Language Enrichment for Resilient MDE

Yasir Imtiaz Khan and Matteo Risoldi

University of Luxembourg, Laboratory of Advanced Software Systems
6, rue Richard Coudenhove-Khalergi – L-1359 Luxembourg
{yasir.khan,matteo.risoldi}@uni.lu

Abstract. In Model-Driven Engineering, as in many engineering approaches, it is desireable to be able to assess the quality of a system or model as it evolves. A resilient engineering practice systematically assesses whether evolutions improve on the capabilities of a system. We argue that to achieve a systematic resilient model-driven engineering practice, resilience concepts should be first-class citizens in models. This article discusses how DREF, a formal framework defining resilience concepts, can be integrated with other modeling languages in order to pursue a resilient development process.

Keywords: resilience, language, metamodel, composition, enrichment.

1 Introduction

Many current development methodologies for software systems support iterative refinements and/or incremental developments. This is well suited to respond to needs such as changing requirements, optimization of development resources and early detection of problems. This has typically been true for techniques known as "agile", but trends are developing in order to bring these qualities to approaches that have traditionally a reputation of being less flexible, like Model-Driven Engineering (MDE) [14,2]. Iterations and incremental development have thus acquired the status of current practices in MDE. In most cases it is desirable for the developers to be able to assess the quality of the system as it evolves. In particular, it is interesting to know whether each new version of the system satisfies the requirements better than the previous one. We will call a system evolution process that improves quality with each new version a *resilient evolution process.* A resilient evolution process is especially desirable for dependable systems, where keeping or improving the satisfaction of properties is highly critical.

Achieving a resilient evolution process requires having quality metrics of the system, and assessing their variation during evolution. While this could in principle be done informally, we argue that a systematic practice of resilience calls for a well-defined set of concepts such as required and achieved satisfiability, failures or tolerance levels. Moreover, we argue that development and quality assurance would benefit from resilience concerns being "natively" included in languages and tools.

DREF [7] is a formal framework that precisely defines the fundamental concepts underlying dependability and resilience of ICT systems. It is oriented to describe how the satisfaction of properties of a system changes over a system evolution axis. The evolution axis can represent different types of evolutions, e.g., different versions of a system, different products in a product line, or even different states in the runtime evolution of a system. DREF proposes measures of satisfiability at various granularity levels, including concepts like different observers, property and/or observer weights, and tolerance thresholds. It defines the concept of *resilient evolution process* as an evolution process of a system that improves its capabilities, increasing overall satisfiability and reducing failures. DREF is rather generic and leaves the definition of details like the exact nature of the system under study, of its properties or of the methods used to assess satisfiability to the user. Therefore it can be applied to a wide range of systems and evolution processes.

The formal definition of DREF has been given [7]. A prototype metamodel for a DREF Domain-Specific Language (DSL) has been defined [7,16] in order to tailor DREF to the Model-driven engineering methodology. In this article, we discuss how the DREF metamodel can be used with other existing modeling languages in order to enrich them with resilience concerns. We will discuss a few approaches to associate a model with a DREF specification, and discuss their advantages and disadvantages. We will also show a simple case study where a model expressed in Algebraic Petri Nets (APNs) undergoes a number of evolutions, with DREF being used to assess the resilience of the evolution process. The goal of this article is not to introduce new language composition techniques. It is rather to give practical advice on the advantages and disadvantages of some existing composition techniques in systematic DSL enrichment, required for pursuing a resilient model-driven development practice.

2 Background and Previous Work

Resilience is a concept that is strictly related to evolution. According to [13], resilience is defined as *"[t]he persistence of service delivery [...] when facing changes"*. These *changes* can be environmental or intrinsic. Some of these changes are part of the planned behavior, while others are not and may be regarded as faults.

Generally speaking, the term *resilient* is frequently intended as the system being good at remaining – or returning – in an acceptable range of operation despite disruptive changes during its evolution at runtime. This view of resilience is somewhat akin to fault tolerance.

However, systems and models are also subject to evolution during their development phase, where an initial version goes through a series of evolutions generally aimed at improving its capabilities – among other things, the satisfaction of its requirements and properties. But it is often not trivial to understand whether or not an evolution has actually brought an improvement in requirement satisfaction. Behaviors may be so complex that a modification may potentially have a positive impact on some properties and a negative impact on others, and the net result may be difficult to quantify objectively. In this respect, we argue that it is desirable to speak about the *resilience of the evolution process itself* that takes place during the development phase. This "view" of resilience is tantamount to measuring whether the evolution process aims in the direction of a general improvement, defined in terms of how well the system is satisfying required properties.

One might imagine going about assessing the resilience of an evolution process by ad-hoc techniques where some metrics are identified and repeatedly used to calculate property satisfaction during evolution. We argue however that for the model-driven engineering community, requirement satisfaction and resilience are attributes that should be first-class citizens in a model, and thus they should appear as part of a specification: i) in an explicit way and ii) with a precise definition. The formal definition of DREF [7] tackles point ii); in the context of model-driven engineering, we propose to tackle point i) by composing modeling languages with DREF, creating models that explicitly include resilience concepts.

2.1 The DREF Metamodel

DREF (Dependability and Resilience Engineering Framework) [7] is a formal framework that precisely defines the fundamental concepts underlying dependability and resilience of ICT systems. It allows to quantify variations in the level of property satisfaction over an evolution axis. DREF is based on the following core concepts.

– An **entity** is anything of interest that is considered. An entity could be, for example, a program, a database, a person, a hardware device, or a development process.
– A **property** is a basic concept used to characterize an entity. It can be, for example, an informal requirement or a logic formula.
– An **evolution axis** is a set of values that are used to index a set of entities and/or a set of properties. Each index corresponds to a "version", or a stage in the evolution of entities and properties.
– An entity will generally have to satisfy some property. This fact is expressed with a **satisfiability function**, defined as follows. Let Ent be a set of entities and $Prop$ a set of properties. The satisfiability of properties by entities is a function $sat : Prop \times Ent \rightarrow \mathbb{R} \cup \{\perp\}$. The sat function can be defined arbitrarily depending on the application. For example, if a property can only be "satisfied or unsatisfied" (like, e.g., in model checking), the codomain of sat might be $\{0, 1\}$; whereas if the satisfiability is a more nuanced concept (like, e.g., in a performance measurement), it could assume any value in $\mathbb{R}$, or a subset thereof. Semantically, sat quantifies how much an entity satisfies (or not) a property. The satisfiability function sat is a partial function, and can be defined only for a subset of $Prop \times Ent$, meaning that for some entity/property pairs a satisfaction value could be not expected (in other

words, some properties might be applicable only to some entities and not others). Moreover, the ⊥ value accounts for the cases where the satisfiability value is not computable.

Using the above concepts in the engineering process, it is possible to get an assessment not only of the extent to which an entity satisfies its properties, but also of how this satisfiability changes during subsequent evolutions of the entity.

The DREF framework also defines a number of other concepts which are useful for dependability and resilience. We will not give a complete definition for them all as this is not the goal of this paper, they are fully defined in [7]. They include *nominal satisfiability* (a satisfiability level that has to be reached for an entity to be considered dependable), *tolerance thresholds* (a satisfiability level below nominal satisfiability but still within operational limits) and *failures* (the difference between the measured satisfiability of an entity and its nominal satisfiability).

In order to use the DREF framework in an MDE context, a metamodel has been given for DREF concepts. This metamodel defines the abstract syntax of a DREF DSL, and can be used to create a DREF specification referring to a model expressed in some other language. This raises the question of how a DREF specification integrates with a model in a different language. In particular, the entities and properties of the DREF framework should be somehow expressed in the other language, so that the concepts of entity and property should bridge the two languages.

A fragment of the class diagram for the DREF metamodel is shown in Figure 1. This fragment focuses on the DrefEntity and DrefProperty metaclasses that are – if we may borrow the aspect-oriented terminology – the “join points” between DREF and other languages. A complete description of the metamodel and its associated constraints is given in [16].

Remark that DrefEntity contains an abstract ModelEntity metaclass. The latter represents the actual entity in the model expressed in the other language. We will see that there are three ways in which we can link this metaclass to the other language. A similar structure is present for the DrefProperty metaclass, that contains the abstract ModelProperty metaclass.

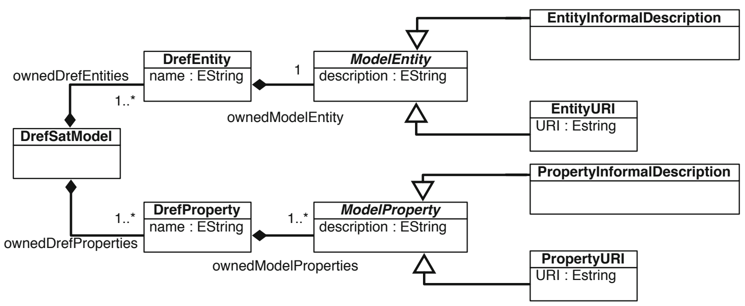


Fig. 1. A fragment of the DREF metamodel

3 Related Work

Enriching languages with new concerns has been done in a number of ways. In Aspect-Oriented Programming [12] (AOP), languages are extended with cross-cutting concerns given in a separate modular specification. AOP can be seen as a generic composition mechanism where a concern can be added to a model. It needs compilers and platforms that support aspect execution, and is appropriate for general purpose languages where the necessary constructs to weave aspects in a model can be added to the language once and used for different concerns. A popular example of AOP language is AspectJ [1].

Another approach to extending languages is by embedding a (generally small) sublanguage in a host language. The sublanguage will typically treat a specific concern, and may therefore be considered a Domain-Specific Language (DSL). Approaches exist that embed DSLs in host languages using, among others, keyword extension [3], role bindings [4] and term representation composition [8]. Embedded DSLs may be used in conjunction with AOP for the generation of code [4].

In the field of DSLs, language composition is mostly sought for modularity and reuse. DSLs bring usability by defining small, dedicated sets of concepts, but tend to have a moderate to high cost of development and deployment. This effort can be minimized by reusing predefined DSLs for the definition of more complex ones. This is particularly useful when families of languages need to be developed. Techniques to specify the composition include ad-hoc techniques based on metamodel [15,5] and domain [6,10] composition, or more systematic techniques for DSL reuse in families of languages [19,18].

This paper is based on DSL-oriented approaches, in particular on metamodel-based techniques, as they best suit our current applications and goals. It must be noted that DSL embedding or AOP composition could be pursued, provided that the DREF DSL is specified with an appropriate syntax and semantics.

4 Composing DREF with other Languages

We will discuss three ways in which DREF and another language can be used together. The first is using DREF stand-alone on the side of the modeling language. The second is composing DREF with a modeling language through metamodel parameterization. The third (which can actually be done in two different ways) is composing with metamodel interfacing. Note that the second and third techniques imply that a metamodel is available for both languages, and that they are homogeneous.

Also remark that DREF does not have an executable semantics; it is a conceptual framework designed to describe sets of organized data, with no concept of execution. Furthermore, the semantics of its concepts is intentionally very abstract, to allow the user to define it appropriately in the context of language composition. Thus we will limit ourselves to consider the composition of DREF with other languages on a syntactical plan (i.e., by integrating metamodels) as the semantic aspects are difficult to foresee in a general way.

4.1 Approach 1: Using DREF Stand-Alone

This strategy keeps the DREF specification and the model separate. The model is created using its own modeling language, and the properties are written in an appropriate property specification language. Then, the DREF specification is created on the side. The link between DREF and the model is done using the metaclasses in the DREF metamodel that inherit the ModelEntity and ModelProperty metaclasses (Figure 1). For the entity, there are two possibilities. Either the entity has been saved in a file, and thus an instance of EntityURI is created in the DREF specification pointing to said file (via the URI attribute); or if no such file exists (for example, if the entity is an hardware device), an instance of EntityInformalDescription is created that simply describes textually (via the inherited description attribute) what the entity is. Likewise for properties, either a property file is pointed to by a PropertyURI instance, or an informal description is given by a PropertyInformalDescription instance. Subsequent evolutions of the entity and properties will be linked to further instances of said metaclasses.

This strategy requires no language engineering effort; the DREF metamodel can be used as-is, and the modeling language must not undergo any modifications. Another advantage is that editors for the modeling language will require no re-engineering to read the models, as these continue to be expressed in their supported modeling language. However, there is no proper integration here. The resilience specification is separate from the model, and its consistency with the model has to be ensured manually. Also, if the goal is to enrich a language to natively support resilient engineering, this approach does not achieve it.

4.2 Approach 2: Metamodel Parameterization

Real integration between DREF and another modeling language can instead be achieved through metamodel composition. The general idea is that, instead of creating instances of ModelEntity and ModelProperty in the DREF specification, it should be possible to create instances of the appropriate metaclasses coming from the modeling language.

One way to achieve this composition is through *metamodel parameterization*. This strategy takes two metamodels as inputs for a metamodel transformation, producing a third metamodel which is the composition of the two. This typically involves building and executing the transformation with a suitable language transformation framework such as ATL.

An example of this type of composition has been defined formally in [15]. In it, a part of a metamodel is marked as a *formal parameter*, and it is replaced by an *effective parameter* which redefines the elements in the formal parameter. More precisely, simplifying a bit the definitions in [15]: let MM be the universe of metamodels; $mm \in MM$ a metamodel; we can define a *formal parameter* $fp \in MM$ in mm ($fp \subseteq mm$) acting as a template for possible replacements. In our case, fp is made of ModelEntity and ModelProperty.

Let us now consider a metamodel $ep \in MM$, called *effective parameter* that redefines at least the elements in fp. The parameterization is then defined as:

$$mm' = mm[fp \overset{\varphi}{\leftarrow} ep]$$

where $\varphi : fp \rightarrow ep$ is a total mapping function between fp and ep, and mm' is the metamodel resulting from the parameterization. In our case, ep is made of the metaclasses from the modeling language that model entities and properties, and mm' will be the composed metamodel of DREF plus the modeling language.

This approach offers a fine granularity of control over the detailed definition of the mapping function φ, and has the advantage of treating several possible cases of composition (e.g. solving possible ambiguities with respect to attribute composition, containment relationships, constraint violation etc.). However, it requires a high level of language engineering effort in order to define the composition. Also, the approach would likely break compatibility of the resulting metamodel with the editors for the modeling language. It is worth following this type of approach when some degree of generality is desired with respect of possible types of composition. In the case of DREF, however, the simplicity of composition rather suggests adopting the *metamodel interfacing* composition strategy, described in the following paragraphs.

4.3 Approach 3: Metamodel Interfacing

In the case of DREF, there is a very simple type of composition, where one metaclass must replace an abstract metaclass that only participates in a containment relationship as the containee. Under this assumption, we don't need to treat all possible composition problems and we can choose a strategy that is restricted to this very particular type of composition. Metamodel interfacing [5] proposes to interface two metamodels by creating a third metamodel called *interfacing metamodel* that contains references to elements in both metamodels, and establishes the desired relationships.

There are two ways we can interface the DREF metamodel with another metamodel: through reference, or through inheritance.

In Metamodel Interfacing through Reference, metaclasses in the interfacing metamodel inherit from one metamodel and reference the other. A generic example is represented in Figure 2 and defined as follows. Let:

- mm_{dref} be the DREF metamodel we showed in Figure 1;
- $mm_{ent} \in MM$ the metamodel of the modeling language which contains the definition of the entities;
- $mm_{prop} \in MM$ the metamodel of the property language which contains the definition of the properties.

Let $\mathsf{Entity} \subseteq mm_{ent}$ be the metaclass modeling entities, and $\mathsf{Property} \subseteq mm_{prop}$ the metaclass modeling properties. The interfacing metamodel $mm_{int} \in MM$ contains:

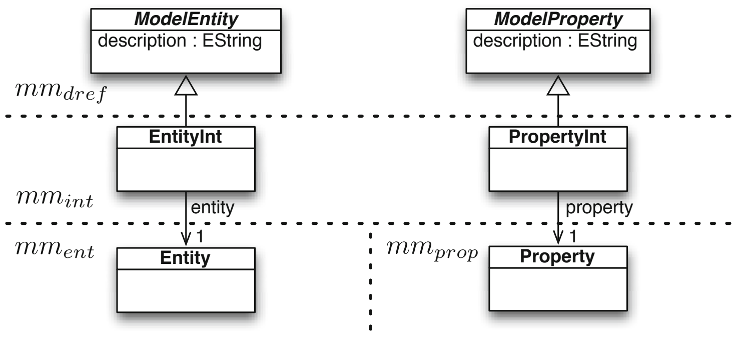


Fig. 2. Example of interfacing metamodel through reference

- a metaclass EntityInt that inherits from ModelEntity in mm_{dref} and has a reference to Entity in mm_{ent};
- a metaclass PropertyInt that inherits from ModelProperty in mm_{dref} and has a reference to Property in mm_{prop};

When modeling, it will be possible to create instances of EntityInt (resp. PropertyInt) as part of the DREF model and to reference existing instances of Entity (resp. Property).

With this technique, the focus of the model stays on DREF. It is recommended to use it when the models for entities and properties already exist. It has the advantage of not modifying the metamodel for entities and properties, thus not breaking compatibility with existing tools and not requiring a big language engineering effort (only mm_{int} has to be created). Moreover, the references between the two models are actually stored in the DREF model.

In Metamodel Interfacing through Inheritance, instead, the metaclasses in the interfacing metamodel inherit from both metamodels, using multiple inheritance. A generic example is represented in Figure 3 and defined as follows.

Given the same definitions as in the previous paragraph for mm_{dref}, mm_{ent}, mm_{prop}, *Entity* and *Property*; the interfacing metamodel $mm_{int} \in MM$ contains:

- a metaclass EntityInt that inherits both from ModelEntity in mm_{dref} and from Entity in mm_{ent};
- a metaclass PropertyInt that inherits both from ModelProperty in mm_{dref} and from Property in mm_{prop};

When modeling, it will be possible to create instances of EntityInt (resp. PropertyInt) that are at the same time part of the DREF model and of the entity (resp. property) model, enabling the specification of resilience and entities (resp. properties) directly in the same model.

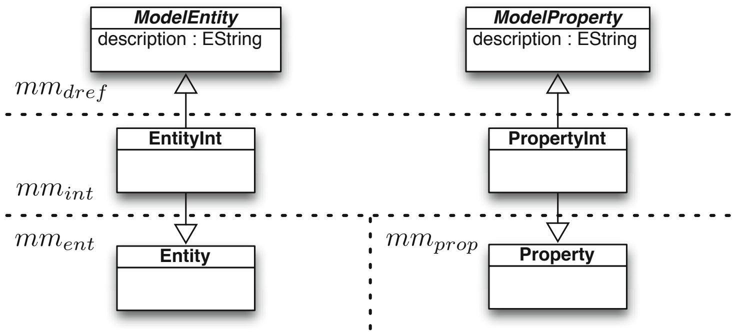


Fig. 3. Example of interfacing metamodel through inheritance

With this technique the same integrated model will contain the entity/property model and the DREF specification. Moreover, a single model will be able to contain all evolutions of entities and properties, indexed by the evolution axis of the DREF specification. This approach achieves full integration, and is better used when designing a language from scratch or if the goal is to enrich a DSL with native support for resilient engineering concepts. However, the inconvenient is that the resulting models will likely not be compatible with tools made to interpret stand-alone entity or property models. Also, care has to be taken if there are syntactic or semantic conflicts between the metaclasses (e.g. if the ModelEntity and the Entity metaclasses have conflicting attributes).

5 Composition Example: DREF + APN

We will now show an example of how we used the metamodel interfacing technique to compose DREF with algebraic Petri nets (APNs), using the APN metamodel from the AlPiNA model checker [9].

Composition through Reference: Figure 4(a) shows the interfacing metamodel composing DREF with APNs through reference, creating a new metamodel that we will call DREFAPN$_r$ (r for *reference*). This metamodel references elements in three other metamodels:

- The DREF metamodel (drefv2 in Figure 4)
- The APN metamodel (apnmm in Figure 4)
- The AlPiNA property language metamodel [9] (propertymm in Figure 4)

The APNModelEntity metaclass inherits from ModelEntity in the DREF metamodel, and references the APN metaclass in the APN metamodel (representing an algebraic Petri net).

The APNProperty metaclass inherits from ModelProperty in DREF and references the PropertiesDeclaration metaclass in the AlPiNA property language metamodel (representing a property declaration).

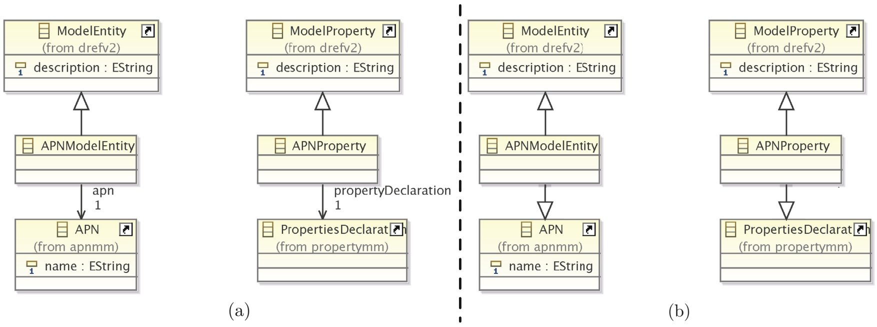


Fig. 4. Interfacing metamodel between DREF and APNs: (a) through reference, and (b) through inheritance

Composition through Inheritance: Figure 4(b) shows the interfacing metamodel that interfaces DREF with APNs through inheritance, creating a new composed metamodel that we will call DREFAPN$_i$ (i for *inheritance*). This metamodel references elements from the same three metamodels as the interfacing through reference approach.

The APNModelEntity metaclass inherits from ModelEntity in the DREF metamodel *and* from the APN metaclass in the APN metamodel (representing an APN). Instances of APNModelEntity will be at the same time a part of the DREF model and the root of an APN specification.

The APNProperty metaclass inherits from ModelProperty in DREF and from the PropertiesDeclaration metaclass in the AlPiNA property language metamodel (representing a property declaration). Instances of APNProperty will be at the same time a part of the DREF model and the root of a property declaration.

6 Case Study: Resilient Evolution of a Car Crash System

We experimented using the discussed integration approaches for the resilient evolution of a car crash emergency management system modeled using APNs [11]. In this system, reports on a car crash are received and validated, and a *superobserver* (i.e. an emergency response team) is assigned to manage each crash.

Three versions of the car crash system have been modeled, with different levels of satisfaction of provided properties. In this example, rather than developing a language from scratch, we took an existing language (with existing tool support), which are APNs, and enriched it with DREF. Therefore, we tried the stand-alone (Section 4.1) and metamodel interfacing (Section 4.3) approaches. We used the Eclipse Modeling Framework (EMF) [17] for metamodel creation and editor generation. This example does not use all of DREF concepts and features, and is intentionally very simple so as to clearly focus on the language composition rather than on a complex resilience specification.

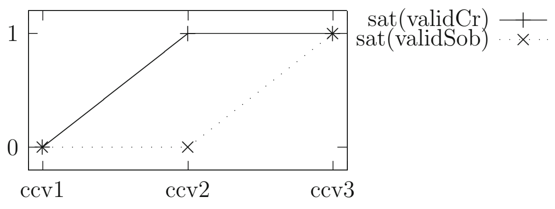


Fig. 5. Measured satisfiability for the car crash system

6.1 Entities and Properties

The set of entities *Ent* is comprised of the three versions of the car crash system (three APNs) named as follows:

$$Ent = \{\mathsf{ccv1}, \mathsf{ccv2}, \mathsf{ccv3}\}$$

The set of properties $Prop \subset PROPERTY$ is comprised of two properties:

$$Prop = \{\mathsf{validCr}, \mathsf{validSob}\}$$

that are informally specified as follows (see [11] for terminology):

- *validCr*: A crisis can only be assigned to a superobserver if its report has been validated
- *validSob*: A crisis can only be assigned to a superobserver that is capable to handle it

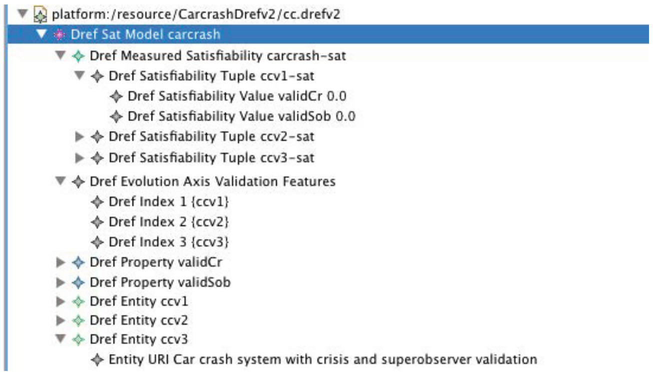
and that have a formal specification in the AlPiNA property language.

The system evolves over an evolution axis where guards are added with each version to improve property satisfaction. The first version, *ccv*1, has no guards; *ccv*2 has a guard ensuring the satisfaction of *validCr*; *ccv*3 has guards for both properties. There are thus three index points on the evolution axis, corresponding respectively to *ccv*1, *ccv*2 and *ccv*3.

The properties being boolean expressions, their satisfiability is a boolean function $sat : Ent \times Prop \to \{0, 1\}$ that was evaluated using the AlPiNA model checker. Figure 5 shows the satisfiability values calculated by AlPiNA for the three versions.

6.2 DREF Model without Composition (Stand-Alone)

Using EMF to create a stand-alone DREF specification for the car crash system is straightforward. As we said, no intervention on the language metamodels is needed. The APN models can be created using the AlPiNA built-in editor. For the DREF specification, EMF can generate an editor from the DREF metamodel. Using this editor, we could create a DREF specification that references the entities and properties of the APN models, indexes them on an evolution axis,

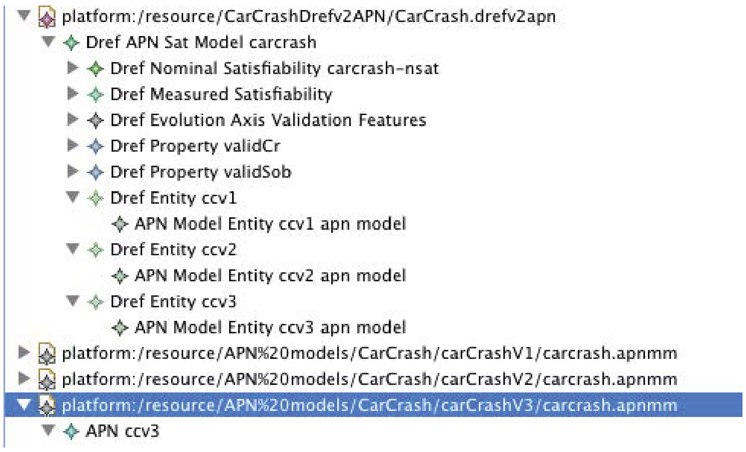


```
▼ platform:/resource/CarcrashDrefv2/cc.drefv2
  ▼ Dref Sat Model carcrash
    ▼ Dref Measured Satisfiability carcrash-sat
      ▼ Dref Satisfiability Tuple ccv1-sat
          Dref Satisfiability Value validCr 0.0
          Dref Satisfiability Value validSob 0.0
      ▶ Dref Satisfiability Tuple ccv2-sat
      ▶ Dref Satisfiability Tuple ccv3-sat
    ▼ Dref Evolution Axis Validation Features
        Dref Index 1 {ccv1}
        Dref Index 2 {ccv2}
        Dref Index 3 {ccv3}
    ▶ Dref Property validCr
    ▶ Dref Property validSob
    ▶ Dref Entity ccv1
    ▶ Dref Entity ccv2
    ▼ Dref Entity ccv3
        Entity URI Car crash system with crisis and superobserver validation
```

Fig. 6. Car Crash DREF satisfiability model in Eclipse, using only the DREF metamodel (no composition)

and associates them to their satisfiability values. Figure 6 shows a screenshot of the DREF editor. It is possible to see the entities definition (at the bottom), their indexing (in the middle) and some of the satisfiability values (at the top). Clicking on the Entity URI instances reveals the URI of the corresponding file.

The result of this approach is a set of APN files containing the models and the properties, and a DREF file containing the resilience specification. The association between the two is not immediately apparent, relying on the Entity URI attributes. The compatibility of the APN models with AlPiNA is full (they are original AlPiNA models).



```
▼ platform:/resource/CarCrashDrefv2APN/CarCrash.drefv2apn
  ▼ Dref APN Sat Model carcrash
    ▶ Dref Nominal Satisfiability carcrash-nsat
    ▶ Dref Measured Satisfiability
    ▶ Dref Evolution Axis Validation Features
    ▶ Dref Property validCr
    ▶ Dref Property validSob
    ▼ Dref Entity ccv1
        APN Model Entity ccv1 apn model
    ▼ Dref Entity ccv2
        APN Model Entity ccv2 apn model
    ▼ Dref Entity ccv3
        APN Model Entity ccv3 apn model
▶ platform:/resource/APN%20models/CarCrash/carCrashV1/carcrash.apnmm
▶ platform:/resource/APN%20models/CarCrash/carCrashV2/carcrash.apnmm
▼ platform:/resource/APN%20models/CarCrash/carCrashV3/carcrash.apnmm
  ▼ APN ccv3
```

Fig. 7. Car Crash DREF satisfiability model in Eclipse, using the DREFAPN$_r$ metamodel

6.3 DREF Model Composed with APNs through Reference

Figure 7 shows the same DREF model, but made using the DREFAPN$_r$ metamodel.

It is almost identical to the previous one, except for the fact that, instead of simply having a reference to a file, we have a reference to the actual APN model. At the bottom of the figure, the three APNs (previously created using the AlPiNA editor) are loaded in the model as an external resource.

The result of this approach is rather interesting, as the APNs continue to exist as separate files (thus keeping compatibility with the AlPiNA editor), but at the same time it is possible to load them together with the DREF specification in an integrated model. This editor is able to edit APN models, however it is still necessary to create them with AlPiNA first.

6.4 DREF Model Composed with APNs through Inheritance

Figure 8 shows a fragment of the editor obtained from the DREFAPN$_i$ metamodel. In this case, this editor is able to create both the DREF specification and the APN (APN editing is shown in the popup menu). We thus have a fully integrated language and editor; the result of this approach is a single model containing the APNs and the DREF specification. However, the resulting models are not compatible with AlPiNA out of the box. In principle this compatibility could be achieved through model transformations, however the added effort of writing the transformations would be considerable.

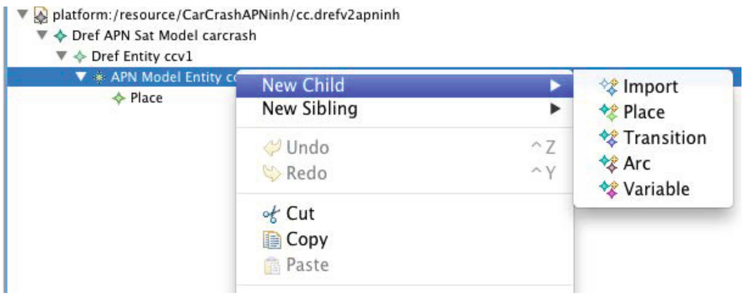


Fig. 8. Car Crash DREF satisfiability model in Eclipse, using the DREFAPN$_i$ metamodel

7 Conclusion

We have discussed a few techniques to enrich languages with resilience concepts by integrating them with DREF. Three approaches have been discussed and two of them have been experimented. We can draw the following conclusions and recommendations concerning the amount of effort, compatibility with preexisting tools, and the circumstances when the approach is appropriated.

The stand-alone approach (Section 4.1) does not require any metamodel editing effort, and can simply reference existing models using files or informal descriptions. It does not introduce compatibility issues, but requires effort in keeping specifications consistent. It is useful for occasional, non-systematic assessments of resilience.

The metamodel interfacing through reference approach (first part of Section 4.3) brings limited integration of DREF with other languages, by allowing a DREF model to reference actual models expressed in other languages, while maintaining separate models. This has the advantage of building a comprehensive model, keeping track of the association between the DREF specification and the entities and properties. At the same time it leaves untouched the entity/property metamodels, thus preserving compatibility with existing tools. It requires some effort in editing an interface metamodel. It is the best compromise when wanting to introduce resilience in a modeling chain without breaking the compatibility with existing toolkits.

The metamodel interfacing through inheritance approach (second part of Section 4.3) is an actual full integration of DREF with another language, where the constructs of the different metamodels coexist in the same space. This allows actual enrichment of a DSL with resilience constructs, at the cost of creating a metamodel which may be incompatible with previously existing tools. It is better suited when wanting to design a language with resilience support from scratch, or when a major language revision is foreseen anyway.

Finally, we think that the metamodel parameterization approach (Section 4.2) requires too much effort to be useful in this type of language composition, and is better suited to cases in which the other approaches fall short (i.e., when the nature of the composition presents a risk of conflicts and calls for a finer control over the composition semantics).

The metamodels discussed in this document are available for download at `http://wiki.lassy.uni.lu/@api/deki/files/499/=drefv2metamodels.zip`

Acknowledgement. This work has been supported by the National Research Fund, Luxembourg, project MOVERE C09/IS/02.

References

1. AspectJ team. The AspectJ project, `http://www.eclipse.org/aspectj/` (visited on May 9, 2012)
2. Atkinson, C., Kuhne, T.: Model-driven development: a metamodeling foundation. IEEE Software 20(5), 36–41 (2003)
3. Cuadrado, J., Molina, J.: A model-based approach to families of embedded domain-specific languages. IEEE Transactions on Software Engineering 35(6), 825–840 (2009)
4. Dinkelaker, T., Wende, C., Lochmann, H.: Implementing and Composing MDSD-Typical DSLs. Technical Report TUD-CS-2009-0156, Technische Universität Darmstadt (October 2009)

5. Emerson, M., Sztipanovits, J.: Techniques for Metamodel Composition. In: OOPSLA 6th Workshop on Domain Specific Modeling, pp. 123–139 (2006)
6. Estublier, J., Vega, G., Ionita, A.: Composing Domain-Specific Languages for Wide-Scope Software Engineering Applications. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 69–83. Springer, Heidelberg (2005)
7. Guelfi, N.: A formal framework for dependability and resilience from a software engineering perspective. Central European Journal of Computer Science 1, 294–328 (2011), doi:10.2478/s13537-011-0025-x
8. Hofer, C., Ostermann, K.: Modular domain-specific language components in scala. In: Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, pp. 83–92. ACM, New York (2010)
9. Hostettler, S., Marechal, A., Linard, A., Risoldi, M., Buchs, D.: High-Level Petri Net Model Checking with AlPiNA. Fundamenta Informaticae 113(3-4), 229–264 (2011)
10. Ionita, A.D., Estublier, J., Leveque, T., Nguyen, T.: Bi-dimensional composition with domain specific languages. e-Informatica Software Engineering Journal 3(1) (2009)
11. Khan, Y.: A formal approach for engineering resilient car crash management system. Technical Report TR-LASSY-12-05, University of Luxembourg (2012)
12. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
13. Laprie, J.-C.: From dependability to resilience. In: Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), page Fast Abstracts, Anchorage, USA (2008)
14. Ludewig, J.: Models in software engineering – an introduction. Software and Systems Modeling 2, 5–14 (2003), doi:10.1007/s10270-003-0020-3
15. Pedro, L.: A Systematic Language Engineering Approach for Prototyping Domain Specific Languages. PhD thesis, Université de Genève, Thesis # 4068 (2009)
16. Risoldi, M.: A metamodel for a DREF DSL. Technical Report TR-LASSY-12-03, University of Luxembourg (2012), `http://wiki.lassy.uni.lu/Special:LassyBibDownload?id=3169`
17. The Eclipse Foundation. The Eclipse Modeling Framework Project (2012), `http://www.eclipse.org/modeling/emf/` (visited on May 16, 2012)
18. Voelter, M.: A family of languages for architecture description. In: 8th OOPSLA Workshop on Domain-Specific Modeling, DSM 2008 (2008)
19. White, J., Hill, J.H., Gray, J., Tambe, S., Gokhale, A.S., Schmidt, D.C.: Improving domain-specific language reuse with software product line techniques. IEEE Softw. 26(4), 47–53 (2009)