2008 International Conference on Software Testing, Verification, and Validation

Detecting and Repairing Inconsistencies Across Heterogeneous Models

Dimitrios S. Kolovos, Richard F. Paige and Fiona A.C. Polack
Department of Computer Science
The University of York
YO10 5DD, York, UK
{dkolovos,paige,fiona}@cs.york.ac.uk

Abstract

With the advent of Domain Specific Languages for Model Engineering, detecting inconsistencies between models is becoming increasingly challenging. Nowadays, it is not uncommon for models participating in the same development process to be captured using different modelling languages and even different modelling technologies. We present a classification of the types of relationships that can arise between models participating in a software development process and outline the types of inconsistencies each relationship can suffer from. From this classification we identify a set of requirements for a generic inconsistency detection and reconciliation mechanism and use a case study to demonstrate how those requirements are implemented in the Epsilon Validation Language (EVL), a task-specific language developed in the context of the Epsilon GMT component.

1 Introduction

A Model Driven Development (MDD) process typically involves maintaining and managing a number of models that represent different, but often overlapping, aspects of the system. With the advent of Domain Specific Languages, it is not uncommon for models participating in the same process to be captured using different modelling languages and even diverse technologies (e.g. MOF [21], EMF [13], MS-DSL [3]). Moreover, each model typically exists in more than one versions, reflecting different levels of refinement or evolution.

The existence of multiple models that are maintained manually, and usually by different actors, increases the risk of inter-model inconsistency [6]. While internal model consistency management is a subject that has been extensively studied, little work has been done on automating consistency checking across different models, particularly when they are expressed using different modelling languages. In the absence of efficient automated mechanisms, significant effort needs to be spent to ensure that models do not become unsynchronized and contradictory.

In this paper we attempt to establish a classification of the different types of relationships that exist between models participating in an MDD process and identify the types of inconsistency each relationship typically suffers from. The aim of this classification is to enable us to elicit requirements for a general-purpose inter-model inconsistency detection and reconciliation mechanism. We present a prototype of such a mechanism in the form of a task-specific model management language, the Epsilon Validation Language (EVL) [10], developed in the context of the Epsilon component [2] of the Eclipse GMT project [1], that enables model engineers to express and evaluate constraints across models captured using different languages and modelling technologies.

The rest of the paper is organized as follows. In Section 2 we present a classification of model relationships and outline the inconsistencies each one is prone to. In Section 3 we summarize the techniques proposed for managing inter-model inconsistency, highlight their advantages and shortcomings, and propose a set of desirable characteristics for a generic inter-model inconsistency detection and repairing mechanism. In Section 4 we provide a brief discussion on EVL and Epsilon. In Section 5 we illustrate a case study in which we use EVL to detect and repair occurrences of contradiction and incompleteness between models expressed using two different modelling languages. In Section 6 we discuss on related work and in Section 7 we conclude and provide directions to further work on the subject.

978-0-7695-3127-4/08 $25.00 © 2008 IEEE
DOI 10.1109/ICST.2008.23

2 A Classification of Inter-Model Inconsistency

Depending on the conceptual relationships that exist between two models, different types of inconsistency can arise. In this section we attempt to establish an approximate classification of the types of relationship models can have with each other. Initially we present the different conceptual relationships and then discuss different approaches for establishing these relationships between particular model elements of the models involved. As well, we describe the additional types of inconsistencies each approach may introduce.

In alignment with the basic Model Engineering principle that *everything is a model* [17], we make no distinction between models expressed using traditional modelling languages (e.g. UML) and programs expressed using programming languages (e.g. Java). As elaborated in [5] both types of artefacts can be viewed and managed as *models*, each conforming to the abstract syntax of its own language.

2.1 Conceptual Relationships

In this section we focus on the types of high-level relationships models may be involved in without taking into consideration the techniques with which those relationships are implemented (in terms of their abstract or concrete syntaxes).

With the exception of the *Aspects Of* and *Weaves* relationships which as discussed in the sequel involve at least three models, all other relationships are binary but non-exclusive relationships. To reduce repetition, we refer to two exemplar models participating in binary relationships as A and B that conform to the MMA and MMB metamodels respectively.

2.1.1 Uses

This is a relationship where model elements from A refer to model elements of B. For instance A can be a model of a system and B a model of an external library that A uses. A and B do not need to conform to the same metamodel. For example, methods of a Java program can use methods defined in a C library.

2.1.2 Extends

This is a relationship where model elements of A extend model elements of B. Extension implies that A and B conform to the same metamodel or the metamodel of A is an extension of the metamodel of B. For example, a core UML model can be extended by another UML model that contributes a number of new classes that provide additional functionality to the modelled system.

2.1.3 Refines

In this case model B refines A by capturing more concrete design decisions than A. A and B need not be expressed in the same language. For example, a UML design is often refined by a Java program. Refinement has been extensively studied in the formal methods community [18, 23].

2.1.4 Complements

In this case A and B describe two different views of the same system and possibly include little overlap with each other. A and B need not be expressed in the same language. An example of this case could involve two models that describe the functions of two different departments of the same organization. While the two models are not expected to overlap significantly - as they don't describe the same department - they may both contain overlapping information when modelling close to the common boundaries of their scope (e.g. relationships with upper management).

2.1.5 Alternative For

In this case A and B describe two largely overlapping views of the system. For example, a model describing the structure of the object model of a system and a model capturing the structure of the database that the system uses to persist its data are significantly overlapping. This type of relationship can be introduced by an earlier refinement activity after which the two models have not been actively maintained in a synchronous mode or by two different groups of individuals describing the same system.

2.1.6 Aspect Of

In this case model B captures additional information related to elements of model A. Maintaining such information in an aspect model (B) instead of capturing it directly inside the core model (A) can be decided either because the metamodel of A does not provide the necessary slots for capturing such information, or to achieve a better separation of concerns and maintain A clean from non-essential information. For example, model A may be capturing the steps of a process while model B captures performance-related information about each process step.

2.1.7 Aspects Of

In this case A and B have an *Aspect Of* relationship with a third model (C). The reason why we choose to add this type of relationship is that while A and B may be consistent with C when examined separately, they may contradict each other (e.g. specify contradicting behaviour) when viewed as a whole.

2.1.8 Weaves

In this case a separate model, which is called the *weaving model* [16], is used to link a number of otherwise unrelated models. For example, a weaving model can capture correspondences between elements of two models so that they can later on be used to perform model merging.

2.2 Linking Approaches

To realize the relationships discussed above, model elements belonging to different models have to be physically linked with each other in a way that enables navigation among them. In this section we discuss two alternative approaches for establishing links between model elements.

2.2.1 Explicit Linking

In this case, model elements of A are directly linked to model elements of B using a unique identification scheme provided by the underlying modelling technology/platform (e.g. XMI identities for EMF [13] and MDR [24]) that enables seamless navigation between elements belonging to different models.

2.2.2 Implicit Linking

In this case, elements from A refer to elements of B by using informal identifiers. The difference of this approach to the previous is that implicit knowledge is required to resolve the target elements such identifiers point at. For instance, naming a class *Invoice* and a table *T_Invoice* demonstrates linking between the class and the table, which however requires additional knowledge (i.e. that classes are linked to tables with the specific naming convention).

2.3 Inconsistency Classification

Having identified a rough classification for model relationship types, we now categorize the inconsistencies that can arise between models and provide typical examples for each category. Then, in Table 1 we provide a summary of the types of relationships that can trigger each type of inconsistency.

2.3.1 Unresolvable References

This type of inconsistency occurs when references - either explicit or implicit - cannot be resolved to actual elements in the target model. Broken explicit references are relatively easy to capture as they are based on the unique identifier scheme discussed in section 2.2. Implicit references on the other hand are much more challenging to identify as they typically involve a more complex - and often ad-hoc - resolution algorithm.

2.3.2 Incompleteness

Incompleteness arises in a number of cases. For example, when annotating a process model with performance-related information using an aspect model, it may be required that the aspect model defines the performance of each sub-processes of the process model. Failure to capture performance information for some of the sub-processes results to incompleteness. In another example from the *Alternative For* relationship, to persist an object model into a database, it is essential that each Class of the object model corresponds to a table in the database model. While there may be tables to which no class corresponds (e.g. system tables), the reverse is not acceptable.

However, as discussed in [19], incompleteness does not always imply inconsistency; instead it may represent *uncertainty*. In this case, the models are not complete, not due to unintentional omissions but because at the time there is not enough information on decisions to be made for the missing aspects.

2.3.3 Contradiction

In this type of inconsistency, the related models capture contradicting decisions about the same aspect of the modelled system. For example, an abstract model may specify that an operation only applies to positive integers while its refinement specifies the exact opposite.

2.3.4 Misuse

Misuse arises in the case a model uses or extends some other model in non-intended/valid ways. For example, a Java program may use an existing C method but invoke it with the wrong type of parameters.

2.3.5 Redundancy

Although it is arguable that redundancy is a form of inconsistency itself, when unintended it can easily lead to inconsistencies when one of the two models is updated [26].

3 Approaches to Managing Inter-Model Inconsistency

Two different approaches are generally recognized for achieving inter-model consistency: constructive and analytical [20].

Constructive approaches maintain consistency between models by actively modifying the involved models so that they remain consistent with each other. A typical constructive approach is model transformation where a set of input models are transformed into a set of consistent output models using well-defined mapping rules specified in languages

	Uses	Extends	Refines	Complements	Alternatives	AspectOf	AspectsOf	Weaves
Unresolvable	x	x				x	x	x
Incompleteness		x	x		x	x		x
Contradiction		x	x	x	x		x	
Misuse	x							
Redundancy				x			x	

Table 1. Relationship-Inconsistency Types Matrix

such as QVT [4] or ATL [14]. However, constructive approaches are not always applicable. While they apply to certain kinds of model relationships such as *refines*, and *is alternative for*, they do not apply to others such as *uses*, *aspect of*, *aspects of* in which the models involved already exist. Moreover, even for the relationships to which they apply, after the initial transformation the target models often follow their own lifecycle and undergo changes which render re-derivation from their source models impractical - if possible at all.

For the types of relationships to which constructive approaches are not applicable, analytical approaches need to be employed. Unlike constructive approaches, analytical approaches do not modify the models involved; instead they only reveal inconsistencies that users need to address in order to maintain the models consistent among each other. While in the related field of intra-model consistency OCL [22] is the indisputable standard, it cannot be used for checking consistency across several models since by design OCL is limited to navigating only one model at a time [12].

3.1 Requirements for an Inter-Model Inconsistency Detection Mechanism

The rough classification of types of relationships between models has demonstrated that there are cases where more than one models, potentially defined using several modelling languages, can be involved in a relationship that involves inconsistencies of several types. Also, while explicit linking implies the use of a common modelling technology that implements the identifier resolution scheme, with implicit linking it is possible to establish relationships between models of different modelling technologies as well.

Therefore, it is essential for a generic inter-model consistency detection mechanism to be able to navigate and query multiple models of potentially diverse metamodels and modelling technologies within the same context.

4 The Epsilon Validation Language

As discussed earlier, the powerful model querying and navigation mechanisms OCL provides are also much desirable in the context of cross-model inconsistency detection;

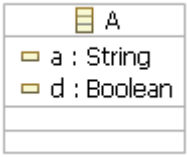

Figure 1. The MMa metamodel

however OCL is limited by design to capturing constraints in the context of a single model. This, and other, limitations of OCL have motivated us to develop the Epsilon Object Language (EOL) [12], which extends the OCL expression sub-language by adding a number of features such as multiple model access, statement sequencing, model modification capabilities and conventional programming constructs such as looping and branching statements. While EOL can be used in a standalone fashion, its purpose is to be used as a core language atop which task-specific languages can be built. Currently, six languages for tasks such as transformation, validation, merging, comparison and text-generation have been built atop EOL.

In the context of this paper, we focus on the Epsilon Validation Language (EVL) [10], which addresses the problem of model validation. As EVL is based on EOL, it inherits the capability to navigate - and thus express constraints - over multiple models concurrently. In this section we provide a brief overview of the syntax of EVL using a simple yet motivating example. For our example we use the two simple metamodels displayed in Figures 1 and 2 respectively. Our aim is to check that for two models *Ma* and *Mb* that conform to *MMa* and *MMb* respectively, for each *a*, instance of *A* in *Ma* that has its *d* attribute set to true, there is at least one *B* or *C* in *Mb* such that their *b* or *c* property (respectively) equals *a.a*. The constraint is artificially complicated to demonstrate that cross-model constraints are not only about simple feature based mappings but can - and in our experience often do - become so complex that they requires a fully-fledged expression language to capture effectively. Listing 1 demonstrates an EVL module that implements the aforementioned constraint.

Figure 2. The MMb metamodel

Listing 1. Exemplar EVL cross-model constraint

```
1  context Ma!A {
2
3    constraint ExistsBorC {
4
5      guard : self.d = true
6
7      check : Mb!B.allInstances.exists(b|b.b = self.a)
8        or Mb!C.allInstances.exists(c|c.c = self.a)
9
10     message : 'No equivalent B or C exists for '
11       + self.a
12
13     fix DeleteA {
14       title : 'Delete ' + self.a
15       do {
16         delete self;
17       }
18     }
19
20     fix CreateB {
21       title : 'Create a B'
22       do {
23         var b : new Mb!B;
24         b.b := self.a;
25       }
26     }
27
28     fix CreateC {
29       title : 'Create a C'
30       do {
31         var c : new Mb!C;
32         c.c := self.a;
33       }
34     }
35   }
36 }
```

Line 1 defines the context (*Ma!A*) in which the contained constraint is specified; all the elements of type *A* that belong to the model *Ma*. Line 3 defines the actual constraint which has three parts. In Line 5 the *guard* of the constraint limits its applicability to only those instances of *A* that have their *d* property set to *true*. In Line 7 the *check* part of the constraint iterates all the instances of B and C in Mb (Mb!B and Mb!C respectively) and checks that there is at least one that satisfies the constraint. If the constraint fails for some instance of A, the *message* part is evaluated that provides a human-understandable description of the error.

A distinctive feature of EVL, is its ability to capture and execute user-defined *fixes* that can also rectify an identified inconsistency. In this example, we specify that three possible solutions are to delete the element from Ma (line 13), or create the required instances of B (line 20) or C (line 28) in Mb.

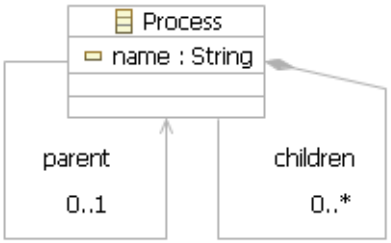

Figure 3. The ProcessLang Metamodel

5 Case Study

In this case study we demonstrate using EVL to detect and repair occurrences of incompleteness and contradiction in an *AspectOf* relationship between two models. In our example we use the simplified *ProcessLang* metamodel, which captures information about hierarchical processes. To add performance information in a separate aspect we also define the *ProcessPerformanceLang* metamodel. The two metamodels are displayed in Figures 3 and 4 respectively.

There are two constraints that we need to check in this example: that each *Process* in a process model (*PM*) has a corresponding *ProcessPerformance* in the process performance model (*PPM*), and that the *maxAcceptableTime* of a process does not exceed the sum of the *maxAcceptableTimes* of its children. We achieve this with the *PerformanceIsDefined* and the *PerformanceIsValid* EVL constraints displayed in Listing 2.

Listing 2. Exemplar EVL module containing a cross-model constraint

```
1  context PM!Process {
2
3    constraint PerformanceIsDefined {
4
5      check {
6        var processPerformances :=
7          PPM!ProcessPerformance.
8          allInstances.select(pt|pt.process = self);
9
10       return processPerformances.size() = 1;
11     }
12
13     message {
```

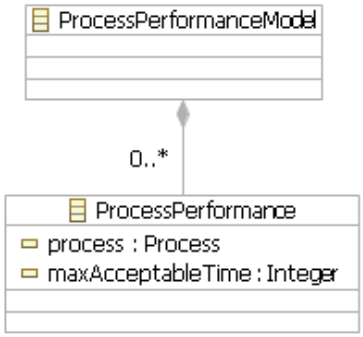

Figure 4. The ProcessPerformanceLang Metamodel

```
14      var prefix : String;
15      if (processPerformances.size() = 1) {
16        prefix := 'More than one performance info';
17      }
18      else {
19        prefix := 'No performance info';
20      }
21      return prefix + ' found for process '
22        + self.name;
23    }
24
25    fix {
26      title : 'Set the performance of ' + self.name
27
28      do {
29        for (p in processPerformances.clone()) {
30          delete p;
31        }
32        var maxAcceptableTime : Integer;
33        maxAcceptableTime := UserInput.
34          promptInteger('maxAcceptableTime', 0);
35        var p :
36          new PPM!ProcessPerformance;
37        p.maxAcceptableTime := maxAcceptableTime;
38        p.process := self;
39      }
40    }
41  }
42
43  constraint PerformanceIsValid {
44
45    guard : self.satisfies('PerformanceIsDefined')
46      and self.children.forAll
47        (c|c.satisfies('PerformanceIsDefined'))
48
49    check {
50      var sum : Integer;
51      sum := self.children.
52        collect(c|c.getMaxAcceptableTime())
53        .sum().asInteger();
54      return self.getMaxAcceptableTime() >= sum;
55    }
56
57    message : 'Process ' + self.name +
58      ' has a smaller maxAcceptableTime '
59        + 'than the sum of its children'
60
61    fix {
62      title : 'Increase maxAcceptableTime to ' + sum
63      do {
64        self.setMaxAcceptableTime(sum);
65      }
66    }
67
68  }
69
70  }
71
72  operation PM!Process getMaxAcceptableTime()
73    : Integer {
74    return PPM!ProcessPerformance.
75      allInstances.selectOne(pt|pt.process=self)
76        .maxAcceptableTime;
77  }
78
79  operation PM!Process setMaxAcceptableTime
80    (time : Integer) {
81    PPM!ProcessPerformance.allInstances.
82      selectOne(pt|pt.process=self).maxAcceptableTime :=
83      time;
84  }
```

In line 5, the check part of the *PerformanceIsDefined* constraint calculates the instances of *ProcessPerformance* in the *ProcessPerformanceModel* that have their *process* reference set to the currently examined *Process* (accessible via the *self* built-in variable) and stores it in the *processPerformances* variable. If exactly one *ProcessPerformance* is defined for the *Process*, the constraint is satisfied. Otherwise, the *message* part of the constraint, in line 13, is evaluated and an appropriate error message is displayed to the user.

Note that the *processPerformances* variable defined in the *check* part is also used from within the *message* part of the constraint. As discussed in [10], EVL provides this feature to reduce the need for duplicate calculations, as our experience has shown that the message for a failed constraint often needs to utilize side-information collected in the *check* part.

To repair the inconsistency, the user can invoke the *fix* defined in line 25 that will delete any existing *ProcessPerformance* instances and create a new one with a user-defined *maxAcceptableTime* obtained using the *UserInput.promptInteger()* statement of line 33.

Unlike the *PerformanceIsDefined* constraint, the *PerformanceIsValid* constraint, line 43, also defines a *guard* part (line 45). As discussed in [10], the guard part of a constraint is used to further limit the applicability of the constraint beyond the simple type check performed in the containing *context*. In this rule, we need to check the validity of the *maxAcceptableTime* of a *Process* only if one has been defined in the *ProcessPerformanceModel*. Therefore, in the guard part of the constraint we specify that this constraint is only applicable to *Processes* which, both they and they

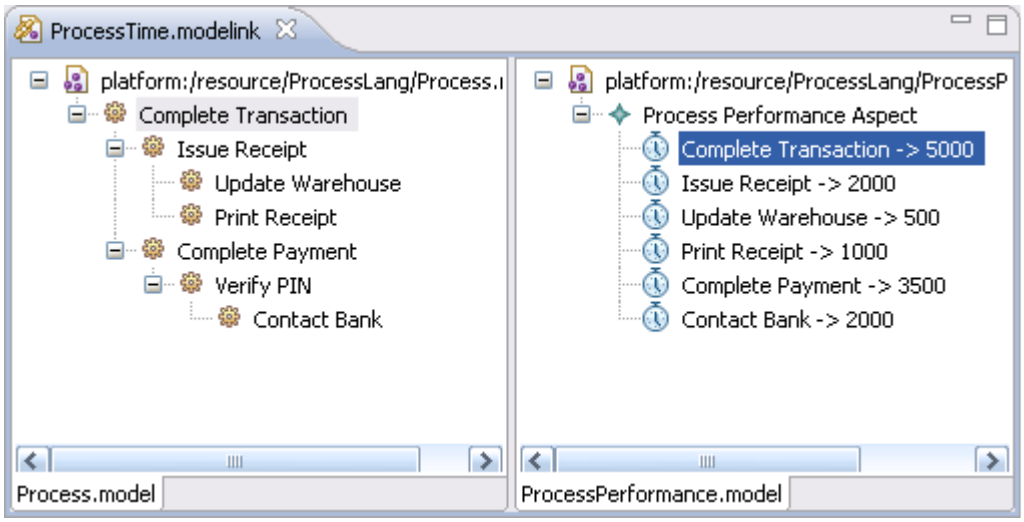

Figure 5. Exemplar Process and ProcessPerformance models

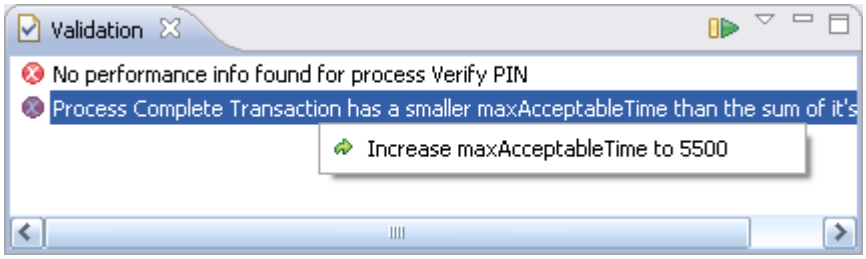

Figure 6. Screenshot of the validation view reporting the identified inconsistencies

children, satisfy the *PerformanceIsDefined* constraint.

The check part of the constraint retrieves the *maxAcceptableTime* of the process and that of its children and compares them. As the *Process* itself does not define performance information, retrieval of the value of the *maxAcceptableTime* of the respective *ProcessPerformance* object is implemented using the user-defined *getMaxAcceptableTime()* operation that is defined in line 72. In case the constraint is not satisfied, the user can invoke the *fix* defined in line 61 to repair the inconsistency by setting the *maxAcceptableTime* of the process to the *sum* calculated in line 51

To demonstrate the evaluation of these constraints we use two exemplar models that conform to the *ProcessLang* and *ProcessPerformanceLang* metamodels. A visual representation of the models is displayed in Figure 5. Evaluating the constraints in the context of those two models reveals two problems which are reported to the user via the view displayed in Figure 6. Indeed by examining the two models of Figure 5, we see that there is no ProcessPerformance linked to the *Verify PIN* process and also that the *maxAcceptableTime* of *Complete Transaction* (5000) is less than the sum of the *maxAcceptableTimes* of its children (2000 + 3500).

6 Related Work

In [25] work on checking consistency between UML models using description logic is discussed. In our view, there are two shortcomings of this approach. First it applies only to UML, which can prove restrictive in case non-UML models exist in the context of the process. Moreover, description logic languages such as *Loom* and *Racer* used in that work are sometimes seen as challenging to use by contemporary developers with an object-oriented background.

Xlinkit [7, 8] is a generic distributed XML document consistency checking toolkit. In [9] Xlinkit is used to perform inter-consistency checking between UML models exploiting the fact that UML models are eventually stored as XML documents following the XMI format. While this approach makes inter-model consistency checking feasible for other types of models (non UML), we argue that the concrete syntax (XML) is an inappropriately low level of abstraction on which meaningful constraints should be defined (especially for complex models).

In [15], the ATL model transformation language [14] is used to detect inconsistencies in a single model by transforming it into a *problem* model. Since ATL supports mul-

tiple model access from the context of the same transformation, the same technique can also be used to detect inconsistencies across different models. However, this approach does not support reconciling inconsistencies and also it is less well-suited to the task compared to a task-specific language such as EVL.

7 Conclusions and Further Work

In this paper we have attempted to classify the types of relationships that can appear between models participating in a Model Driven Development process and the types of inconsistencies each relationship typically suffers from. The result of our study revealed that to manage all those different types of inconsistency a mechanism that provides access to models of diverse metamodels and a powerful query and navigation language is required. Through a case study we have demonstrated that the Epsilon Validation Language possesses those characteristics and can thus be used to discover and repair diverse types of inconsistencies between models of diverse metamodels and technologies.

We recognize that providing the user of a constraint language, such as EVL, with the expressive power of an imperative language, such as EOL, has both benefits and drawbacks. As seen in the case study, using the imperative features of EOL complex statements can be decomposed into sequences of simpler statements and user-defined operations that make the constraints more understandable and maintainable. On the other hand, nothing prevents the engineer from modifying the involved models from within the context of a *guard* or a *check* part of a constraint (which is undesirable in most cases). Also, as fixes are user-defined it cannot be guaranteed by the execution engine that they do repair the inconsistency they are supposed to address and thus the developer is responsible for ensuring this. Our experiences from using both EVL and the side-effect free, purely declarative OCL to define the same constraints in a case study presented in [10], have shown that the benefits of EVL outweigh the aforementioned weaknesses.

With regard to further work, through our experimentation, we have discovered that there are cases where it is useful to establish a preliminary match trace linking elements that should be checked against each other in a separate step before evaluating the validation specification. We have observed that there are cases - particularly where homogeneous models are involved - where such a tactic reduces the amount of validation-specific code and also speeds up the execution time. We are currently experimenting with using the Epsilon Comparison Language (ECL) [11] for this purpose and expect to report on this subject soon.

8 Acknowledgements

The work in this paper was supported by the European Commission via the MODELPLEX project, co-funded by the European Commission under the "Information Society Technologies" Sixth Framework Programme (2006-2009).

References

[1] Eclipse GMT - Generative Modeling Technology, Official Web-Site. http://www.eclipse.org/gmt.

[2] Extensible Platform for Specification of Integrated Languages for mOdel maNagement (Epsilon). http://www.eclipse.org/gmt/epsilon.

[3] Microsoft Domain Specific Languages Framework, Official Web-Site. http://msdn.microsoft.com/ vstudio/ teamsystem/ workshop/ DSLTools/ default.aspx.

[4] QVT Partners Official Web-Site. http://qvtp.org/.

[5] Anneke Kleppe. A Language Description is More than a Metamodel. In *Proc. 4th International Workshop on Software Language Engineering*, Nashville, USA, October 2007. (to appear).

[6] Anthony Finkelstein. A Foolish Consistency: Technical Challenges in Consistency Management. In *Proc. 11th International Conference on Database and Expert Systems Applications*, volume 1873 of *Lecture Notes In Computer Science*, pages 1 – 5, 2000.

[7] Christian Nentwich, Licia Capra, Wolfgang Emmerich and Anthony Finkelstein. xlinkit: A Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2(2):151–185, May 2002.

[8] Christian Nentwich, Wolfgang Emmerich, Anthony Finkelstein and Erns Ellmer. Flexible Consistency Checking. *ACM Transactions on Software Engineering and Methodology*, 12(1):28–63, 2003.

[9] Clare Gryce, Anthony Finkelstein, and Christian Nentwich. Lightweight Checking for UML Based Software Development. In *Workshop on Consisteny Problems in UML-based Software Development*, 2002.

[10] Dimitrios S. Kolovos, Richard F. Paige and Fiona A.C. Polack. On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages. In *Proc. Dagstuhl Workshop on Rigorous Methods for Software Construction and Analysis*, 2007.

[11] Dimitrios S. Kolovos, Richard F. Paige, Fiona A.C. Polack. Model Comparison: A Foundation for Model Composition and Model Transformation Testing. In *Proc. 1st International Workshop on Global Integrated Model Management (GaMMa), ACM/IEEE ICSE 2006*, pages 13 – 20, Shanghai, China, 2006. ACM Press.

[12] Dimitrios S. Kolovos, Richard F.Paige and Fiona A.C. Polack. The Epsilon Object Language. In *Proc. European Conference in Model Driven Architecture (EC-MDA) 2006*, volume 4066 of *LNCS*, pages 128–142, Bilbao, Spain, July 2006.

[13] Eclipse.org. Eclipse Modelling Framework. http://www.eclipse.org/emf.

[14] Frédéric Jouault and Ivan Kurtev. Transforming Models with the ATL. In Jean-Michel Bruel, editor, *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, volume 3844 of *LNCS*, pages 128–138, Montego Bay, Jamaica, October 2005.

[15] Frédéric Jouault, Jean Bezívin. Using ATL for Checking Models. In *Proc. International Workshop on Graph and Model Transformation (GraMoT)*, Tallinn, Estonia, September 2005.

[16] Jean Bezevin, Frederic Jouault and Patrick Valduriez. First Experiments with a ModelWeaver.

[17] Jean Bezivin. On the Unification Power of Models. *Software and System Modeling (SoSym)*, 4(2):171–188, 2005.

[18] Jim Woodcock and Jim Davies. *Using Z : Specification, Refinement, and Proof*. Prentice Hall, March 1996.

[19] Marsha Chechik, Benet Devereux, Steve Easterbrook. Implementing a Multi-valued Symbolic Model Checker. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001*, LNCS, pages 404–419, Genova, Italy, April 2001.

[20] Monique Snoeck, Cindy Michiels and Guido Dedene. Consistency by Construction: The Case of MERODE. In *International Workshop on Conceptual Modeling Quality*, 2003.

[21] Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification. http://www.omg.org/cgi-bin/doc?ptc/03-10-04.

[22] Object Management Group. UML 2.0 OCL Specification. http://www.omg.org/docs/ptc/03-10-14.pdf.

[23] Samir Chouali, Maritta Heisel, Jeanine Souquiéres. Proving Component Interoperability with B Refinement. Technical report, LORIA, April 2005. http://hal.inria.fr/inria-00000171/en/.

[24] Sun Microsystems. Meta Data Repository. http://mdr.netbeans.org.

[25] Tom Mens, Ragnhild Van Der Straeten, and Jocelyn Simmonds. Maintaining Consistency between UML Models with Description Logic Tools. In *Sixth International Conference on the Unified Modelling Language - the Language and its applications, Workshop on Consistency Problems in UML-based Software Development II*, 2003.

[26] WenQian Liu, Steve Easterbrook and John Mylopoulos. Rule-Based Detection of Inconsistency in UML models. In *Proc. Workshop on Consistency Problems in UML-Based Software Development*, Dresden, Germany, October 2002.