In this paper, we define a method to derive all the possible stories conforming to the specification of a policy, by constructing visual rewriting rules whose firing is guarded by some temporal constraint, derived from the temporal annotations associated with the TMSD in the policy. In particular, we assume the availability of a universal clock associated with a policy and of timers set according to the intervals specified in the policy, and generate a set of rules describing the possible evolutions of the state of instances of that type, conditioned on the time marked by the clock. For each evolution triggered by some external event, the event must occur at a time within the interval in which the antecedent of the rule is valid, while for evolutions dictated by time-outs, the rule is assumed to be applied exactly at the time separating the interval of validity of the antecedent and the interval associated with the consequent.

The generated rules can be used for different purposes, such as to simulate possible runs of the system, to assess temporal properties of the system via model-checking, or to perform static analysis to detect conflicts in the policy. Moreover, we generate rules whose execution results in the violation of some constraint, so that they can be used to test the correct implementation of the constraint-checkers (if an execution is simulated, the constraint checker should detect the violation).

One aim of developing the relatively simple form of the visual language of TMSDs for policy specification is to provide a means of representation acting both as a formal specification due to the underlying theoretical machinery and as an accessible front-end for use by all stakeholders, including policy makers. The definition of the theoretical aspects is propaedeutic to the production of interactive tools, to be tested,

together with the notation, on the stakeholders.

Section II provides background on SDs and their timed and modeling extensions, introducing a running example of policy specification. Section III presents a notion of rewriting for TMSDs, whilst Section IV illustrates the procedure to generate rewriting systems defining the language of the stories for a policy. In Section V we prove the correctness of this procedure and discuss how to construct invalid sequences. We discuss related work in Section VI and conclude in Section VII.

II. Background on TMSDs

An *Euler diagram* (ED) is a collection of labelled simple closed curves in the plane, decomposing it into connected *minimal regions*. A *zone* is a region *inside* one set of curves and outside the remaining curves; zones may be *shaded*. All diagrams have a "boundary contour", drawn as a rectangle and labelled by U; all regions are inside U. The semantics of EDs are given by asserting that the interior of the curves represent sets; this extends naturally to the intersection, containment and disjointness of sets; shading places an upper bound on the set cardinality (a shaded zone denotes an empty set in an ED).

A *unitary spider diagram*[1] (SD) is an ED augmented with *spiders*, i.e. trees whose vertices (called *feet*) are placed in zones; no two vertices of the same tree lie within the same zone. An *existential* spider, depicted as a circular dot, denotes some generic element; a *constant* spider, depicted as a squared dot, denotes a specific individual. The *habitat*, i.e. the set of zones inhabited by a spider, determines the set containing the element represented by the spider: if a zone is shaded, the only elements in the represented set are those represented by the spiders in it. Definition 1 formalises these concepts.

Definition 1 (Spider Diagram): A *unitary spider diagram* d is a tuple (C, Z, sh, S, h) such that:

- $C = C(d)$ is a finite set of *curve* labels with *boundary curve label* $U \in C$.
- $Z = Z(d) \subseteq \{(X, C \setminus X) \mid X \subseteq C \wedge U \in X\}$ is a finite set of *zones*, with $(\{U\}, C \setminus \{U\}) \in Z$, and $c \in C \implies \exists X \subseteq C[c \in X \wedge z = (X, C \setminus X) \in Z]$.
- $sh : Z \to \mathbb{B}$ is a Boolean *shading* function on zones. A zone z for which $sh(z) =$`true` is said to be *shaded*.
- $S = S(d)$ is a finite set of *spider* labels, partitioned into the subsets of *existential*, S_e, and *constant*, S_c, spiders.
- h is the *habitat* function $h : S \to \mathcal{P}(Z) \setminus \{\emptyset\}$ Each unique pair $(s, z) \in S \times Z$ with $z \in h(s)$ is called a *foot* of s. Let $F = F(d)$ denote the set of all feet in d.

We call diagram $e_d = (C, Z, sh)$ the *underlying ED* for d.

The notion of SDs was extended in [1], [2] to incorporate temporal specifications by annotating elements of d with *intervals* and *interval specifications*. The latter generalise intervals to permit the use of variables (distinguished between *time variables* ranging over timestamps, and *arithmetic variables*, ranging on natural numbers), together with sets of constraint over these variables. A *valid* assignment of values to variables satisfies the conjunction of all the constraints over a set of interval specifications. A *timed-SD* is an SD d together with a function ω assigning interval specifications to every curve, zone, shaded zone, spider, or spider foot in d. Given a timed-SD d and an assignment $\mathcal{A}$ which is valid for all the interval specifications in d, $\mathcal{A}$ is *time-consistent* if it respects the natural constraints on the lifetime of diagram syntax (i.e. feet can only exist within the time period that both the spider and zone live; shading can only be present when its zone lives; zones must live within the lifetime of the curves).

Fig. 1 shows an example of timed-SD, based on the law about coming of age in Italy, passed on March 10, 1975 and still valid on the day the diagram was drawn. The annotations indicate that `paolo` (a specific person represented by a constant spider) became of age on November 29, 1978 and will remain so until some moment in the future (he is still alive at the time of drawing, as indicated by the constraint on the temporal variable Y). Any `person`, represented by an existential spider, born after March 10, 1975 (the birth date providing an assignment to the temporal variable Z) will (or has) become of age on the day after his or her 18th birthday (the exact number of days depending on the number of leap years to that date, as calculated by a function $leap$) and will remain (or has remained) of age for the rest of his or her life, as indicated by the temporal variable Q. Note that even if Y and X have equal constraints, they can receive different assignments, one at the time `paolo` ceases to be represented, and one when (and if) the law changes. All intervals and interval specifications are given at the granularity of days, as indicated by the subscript `D`. The operator $\oplus$ denotes that the number of units (at the granularity level indicated by the subscript) in the arithmetical expression on the right must be added to the timestamp or temporal variable on the left.

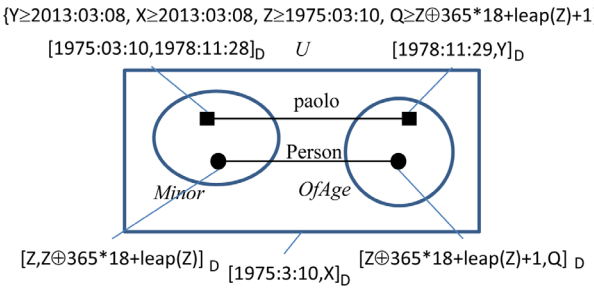

Fig. 1. A timed-SD concerning coming of age in Italy.

A *basic timed-SD* associates an interval specification with an entire diagram instead of individual elements (i.e. the same interval is assigned to every diagrammatic element in d). A sequence $\mathcal{S}$ of basic timed-SDs is *contiguous*, with respect to an assignment $\mathcal{A}$, if $\mathcal{A}$ produces contiguous intervals for consecutive diagrams in $\mathcal{S}$. The resulting sequence $\mathcal{S}'$, has the same diagrams as $\mathcal{S}$, but annotated with intervals evaluated according to $\mathcal{A}$. Then, $\mathcal{S}'$ represents the evolution of a system over a time period, with each diagram d'_i in $\mathcal{S}'$ specifying the system configuration over the interval of existence for d'_i.

A second extension to SDs, permits the expression of type vs. instance information, providing a closer link to the object-oriented modelling paradigm than standard SDs [3], [1]. In a *type*-SD d, all spiders are constant spiders named by types and all curves are named by states. In an *instance*-SD d' all spiders

[1]In this paper, we deal only with unitary SDs for simplicity.

are constant and represent instance identifiers, all curves are named by states, no zone is shaded and every spider has exactly one foot. Then d', together with a surjective function Θ associating each spider in d' with a spider in a type-SD d, is an *instance-SD for* d. Type-SDs place constraints over the admissible states for a set of types, while instance-SDs present configurations of instances in some states, to be checked for conformance to the constraints on their respective types.

The two extensions of SDs were brought together in [1] to define timed-SD policy specifications (or *policy* for short); we paraphrase the definition from there.

Definition 2 (Policy): A *policy* is a construct $\Pi = (\textit{validity}, \textit{trigger}, \textit{condition})$ where:

1) *validity* is an interval specification $[P, Q]$, where each of P and Q is either a fixed timestamp or a temporal variable associated with an event.
2) *trigger* is a type-SD d annotated with W, W being either the special temporal variable *WHEN*, or a fixed timestamp such that: a) if P is also a timestamp, $P \leq W$; b) if Q is also a timestamp, $W \leq Q$.
3) *condition* is a set of sequences of basic timed type-SDs, where each sequence is bound to be contiguous, and the only admitted time variable is *WHEN*, and only if it appears in the trigger. For simplicity, we will assume an identity between the time at which the trigger occurs and the starting time of each sequence.

Fig. 2 shows an example of a policy for a parking system, where: (a) the policy is valid from the time that the meter is in place until a policy change; (b) the trigger is the event that a car starts enjoying free parking; (c) the car parking state may evolve as follows: after a car enters the free parking state it can remain there for any length of time X up to 60 minutes, after which it must either be running or be toll-parking for at least 120 minutes (indicated by the constraint at the bottom) before it can return to free parking. We adopt a slight modification with respect to the analogous policy presented in [1], by explicitly representing within the policy the constraint on the next possible occurrence of the trigger, in order to simplify the generation of rules, which would otherwise require the management of a special case. This, however, does not dictate that the car return to free parking, as Y has only a lower limit.

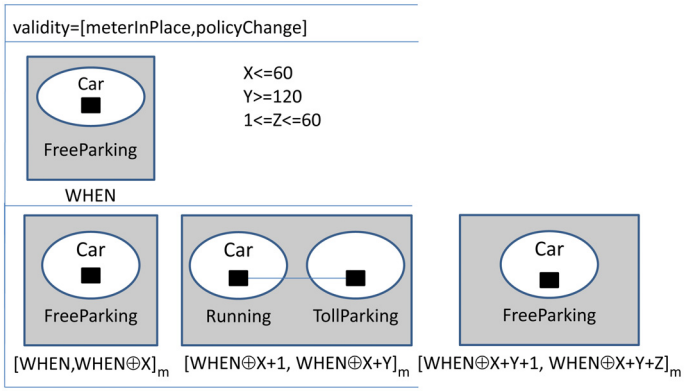

Fig. 2. A simple example of a policy for a parking system.

In this simple case, the condition presents only one sequence and the interval specifications make use of the arithmetic variables X, Y and Z, and of the variable *WHEN*, also used to annotate the type-SD in the trigger. The sequence at the top of Fig. 3 is an example of a valid *story* for the policy (i.e. a contiguous sequence of instance-SDs satisfying the corresponding type-SDs), whilst the sequence at the bottom is not a valid story, since the instance $ZX12$ of type Car returns to the *FreeParking* state before it is permitted to do so.

III. Rewriting systems for SDs

In order to provide a mechanism to test policies, we need to introduce a notion of transformation of instance-SDs through which to model their dynamic behaviour and check their consistency with policies. In particular, due to the characteristics of policies, we only need to consider instance-SDs, which present only curves, zones (without considering the shading function) and spiders, where a spider is composed of a single foot living inside some zone. We adopt a simple model for spider diagram rewriting, based on the following rule schemes for atomic operations:

- **Spider creation.** A foot of a new spider is added to a zone.
- **Spider deletion.** The single foot of an existing spider is removed from the diagram.
- **Spider movement.** The single foot of a spider is moved from one zone to another zone.

Fig. 4 gives a visual representation of the three rule schemes for spider creation, deletion and movement in the form of partial morphisms between typed graphs, where the types represent the different sorts of elements. Since spiders are composed of single feet, we can simply present the modifications of the relation of a spider with the single zone it inhabits. Each scheme is associated with a rule signature. In particular, we define three signatures, shown in Table I, and describe their effect when applied to a diagram $d_1 = (C_1, Z_1, S_1, h_1)$ by presenting the variations from d_1 in the resulting diagram $d_2 = (C_2, Z_2, S_2, h_2)$. Which diagram plays the role of d_1 or d_2 is derivable from the orientation of the arrows in Fig. 4.

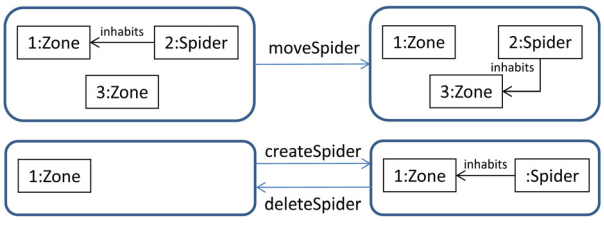

Fig. 4. The visual rule schemes for spider creation, deletion or movement.

The presence of the same identifier in the left- and right-hand sides of a scheme indicates that the element is preserved in the transformation. An element not identified across the two sides indicates that the element is created, if it appears only on the right-hand side, or deleted, if it appears only in the lefthand sides. All elements not mentioned in a rule scheme are left unchanged by the application of a concrete rule following

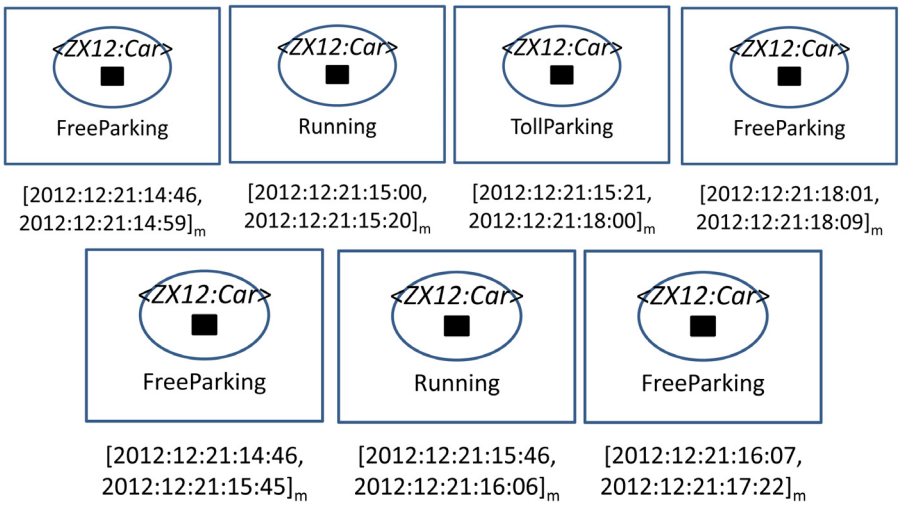

Fig. 3. Examples of valid (top) and invalid (bottom) stories for the policy in Fig. 2.

that scheme. All the curves and zones are always preserved, as these represent the states on which the policy is defined.

The effect on the habitat function is described with reference to its representation as a collection of pairs (spider, zone). Each scheme is instantiated with elements of the sorts indicated in its arguments. Concrete rules are defined by instantiating the parameters in the rule schemes, as indicated in Table I. In the signature, `Label` indicates an element which is a label, either already present in d_1 or to be introduced in it, `SoL` indicates a set of labels and `Pair` constructs pairs of elements.

TABLE I. SIGNATURES AND EFFECT FOR RULE SCHEMES.

1) **createSpider**(`Label`,`Pair`(`SoL`, `SoL`))
   createSpider($l, (X_1, Y_1)$)=d_2), where $l \notin S_1$, $S_2 = S_1 \cup \{l\}$ and $h_2 = h_1 \cup \{(l, (X_1, Y_1))\}$
2) **deleteSpider**(`Label`)
   deleteSpider(l)=d_2, where $l \in S_1$ and $S_2 = S_1 \setminus \{l\}$ and $h_2 = h_1 \setminus \{(l, h_1(l))\}$
3) **moveSpider**(`Label`, `Pair`(`SoL`, `SoL`), `Pair`(`SoL`, `SoL`))
   moveSpider($l, (X_1, Y_1), (X_2, Y_2)$)=$d_2$, where $l \in S_1$ and $h(l) = (X_1, Y_1)$ and $h_2 = h_1 \setminus \{(l, (X_1, Y_1))\} \cup \{(l, (X_2, Y_2))\}$.

The rewriting rules presented here differ from reasoning rules[2] for SDs, see e.g. [4], in that reasoning rules are sound, (i.e. the syntax changes indicated by the rule imply logical inference) and are used to derive correct implications out of a state of affairs represented by an SD. On the contrary, rewriting rules are not required to be sound, as they are transformation rules modeling possible evolutions of a state of affairs. In principle, the integration of rewriting and reasoning rules can be used to derive updated information out of each new configuration. A *Delete Spider* rule is also included in the reasoning system of [4], where it deletes all feet of a spider, if none of them is included in a shaded zone. This is equivalent to rewriting according to `deleteSpider` in the context of instance-SDs (no zone is shaded, all spiders have a single foot).

[2]Which also actually define rule schemes.

In the context of policy modeling, we assume that all rewriting processes are conducted with reference to an ED which remains stable during the whole process and which is derived from merging all of the underlying EDs of the SDs in the policy, as described in Section IV.

IV. GENERATING REWRITING SYSTEMS FROM POLICIES

In order to generate a rewriting system for the production of stories conformant to a policy, we enhance timed modeling SDs (i.e. instance-SDs and type-SDs) with two new types of element: `Timer`, with an attribute `end`, and `Clock`, providing a method `now()` which performs time observations and returns the corresponding timestamp. In particular, we associate timers with interval specifications in a policy. An assignment α, defined for the variables in a specification s, induces a valuation $\mathcal{V}_\alpha$ of s, from which an interval $i_s = \mathcal{V}_\alpha(s)$ is derived. A corresponding instance of `Timer` will have the value of `end` set to the latest timestamp admissible for i_s, or `UNDEFINED` if there is no such timestamp, i.e. the end of the interval is unconstrained. A special *interval* timer τ_{val} is defined to store information on the validity interval $[P, Q]$ for the policy. If Q is a variable associated with an event, then τ_{val}.`end=UNDEFINED`. A single instance of `Clock`, χ, is present in each diagram, marking a universal time. A rewriting process starts with an axiom, which is a basic timed instance-SD composed of the diagram in the policy trigger together with a timer defined according to an observation on χ. Namely, if the trigger is associated with the temporal variable $WHEN$, then this variable is assigned the value returned by performing `now` within χ, possibly determining the value of `end` for the timer. The timer will then be replaced with new ones, as the transitions progress through the diagrams in a sequence.

Rule application is conditioned on satisfying the temporal constraints set by the policy. A condition is uniformly imposed: each rule is allowed to fire only if the timestamp returned by χ.`now()` is within the validity interval for the policy, as recorded in τ_{val}. We do not present conditions of this latter type, nor do we explicitly represent τ_{val}, in the concrete representation of rules. Transitions are considered to be instantaneous, but each application of a rule is preceded

by a time observation, i.e. an invocation of `now()` on χ. A condition of *time progression* holds: for a sufficient long iteration of rules, χ.`now()` will return increasing values.

We present the procedure for generating rules from a policy with reference to the pseudocode in Table II. The procedure starts by initialising the `ED` variable to the ED underlying the *trigger* type-SD, as evaluated by the `createED()` function, omitting the spiders and the shading in it. After that, the procedure iterates through all of the sequences in the condition to *merge* all of the underlying EDs in the policy. To this end, for each `basic` diagram in a sequence the function `EDmerge()` is applied, having as arguments the current diagram in `ED` and the ED underlying the basic diagram under consideration, stripped of shading, as resulting from the application of `extractED()`. The overall effect is to construct an ED comprising the set of all the curves appearing at least once in the policy and a minimal set of zones which *cover* the zones of the underlying EDs[3] in the policy (i.e. all of the zones appearing in any underlying ED in the policy can be obtained by removing some of the curves of the constructed ED, and no zone appears in the constructed ED which is not in the underlying ED for some diagram in the policy). Through this merging proccess, all the generated rules are defined on the same underlying ED. As an example, Fig. 5 shows the underlying ED on which the car parking policy is defined.

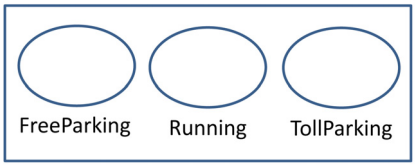

Fig. 5. The merged ED for the parking policy of Fig. 2.

The rest of the procedure creates a collection of rules whose application produces the admissible changes in habitats of spiders according to the sequences in the policy conditions. Each rule is defined on the underlying ED produced by the previous steps. We assume that each sequence in *condition* starts with the trigger and we add a rule to `RSPS[i][k][s]` for each change of habitat of the spider `s` derived from a comparison of the two SDs, named `L` and `R`, obtained with `extractSD` from the k-th pair of the i-th sequence of *condition*[4]. To enable this comparison the procedure computes the `habitat` of `s` in the two diagrams `L` and `R`. Each rule is an instantiation of one of the schemes of Table I and creates, moves or deletes a spider, according to its presence/absence or change of habitat in the two diagrams. Note that we are deriving rules at the instance level, i.e. relations between pairs of instance-SDs, from the comparison of pairs of type-SDs. Since in a type-SD a spider can have feet in different zones, we have to produce a rule for each possible variation in the habitat of the instance spider compatible with the habitat of the corresponding type spider. For example, the second diagram in the sequence in the condition of Fig. 2 presents two zones in the habitat of the `Car` spider, while in the other two diagrams the spider can inhabit only one zone. As a consequence, the procedure produces two movement rules for both the first and the second pair, one for each zone in the habitat. For many-footed spiders, this creates also identical rules, used for subsequent merging, but then removed from the final collection.

Besides rules derived from consecutive pairs of SDs, we need to derive rules for each diagram in the sequence which presents more than one zone in the habitat of a type spider, as this indicates that instances can be in any of those zones during a certain interval. Hence, the procedure creates a rule for each possible movement between these zones. We store such rules in the `RSPS[i][k][s]` position, where k denotes the pair where the diagram with multiple habitats first occurs. For example, from the second diagram in the condition of Fig. 2 we derive rules which allow a car to move between the states *TollParking* and *Running*.

Since the creation, deletion or movement of several spiders is possible, but the rules so far consider only single spiders, the function `mergeRules()` is used to produce rules resulting from the overlapping of all the rules in `RSPS[i][k]` (i.e. those created for different spiders in the same pair). In particular, each merged rule presents in its `L` (resp. `R`) component a foot for each spider, which appeared in one of the zones in the `L` (resp. `R`) component for at least one of the original rules in the set. The merging process considers only rules created for different spiders, i.e. it does not merge rules for the same spider derived from a diagram with multiple zones in its habitat. Since all the rules operate on the same underlying ED and each rule concerns a different a single-footed spider, the overlapping results from placing each spider foot in the corresponding zone, after renaming identifiers to avoid naming coflicts. The set of merged rules for the k-th pair is then placed in `RSP[i][k]`. Finally, `makeSet()` flattens all the rule sets in `RSP[i][1]`,...,`RSP[i][l]` for all the pairs in the i-th sequence of length $l+1$ into a single set, placed in `RS[i]`.

To avoid cluttering the presentation, we have considered only the construction of rules, but actually all the generated rules are augmented with suitable time conditions. Indeed, if `cnt` is the position of the first diagram in a pair in a sequence, we have a corresponding interval specification in `intervals[sequence][cnt]`. A partial valuation of these intervals is obtained by performing a time observation (i.e. an invocation of `now()`) at the occurrence of the trigger, and assigning the resulting timestamp to the *WHEN* variable, if it exists. In particular, this observation fixes the start time of the interval at `intervals[sequence][1]`. If W for the trigger is a timestamp, then the sequence is assumed to start when W equals `now()`. For all rules where the left-hand side is generated from a diagram annotated with an interval ending with a variable subject to a constraint, we condition the application of the rule to a check that the time of application is within that constraint. If the two diagrams are such that the first is annotated with an interval ending at a specific value (under the valuation established by the time observation associated with the trigger), then the rule is associated with a condition that enables its application exactly at the time marking the start of the interval associated with the second diagram.

As to complexity, the rule space grows with the sum of the products of the habitat sizes between consecutive diagrams for each spider, and with the product of the number of spiders.

[3]We only require a covering set of the set of zones in which spiders appear, but this construction is more natural.

[4]A policy might refer to several type spiders and define possible combinations of their habitats.

TABLE II. THE PSEUDOCODE OF THE PROCEDURE FOR RULE GENERATION.

```
procedure generateRules(Policy (validity,trigger,condition)):RuleSet[] ::=
ED=extractED(trigger);
foreach sequence in condition { foreach basic in sequence { ED=EDmerge(ED,extractED(basic)); } }
foreach sequence in condition {
  RS[sequence] = new RuleSet[size(sequence)]; cnt = 1;
  while cnt < size(sequence) {
     L := extractSD(sequence[cnt]); R = extractSD(sequence[cnt+1]);
     RSP[sequence][cnt] = new RuleSet[numberOfSpiders(L)];
     foreach spider in L {
        RSPS[sequence][cnt][spider] = new RuleSet();
        foreach oldZone in habitat(L,spider) {
           foreach newZone in habitat(R,spider) { RSPS[sequence][cnt][spider].add(new Rule(ED,moveSpider,oldZone,newZone)); }
        }
        if (habitat(R,spider)==null) { RSPS[sequence][cnt][spider].add(new Rule(ED,deleteSpider,spider)); }
        foreach (zone1,zone2) in habitat(L,spider) { RSPS[sequence][cnt][spider].add(new Rule(ED,moveSpider,zone1,zone2)); }
     }
     foreach spider in R \ L {
        foreach newZone in habitat(R,spider) { RSPS[sequence][cnt][spider].add(new Rule(ED,createSpider,spider,newZone)); }
     }
      RSP[sequence][cnt] = mergeRules(RSPS[sequence][cnt]);  L = R;  cnt = cnt+1;
  }
  RS[sequence]=makeSet(RSP[sequence]);
} return RS;
```

The procedure for rule construction is guaranteed to terminate, since it only depends on the diagrams in the policy condition. While loops can start during rule application, time progression will make them terminate if the policy validity limit is reached.

A. Applying the procedure to the running example

We present the generated rules for the policy of Fig. 2, based on the underlying ED of Fig. 5. Figs. 6 and 7 each show one rule for each pair of rules obtained by instantiating the `moveSpider` rule scheme via the procedure `generateRules()` on the example policy. We use a concrete presentation for rules, where the universal clock is represented by the icon of an analogical clock, and timers are represented by an agenda icon. All time information is expressed in terms of minutes. The rule in Fig. 6 models a situation where a car leaves the free parking state to enter the running state (the other rule in the pair models transition to the toll parking state). Conforming to the first two diagrams in the policy condition, this can occur at any time before the end of the parking permit period (i.e. for any $\mathtt{T} \leq \mathtt{E1}=\mathtt{WHEN} \oplus 60$). The existing timer is removed and a new one is started, recording the fact that for 120 minutes, after the time `T` when the rule is fired, the car cannot enjoy free parking.

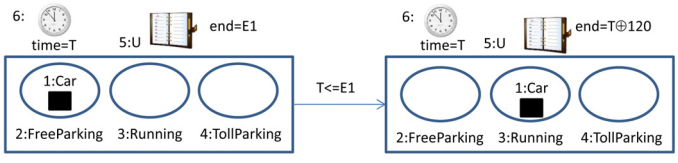

Fig. 6. A rule for leaving the free parking state.

The rule in Fig. 7 models the return of a car to the free parking state from toll parking, the other rule in the pair modeling return from the running state. This can only occur if the current time is more than 120 minutes after the time that the car was last observed in that state, as required by the interval specification associated with the second diagram in the condition. Again, the current timer is deleted and a new one is created which will be used to check that the car does not stay longer than 60 minutes in that state.

Fig. 8 shows one of the two rules derived from the presence of a two-footed spider in the second diagram of the sequence, the other rule being symmetrical. In this case, the timer is preserved and no check is performed on the current time.

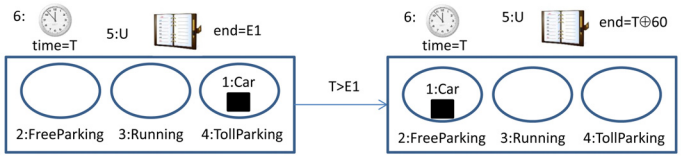

Fig. 7. A rule for returning to the free parking state.

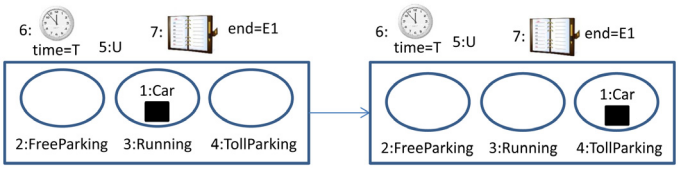

Fig. 8. One rule for alternating between states other than free parking.

Since there is only one sequence in the condition, and only one spider, all the produced rules derive from the rule sets at `rules[seq1][cnt][Car]`, for `cnt`=1,2. In all the considered cases, rules appear in pairs due to the existence of spiders with two feet in the second diagram in a sequence. As only one spider appears in the policy, no rule merging occurs.

Fig. 9 presents a more complex parking policy, for cars with trailers. Both the car and the trailer can park freely for at most 60 minutes. According to the first sequence, the car has then to leave, while the trailer can remain in free parking for other 5 hours at most[5]. They can then be both running, and are not allowed to use free parking again until one day has passed from the first entrance. For the second and third sequences, if car and trailer leave free parking together, to be either running

[5]All interval specifications are meant to refer to minutes.

or toll-parking, they can re-enter the free parking state after 12 hours from the first entrance.

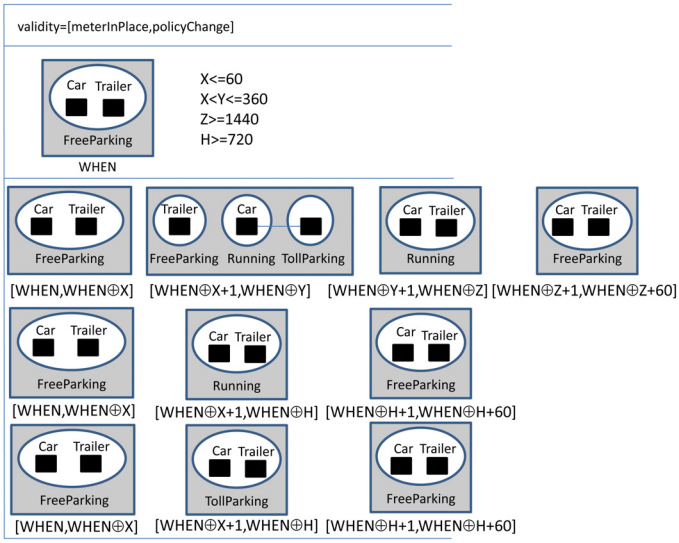

Fig. 9. A policy for parking of car and trailers.

The execution of `generateRules` on this policy produces three sets of rules, one for each sequence, all of them defined on the underlying ED extracted from the second SD in the first sequence. Each rule describes the movement of at least one spider, but all the rules generated for the second and third sequence model the simultaneous movement of both spiders. Since the car type spider has two zones in its habitat in the second SD of the first sequence, the first two pairs of diagrams in the first sequence generate two rules each, and another two rules are generated for the second diagram. In the first pair of rules, the trailer remains fixed, while the car moves to one of the running or toll parking states. For the rules generated from comparison of the second and third diagram, the trailer moves to the running state in any case, whilst the car remains in the running state, or moves to it. Finally, in the rules generated from the second diagram the car can alternate between the running and toll parking states.

The first two rules from the top in Fig. 10 are generated for individual spiders during the execution of `generateRules` by comparing the second and third diagram in the first condition of the policy of Fig. 9. The rule at the bottom is produced from the first two, which are then removed from the collection, during the execution of `mergeRules`.

V. Correctness of the Construction

In order to ensure that the generated rules are sufficient for the exploration of the entire state space, we consider the structure of policies, under the assumption that the time of the trigger establishes a common reference for all the time constraints in the policy. We observe that each rule originates either from the difference between two consecutive diagrams in a sequence, or from the presence of more than one foot for the same spider. Each type of rule can have some specific form of effect on timers and of checks on the value read on the universal clock, modeling a time observation. In the following, we assume that the SDs in the policy have been normalised to

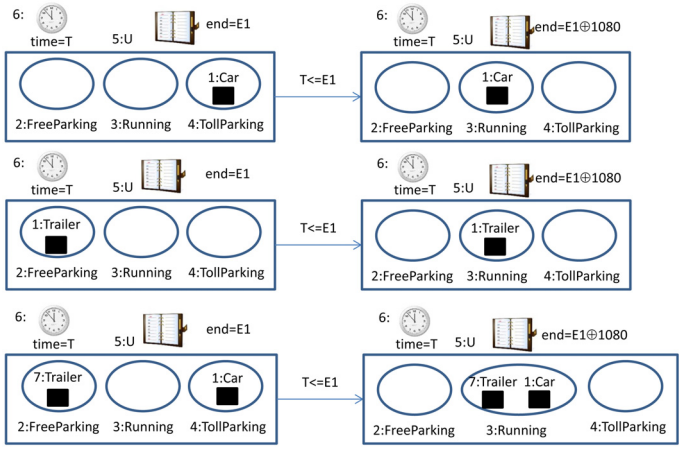

Fig. 10. Rules for individual spiders and merged rule for the car/trailer policy.

the same underlying ED, by the application of `mergeED()`. Theorem 1 systematises these observations.

Theorem 1: Given a policy Π containing only unitary SDs[6], the procedure `generateRules` generates a collection of rules sets, which collectively allow the construction of all and only the possible stories satisfying Π.

Proof: Let Π be a policy; then any story satisfying Π must start with an instance-SD conforming to its trigger and be constituted of a sequence Σ of timed instance-SDs, conforming with one of the sequences $C_1, \ldots, C_n$ of type-SDs in the condition of Π. Let Tr denote the trigger of Π and C_l be the sequence (of length k) to which Σ conforms. Note that Tr and the first timed-SD in C_l consist of the same SD, but with distinct temporal annotations. We represent Σ as a chain of transitions $d^1 \Rightarrow \ldots \Rightarrow d^n$, where d^1 is an instance-SD conforming with Tr and $d^i \neq d^{i+1}$ for $i = 1, \ldots, n$, with each transition occurring in correspondence with a time observation t_i, such that the associated intervals are of the form $I^i = [t_{i-1}, t_i]_u$ and $I^{i+1} = [t_i \oplus 1, t_{i+1}]_u$, with u the minimum granularity for a time observation. Let us consider that the trigger occurs at time t_0 (i.e. either W is the timestamp t_0 or $WHEN$ is assigned the value t_0).

For stories with exactly one instance s of type θ (identifying an instance and its type with the spiders s and θ in the corresponding instance- and type-SDs), each diagram differs from the previous one only for the definition of the habitat $h(s)$, i.e. s moves from one zone (state) to another. Let $(X_1, Y_1) = h^i(s)$ in d^i and $(X_2, Y_2) = h^{i+1}(s)$ in d^{i+1}. We have then three cases: (1) (X_1, Y_1) and (X_2, Y_2) are both in $h_j(\theta)$ for a certain diagram of C_l, say d_j, and the concatenation $[t_{i-1}, t_{i+1}]$ of the intervals I^i and I^{i+1} is within a valuation for the interval specification I_j for d_j; or (2) $(X_1, Y_1) \in h_j(\theta)$ in d_j and $(X_2, Y_2) \in h_{j+1}(\theta)$ in the contiguous diagram d_{j+1} of C_l, with I^i within a valuation of I_j and I^{i+1} within a valuation of I_{j+1}; or (3) $(X_1, Y_1) \in h_j(\theta)$, $(X_2, Y_2) \in h_{j+1}(\theta)$ and $\{X_1, Y_1), (X_2, Y_2)\} \cap (h_j(\theta) \cap h_{j+1}(\theta)) \neq \emptyset$, (i.e. at least one of the zones inhabited by s is in the habitat of θ for two contiguous type-SDs in C_l) and $[t_{i-1}, t_{i+1}]$ is within the concatenation of the valuations for I_j and I_{j+1}. In all these

[6]By definition these are basic timed type-SDs.

cases, the valuations of I_j and I_{j+1} must be consistent with the constraints on them and the timestamp t_0 for the trigger.

For each of these cases, `generateRules` will have placed, in the rule set for sequence C_l, a rule for the corresponding transition; hence all stories can be generated. If the policy concerns more than one type, two consecutive diagrams in the story differ for the habitat of at least one spider, and the same argument can be used. If more than one spider changes, each movement corresponds to the activation of some generated rule, since they derive from all admissible configurations of spiders. As no rule allows any temporal constraint to be violated or transition to a diagram not conformant to a condition, only admissible stories can be generated. ■

The analysis of the possible consecutive pairs of basic timed instance-SDs in a valid story also dictates how to generate rules which give rise to invalid stories. Based on the discussion in Theorem 1, a transition $d^i \Rightarrow d^{i+1}$ cannot occur in a valid story according to a sequence C_l for one of the following reasons: (1) the overlapping of the intervals for d^i and d^{i+1} is within the interval I_j resulting from the valuation of a single diagram d_j in C_l, but one of $h^i(s)$ or $h^{i+1}(s)$ is not consistent with $h_j(\theta)$; or (2) d^i conforms to d_j and d^{i+1} conforms to d_{j+1}, but no consistent valuation exists for the interval specifications I_i and I_j; or (3) there is no pair d_j, d_{j+1} of contiguous diagrams in C_l such that d^i conforms to d_j and d^{i+1} conforms to d_{j+1}. Based on these observations we can construct a set of *erroneous* rules by taking each left-hand side L of a correct rule generated according to a given sequence and associating as R each diagram with a position of the spider foot different from the positions in any other diagram which is in a correct R for L, regardless the constraint. Also, we generate a copy of each valid rule for L, but with a condition which is the negation of the constraint associated with the permanence in the state indicated by L (so that it will stay in L for either too short or too long a period). Theorem 2 ensues.

Theorem 2: A sequence of diagrams is an invalid story for a policy *iff* it results from a sequence of transitions including at least one application of an erroneous rule.

VI. Related work

Time-based specifications usually deal with intervals to model uncertainty about the actual occurrence of an event. In the clock-synchronous semantics of Statemate events can only occur when a clock ticks [5]. This view was adopted also in [6] to integrate time in graph transformations, by introducing a specific attribute updated by clock messages. Temporal aspects have been considered in the automatic generation of controller systems for timed automata, where the admissible transitions are restricted to satisfy some property (see e.g. [7]). In this case, however, one has a specific automaton on which to derive properties, rather than an abstract description of a collection of admissible behaviours of instances of some specific type as defined by a policy, which would combinatorially explode if defined through a single automaton. In policies, what is modeled is the possible persistence of an element in a state over a period, rather than the occurrence of specific transitions triggered by any type of events. In this line, the work in [8] capitalises on [6] translating graph transformation systems into transition systems to be input to a model checker, restricting valid execution paths to time-ordered transformation sequences. In general, several techniques have been developed to derive inputs for model-checkers from the specification of system behaviour in the form of graph transformations (see [9] for a comparison of two general approaches to the development of these techniques). The simple form of the spider diagram transformation needed to simulate and test policies should enable modelers to adapt such existing techniques. Xie [10] showed that UML sequence diagrams allow a better understanding of the functional logic of a multi-threaded program than UML state diagrams, which suggests that condition sequences in a policy can also facilitate understanding.

VII. Conclusions

We have described a process for generating rewriting systems from policies specifying admissible behaviours of instances of types. The resulting systems can be used for simulating valid or invalid stories, to be used when reasoning on the properties of policies. The development of such a formal visual specification language for policies and stories would enable stakeholders to access the information via the formal notation itself, rather than via its translation into another form, thereby potential reducing communication errors. User studies will be needed to assess the value of the adopted notations. The approach is based on the well-established basic notation of Spider Diagrams, which are well suited to express relevant relationships (e.g. set membership and containment), annotated with calendar intervals to express temporal constraints. However, the approach could be applied also to other types of notational system and temporal models. The opposite path, of checking sets of rules against policies, or reconstructing policies from rules is the subject of future work. We also plan to extend the approach to compound diagrams, allowing more compact policies, for example replacing the last two sequences of Fig. 9 with one sequence having as second SD the disjunction of the corresponding SDs in the two sequences.

References

[1] P. Bottoni and A. Fish, "Extending Spider Diagrams for policy definition," *JVLC*, vol. 24, no. 3, pp. 169–191, 2013.

[2] ——, "A visual language for temporal specifications based on spider diagrams," *ECEASST*, vol. 41, 2011.

[3] ——, "Policy specifications with timed spider diagrams," in *Proc. VL/HCC'11*. IEEE, 2011, pp. 95–98.

[4] J. Howse, F. Molina, J. Taylor, S. Kent, and J. Gil, "Spider diagrams: A diagrammatic reasoning system," *JVLC*, vol. 12, no. 3, pp. 299–324, 2001.

[5] R. Eshuis, D. N. Jansen, and R. Wieringa, "Requirements-level semantics and model checking of object-oriented statecharts," *Requir. Eng.*, vol. 7, no. 4, pp. 243–263, 2002.

[6] S. Gyapay, D. Varro, and R. Heckel, "Graph transformation with time," *Fundamenta Informaticae*, vol. 1, pp. 1–22, 2003.

[7] O. Maler, A. Pnueli, and J. Sifakis, "On the synthesis of discrete controllers for timed systems (an extended abstract)," in *Proc. STACS'95*, 1995, pp. 229–242.

[8] S. Gyapay, Á. Schmidt, and D. Varró, "Joint optimization and reachability analysis in graph transformation systems with time," *Electr. Notes Theor. Comput. Sci.*, vol. 109, pp. 137–147, 2004.

[9] A. Rensink, A. Schmidt, and D. Varró, "Model checking graph transformations: A comparison of two approaches," in *Proc. ICGT 2004*, ser. LNCS. Springer, 2004, vol. 3256, pp. 226–241.

[10] S. Xie, "Evaluating and refining diagrams that support the comprehension of concurrency and synchronization," Ph.D. dissertation, University of Georgia, 2008.