On the Challenges of Model Decorations for Capturing Complex Metadata

Horacio Hoyos Rodriguez
University of York
UK

Athanasios Zolotas
University of York
UK

Dimitris Kolovos
University of York
UK

Richard F. Paige
University of York
UK
McMaster University
Canada

Abstract—Model decorations have proven useful as an extension mechanism to provide bespoke language extensions for particular scenarios in the language's domain. However, the current state of the art has only explored extension mechanisms that allow capturing basic metadata, e.g. additional attributes. In this paper we explore the challenges encountered when decorations must capture more complex metadata, in particular metadata that targets model management operations. Additionally, we provide an initial take on these challenges through automated language extension generation. The generated extensions provide enhanced model decoration capabilities that can support metadata of higher complexity.

Index Terms—model decorations, auto-generation, DSL, challenges

I. Introduction

When a modelling language is designed, it is impossible to anticipate all scenarios in which the language will be used [1]. In some scenarios, additional information might be required in order to bridge the gap between intended and real use. Some languages (like UML) have language constructs that allow users to capture additional information in the model (UML profiles allow users to define stereotypes that can add additional properties to an element) for a specific scenario. However, this is not always the case. Extending the language to meet these scenarios can lead to model pollution [2], reduce the flexibility, and require continuous changes to supporting tools (e.g. editors, verification scripts, etc.).

The purpose of extending the language is to be able to capture additional information (metadata). One possible approach is to define a decorator language that extends the base language by adding constructs that allow capturing the new information. When the decorator language is constructed with the same technologies as the original language, the additional classes can have direct references to the original classes [1]. As a result, new models can be created that capture both the original and the extended language concepts. This approach has the advantage that the additional information is seamlessly captured in the model. The drawback is that the original language is not aware of these extensions and additional tooling is required for also providing seamlessly navigating between elements form the original language and elements of the metadata (e.g. [1]).

Another approach is to capture the metadata in a separate artefact. In this case the decorator language also extends the base language by adding constructs that allow capturing the new information. Since the metadata is kept separate seamlessly navigation is not possible. However, the metadata can be considered a separate model which can then be used as input for model management operations. This approach can be usefull when there exists a set of exiting base models that cannot be modified to add the additional metadata.

Different approaches [1], [3], [4], EMF Facet[1], have been proposed to support decorator languages. Albeit providing solutions that target different concerns, their common driver is to allow users to provide metadata that complements the model data. In this paper we explore why the challenges associated when the metadata is also to be used to guide model management operations. We understand model management operations as activities that consume information captured in the model, e.g. model validation, model-to-model transformations, model-to-text generation, etc. We will use a particular language and scenario to provide context for the challenges and to present initial ideas on how to provide solutions for these challenges. Our example a uses separate artefact for metadata, but we think techniques can be applied in both cases.

Section II introduces the domain that motivated this work and Sect. III describes the metadata we are interested in and presents the challenges of capturing complex metadata. Section IV gives an overview of our solution to these challenges and finally Sect. V concludes.

II. GSN and GSN Patterns

The development of assurance cases is a key part of engineering critical systems. Assurance cases are now an accepted part of certification arguments in areas such as defence, rail, automotive and aviation. An assurance case presents a structured argument aimed at ensuring that the safety or security of a system can be demonstrated with respect to evidence (e.g., tests, inspections). Assurance cases (or safety cases and security cases) are typically specified

[1] https://www.eclipse.org/facet/

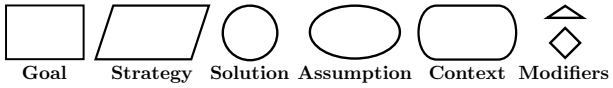


Fig. 1. Goal Structuring Notation [8]

.

using structured languages, including Goal Structuring Notation (GSN) [5] or the OMG's standard Structured Assurance Case Metamodel (SACM) [6]. "Indeed, the development and acceptance of a safety case is a key element of safety regulation in many safety critical sectors. However, safety cases are typically constructed manually, since many tools rely on diagrammatic drawing support input (i.e. 'boxes and arrows'). This is time-consuming and expensive, especially when we are dealing with large amounts of artefacts and iterative software development" [7].

In our work as part of the SECT-AIR project we were looking into developing tools that would reduce the amount of manual work required to build safety cases. In particular, we were interested in how the safety cases could be auto-generated and how the information required to generate them could be captured or retrieved from other sources. This paper presents work on the former.

The Goal Structuring Notation (GSN) [8] is a well-established graphical argumentation notation to represent safety arguments in a structured way. It can be used to explicitly present "the individual elements of any safety argument (requirements, claims, evidence and context) and (perhaps more significantly) the relationships that exist between these elements (i.e. how individual requirements are supported by specific claims, how claims are supported by evidence, etc.)" [5]. The GSN is widely adopted within safety-critical industries (i.e. aerospace, railways and defence) for the presentation of safety arguments within safety cases.

A. The GSN Language

Fig. 1 presents some elements of the GSN notation, in particular those used in the examples that follow. A Goal represents a claim about the system. When a claim can only be supported by other claims, a Strategy is used to break down a goal into subgoals. When a claim can be directly supported by evidence, a Solution is used. Assumption and Context elements can be used to provide additional information about when and how a claim can be made. The modifiers are used to state that an element needs to be instantiated (triangle) or that it is undeveloped (diamond). Edges between elements representSupportedBy relations and can either be simple, choice (diamond) or multiple (circle).

B. GSN Patterns

In some particular domains it is common that parts of the safety arguments follow a similar structure. The appearance of these repeated structures suggests that they are a correct, comprehensive and convincing set of arguments that can be reused in the future. These successful structures can be captured as GSN patterns [9].

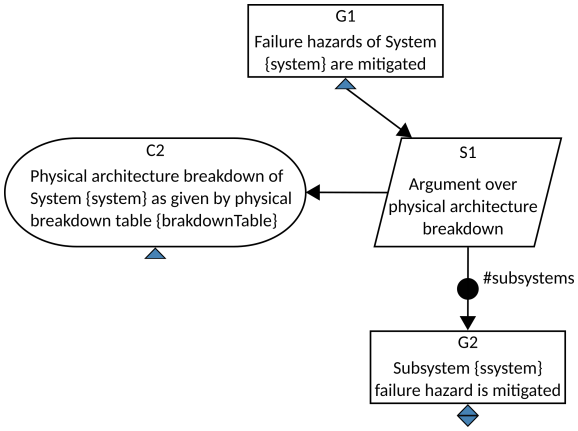


Fig. 2. Physical Decomposition Pattern [10]

All elements of a GSN pattern are uninstantiated (i.e. they represent a to-be) and considered to be abstract. GSN uses the Role mechanism (represented by text inside curly braces) to indicate that parts of a label (names, descriptions, etc) are replaceable [5]. For the structure, relations (edges) can be marked optional, with a required cardinally, or as choices to convey that the section of the pattern under the edge has structural restrictions.

Fig. 2 presents an example of a GSN pattern. In Goal $G1$ the $\{system\}$ role allows the name of the system to be replaced and in $G2$ the $\{ssystem\}$ role can be used to place the name of the different subsystems in the system. The $G1$-$S1$-$G2$ structure shows that the claims about the system are suported by claims about the system's subsystems. The supported by relation between $S1$ and $G2$ is labelled #subsystems to accommodate that a system can be divided into any number of subsystems.

C. Pattern Instantiation

A GSN pattern is used by applying it to a specific element of an existing GSN model. During instantiation, the element is replaced by copies of the pattern's elements. Further, the roles are replaced by specific values and optional values, multiplicities and choices are resolved. This means that a sub-structure of the graph can be copied 0, 1 or multiple times. From this, it follows that pattern instantiation requires the existence of metadata that provides values for the roles, as well as information on how to resolve optional, multiplicities and choices. We consider instantiation a model management operation on the pattern, the existing model and the metadata.

III. Decorating GNS to capture pattern meta-data

In this section we discuss how decorations could be used to support GSN Patterns and while doing so, we present

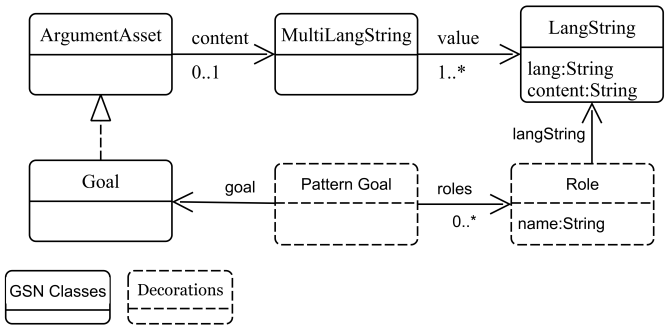


Fig. 3. Simplified extract of the GSN Metamodel with decorators.

the challenges that we have found when working with complex metadata. Although the GSN standard discusses patterns and defines how roles, optionals, multiplicities and choices can be defined, the current GSN language abstract syntax does not provide constructs to capture this information. Effectively, the information is stored as strings. As a result, any model operation needs to parse the text in order to extract the correct values. For example, in the case of roles, text inside curly braces can be extracted to know what metadata must be captured in the model or additional artefacts.

A. Capturing role metadata

Continuing with the role example, in order to capture the exact roles, we need to extend the description text attribute in GSN elements so that role can be captured. Fig. 3 presents a simplified extract of the GSN metamodel and a possible decoration for capturing roles. The shown classes depict how the content of a goal (the text) is made up of multiple LangStrings (to support different languages). In order to explicitly capture role information we could decorate the LangString class. Using the approach from [1] we defined a Pattern Goal class that captures the use of goals in patterns and a Role class that allow us to capture the differnt roles used in a goal.

As a result, for the pattern in Fig. 2 we could decorate G2 with a Pattern Goal instance and one Role with name ssystem. The proposed decoration effectively captures the role information. But what about additional master data, i.e. the role value(s)?

For pattern instantiation the above decoration is not enough because it does not capture role value(s). We could modify the Role class and add a value:String attribute to capture the value to use for the role during instantiation. However, this solution does not consider the semantics of GSN pattern instantiation as explained next.

B. Capturing complex metadata

The pattern semantics indicate that during pattern instantiation multiple copies of an element can be created. This happens whenever the pattern has a multiplicity relation. In such cases, the complete sub-tree, starting with the first node below the multiplicity, must be duplicated as many times as the multiplicity role is defined.

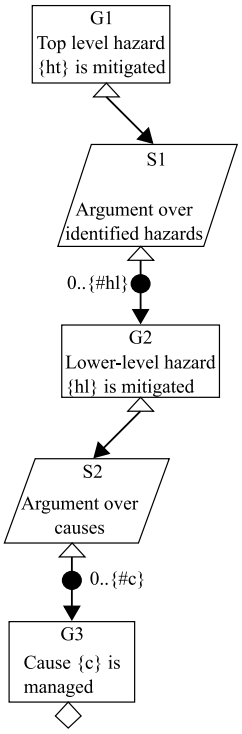


Fig. 4. Multi-level pattern [10]

Consider the pattern in Fig. 4 adapted from [10][2]. The pattern can be used to argue that hazards, their parts, and their identified causes have been mitigated. The top-level claim (G1) is developed over the lower-level hazards (G2) that constitute the to the top-level hazard. The pattern captures the argument implicit in a hazard table in which hazards are linked to their causes (G3).

This pattern has two multiplicity relations: decomposition of the top-level hazard into other hazards, and the causes linked to a hazard. Further, these relations are nested. During instantiation, the sub-tree starting at G2 would need to be replicated hl times. And for each of this copies, the subtree below G3 would need to be replicated c times. As a result, when capturing role values, each {ht} role value would need to be associated with many {hl} values, and each {hl} value associated with many {c} values. Thus, additional constructs are required in the decoration language in order to capture the instantiation semantics. That is, the decorator metadata has additional complexity.

One possible solution is to mimic this tree structure by allowing roles to have children and a parent, as presented in Fig. 5. The main problem with this approach is that it is too general. That is, any role structure is allowed to be captured. For example, we could put c role metadata

[2]The original pattern also links hazards to their mitigations and eventually to safety requirements.

under ht role metadata, which is clearly not what is required. Therefore, this solution would require additional validation to ensure that the metadata structure is well-structured in order to be used for instantiation.

Ideally, we would like a decorator model with more detailed information thus simplifying use of the complex metadata. For example, we could add specific pattern elements and roles to the decoration language, as presented in Fig. 6. However, these decorations would need to be created per pattern and as more roles are used, the complexity of modelling the required relations increases.

C. Overview of the challenges

Complex metadata, like those required for guiding model management operations, place additional requirements on the decoration language. As discussed, a general decoration language is not sufficiently expressive to represent complex metadata and requires additional validation activities. Second, the required decorations can even per-model basis, turning the decoration activity in much more time-consuming. A side effect of this fine-grained granularity is that reuse of decoration languages is reduced. In a nutshell, when metadata is tightly coupled to the semantics of model operations, the ability to define more fine-grained decorations and model more complex relations is required.

IV. Generating decorators

This section presents one possible approach to solve the challenges of creating decoration languages to manage complex metadata. Our approach is based on the idea of auto-generation of decoration languages. The auto-generation approach provides some key features:

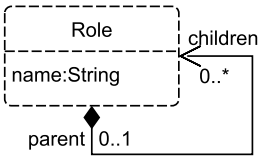


Fig. 5. Adding a tree-like structure to role decorations.

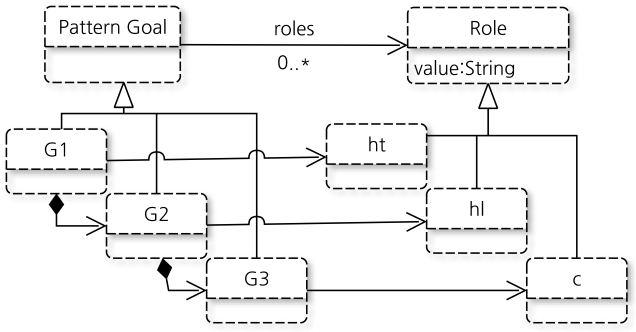


Fig. 6. Adding pattern specific elements.

- Facilitates creating fine-grained decorations, even within the same domain.
- Speeds up development of tools (e.g. editors) to construct decoration models.
- Can handle complex metadata (e.g. complexity based on the language semantics).

Our prototype has been developed for GSN Patterns, but it can be easily replicated in other domains.

A. Understanding the domain

For GSN Patterns, we identified that the complex metadata is structured as a tree where branches are related to the SupportedBy relations' multiplicity, optionality or selection. Since this tree structure is a result of the semantics of GSN Pattern instantiation and not from a particular pattern, we use it as a base structure for the auto-generated decoration models. Thus, decoration languages for GSN patterns follow a tree like structure in which each node can capture specific role information.

In our approach we call the role:value pairs a link and each node in the tree can have 0 or more links. We also identified that roles are often reused throughout the pattern in order to describe relations between them. For example, in the pattern in Fig. 2 the system role is used in G1 and C2. As a result, since links are captured for each node this would result in duplicate information being captured. Duplicate information can be the source for errors and can lead to inconsistencies. For this reason, our approach supports the concept of variables as in most programming languages: variables are used to store information to be referenced and manipulated. Finally, apart from roles we also need to be able to capture multiplicity and selection information.

Designing a Domain Specific Language (DSL), in this case the decoration language, is not a trivial task. Considerations like the framework used for development, selecting the concepts captured by the language, and the level of abstraction, among others, must be taken in order to reduce the risk associated with developing a DSL [11]. The main characteristics of the GSN Pattern decoration language are defined by the roles and by the relations between the pattern elements. How these characteristics are captured by the decoration language syntax, is presented in the next sections.

Given that the nature of links is to capture text values, we opted for creating decoration languages that use textual notation. Fot this reason, we selected the Xtext framework [12] as our target platform. That is, the auto-generated decoration languages are specified as Xtext concrete syntax instances. These instances can then be used to generate editors that support the decoration language. We will initially discuss the decoration language syntax using EBNF and then show how we can do it in Xtext.

B. A GSN Pattern decoration template

Our first step was to define a language template that captures the tree structure and the existence of links. The template define the basic constructs that all specific GSN Pattern decoration languages will have. The proposed language is indentation-based (i.e. structure is defined by indentation), thus the tree structure is represented by indentation levels. The template is presented in Listing 1 in a textual representation (EBNF). We use the <xxx> notation to represent a replaceable concept.

Listing 1. Template for GSN Pattern Languages

```
1 data ::= variable* <gsnnode>*;
2 variable ::= ID '=' STRING;
3 <gsnnode> ::= (count <branchnode>) | <node>;
4 <branchnode> ::= '<name>_br' ':' <node>*;
5 <node> ::= '<node_name>' ':'<link>* gsnnode*;
6 <link> ::= '*<role>' '=' ID | STRING;
7 count ::= <max> | (<min>,<max>)
```

The Pattern (meta)data (line 1) is composed of variables and gsnnodes. A variable (line 2) has a name, given by an ID and a value of type String. A gsnnode can either be a branch node or a simple node. Branch nodes are used to represent nodes that follow a multiplicity relation. Branch nodes have multiplicity limits given by their count and can only contain other nodes (line 4). Count is loosely defined as either a max, or min,max replaceable concepts so there is flexibility for fixed and dynamic (role) counts. The proposed min/min-max representation can capture both multiplicity (*min..max*) and selection (*min* of *max*) information. A node (line 5) has a name, and can have multiple links and nested nodes. Finally, a link allows paring a role to its value. Links are prefixed with an '*' to differentiate them from nodes. The value can either be a String or the name of a previously defined variable (ID).

Replaceable concepts are intended to be substituted by specific pattern values when auto-generating the GSN pattern's language. Each of the nodes in the pattern should have a corresponding node construct. Each of the multiplicity relations should have a corresponding branchnode construct. For each of the roles on each node, a <link> construct with the roles name should be present.

Listing 2 presents the decoration language specification for the pattern in Fig. 4. Branches from goals G1 and G2 have been defined as branch nodes: g2_br and g3_br. In both cases the ability to have multiple subtrees is specified by indicating that the construct can appear multiple times ('*' in EBNF). Simple nodes for G1, G2 and G3 (lines 2, 6 and 10) have definitions for roles ht, hl and c respectively. There is also an explicit capturing of the multiplicities via hl (line 5) and c (line 9). Note that using the proposed structure for decoration languages, the node names, roles and structure are hard-coded into the language, giving little room for errors when capturing metadata (i.e. using the language to create models).

Listing 2. Simple Pattern DSL

```
1 data ::= 'G1' ':' g1 s1;
2 g1 ::= '*ht' '=' ID | STRING;
3 s1 ::= 'S1' ':' hl g2_br*;
4 g2_br ::= 'G2' ':' g2 s2;
5 hl ::= 'hl_count' '=' INT;
6 g2 ::= '*hl' '=' ID | STRING;
7 s2 ::= 'S2' ':' c g3_br*;
8 g3_br ::= 'G3' ':' g3;
9 c ::= 'c_count' '=' INT;
10 g3 ::= '*c' '=' ID | STRING;
```

C. Matching domain semantics to grammar constructs

So far we have presented the GSN Pattern languages in EBNF notation. However, we stated that we target the Xtext framework. In this section we present how the GSN Pattern instantiation semantics can be mapped to grammar constructs supported by Xtext.

The proposed template and example language only discuss the case of multiplicities in relations. However, GSN pattern instantiation semantics also accounts for optional and select relations. Optional relations depict the case in which the target node (and its sub-tree) of the relation can either be present or not. Selection relations depict the case in which n of the target nodes (and its sub-tree) of the relation can be present or not. The mapping between multiplicities and optionals, and grammar constructs is as follows:

- **Option** These relations are represented graphical in GSN with an empty circle in the relation. Since the relation indicates present or not, we use the Xtext optional construct, denoted by a question mark (?), to represent it. That is, if a branchnode is optional it will be followed by a question mark: <>_br?.
- **Selection** These relations are represented graphically in GSN with a solid diamond relation with one source and multiple targets. Additionally, the label of the relation is of the form n of m, which indicates an n of m selection. Both n and m can be replaced by roles. In Xtext we use the Alternatives construct, denoted by a bar (|), to represent it. Note that the alternative construct can list the alternatives but can't enforce the limits.
- **Multiple** These relations are represented graphically in GSN with an filled circle in the relation. Additionally, the label of the relation is of the form n..m or m, which indicates an n to m or 0 to m multiplicity. In Xtext we use the zero or more construct, denoted by an asterisk (*), to represent it. As with the selection case, we cannot enforce the limits via the grammar.

Since limit information for both selection and multiplicity relations is captured in the model, this information can be used to provide instant validation on the generated editors.

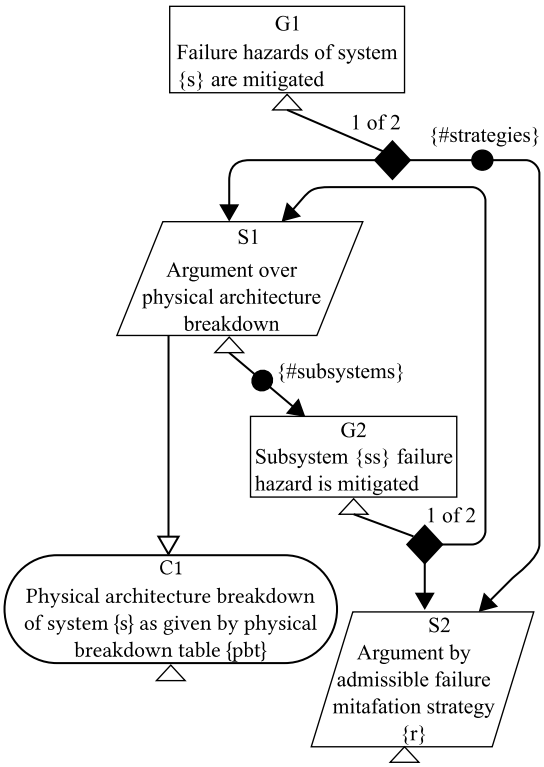


Fig. 7. Extended/Hierarchical physical decomposition pattern, adapted from [10].

D. Example

Fig. 7 presents the Extended/Hierarchical physical decomposition pattern (EPDP), adapted from [10]. The intent of the pattern is to assure that failure hazards of a system have been sufficiently mitigated. An excerpt of the automatically generated language in Xtext is presented in Listing 3, which includes some Xtext features that are important in our approach.

The BEGIN and END keywords are used to specify that the language is indentation based. In Xtext rules can be assigned to variables to ease retrieving information. For example, in line 2 we store all the Vars in a variable called vars. When inspecting the parsed tree, these variable names can be used to easily replace their values when used elsewhere. Line 44 shows another important feature of Xtext. The name inside square brackets indicates that a variable reference (VarRef) must point to a previously defined variable (Var). This will guarantee that only previously defined variables are used, reducing the number of errors.

Listing 3. EPD pattern language in Xtext

```
1  data:
2    (vars+=Var)*
3    top=g1;
4  g1:
5    'G1'':'
6      BEGIN
7      '*s' '=' s=(STRING | VarRef);
8      g1s1=s1 | (count=s2_count g1s2+=s2_br*);
9      END;
10 s1:
11   'S1:'
12     BEGIN
13     s1c1=c1 (count=g2_count s1g2+=g2_br)*;
14     END;
15 c1:
16   'C1'':'
17     BEGIN
18     '*s' '=' s=(STRING | VarRef);
19     '*pbt' '=' pbt=(STRING | VarRef);
20     END
21 g2_br:
22   BEGIN
23     top=g2 (g2s1=s1 | g2s2=s2);
24   END
25 g2_count:
26   'subsystems_count' '=' INT;
27 g2:
28   'G2'':'
29     BEGIN
30     '*ss' '=' ss=(STRING | VarRef);
31     END
32 s2_gb:
33   top=s2
34 s2_count:
35   'strategies_count' '=' INT;
36 s2:
37   'S2'':'
38     BEGIN
39     '*r' '=' r=(STRING | VarRef);
40     END
41 Var:
42   name = ID '=' value=STRING;
43 VarRef:
44   ref = [Var];
```

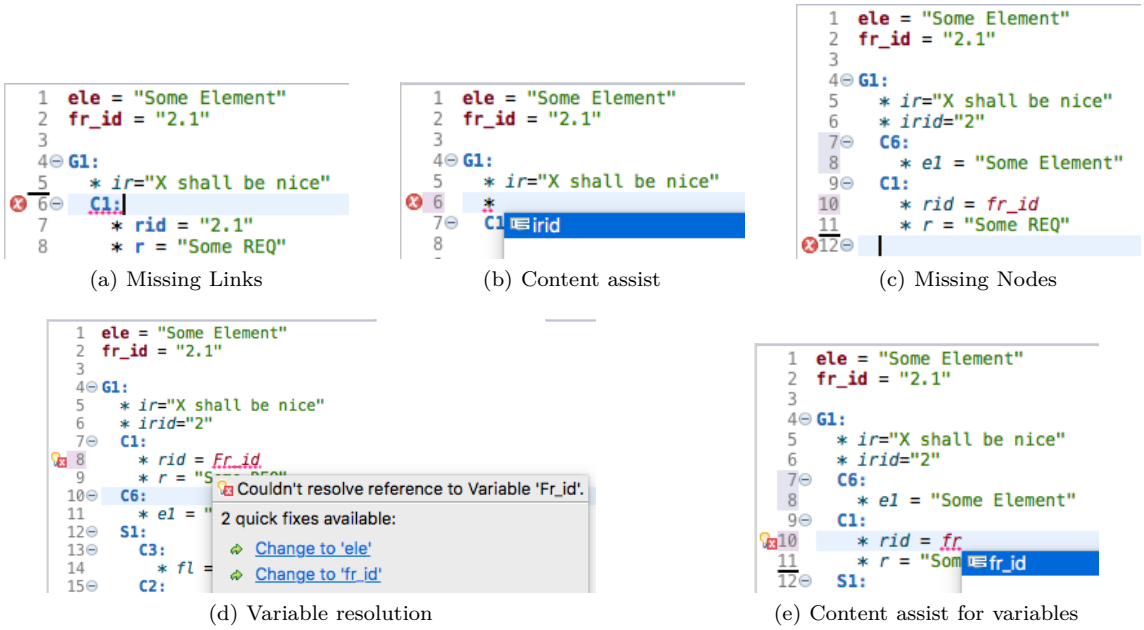
E. The generated editor

Fig. 8 presents screenshots of the generated Pattern language editor for another Pattern. In Figure 8a missing links are marked as errors. By using content assist (Figure 8b) users can find which links are missing. Further, only the links required by the specific node are allowed thus eliminating errors. Similarly, in Figure 8c the absence of required nodes (in this case S1) is also highlighted as an error. As with links, the content assist will correctly list/add the missing nodes for that specific section of the pattern. Finally, when using variables, the editor highlights unknown variables as errors. In this case, the Fr_id variable has not been declared. The editor can suggest possible replacements and as before, the content assist can also be used (Figure 8e).

To evaluate our approach, we are planning to run a comparative study with safety engineers. The main objective would be to measure the gains, if any, of using auto-generation. We are also interested in a qualitative evaluation about the proposed syntax and language structure.

V. Conclusions

Model decorations are key to allow DSL to be used beyond their intended use. This paper presented the challenges associated when the model decorations need to handle complex metadata and why the state of the art



(a) Missing Links (b) Content assist (c) Missing Nodes

(d) Variable resolution (e) Content assist for variables

Fig. 8. Pattern aware meta-data editor

is not sufficiently expressive. For example, when metadata affects the execution model management operations on the decorated models.

We also presented a possible solution for the challenges which is based on the automatic generation of the decoration languages. The generation can take into consideration the semantics of the domain and the requirements of the model management operations. Auto-generation allows complex decoration languages to be generated for more fine-grained scenarios and reduces the manual labour that entails the creation of multiple decoration languages.

Acknowledgment

This work was supported by the SECT-AIR project, funded by the Aerospace Technology Institute and Innovate UK, as project number 113099.

References

[1] D. S. Kolovos, L. M. Rose, N. Drivalos Matragkas, R. F. Paige, F. A. C. Polack, and K. J. Fernandes, "Constructing and navigating non-invasive model decorations," in Theory and Practice of Model Transformations, L. Tratt and M. Gogolla, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 138–152.

[2] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "On-demand merging of traceability links with models," in 2nd ECMDA Traceability Workshop (2006), Proceedings of the, 2006.

[3] P. Langer, K. Wieland, M. Wimmer, and J. Cabot, "From uml profiles to emf profiles and beyond," in Objects, Models, Components, Patterns, J. Bishop and A. Vallecillo, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 52–67.

[4] H. Bruneliere, J. Garcia, P. Desfray, D. E. Khelladi, R. Hebig, R. Bendraou, and J. Cabot, "On lightweight metamodel extension to support modeling tools agility," in Modelling Foundations and Applications, G. Taentzer and F. Bordeleau, Eds. Cham: Springer International Publishing, 2015, pp. 62–74.

[5] S. A. C. W. Group, "Goal structuring notation community standard," Online, The Safety-Critical Systems Club, 2018.

[6] OMG, "Structured assurance case metamodel," https://www.omg.org/spec/SACM/2.0/PDF, 2018. [Online]. Available: https://www.omg.org/spec/SACM/About-SACM/

[7] E. Denney, G. Pai, and J. Pohl, "Automating the generation of heterogeneous aviation safety cases," NASA, Tech. Rep. NASA/CR-2011-215983, 2012.

[8] T. Kelly and R. Weaver, "The goal structuring notation – a safety argument notation," in Proc. of Dependable Systems and Networks 2004 Workshop on Assurance Cases, 2004.

[9] T. Kelly and J. McDermid, "Safety case patterns-reusing successful arguments," in IEE Colloquium on Understanding Patterns and Their Application to Systems Engineering (Digest No. 1998/308), 4 1998, pp. 3/1–3/9.

[10] E. W. Denney and G. J. Pai, "Safety case patterns: Theory and applications," NASA, Tech. Rep. 20150004086, 2015.

[11] U. Frank, Domain-Specific Modeling Languages: Requirements Analysis and Design Guidelines. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 133–157. [Online]. Available: https://doi.org/10.1007/978-3-642-36654-3_6

[12] M. Eysholdt and H. Behrens, "Xtext: Implement your language faster than the quick and dirty way," in Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 307–309. [Online]. Available: http://doi.acm.org/10.1145/1869542.1869625