Resource-aware Policies[☆]

Paolo Bottoni[a,*], Andrew Fish[b,**], Alexander Heußner[c,**],
Francesco Parisi Presicce[a]

[a] *Dipartimento di Informatica, "Sapienza" Università di Roma, Italy.*
[b] *School of Computing, Engineering and Mathematics, University of Brighton, UK.*
[c] *Otto-Friedrich-Universität Bamberg, Germany.*

Abstract

In previous papers, we proposed an extension of Spider Diagrams to object-oriented modeling, called Modelling Spider Diagrams (MSDs), as a visual notation for specifying admissible states of instances of types, and for verifying the conformance of configurations of instances with such specifications. Based on this formalisation, we developed a notion of transformation of MSDs, modeling admissible evolutions of configurations. In the original version of MSD, individual instances evolve independently, but in reality evolutions often occur in the context of available resources, so transformations must be extended to take this into account. In this paper we provide an abstract syntax for MSDs, in terms of typed attributed graphs, and a semantics for the specification of policies based on notions from the theory of graph transformations, and we associate with them a notion of resources. We also introduce a synchronisation mechanism, based on annotation of instances with resources, so that the transformations required by a policy occur with respect to available resources. In particular, resources can be atomically produced or consumed or can change their state consistently with the evolution of the spiders subject to the policy.

1. Introduction

In the context of requirements engineering, the expression and understanding of policy specifications by various, possibly non-technical, stakeholders, may be

[☆] We are happy to contribute to an homage to S.-K.Chang celebrating his achievements in the foundation and conduction of the Journal of Visual Languages and Computing, for many years together with the late Stefano Levialdi, and in the establishing the whole area of research of Visual Languages. This work brings together two areas of research, diagrammatic reasoning and graph transformations, which, although never directly investigated by S.-K., fit very well with his vision that "cooperative and interdisciplinary research can lead to a better understanding of the visual communication process for developing an effective methodology to design the next generation of visual languages" (from the preface to the groundbreaking 1986 volume on Visual Languages curated by him). Thanks, S.-K., for starting the field!

[*] Principal corresponding author
[**] Corresponding author

favoured by the adoption of diagrammatic, intuitive representations. In [1] we presented a framework for expressing *temporal* policies that restrict admissible evolutions of the state of instances of types, based on an underlying extension of Spider Diagrams for representing types or their instances. In this framework, in contrast to classical Spider Diagrams, curves represent admissible states, and additional temporal information is provided by suitable annotations of the graphical elements. A temporal policy then specifies over which periods an instance of a given type can be in some given state. However, these extensions may not be sufficient in practice, if the evolution of an instance, hence the viability of a policy, depends essentially on the (un-)availability of some resource.

We use, as running examples, a number of scenarios derived from actual parking policies, along the lines of the examples that we have used as testbed for our definition of policies [1, 2] and which permit a number of variations.

Running Example. *The Fiumicino airport in Rome has recently put in place a policy to constrain the time during which a private car collecting passengers can stay in the arrival area. The car's registration plate is photographed when entering the area. If the car does not leave within 15 minutes, a fine will be issued to the owner. To avoid this penalty, the owner can: (1) park in a special 'free parking' zone, where the car is allowed to park for 15 minutes. (2) leave the restricted area, or (3) enter a 'toll parking' zone, where the car can stay indefinitely (actually, no longer than 24 hours). When entering a parking zone, the owner must collect a ticket while the plate is photographed again, and the car is recognised to have left the area when the ticket associated with that plate is discharged at the exit. If the car stays in the free parking zone for more than 15 minutes, a fee must be paid, and registered on the ticket, otherwise the car will be subject to a fine when it is recognised as leaving the arrival area. In either case, if the duration of the stay exceeds the time that has been paid for, a fine will be issued. No car will be admitted to any parking zone which does not offer available parking spaces, so that the possibility for a car to comply with the policy by staying in a parking place is ultimately subject to the availability of the space. Buses can only load and discharge in suitable spaces, while taxis have to enter a reserved area. Payment can be performed in different ways, e.g. cash, credit card, company subscriptions, or through a mobile application.*

We do not consider here the timing aspect, as it has been extensively treated in [1], but we focus on two important features of policy modelling:

(1) A policy (implicitly or explicitly) defines *admissible sequences* of operations or states. For example, a car having enjoyed a free parking period must either leave the area or buy some time before leaving the area anyway. A car cannot enjoy a free parking period after paying a ticket unless it exits and re-enters the controlled area.

(2) In order to comply with a policy, *resources* of some kind must be available. For example if the driver wants to buy some parking time he or she must have some paying instrument available, and the airport system must provide

suitable devices to collect the payment, e.g. parking meters, or connection with the application database. On the parking side, a car which cannot find a parking space, of any type, must necessarily leave the controlled area. The *state* of these resources change either as a consequence of an action executed by some car following the policy, e.g. a parking place is occupied or abandoned by a car, or for independent reasons, e.g. a set of parking places is excluded from usage due to roadworks or security reasons on special occasions.

Running Example (contd.). *Parking regulations in the City of Rome distinguish between three types of public parking places alongside streets: completely free places, short-term free parking places, and toll parking places where parking time must be paid in advance. Short-term parking places cannot be continually occupied by the same car for more than three hours, and no extra time can be bought for these places, but the car can leave the place and reoccupy it, placing an indication of the time at which the new occupation starts. Compliance to this policy is not controlled by automatic means but inspectors can check it at any time (looking for cars which have overstayed their free or paid period). In order for this policy not to be repelled by courts, as appellants can lament the lack of free spaces, the city council must guarantee that in each neighbourhood a certain ratio of (completely or short-term) free to toll places is respected.*

We use this example as an indication of the fact that policies may have conditions of *validity*, out of which they cannot be enforced. These conditions may refer to availability of resources, as well as to specific *events* which activate them or not. We also remark that policies apply to well-defined *types* of elements, e.g. ambulances, police, or firefighter cars are not subject to parking restrictions, and that individual instances start being subjected to a policy only when some *trigger event* occurs, e.g. a car enters the Fiumicino controlled area.

Running Example (contd.). *Garages of commercial malls offer free parking to all customers, but customers may have to provide evidence of a purchase at the mall (upon leaving) to qualify for the use of the free parking. Internally, the mall can reserve spaces closer to the shopping area for pregnant women or families with children, as well as for disabled people, all states which can be easily demonstrated. Multi-storey car parks can reserve different areas for long vehicles and for rental cars, and they do not admit methane-propelled cars in covered zones. In all these cases, availability of a resource is constrained by some property of the car (or of its owner), or some combination of properties. Other similar cases include private car-parks which can only be entered by customers who possess a parking permit or membership card.*

The above are cases where access conditions may differ for elements with specific properties (which can be assessed before or after using the resource). Conformance to a policy may then depend on such properties and on synchronisation of the changes in state dictated by the policy with changes in the availability of multiple resources at the same time.

The formal model for *resource-aware* policies presented here extends our previous policy framework along two directions: On the one hand, we provide a more general notion of policy in see Section 2, of which temporal policies are a special case. On the other hand, after introducing our notion of resources in Section 3, we consider, in Section 4, how (un-)availability of resources affects conformance to a policy. Discussion of related work is postponed until Section 5, after which Section 6 concludes the paper.

Three recent lines of research concur in this work: (1) the extension of Spider Diagrams to the world of OO modelling, in particular through the definition of policies; (2) the definition of an abstract representation of Spider Diagrams in terms of Spider Graphs [3], from which we derive a formal notion of *conformance* of an instance to a policy; (3) the notion of annotation as a flexible way for connecting different domains. In addition, we introduce a generic notion of resource and consequently enrich the notion of policy. We use annotations to relate elements of the domain for which the policy is defined with resources needed for being conformant to the policy, as expressed by global constraints. We define a process of synchronisation between elements and resources in transformations which ensures conformity to the policy.

In particular, we adopt a simplified version of the notion of Spider Graphs, which we have developed to set the logical formalism of Spider Diagrams within the framework of attributed typed graphs and we provide a number of constructs for expressing and reasoning on policies. We introduce an original notion of synchronisation of elements subject to a policy with the needed resources, via the annotation mechanisms presented in [4]. In the resulting setting, systems, modeled as Spider Graphs, can evolve according to a policy under constraints represented by resource availability, while considering resource evolution only in the context of system evolution. Annotations are used both in the representation of constraints and in the construction of synchronised rules, modelling the concurrent evolution of systems and resources. System evolution is modelled through transformation rules ensuring conformity with a policy. Rules can be derived from the policy specification following a procedure described in [2].

The idea of constraining transformations on resources presented in Section 4 is general and could be used in any domain in which policies determine admissible transitions. We point the reader to [3] for detailed motivations for the use of Modelling Spider Diagrams, and on the introduction of Spider Graphs.

2. Formal setting: Modelling Spider Diagrams, Graphs & Policies

Graphs and typed graphs. Following [5], a graph is a tuple (V, E, s, t), with V and E finite sets of *nodes* and *edges*, and functions $s : E \to V$, $t : E \to V$ mapping an edge to its source and target. A *graph morphism* $m : G \to H$ is given by a pair of functions $m_V : V_G \to V_H$ and $m_E : E_G \to E_H$ preserving sources and targets of edges. Morphism composition is denoted by $\circ$, where $m_1 \circ m_2$ indicates that m_1 is applied to the result of the application of m_2.

A *graph transformation rule* is a span of graph morphisms $L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R$, and is applied following the Double Pushout (DPO) Approach. Figure 1 (left) shows

a DPO direct derivation diagram. Square (1) is a pushout (i.e. G is the union of L and D through their common elements in K), modelling the deletion of the elements of L not in K, while pushout (2) adds to D the new elements, i.e. those present in R but not in K, to obtain H as the result of the rule application via the *matching morphism* $m_L : L \to G$, through the *comatching morphism* $m^* : R \to H$. The image $m_L(L)$ is called the *match* of L in G.

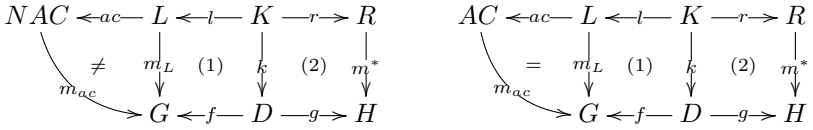


Figure 1: DPO Direct Derivation Diagram for rules with NACs (left) and ACs (right).

An *atomic graph constraint* (or simply *constraint*) is a morphism $c : P \to C$, *satisfied* by a graph G (denoted with $G \models c$) if for each morphism $m_p : P \to G$, a morphism $m_c : C \to G$ exists, with $m_c \circ c = m_p$.

In the simplified definition adopted here, an *atomic application condition* (or simply *application condition*) for $L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R$ is a morphism $ac : L \to AC$, such that the rule is applicable on a match $m_L(L)$ if a morphism $m_{ac} : AC \to G$ exists such that $(m_{ac} \circ ac)(L) = m_L(L)$ (see Figure 1 (right)). A *negative application condition* requires that no such m_{ac} exists (see Figure 1 (left)).

In a *type graph* $TG = (V_T, E_T, s^T, t^T)$, V_T and E_T are sets of node and edge types, with $s^T \colon E_T \to V_T$ and $t^T \colon E_T \to V_T$ defining source and target node types for each edge type. A graph $G = (V, E, s, t)$, with $V \cap V_T = \emptyset$ and $E \cap E_T = \emptyset$, is *typed on* TG if equipped with a (total) graph morphism $tp \colon G \to TG$, where $tp_V \colon V \to V_T$ and $tp_E \colon E \to E_T$ preserves the typing of the images for s and t i.e. $tp_V(s(e)) = s^T(tp_E(e))$ and $tp_V(t(e)) = t^T(tp_E(e))$.

Attributed typed graphs. As far as attributes are concerned, we partition V into V_G and V_D (D for *data*), the sets of *graph* and *value* nodes, respectively, and E into E_G and E_A (A for *attribute*). *Graph edges* in E_G are equivalent to those for non-attributed graphs, while an *attribute edge* in E_A defines the assignment of a value to an attribute of a node. Moreover, $s = s_G \cup s_A$, with $s_G \colon E_G \to V_G$ and $s_A \colon E_A \to V_G$, and $t = t_G \cup t_A$, with $t_G \colon E_G \to V_G$ and $t_A \colon E_A \to V_D$. In a similar way, the type graph TG has distinct sets V_T^G and V_T^D of graph and value node types respectively, as well as distinct sets E_T^G and E_T^A for graph and attribute edge types. Given $t \in V_T^G$, all nodes of type t are associated with the same subset $A(t) \subset E_T^A$ of edge types, corresponding to the set of attribute names for t. Values in V_D range over the disjoint union of the set of sorts in a *data signature* $DSIG$. All morphisms are partial on the edges in E_A and extended with a collection of identities on V_D (a morphism may change the assignment of a value to an attribute, but value elements cannot change). In this paper, we consider all values to be strings. Moreover, we informally exploit the notion of *node type inheritance* from [6], whereby a node of a supertype in a graph can be substituted with a node of any subtype preserving all the

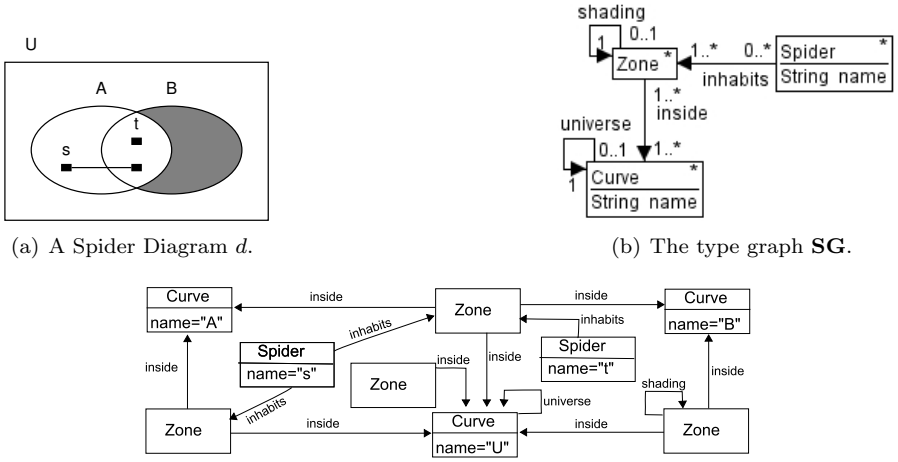
edges (both graph and attribute ones) touching the original node, to refer to commonalities among subtypes.

A *(graph) domain* $\mathcal{D}(\mathcal{TG}, C, DSIG)$ is defined as the set of graphs typed on a type graph $\mathcal{TG}$, complying with a set of constraints C and a data signature $DSIG$. In the following we adopt a UML-like representation of attributed nodes, presenting attributes and their values in a separate compartment.

Spider Diagrams. A *Spider Diagram* (SD) permits the representation of predicates (represented by *curves*) on elements (represented by *spiders*) of some universe of discourse (represented by *the universe* curve) by placing spiders' *feet* in *zones* (intuitively, the region *inside* a set of curves, and outside the remaining curves) on an underlying Euler diagram (see Figure 2(a)). Each curve and spider is uniquely identified by its *label*. Intuitively, a spider with a foot in a zone (i.e. *inhabiting* the zone) indicates that the element represented by the spider can belong to the intersection of all the sets represented by the curves that the zone is inside. A *shaded* zone indicates that no elements can be in the corresponding set intersection except for those with an explicit representation as a spider inhabiting the zone (a spider represents an element in exactly one of the zones that it inhabits). Disjoint non-overlapping curves represent disjoint sets and the nesting of curves represents the subset relation. In this paper we consider only unitary SDs (i.e. we do not consider diagrams joined with logical connectives) and we provide a simplified definition, ignoring relations among spiders, to streamline the approach without unnecessary details. See [3] for a detailed formal definition of SD. We denote the class of all SDs by $\mathcal{SD}$.

Spider Graphs. Figure 2(b) presents the type graph **SG**, a simplified version of the one introduced in [3] to define a semantics for SDs in terms of graph transformations, which characterises, together with a set of constraints, the family of *Spider Graphs* (SGs). This version corresponds to the presentation of SDs above, not including relations between spiders. In SGs, curves, spiders and zones are modelled as nodes of the corresponding types, the first two characterised by a `name` attribute. The modelling of spiders inhabiting a zone and of zones being inside a curve is realised with edges of the, suitably named, `inhabits` and `inside` types, while the properties of being shaded for a zone and of being the unique universe curve are modelled by the `shading` and `universe` edge types. Figure 2(c) shows the SG equivalent to the SD in Figure 2(a).

Modelling Spider Diagrams and Graphs. The extension of SDs to object-oriented modelling, called Modelling Spider Diagrams (MSDs), permits the expression of type versus instance information, thus providing a closer link to the object-oriented modelling paradigm than standard SDs [1]. To this end, one refers to a *name domain*, $\mathcal{N}_D$, as resulting from a collection of names of types, N_T, names of states, N_S, names of attributes, N_A, string representation of attribute values N_V, and identifiers of instances (of the types in N_T), N_I, associated with proper specifications of each name. Syntactically, in a *type*-SD d', all spiders have names in N_T and all curves have names (labeled either with the name of a state



(a) A Spider Diagram d.

(b) The type graph **SG**.

(c) The Spider Graph G_d, corresponding to d.

Figure 2: From Spider Diagrams to Spider Graphs

or with the name of an attribute followed by its value) in $N_S \cup (N_A \cdot \text{`@'} \cdot N_V)$, whilst in an *instance*-SD $d$, spiders have names in $N_I$, curves have names in $N_S \cup (N_A \cdot \text{`@'} \cdot N_V)$, no zone is shaded and every spider inhabits exactly one zone. From the semantical point of view, interpretations of an MSD are consistent with the specifications in $\mathcal{N}_D$ for the corresponding names.

A type-SD places constraints over the admissible states for instances of a set of types (represented by spiders), while instance-SDs present configurations of instances in some states; intuitively, an instance-SD d *conforms to* a type-SD d' if the set of curves $C(d)$ in d is a subset of the set $C(d')$ of curves in d', each spider s in d is named by an instance of a type providing a name for a type-spider s' in d', and s inhabits a zone z which corresponds to a zone z' inhabited by s', in the sense that, given the set I' of curves z' is inside, z is inside the set $I' \cap C(d)$. For a formal definition see [1].

Example 1. *The SD in Figure 2(c) cannot be an instance-SD since it has a spider with two feet, whilst it cannot be a type-SD without an additional mapping of labels to relevant domain names.*

In order to accommodate the restriction of SDs to MSDs, hence defining MSGs, we put the following constraints on the composition of the `name` attribute for *Curve* and *Spider*, which can be easily verified by a suitable parser.

1. For a type-MSG:

   (a) For a *Spider*, the value of the attribute `name` starts with the prefix '`TYPE`', followed by the actual name of the type (an element in N_T), separated by '`:`'. The function $typTNm : String \to String$ extracts

the actual name of the type (i.e. the part of the string after '`:`') given a well-formed value of the `name` attribute for a spider.

(b) For a *Curve*, the value of the attribute `name` starts with a prefix indicating the *family* of the curve (i.e. one of '`STATE`' or '`ATTRIBUTE`'), followed by '`:`'. In the first case, the prefix is followed by the actual name of the state (an element in N_S). In the second case, an infix string reports the name of the attribute (an element in N_A) and the final suffix, separated by '`@`', is a representation of the actual value of the attribute (an element in N_V). The functions $fmlyNm : String \to String$, $attrNm : String \to String$, and $vlueNm : String \to String$ extract the corresponding components, given a well-formed value of the `name` attribute for a curve.

2. For an instance-MSG:

(a) For a *Spider*, the name is composed of the prefix '`INSTANCE`', followed by the name of a type (an element in N_T), followed by a string uniquely identifying the instance (an element in N_I), each separated by '`:`'. The functions $typINm : String \to String$ and $instNm : String \to String$ extract the corresponding parts (i.e. the two parts separated by '`:`', after dropping the prefix '`INSTANCE:`'), given a well-formed value of the `name` attribute for a spider.

(b) For a *Curve*, the name is constructed in the same way as above.

Notice that a type-MSG can present more than one spider with the same type label, indicating that the policy refers to configurations of as many instances, as there are spiders with the same type name, while names for curves are unique. In an instance-MSG, all names are unique. In the following, we will consider policies concerning the evolution of single instances for any type.

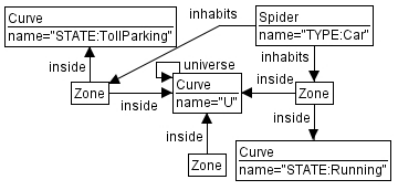
We can now provide an adequate notion of conformance of an instance-MSG to a type-MSG in terms of graph morphisms between graphs typed on the type graph for $\mathcal{SG}$, with the additional constraints on the valuation of the name attributed discussed above.

Definition 1 (Conformance of instance-MSGs to type-MSGs)**.** *We say that an instance-MSG G* conforms *to a type-MSG G'* iff *there exists a total* conformance morphism $gc : G \to G'$ *such that:*

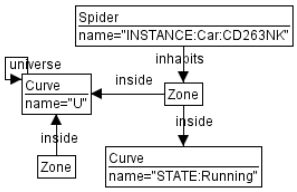
- *for each instance spider* $\mathbf{is}$ *in G,* $gc(\mathbf{is}) = \mathbf{ts}$ *is a type spider and* $typINm(\mathbf{is}) = typTNm(\mathbf{ts})$.
- *gc is surjective on the set of type spiders in G'.*
- *for each curve spider* $\mathbf{ic}$ *in G,* $gc(\mathbf{ic}) = \mathbf{tc}$ *is a curve with* $\mathbf{ic}$.name $=$ $\mathbf{tc}$.name.
- *for each zone node* $\mathbf{iz}$ *in G,* $gc(\mathbf{iz}) = \mathbf{tz}$ *is a zone such that for each edge* $\mathbf{ie}$ *which is incident with* $\mathbf{iz}$*, if* $\mathbf{ie}$ *is incident with node* $\mathbf{n}$ *(representing a curve or spider) then* $gc(\mathbf{ie})$ *is also incident with* $gc(\mathbf{n})$.

Informally, Definition 1 requires that each spider node in the instance-SG, G, inhabits a zone which is the projection, according to the curves present in G of one of the zones that the corresponding spider inhabits in the type-MSG, G'.

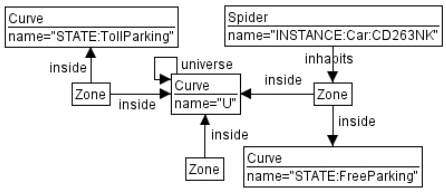
Running Example (contd.). *For the type-MSG G' in Figure 3(top) a Car can be in state either* `Running` *or* `TollParking`*. The instance-MSG G_1 (bottom-left) conforms to G', while G_2 (bottom-right) does not, as there is no possible conformance morphism providing a suitable image in G' for the* ***`inhabits`*** *edge, as this should touch a zone inside a curve named '*`STATE:TollParking`*', according to Definition 1, but such a zone does not exist in G'. Note that G_1 does not have a '*`STATE:TollParking`*' curve or,* a fortiori*, a zone inside it, while G_2 does. In either case, this is irrelevant to establishing (or not) the conformance morphism, since no spider in G_1 or G_2 inhabits a zone inside this curve.*



(a) Type-MSG G'.



(b) Instance-MSG G_1.



(c) Instance-MSG G_2.

Figure 3: A type-MSG G' (top) and two instance-MSGs, one conforming to G' and one not.

As depicted in Figure 4, the SD families are considered as a front-end visual language, utilised for modelling the evolution of some system and the presentation to the end-users, whilst the SG families are considered as the back-end underlying model which can utilise the advanced graph based machinery for analysis purposes. In this paper, we adapt and extend the machinery of SGs to develop and use Modelling Spider Graphs (MSGs).

Representing Policies. In [1], MSDs were extended with temporal information to *Timed* MSDs (TMSDs) in the context of the definition of temporal policies. We give here a more general definition of policy, which preserves an implicit notion of *step progression*.

Definition 2 (Policy)**.** *A policy on MSDs is a triple $\Pi = (Val, Trg, \Gamma)$ where:*

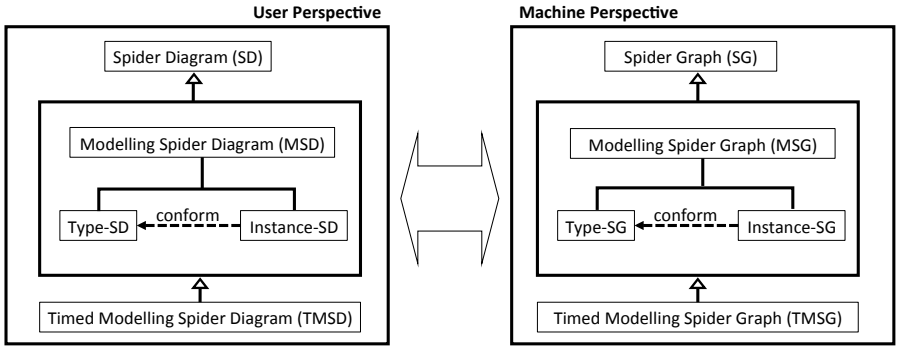


Figure 4: A conceptual view of the relations between the formalisms discussed above: the SD families are considered as front-end user-facing visual languages whilst the SG families are considered as a back-end underlying formalism allowing reasoning based on the theoretical and practical tools developed for graph transformations.

- Val *is a* validity *constraint, indicating the invariant context in which the policy must hold.*
- Trg *is a type-SD, called the* trigger*, which establishes the applicability of the policy.*
- Γ *is a collection of constraints composed of a set* Γ^l*, expressing* local conditions *on the admissible state transitions for an instance subject to the policy, and a disjoint set* Γ^g*, expressing global constraints on the relations between elements in different states.*

In this paper, we express Val using the Object Constraint Language (OCL) syntax [7] and Γ^l by sequences of type-SDs, where each pair of consecutive diagrams depicts admissible state transitions in the application domain.

When type-MSDs are put in a sequence $(d'_1, \ldots, d'_n)$, the difference between two consecutive type-MSDs d'_i and d'_{i+1} describes the admissible evolutions of a sequence of conforming instance-SDs, which is called a *story*. Moreover, if the habitat of a type-spider s' in d'_i comprises more than one zone, also evolutions are admissible in which the habitat of an instance-spider s for that type alternates between any zone in the habitat of s'. Hence, a subsequence σ of instance-SDs, in which spiders change states in consecutive diagrams but continue to occupy states within the habitats of the corresponding spiders in d'_i, can be part of an admissible evolution, with all of the instance-SDs in σ conforming to d'_i. Note that if the habitat of a type-spider s_j in d'_{i+1} extends the habitat of s_j in d'_i, when checking admissibility, we consider the maximal subsequence of instance SDs that conform to d_j, only checking conformance to d'_{j+1} when a spider of an instance-SD inhabits a *new* zone (i.e. one in its habitat in d'_{j+1} but not in d'_j).

Definition 3 (Conformance to a Policy)**.** *A sequence* $(d_1, \ldots, d_n)$ *of instance-SDs* conforms to a policy Π *if: (1) the constraint* Val *is satisfied during the*

whole sequence; (2) each d_i satisfies all of the constraints in Γ^g; (3) d_1 conforms to Trg; (4) each d_{i+1} differs from d_i in the habitat of some spider, in a way that complies with the conditions in Γ^l. Such a sequence of instance-SDs is called a story for the policy.

In [1], a policy's Val was given as temporal interval over which the policy was in place, the trigger was additionally annotated with a temporal variable (WHEN) which recorded the point in time at which an instance started being subject to the policy, and conditions were expressed as sequences of MSDs, each annotated with temporal constraints indicating the time intervals at which the instance could/must conform to that specific MSD. In this paper, in order to focus on the relation between resources and policies, we dispense with time annotations. The timed version of the theory can be immediately recovered and integrated with the resource perspective.

Figure 5 provides an informal overview of the general notion of policy adopted in this paper. We view a policy as a collection of constraints: (i) a validity constraint which determines when the policy should be enforced, within a wider context of the system as a whole; (ii) an activation constraint which specifies the conditions under which an instance triggers the activation of the policy; (iii) a set of global constraints which specifies constraints on all instances within the system; (iv) a set of local constraints that specify the permissible evolutions of instances. In [1] we realised the specific instance of policies for MSDs, with a notion of conformance of sequences of instance-SDs to sequences of type-SDs established, whilst in this paper, we realise the analogues with MSGs.

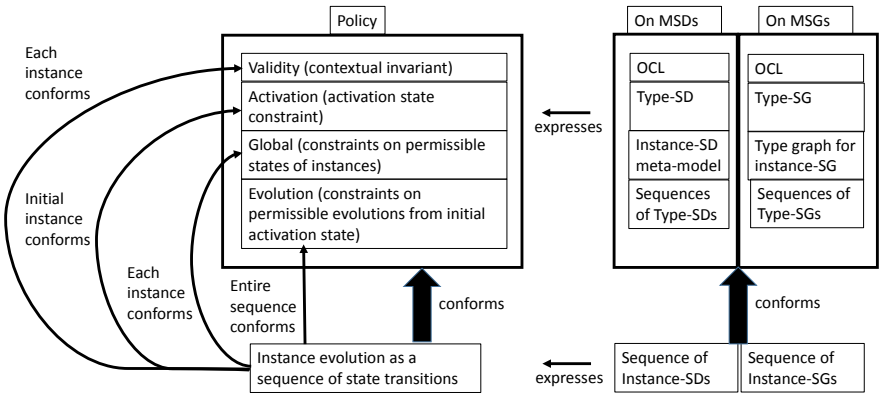


Figure 5: An informal overview of the general notion of policy adopted, the idea of instance evolution conformance to a policy, and their expressions as MSDs or MSGs.

Running Example (contd.). *Figure 6 shows a simple parking lot policy,*

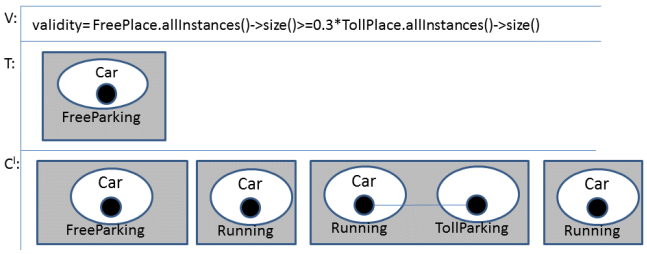


Figure 6: A policy defining admissible transitions between parking or running states of a car.

adapted from [1]. A car becomes subject to a policy when it enters the state of ***FreeParking****, from which it can only transition into the* ***Running*** *state. It can then transition to the* ***TollParking*** *state, and alternate between the latter two states. Finally, the car will be in the* ***Running*** *state again, from which a new transition to the* ***FreeParking*** *state will activate the policy again. The policy implicitly states that a car cannot enjoy free parking again if it has not transited through toll parking, and that no direct change of state from* ***FreeParking*** *to* ***TollParking*** *(or vice versa) is possible. In the version of [1], temporal annotations constrained the duration of free parking, and the amount of time before being able to return to it. The validity constraint indicates that this policy is in place only if there are sufficient number of free parking places, with respect to the number of toll parking places.*

In the MSG setting, we describe Γ^l as a series of partial morphisms between the type-MSGs corresponding to the type-MSDs in Γ^l, where each morphism preserves *Spider* nodes, maps *Curve* nodes to *Curve* nodes with equal names and preserves the *inside* relation for the images of *Zone* nodes, as expressed in Definition 4, where $SP(G)$, $CV(G)$ and $ZN(G)$ denote the sets of `Spider`, `Curve` and `Zone` nodes in a type-MSG G, $INS(G)$ denotes the sets of `inside` edges, and $spNms(G)$ denotes the set of names for spiders in a MSG G.

Definition 4 (Policies in MSGs)**.** *Let $GSEQ = \{G'_1, \ldots, G'_n\}$ be an ordered collection of type-MSGs (possibly with repetitions, but such that $G'_i \neq G'_{i+1}$ for $i = 1, \ldots, n-1$), where $spNms(G'_i) = spNms(G'_{i+1})$ for $i = 1, \ldots, n-1$. Let $MSEQ = \{m^1 : G'_1 \to G'_2, \ldots, m^{n-1} : G'_{n-1} \to G'_n \mid G'_i \in GSEQ\}$ be an ordered collection of partial morphisms between consecutive elements of $GSEQ$. Then, $MSEQ$ defines a collection Γ^l of* local constraints *in a policy Π iff the following hold for all $i \in \{1, \ldots, n-1\}$:*

1. $\forall sp \in SP(G'_i)\,[(m^i_V(sp)) \neq \emptyset) \wedge (m^i_V(sp)).\texttt{name} = sp.\texttt{name}]$.
2. $\forall cv \in CV(G'_i)\,[m^i_V(cv) \neq \emptyset \Rightarrow m^i_V(cv).\texttt{name} = cv.\texttt{name}]$.
3. $\forall zn \in ZN(G'_i)$
   $[m^i_V(zn) \neq \emptyset \Rightarrow \forall e \in INS(G'_i)[zn = src(e) \Rightarrow m^i_E(e) \neq \emptyset]]$.

The elements Val, Trg, and Γ^g in the definition of Π are direct representations in terms of graphs of the corresponding elements for MSDs.

Note that, due to the definition of morphism, the images of edges have their sources or targets in the corresponding images for zones and nodes.

Conformance of a sequence of instance-MSGs to the local constraints in a policy, as defined by a series of morphisms between type-MSG, requires that each square composed with the corresponding conformance morphisms for individual MSGs commutes, as expressed in Definition 5.

Definition 5 (Conformance of instance-MSGs to a policy)**.** *Let* $GSTOR = \{G_1, \ldots, G_m\}$ *be an ordered collection of instance-MSGs (possibly with repetitions, but such that* $G_i \neq G_{i+1}$ *for* $i = 1, \ldots, m-1$*). Let* $MSTOR = \{k^1 : G_1 \to G_2, \ldots, k^{m-1} : G_{m-1} \to G_m \mid G_i \in GSTOR\}$ *be a collection of partial morphisms between elements of* $GSTOR$ *with the same preservation properties as in Definition 4, but such that for each morphism there is at least one spider whose* `inhabits` *edge is not preserved. Let* Γ^l *be a collection of* local constraints *in a policy* Π *as per Definition 4. We say that* $MSTOR$ *conforms to* Π *(or defines a* story *for* Π*)* iff $G_1 \models Trg$ *and the following hold for all* $i \in \{1, \ldots, m\}$*:*

1. $G_i \models Val$ *and* $G_i \models \gamma$ *for each* $\gamma \in \Gamma^g$*.*
2. $\exists G'_j \in GSEQ$ *such that* G_i *conforms to* G'_j *via a morphism* g_i^j *and* G_{i-1} *(if it exists) conforms either to* G'_j *or to* G'_{j-1} *and* G_{i+1} *(if it exists) conforms either to* G'_j *or to* G'_{j+1}*. Furthermore, with reference to Figure 7:*
   (a) *If* G_{i+1} *conforms to* G'_j *then the triangle given by the morphisms* g_i^j, k^i, g_{i+1}^j *commutes.*
   (b) *If* G_{i+1} *conforms to* G'_{j+1} *then the square given by the morphisms* $g_i^j, k^i, g_{i+1}^{j+1}, m_i$ *commutes.*

   and similarly for the triangles or squares formed with the morphisms between G_{i-1}*,* G_i*,* G'_j *and* G'_{j-1}*.*

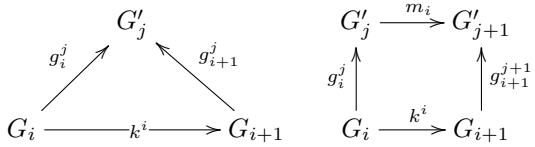


Figure 7: Relations between conformance matches and morphisms in a story.

The above material enables us to present allows us to present an abstract syntax for a specific version of SDs, whose concrete syntax was presented in [8], where zones and curves are necessary to define possible states and conformance is neatly expressed via Definitions 1, 4, and 5, rather than through the previous definition of an ad hoc morphism.

3. Resource Model

We refer to a meta-model of resources, expressed as the UML Class Diagram of Figure 8, where the meta-type `Resource`, with attribute `name`, has subtypes `AtomicResource`, `EvolvingResource` and `BulkResource`. Types derived from `AtomicResource` do not show any inner structure, types derived from `EvolvingResource` are associated with some attribute to specify the state of their instances, while types derived from `BulkResource` have a numeric attribute `quantity`, to reflect the amount available for consumption. The meta-model states the possibility of relations among resources through the meta-association `related`. Specific domains of resources are represented as type graphs where node types are instances of the meta-types above and edge types are instances of `related`. Resources are modelled as *shareable* among different elements or not, with `isShareable()` testing this property.

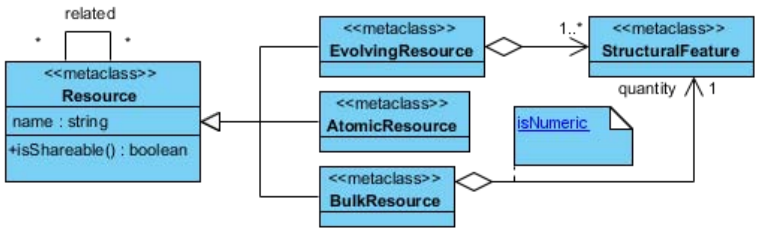


Figure 8: The UML metamodel for the definition of resource domains.

Atomic resources are those which are produced or consumed in a transformation, and cannot be shared within transformations. However, if several copies of the same resource exist, then a number of transformations can occur in parallel as permitted by the number of resources. An example of modelling based on atomic resources is that of an assembly line where individual components are assembled into some piece [9]. In this case, one would not be interested in the properties of the components but only in their availability for the assembling process, and, conversely, only in the fact that some piece is produced. Evolving resources can be created or consumed by some processes, in the same way as atomic resources are, but their usage in a process can depend on their state. Also, they can be involved in processes which preserve their existence, but change their state. We will use resources of this type to model the availability of parking places, also considering additional attributes for special applications. Bulk resources are concentrated in space and a fraction of their total number can be consumed or produced during a transformation. An example is the supply of nutrients available in bulk.

4. Resource-aware Transformations

Annotations. To connect MSGs with resources, we use annotations [4]: first-class entities enriching an application domain $\mathcal{AD}$ with nodes from a contextual

domain $\mathcal{X}$. In [4], we used annotations to express contextual constraints on elements of the application domain and provided a number of constructions for deriving application conditions on transformations in the application domain. Annotations can be flexibly added and removed. Here, we use annotations to model relations between elements subject to a policy (i.e. from the application domain) with the resources (from the contextual domain) needed to conform to the policy. An annotation gives rise to a pattern of the form $\alpha = d \longleftarrow a \longrightarrow x$, with: d a node in $\mathcal{AD}$, a a node of an *annotation* sort, and x a node in $\mathcal{X}$, with $\longleftarrow$ and $\longrightarrow$ suitably typed edges. In this paper we use the *annotatesWith* edge type for $a \longrightarrow x$ and domain-dependent types for $d \longleftarrow a$.

To simplify the presentation, we slightly abuse typing and encode the two orthogonal type systems (of SGs in general and of the typing induced by type-MSGs) into one by introducing subtypes of *Spider* and *Curve*, according to the prefixes of the names for their instances. For example, `CarSpider` will be used to refer to nodes, s, of type *Spider* such that $typTNm(s.\texttt{name}) =$ '`Car`' (and correspondingly for instances, $typINm(s.\texttt{name}) =$ '`Car`') and `StateCurve` will be used to refer to nodes, c, of type *Curve* such that $fmlyNm(c.\texttt{name}) =$ '`STATE`', adapting the notion of conformance accordingly. We provide a new construction integrating annotation (and de-annotation) processes in application domain transformations, keeping the evolution of resources synchronised with that of the annotated element.

Resource-aware Transformations. We model the relation between spiders in MSGs and resources using annotated graphs, and their synchronisation via a combination of graph transformations and (de-)annotation processes. Given an application domain $\mathcal{AD}$ (the *domain subject to the policy*), a domain of resources $\mathcal{R}$, and an annotation sort $\mathcal{A}$, an MSG *with resources* is a construct $MSGR = (V, E, s, t)$ where each $Z \in \{V, E, s, t\}$ is the union of components Z_c, with $c \in \{\mathcal{AD}, \mathcal{R}, \mathcal{A}\}$. In particular, $(V_{\mathcal{AD}}, E_{\mathcal{AD}}, s_{\mathcal{AD}}, t_{\mathcal{AD}})$ defines an MSG over $\mathcal{AD}$, $V_{\mathcal{R}}$ is a collection of nodes[1] from $\mathcal{R}$, and $s_{\mathcal{A}} : E_{\mathcal{A}} \to V_{\mathcal{A}}$, $t_{\mathcal{A}} : E_{\mathcal{A}} \to V_{\mathcal{AD}} \cup V_{\mathcal{R}}$ (so that $\mathcal{R}$ plays the role of the contextual domain $\mathcal{X}$). An edge $e \in E_{\mathcal{A}}$ with $t(e) \in V_{\mathcal{R}}$ is of type *annotatesWith*, while an edge $e \in E_{\mathcal{A}}$ with $t(e) \in V_{\mathcal{AD}}$ is of some domain-dependent type. Let $\mathcal{MSGR}$ denote the class of all MSGRs. We define specialised versions of the notions of constraints and rules. An *atomic constraint on resources* is a morphism $ac : P \to C$, where $P \in \mathcal{MSG}$ and $C \in \mathcal{MSGR}$.

Running Example (contd.). *Figure 9 presents the type graph on which to type the MSGRs modeling the synchronisation of the evolution of MSGs representing the state of cars conforming to the parking policy of Figure 6 with the state of availability of parking places (seen as evolving resources). We rename node types along the lines indicated above to indicate their admissible interpretations and use inheritance from the abstract type* `ParkingPlace` *to indicate aspects common to each type of parking place resource. The relation between*

[1] In this paper we are not interested in relations among resources.

the state of the car (as represented by a `CarSpider`*, which inhabits a* `Zone` *that is inside a* `StateCurve`*) and the usage of resources is defined via the two constraints of the form presented in Figure 10. Parking places are considered as non-shareable resources and we constrain each car to use only one parking place at a time, as per the two forbidden graphs in Figure 11.*

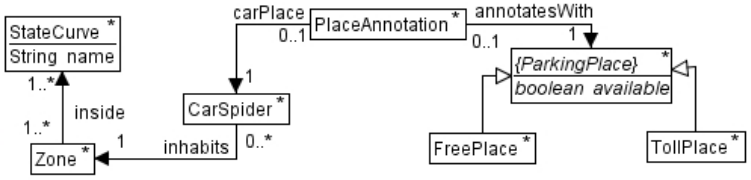


Figure 9: The type graph for the policy example with synchronisation on parking resources.

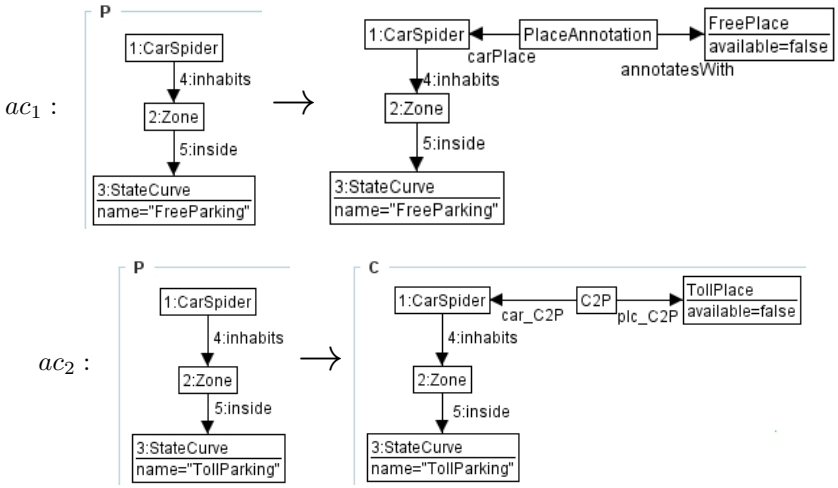


Figure 10: The two constraints relating the state of a car with the usage of parking resources.

As discussed in [2], rules can be derived from a policy Π to model the transformations which bring to the creation of stories conformant to Π. Basically, one creates two types of rules: (1) for each graph G_i in $GSEQ$ and for each spider sp in $SP(G_i)$ a set of rules to allow a spider of type $tpTNm(sp)$ to alternate its habitat between all states inhabited by sp, and (2) for each morphism $m_i : G_i \to G_{i+1}$ in $MSEQ$ and for each spider sp in $SP(G_i)$, a set of rules to allow a spider of type $tpTNm(sp)$ to transition from any of the states sp inhabits in G_i to each of the states that $m_i(sp)$ inhabits in G_{i+1}. For example, Figure 12 shows two rules derived from the policy of Figure 6, namely that allowing the transition from the free parking to the running state (derived from the first morphism) and that for transitioning from running to toll parking (derived from both the second morphism and the third graph).

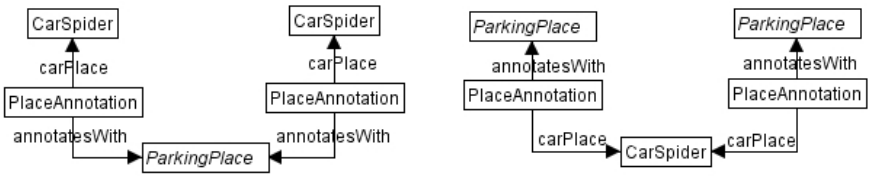


Figure 11: Two forbidden graphs for the annotation of cars with parking places.

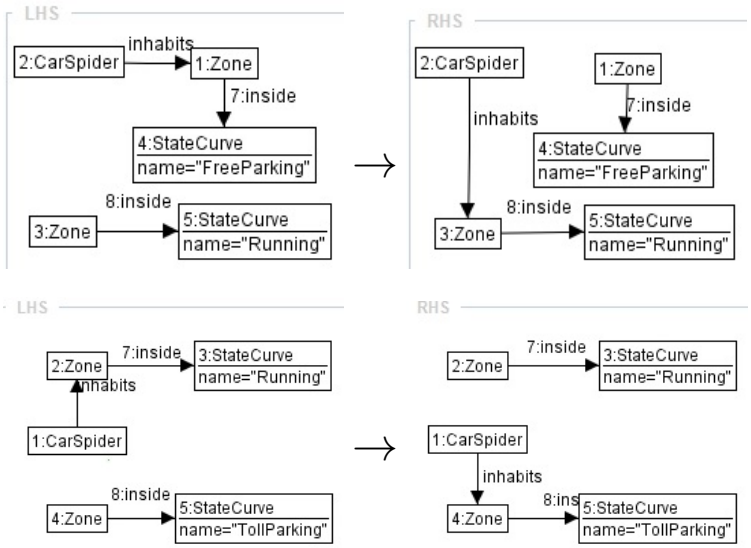


Figure 12: Two rules for allowing transitions conforming to the policy of Figure 6.

An *MSGR transformation rule* is a DPO rule ($L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R$) for which projections on $\mathcal{AD}$, $\mathcal{R}$, and $\mathcal{A}$ provide, respectively, an MSG transformation rule ($L_{\mathcal{D}} \leftarrow K_{\mathcal{D}} \rightarrow R_{\mathcal{D}}$) and a resource transformation rule ($L_{\mathcal{R}} \leftarrow K_{\mathcal{R}} \rightarrow R_{\mathcal{R}}$), with additional annotation components $L_{\mathcal{A}}$, $K_{\mathcal{A}}$ and $D_{\mathcal{A}}$, while the components $l_{\mathcal{A}}, r_{\mathcal{A}}$ of the morphisms l and r restricted to the domains and images of $s_{\mathcal{A}}, t_{\mathcal{A}}$ relate to annotation processes. In particular, annotations in $L_{\mathcal{A}}$ and not in $K_{\mathcal{A}}$ define a de-annotation process, while annotations in $R_{\mathcal{A}}$ and not in $K_{\mathcal{A}}$ define an annotation process. All in all, an MSGR transformation rule models the synchronised evolution of the application and resource domains[2], with synchronisation expressed through annotations.

The construction is such that if the synchronised rule is applied to a "correctly annotated" configuration, i.e. one satisfying some resource constraint, then the resulting configuration has annotations which were only valid in the

[2]In this paper we use Spider Graphs only for modelling the application domain, with states of resources modeled through attributes, rather than via state curves.

source graph removed, and annotations which become valid in the target graph added. We need to indicate the transformation of the state of availability for the parking place in the rule for two reasons: (1) since parking places (and in general resources) can have an evolution which is independent from their use with respect to a policy, a parking place may be unavailable even if there is no car currently parked in that place, for example due to works in the area it is located, so that we need to make sure that a place in the correct state is used; (2) the change of state in the parking place must be contextual with the change of state of the car, either for acquiring or for releasing the resource.

Figure 13 shows the construction of a synchronised rule out of a single application domain rule, $L_{\mathcal{AD}} \leftarrow K_{\mathcal{AD}} \rightarrow R_{\mathcal{AD}}$, and a collection of relevant constraints and resource rules, ensuring that the application of the rule preserves conformance to the policy. We identify left-synchronisation (left of Figure 13) and right-synchronisation (right of Figure 13) processes, based on the observation that application domain rules concern changes of zone of single-footed spiders, constraints refer to the presence of feet in a zone and the consequent annotation with a resource, while resource rules concern change of state or creation or deletion of resources. Intuitively, a left process is concerned with a constraint whose premise has to be present in a graph to which the application domain rule can be applied, but which ceases to be present after rule application. Conversely, a right process is concerned with a premise which must be present after the rule application, but which was not present before.

Each left process involves a constrainst $c_i : P_i \rightarrow C_i$ and a resource rule $L^j_{\mathcal{R}} \leftarrow K^j_{\mathcal{R}} \rightarrow R^j_{\mathcal{R}}$, such that: (1) P_i has a non-empty maximal intersection[3] X^1_i with the left-hand side, $L_{\mathcal{AD}}$, of the application domain rule; (2) C_i has a non-empty intersection X^2_i with the left-hand side of a resource rule, $L^j_{\mathcal{R}}$; and (3) there is a replica of the annotation pattern α relating the images of X^1_i and X^2_i in C_i, via $c_i \circ x^1_c$ and x^2_c, respectively. In particular, due to the structure of rules and constraints, where rules contain only elements from the application domain and resources can only appear in the conclusion of a constraint, X^1_i consists of a graph specifying that some types of spider can be in certain states, while X^2_i concerns the properties of some resource. Then, the left-hand side of the synchronised rule is constructed as colimit of $L_{\mathcal{AD}}$, C_i and $L^j_{\mathcal{R}}$ via X^1_i and X^2_i (informally, this is the union of all the elements present in these graphs, up to identification of elements present in their intersections). Then K^j_i and R^j_i are constructed as pushouts of the corresponding components through their empty intersections and the rule is completed via the morphisms from K^j_i deriving from the universal property of the pushout.

Symmetrically, in a right process, one looks for non-empty maximal intersections Y^1_m and Y^2_m of the premise and conclusion of some constraint $c_n : P_n \rightarrow C_n$ with $R_{\mathcal{AD}}$ and the right-hand side $R^m_{\mathcal{R}}$ of some rule in the resource domain, re-

[3]By *intersection* G_0 of two graphs G_1 and G_2, we mean here that G_0 is such that there are two injective total morphisms $i_1 : G_0 \rightarrow G_1$ and $i_2 : G_0 \rightarrow G_2$. By *maximal* we mean here that any other G with this property has an injective total morphism $i : G \rightarrow G_0$.

spectively, under conditions analogous to those above. The right-hand side of the resulting rule, R_n^m is obtained again via a colimit construction and the rest of the rule is obtained by the pushouts of the L and K components along their empty intersections, component-wise. All the resulting rules (i.e. those produced by all the left and right processes) are then composed via a colimit construction along the original application domain rule.

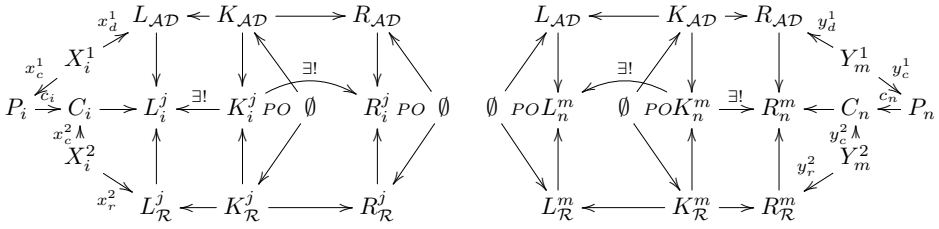


Figure 13: Left and right synchronisation of rules through constraints.

Running Example (contd.). *Considering the two rules of Figure 12, in rule (a) the evolution of the car must be synchronised with the evolution of the free parking place which was occupied by the car and which becomes available to other cars following the application of the rule.*

In order to ensure synchronisation for the first rule, we follow the left-synchronisation process, as shown in Figure 14, where the constraint on free parking (ac_1 from Figure 10), the rule for modifying the car state (Figure 12(a)), and the rule for changing the availability status of a free parking space from occupied to available are placed in positions corresponding to those of Figure 13. We have omitted to represent the empty intersections. As a result, in the first rule, the previously occupied place becomes available so that the existing `PlaceAnnotation` *is deleted, together with its adjoining edges.*

On the contrary, in rule (b) from Figure 12, the car was not originally associated with any parking resource (a negative application condition could be used to ensure this), so an annotation node must be created and the resource part must only describe the evolution of the toll parking resource. This is achieved by applying to rule (b) a right synchronisation process (not shown here) with the rule for the evolution of a toll parking place from available to occupied, considering the constraint ac_2 from Figure 10 on toll parking.

The resulting rules are shown in Figure 15.

Running Example (contd.). *In the Fiumicino policy, a different policy exists for the departure area, differing from that for the arrival area discussed in Section 1 on the management of free parking. In the policy for the arrival area, the ticket for overstaying the free period has to be paid when leaving the zone (and the area), with the car having already left the parking place. In the departure area a 'Kiss & Go' zone is established, where extra time can be bought at any moment during the allowed fifteen minutes. In this case, the car's registration plate is manually entered by the owner, with the car already occupying a space.*

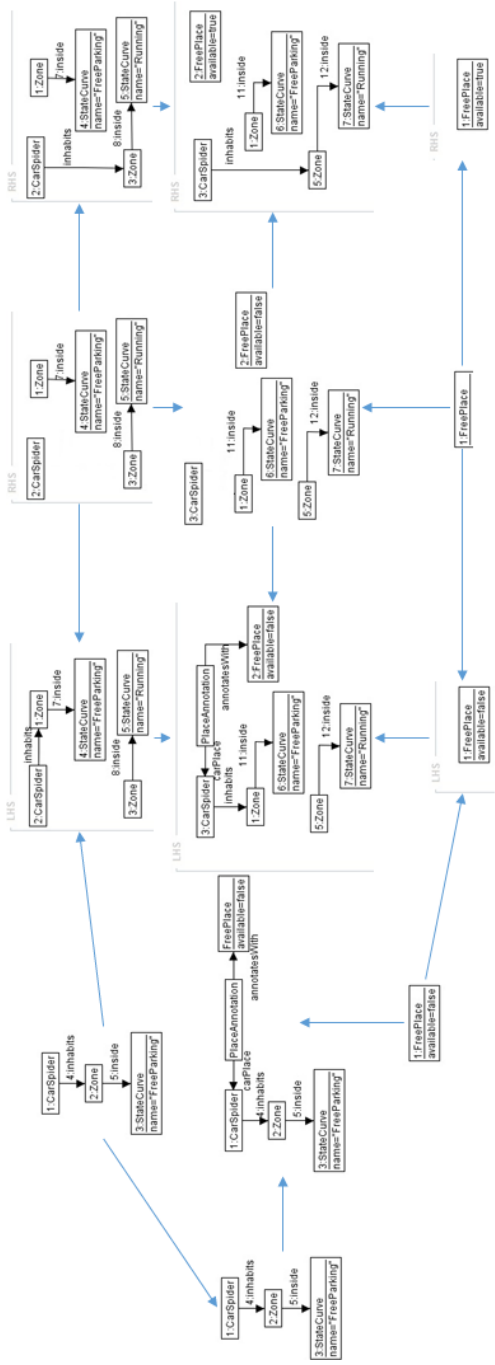
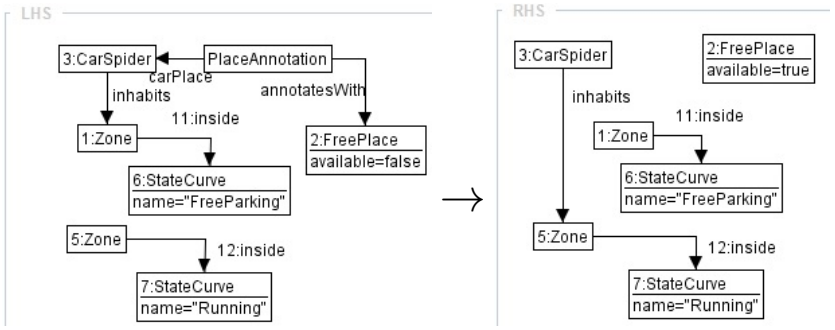
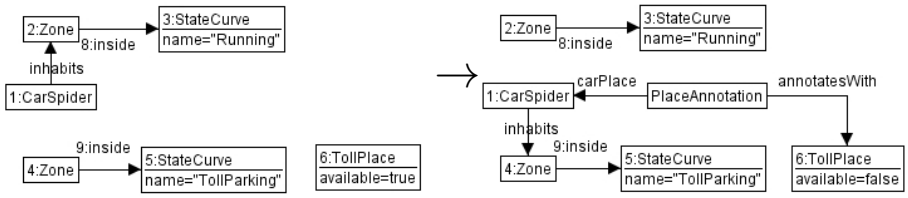


Figure 14: Derivation of the synchronised rule for the transition from free parking to running.



(a) The rule for transition from free parking to running.



(b) The rule for transition from running to toll parking.

Figure 15: Rules for synchronising evolution of parking state with parking place availability.

As a consequence, the car performs a transition from free parking to toll parking, while the parking place retains its state of being occupied, acquired when the car had stopped running to start free parking. To model this policy, the type graph of Figure 9 is refined to include a `Kiss&GoParkingPlace` *and the states for the car are differentiated to express whether the car is in the departure or in the arrival area. The* Γ^l *component of the policy for the departure area is therefore defined by a sequence which allows a transition from free to toll parking. The rule expressing this transition is synchronised with the identity rule preserving the state of being occupied for a* `Kiss&GoParkingPlace`*, since in both the left and right synchronisations the same state for the place appears in the conclusion of the relevant constraint, irrespective of the car's parking state (free or toll).*

Synchronisation over Properties of Spiders. Since attributes, beyond state specifications, may be relevant for resource allocation, we consider now curves, c, for which $fmlyNm(c)$.`name` = '`ATTRIBUTE`' and adopt an analogous convention for subtyping attribute curves with an indication of the represented attribute, as extracted by $attrNm$. Hence, we only report in the `name` attribute the representation of the value component (as would be obtained via $attrNm$) for that attribute. However, we maintain the state as the designated feature with which the policy is concerned.

Running Example (contd.). *Figure 16(a) presents the type graph for situations where parking places can be allocated only to cars not exceeding a certain*

length. In this case, attribute curves represent car lengths, while parking resources are endowed with a property describing the maximum admissible length for a car occupying that place. Figure 16(b) shows how the constraints relating parking states with parking resources are extended when considering lengths. The extension is twofold: (1) we represent the `length` *attribute curve in the premise and the* `maxLength` *property of the parking resource in the conclusion; (2) we state conditions on the attribute values. The constraint states that if a car of length* `Float.parseFloat(Z)` *is in a parking state (i.e. one which is not* `'Running'`*)* [4]*, then its length must be compatible with (i.e. less than) the value* `Y` *of the maximum length for the occupied parking place. Due to inheritance, this represents two constraints, one for each type of parking place. Rules analogous to those of Figure 15 are then built by extending the construction of Figure 13 to incorporate the condition on length. In Figure 16(c), we show the extended version of the rule in Figure 15(b). Note that the construction includes a condition on the maximum length of the toll parking place that becomes occupied.*

The example can be extended to take into account other types of constraints on resource usage. For example a car could have a property describing the type of fuel (e.g. `petrol` or `propane`); then propane or methane fueled cars are allowed to park only in parking places with no roof above them, for which a boolean `covered` property can be provided when describing parking resources.

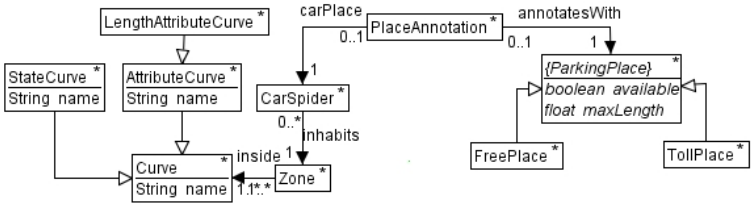
Synchronisation over Multiple Resources. Conformance to a policy could depend on more than one type of resource. We identify three cases in which such a dependence can occur: (*i*) a state transition changes the availability of more than one type of resource; (*ii*) a state transition uses some resource, but requires the availability of some other resource of a different type; (*iii*) the usage of some resource is constrained by the availability of some other type of resource.

As an example of the first case, consider a variant of the classical problem of the dining philosophers, where each philosopher can be either *tranquil*, *thirsty*, *hungry*, *drinking* or *eating*, and resources are represented by *bottles* (for drinking) and *forks* (for eating). In order to move from drinking to eating a philosopher must release the bottle (de-annotation process) and grab a fork (annotation process). The two processes are sequentially independent so that the order in which they are executed is irrelevant (see Theorem 2 of [4]) and require simultaneous synchronisation with the two rules modelling the changes in availability of the two resources.

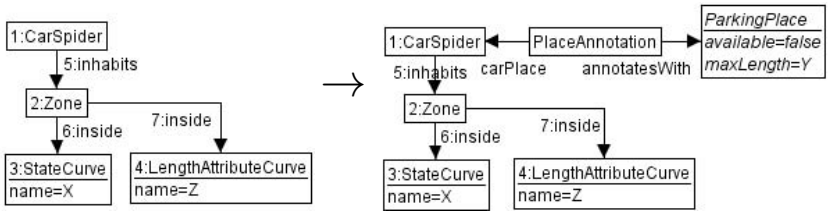
The parking policy provides an example of the second case: moving to the running state for a car would require that the `Petrol` bulk resource associated with it had a non-zero value for the `quantity` property. By abstracting from the negligeable consumption associated with starting the engine no synchronisation is needed with respect to the `Petrol` resource.

The third, and more complex, case typically occurs when access to the resources of a given kind is constrained by the availability of resources of a different

[4] We use an abstract version, to be specialised for the possible values of `X`

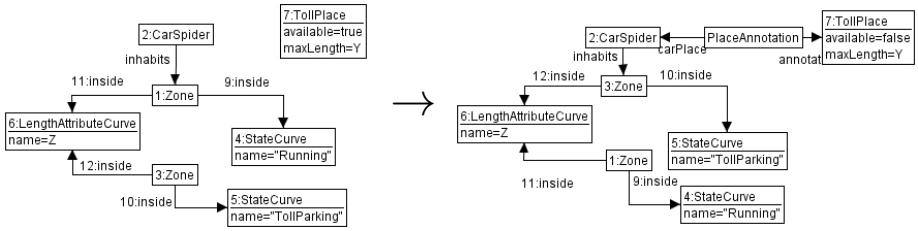


(a) The type graph for the policy example with constraints on length.



`X!='Running' && Float.parseFloat(Z)<=Y`

(b) The abstract version of the constraint for parking with conditions on length.



`Float.parseFloat(Z)<=Y1 && Float.parseFloat(Z)<=Y2`

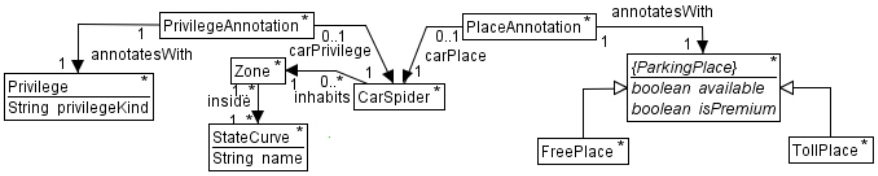
(c) A rule for transitioning from running to toll parking with conditions on length.

Figure 16: Synchronisation over spiders' properties: conditions on lengths.

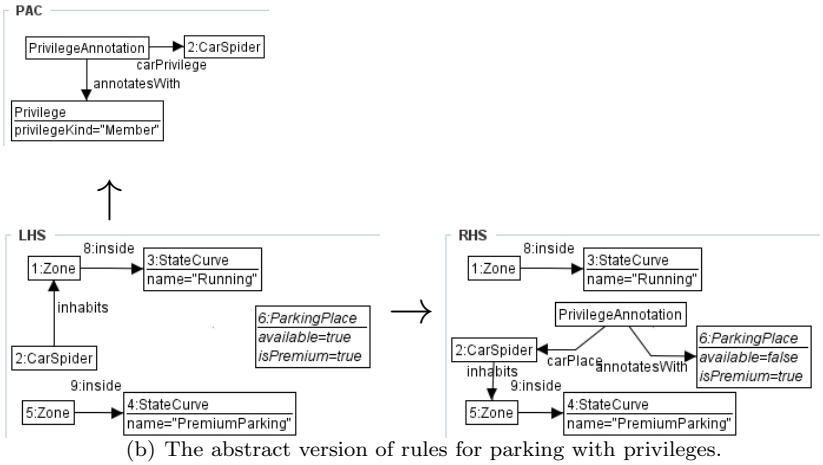
kind, so that one annotation process is dependent on the other. We model such situations by multiple sequential annotation processes. Due to the dependency relation between these resources, each annotation process is considered as occurring on the domain resulting from the previous annotations.

For example, a place could be used only by cars whose owners have some privileges. Figure 17(a) shows the type graph on which to model such a policy, where `Privilege` is a type of atomic resource, whose instances are specifically created for a given entity. The `isPremium` property is inherited by all specialisations of `ParkingPlace`. A constraint, not shown here, states that premium places (for which `isPremium` is set to `true`) are only available to cars with `Member` privileges. Figure 17(b) shows the abstract version of the rules modelling transitions from the running state to the state of occupying a premium place (be it free or toll). The positive application condition, denoted with PAC (of which we show only the projection concerning privileges), requires that a

car moving to occupy a premium place is already annotated with the suitable privilege, as required by the constraint.



(a) The type graph for the policy example with privileges.



(b) The abstract version of rules for parking with privileges.

Figure 17: Synchronisation over multiple resources: parking with privileges.

As in the case of a bulk resource, not resulting in a state transition, no synchronisation is needed for atomic resources which do not get created or destroyed as a consequence of a state transition. In these cases, we can use a construction from [4] where a constraint on the annotations required by the new graph resulting from the application of a rule induces an application condition to be checked on the L component of the domain rule. Conditions on attributes can be orthogonally integrated with any of the three cases described above.

5. Related Work

Forms of synchronisation between different domains are typically expressed, in the graph transformation field, via Triple Graph Grammars (TGGs) [10], where two different graphs (called *source* and *target* graphs) are related by a *correspondence* graph, through graph morphisms.

With respect to triple graphs, annotations offer the possibility of dealing with multiple domains, and to dynamically change the connections between elements, which is not considered as the trace of a transformation, but as additional information associated with an element. A recent proposal extends TGGs to

graph diagram grammars, in which multiple domains can be made to correspond in different ways, taking care of the overall consistency of the relations [11].

Addressing a similar problem, the Query-View-Transform (QVT) approach to model-to-model transformation uses trace elements to maintain the correspondence between elements of different domains [12], making TGGs a suitable candidate for implementing the QVT specification.

These approaches focus on managing correspondences between related elements, whereas a distinguishing feature of annotations is the possibility of dynamically creating and deleting relations between elements in different domains. Moreover, they distinguish between correspondence and application domains, but not between elements and resources.

Synchronisation of the evolution of related models via TGGs has been studied by Hermann *et al.* [13, 14], also with reference to conflicts between transformations in the two domains and constraint propagation across models. A novel feature of our approach is the possibility of expressing constraints and synchronisation across an arbitrary number of domains.

In the context of graph transformation based modelling, other different approaches for policy modelling were proposed. Aspect-oriented graph grammars permit the implementation of global policies on the application of graph transformation systems given by DPO rules [15]. Aspects, expressed as second-order transformations or flattened into AGG models [16], can use attributes of graph elements to influence the application of grammar rules, but there is no explicit resource modelling or evolution of resources independent of other graph entities.

In [17], graph constraints and rules model policies in the context of roles and action, allowing their construction, refinement and abstraction and a simple verification framework on Boolean formulas with a *previous-time* operator. Resources are not explicitly modelled. Lastly, [18] provides a thorough coverage of the use of constraints to generate application conditions on graph transformations, but it does not address the use of constraints for process synchronisation. Our use of constraints distinguishing between application and contextual domains, matches some particular cases from [18] and facilitates analysis of decidability and conflicts.

In the UML world [19], two types of diagrams can be used to represent information analogous to that given by MSGRs. Statecharts (in particular in the protocol version) are the formalism of choice to model state evolution, and guards on transitions could be extended to integrate information on resource availability. However, our approach provides three major advantages: (1) graphs in a policy can refer to configurations of elements (spiders), whereas UML Statecharts refer to single model elements; (2) synchronised evolution of resources can be represented together with evolution of elements subject to the policy, which would require the use of explicit mechanisms of communication between diagrams, or between concurrent regions; (3) an element in a state can undergo the same evolution, irrespective of how it reached the state. On the contrary, in a policy the same state can occur several times, each time with different possible evolutions, depending on the currently active constraint from Γ^l. Timing diagrams mitigate the last problem, by showing possible evolutions along time

in a single diagram, and they allow multiple lifelines in the same diagram, with explicit indications for synchronisation. However, they do not support a distinction between elements subject to a policy and resources. Modeling the evolution of several elements in a single diagram may may lead to unwieldy results, while MSGRs localise the information relevant to each given step in a policy.

6. Conclusions

We have presented an approach to modeling resource-aware policies, in which the ability of a system to conform to a policy depends on the availability of resources. We have modelled the domain of elements subject to a policy as Modeling Spider Graphs (MSGs), a novel abstract representation of the diagrammatic language of Modeling Spider Diagrams, allowing the integration of diagrammatic reasoning and model transformation, while resources are represented as nodes in some domain. The allocation of a resource to an element from the policy domain is realised via annotations. In this setting, we have shown how to synchronise evolutions of elements and policies exploiting constraints modeling the policy/resource dependency, based on a formal construction, and we have considered the case of synchronisation on multiple resources.

Spider Diagrams are a straightforward enough notation to be usable as a front end notation for stakeholders, whilst at the same time being entirely formal objects with precise semantics. At the back end, we consider the natural abstract representation of Spider Diagrams as attributed typed graphs, thereby permitting access to all of the associated facilities and tools already available. The extension of the Modelling Spider Diagrams framework to account for resources is an important step forward in providing sufficiently robust machinery. Whilst the basic underlying approach brings together the diagrammatic logic world of Spider Diagrams with the graph transformation world via Spider Graphs, we extend this further into the modelling context, developing the required theory for the extension to policy specification, and providing an integration of the approach with the framework for annotations bringing together for the first time the three different strands of research. We have presented the new integration with the notation of annotations at an abstract level, which can then be realised in different ways at the concrete level. How to most effectively realise notational extensions incorporating the annotation concepts is the subject of orthogonal ongoing work; for instance, we have proposed the use of different forms of colouring to clearly differentiate between states and attributes [20], or of annotation edges in [20, 8].

The treatment has considered only the evolution of resources as related to the domain policy. Future work will consider also independent evolution of resource policies, so that they have to be represented via MSGs as well. In particular, repair actions must be activated when a resource is destroyed, or its allocation removed. Moreover, the synchronisation machinery will have to be extended to consider the case of simultaneous evolution of different resources, as well as the case of more complex constraints.

[1] P. Bottoni, A. Fish, Extending Spider Diagrams for policy definition, JVLC 24 (3) (2013) 169–191.

[2] P. Bottoni, A. Fish, Policy enforcement and verification with Timed Modeling Spider Diagrams, in: VL/HCC, 27–34, 2013.

[3] P. Bottoni, A. Fish, F. Parisi-Presicce, Spider Graphs: a graph transformation system for spider diagrams, Software and System Modeling 14 (4) (2015) 1421–1453.

[4] P. Bottoni, F. Parisi Presicce, Annotation processes for flexible management of contextual information, JVLC 24 (6) (2013) 421–440.

[5] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer, Fundamentals of Algebraic Graph Transformation, Springer, 2006.

[6] J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, G. Taentzer, Attributed graph transformation with node type inheritance, TCS 376 (3) (2007) 139–163.

[7] OMG, Object Constraint Language Version 2.4, Tech. Rep. formal/2014-02-03, OMG, 2014.

[8] P. Bottoni, A. Fish, A. Heußner, Annotating Spiders with Resource Information, in: Proc. VL/HCC 2014, IEEE Computer Society, 33–40, 2014.

[9] J. de Lara, H. Vangheluwe, Automating the transformation-based analysis of visual languages, Formal Asp. Comput. 22 (3-4) (2010) 297–326.

[10] A. Schürr, Specification of Graph Translators with Triple Graph Grammars, in: WG'94, vol. 903 of *LNCS*, 151–163, 1994.

[11] F. Trollmann, S. Albayrak, Extending Model to Model Transformation Results from Triple Graph Grammars to Multiple Models, in: Proc. ICMT 2015, vol. 9152 of *LNCS*, Springer, 214–229, 2015.

[12] Object Management Group, Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT), http://www.omg.org/spec/QVT/1.1, 2011.

[13] F. Hermann, H. Ehrig, C. Ermel, F. Orejas, Concurrent Model Synchronization with Conflict Resolution Based on Triple Graph Grammars, in: FASE 2012, vol. 7212 of *LNCS*, 178–193, 2012.

[14] H. Ehrig, F. Hermann, H. Schölzel, C. Brandt, Propagation of constraints along model transformations using triple graph grammars and borrowed context, JVLC 24 (5) (2013) 365–388.

[15] R. Machado, L. Foss, L. Ribeiro, Aspects for Graph Grammars, ECEASST 18.

[16] R. Machado, R. Heckel, L. Ribeiro, Modeling and Reasoning over Distributed Systems using Aspect-Oriented Graph Grammars, in: RULE'09, vol. 21 of *EPTCS*, 39–50, 2009.

[17] M. Koch, F. Parisi Presicce, Describing Policies with Graph Constraints and Rules, in: ICGT'02, vol. 2505 of *LNCS*, 223–238, 2002.

[18] A. Habel, K.-H. Pennemann, Correctness of high-level transformation systems relative to nested conditions, MSCS 19 (2) (2009) 245–296.

[19] OMG, OMG Unified Modeling Language (OMG UML) Version 2.5 FTF - Beta 1, Tech. Rep. ptc/2012-10-24, OMG, 2012.

[20] P. Bottoni, A. Fish, A. Heußner, Coloured Modelling Spider Diagrams, in: Proc. Diagrams 2014, vol. 8578 of *LNCS*, Springer, 45–47, 2014.