# Towards Problem Analysis at Runtime – A case of Model-Driven Software Development

Yijun Yu[1], Thein Than Tun[1], Arosha K. Bandara[1], Tian Zhang[3], and Bashar Nuseibeh[1,2]

[1]Department of Computing, The Open University, Milton Keynes, UK
[2]Lero, University of Limerick, Ireland
[3]State Key Laboratory for Novel Software Technology, Nanjing University, China

**Abstract.** Following the "convention over configuration" paradigm, model-driven software development (MDSD) generates code to implement the "default" behaviour that has been specified by a template separate from the input model, reducing the decision effort of developers. On the one hand, developers can produce end-products without a full understanding of the templates; on the other hand, the tacit knowledge in the templates is subtle to diagnose when a runtime software failure occurs. Using a concrete example, we discuss in this chapter the challenges of runtime problem analysis for MDSD, highlighting a potentially valuable research agenda for diagnosing the model-driven developed systems at runtime.

**Keywords** Model-Driven Software Development, Problem Frames

## 1 Introduction

Decades since Alan Turing introduced the computing machine that uses a tape of infinitely long '0' and '1' binary numbers to store data and programs [14], abstraction levels of programs become closer to human understanding of the physical world [9]. High-level programming languages can be automatically translated and optimised into Turing machines by compilers, freeing programmers from thinking in terms of machine instructions [1]. Naturally, one would like to model the physical world, from which code for implementing the machine can be generated, in the same automated way as what the compilers have achieved. This vision motivates model-driven software development methods (MDSD) [6], using an input model much more abstract than the binary code of Turing machines. Figure 1 contrasts Turing machine to the MDSD approaches [5] which attempt to bridge the abstraction gap between human understanding of the physical world to the machine implementation.

For example, our graphical modeling tool to support Problem Frames (PF) [8] was created using MDSD method, starting from a concise domain-specific language for representing or modeling problem diagrams. Given that diagrammatic notations of the PF have been unambiguously defined by researchers, and graphical editing is one of the exemplars of mature MDSD tools, one would assume that automating this creation a straightforward application of MDSD methods.

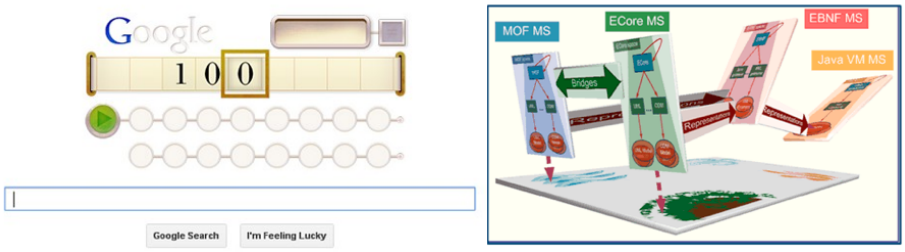



Fig. 1: A bridge between machine and world: (a) Turing machine marked for Alan's 100th years birth; (b) Megamodels excerpt from the Tao of MDSD [5].

However, this assumption needs to be checked, both from a requirements engineering (RE) perspective and from a pratical problem solving perspective. From a RE perspective, we need to analyse the requirements of "developing a graphical modeling tool support for Problem Frames approach", as an exercise of both MDSD and Problem Frames. This exercise serves two purposes. First, it tells whether MDSD directly meets the requirement of "supporting a graphical modeling language"; second, it tells how such MDSD requirements can be analysed by the PF approach. In doing so, we hope to discover a useful pattern in the problem solving practice that relates the MDSD solutions to the requirements. We also hope to improve our understanding in whether one needs to be aware of any generic concern in the MDSD methodology. From a practical perspective, we would like to explore problems that cannot be solved by the current practices of MDSD. If such problems exist, a new methodology needs to be followed by the practitioners to diagnose them.

To demonstrate, a chain of automated tool support from the Eclipse Modeling project[1] will be described as the background whilst discussing the terminology used.

### Background and terminology of MDSD

Many techniques have been proposed for MDSD. The general idea is to have one metametamodel (e.g., OMG MOF) whose instance is a metamodel or a modeling language. An instance of the metamodel is a program in a domain specific or generic language. For an Eclipse modeling project, the metametamodel is called *ecore*, a sublanguage to define metamodels in the XML interchange (XMI) format. Ecore itself is an instance of the ecore metamodel, which we call *self-defining*. In general, an instance of ecore is called *EMF* model, named after the de facto standard in the Eclipse modeling community. All these languages are supported by a chain of EMF tools[2].

Using an analogy to language engineering, EMF corresponds to the abstract syntax of the language without specifying its concrete syntax. The XMI is only

[1] www.eclipse.org/modeling
[2] www.eclipse.org/emf

one concrete syntax to represent EMF, one may also choose another concrete syntax such as a textual DSL language or a graphical language. Transformations can be written to convert text to model (T2M), model to model (M2M), and model to text (M2T), following a suite of OMG modeling standards. Because the Ecore modeling language is a generic implementation of the OMG MOF, diagrammatic languages such as UML can also be fully supported.

As an example, the `xtext` framework[3] is provided to perform the T2M parsing, converting the abstract syntax of a DSL program into its corresponding EMF model. As the by-product of such a transformation, a syntax-highlighting text editor can be generated for editing the DSL program instances. Similarly, `GMF` editors can also be generated for editing the EMF models graphically[4]. These feature-rich graphical editors can be generated from the EMF metamodel, the graph definition models that define the graphical notations, and the mappings between the elements on the Ecore to the presentations.

In a nutshell, generating a graphical editor in MDSD is now feasible by providing the language design in an abstract way using the extended BNF rules, plus the mapping decisions to show the modeling elements in appropriate graphical notations.

## 2 Describing PF modeling as a PF model

Before analysing the general problem, we first describe the requirements and the stakeholders involved in a specific example. In this example, our primary requirement is "a PF graphical modeling tool must allow users to create and edit problem diagrams as defined by the PF researchers". For the PF modeling tool to be developed, this requirement also involves stakeholders such as users who use the PF modeling tool and researchers who define the PF language.

To solve this problem without using the MDSD approach, a Model-View-Controller (MVC) design pattern or a Workpiece frame [8] can be used.

A Workpiece frame is a general class of problems identified by a requirement of users to edit a piece of work through a tool. Any editing problem fits this frame, with no exception to the PF Graphical Modeling Tool (see Figure 2).


Fig. 2: A Work Piece frame and its instantiation for the PF editing problem

[3] http://xtext.itemis.com
[4] www.eclipse.org/gmf

*Basic PF notations* A requirement is represented by a dashed ellipse shape, labelled by the name of the requirement and its abbreviation; and a solution to the problem is represented by a rectangle, marked with double strips on the left. When marked with a single strip on the left, the domain is "designed" by other problem solving steps. A physical domain can also be represented by a rectangle with names and abbreviation labels without the strips. The behaviour type of a domain node can be classified by a letter mark at the lower-right corner of the rectangle. For example, a lexical domain marked with "X" indicates a passive behaviour that does not cause change itself, a biddable domain marked with "B" indicates an active behaviour that can change by itself non-deterministically, a causal domain marked with "C" indicates an active behaviour that is deterministic. Domains can share an interface between each other. The shared interface is represented by an undirected solid link, marked with a letter abbreviating a set of shared phenomena such as events and states. A requirement can constrain a domain's behaviour, indicated by a dashed arrow to the constrained domains; a requirement can also refer to a domain, shown as a dashed link between them.

In fact, a textual or graphical editing tool may already meet this requirement. Most PF diagrams documented so far were specified using either a text editing tool such as LaTeX, or a diagramming tool such as Dia[5]. This fact generates an interesting question: "what can MDSD add to the available solutions for the PF modeling tool requirement" and "who can benefit from MDSD"?

Naturally such an investigation brings us to a new type of stakeholders – "Developer". In fact, a developer opts for the MDSD method mainly because it promises two more quality requirements: "productivity" and "maintainability": it must take little effort for a developer to create a PF modeling tool from scratch; and it must take little effort for a developer to adjust the tool when the researcher makes some refinement to the PF language.

Even with the productivity and maintainability requirements in mind, there is still one alternative meeting these requirements without resorting to the MDSD technology: to customise existing functionalities in graphical editing tool such as Visio, e.g., by creating a new stencil or template for PF notations. In fact, this is what the graphical drawing tool Dia already offered. So, why do we still bother with MDSD?

Let us revisit the initial requirement of the "Users" and the "Researchers". There is one additional requirement "modeling conformance" that a customised general diagramming tool cannot easily meet. "How can one be sure that the modeling elements are uniquely named? How can one check whether there is a single machine node and a single requirement node for the problem frame?" Questions could not stop here. "How can one make sure all the nodes are linked and all the links are connected to certain nodes? How can one make sure the dashed arrows are always from requirement nodes to the domain nodes?" In short, the key advantage of providing PF Modeling Tool through MDSD is the additional capability to satisfy these "domain-specific" modeling requirements. Syntax check aside, syntax highlighting, syntax-driven editing, auto completion,

[5] http://projects.gnome.org/dia/

pattern matching, transformations, and various form of inconsistency checks such as type checks and uniqueness checks, are amongst the various benefits a MDSD derived PF Modeling tool brings about, on top of the graphical editing features such as drag-and-dropping, zooming, panning, layouting, printing, etc. Instead of asking "why bother with MDSD", one would ask "why bother with implementing all these nice features yourself" instead.

In rather abstract terms, such an analysis is summarised by a PF diagram (see Figure 4).

Note that we had quite similar experience in creating other requirements modeling tool using the MDSD approach (e.g., OpenOME for i*). In the following, we discuss several example problems found in practice, during the development time of our research prototype.

## 3 Problems and concerns in MDSD

Given the analysis so far, we established how MDSD benefits the developers in creating and maintaining a PF modeling tool for the PF researchers and users alike.

### 3.1 The "additional alphabet" or "tacit knowledge" concern

Now we go a bit further to look at the dark corners, explaining some concern we experienced about the MDSD approach. A possibly shocking concern we documented here resembles the experience in several non-trivial instances. Therefore it is our belief that this may be a general concern for MDSD development.

The poor experience came from the attempt to stretch the tool to support analysing the requirements problems in two complementary modeling languages, namely PF and i* [16]. While the PF approach focuses on understanding the entailment relationship $W, S \vdash R$ between the requirements $R$, solutions $S$ and the world context domains $W$, the goal-oriented modeling approach focuses on understanding the relationships between the stakeholders (i.e., the "Who") and their intentional requirements (i.e., the "why"). Since their diagramming tools have been both developed using MDSD, we would consider a generalisation of the graphical modeling tool support.

The first attempt was to use the grammar "mixin" feature in `xtext`. By inheriting concrete syntax from both grammars of PF and i*, we obtained such a modeling language that can navigate between them: (1) a requirement node in PF could be expanded into a detailed i* diagram where the requirement is one of the goals; (2) an intention node in i* diagrams (goal, task, resource, softgoal) could be expanded into a PF diagram where the requirement corresponding to the expanded goal. After applying the `xtext` MDSD generation, we then obtained a text-based parser that can transform the concrete syntax into an abstract syntax expressed by the combined EMF model. As a result, the new EMF model was compliant to both the metamodel in PF and the metamodel in

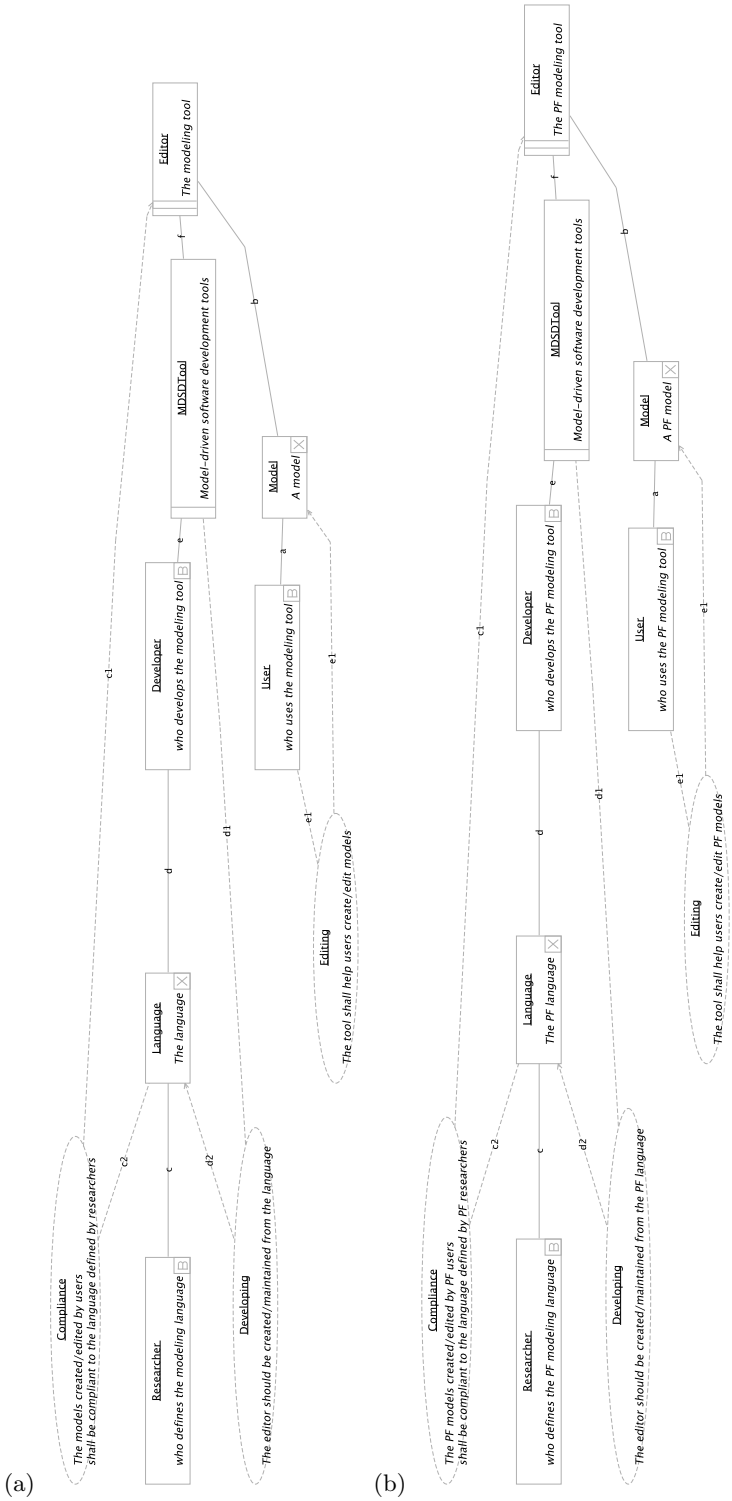



Fig. 3: A Model Driven Software Development frame (a) and its instantiations in PF modeling tools (b).

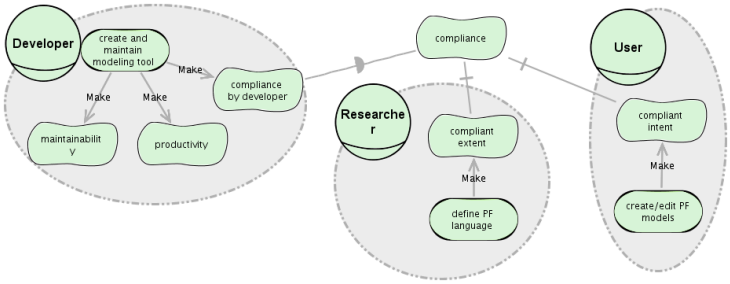



Fig. 4: Requirements of Model Driven Software Development in i*

i*, making it much easier to perform new kinds of analysis such as scoping the contexts of alternatively refined subgoals programmatically [2].

However, several subtle problems arose when the two Java code bases generated from the EMF models were used together, complicating the MDSD experience.

Figure 5 summarises the alphabet concern in the "convention over configuration" MDSD paradigm. By "convention", the template code is generated by instantiating the templates behind the scenes with the input model; by "configuration" users can further modify the generated code according to their individual requirements. The additional alphabet concern applies because neither does the designer of the templates understand the individual users' requirements, nor do the users could fully understand the rationale behind the "default" behaviours. When the two misunderstand each other, a glitch is inevitable. In the following subsections, we document four example problems that are caused by this kind of misunderstanding.

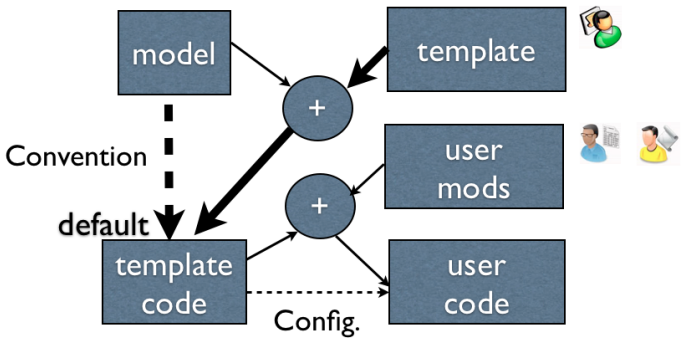



Fig. 5: The additional alphabet (or tacit knowledge) concern in the "convention over configuration" MDSD paradigm. Developers of the templates may not know the requirements of individual programmers, and the individual programmers may not fully understand the rationale behind the default behaviours in the generated template code.

### 3.2 The "detached" requirement problem

The first problem was related to an unwanted behaviour in the graphical editing. As described earlier, a requirement node in PF is an ellipse shape, which should connect to other domain nodes through links by the design of language. However, while moving such a node to an angle not aligned horizontal/vertically with the node on the other side of the link, the link would not be connecting to the requirement node physically, appearing as if it was detached. A search on the developers forum revealed that this problem was to do with the `org.eclipse.draw2d. ChopboxAnchor` class used by default in the generated code, rather than the proper `org.eclipse.draw2d.EllipseAnchor`. The ChopboxAnchor in effect calculates the connection anchors based on a rectangle shaped outline, whilst the EllipseAnchor class uses the ellipse shape instead. After replacing ChopboxAnchor with EllipseAnchor in the generated code, however, we found that the problem not solved. Trace the execution in a debugger, we found that the real problem was rooted deeply in the path resolution mechanism at the time of dynamic class loading. In fact, our customised `uk.ac.open.problem.diagram.edit.parts.NodeEditPart` class generated from the MDSD tool was never invoked. Instead, the GMF runtime system load a `org.eclipse.gef.NodeEditPart` class in the runtime class library of GEF framework. By removing such a "import" statement in the customised class, in effect we instructed the GMF editor to load our class instead, which solved the problem. However, when we did the same for the LinkEditPart class, the IDE automatically inserted the unwanted "import" statement back into the code. Ultimately, we had to explicitly coerce the class by casting the expression to the NodeEditPart class prefixed with our exact package name.

Figure 6 illustrates the "detached" requirement problem in details. First of all, (a) is observed to behave like a Chopbox with respect to the connections to the requirement node, this is highlighted as a "runtime abnormal behaviour". The method implementing this behaviour is all in the generated code. The arrows point backward along the chain of causality. First, the ChopboxAnchor was used in the generated method body, which implements a default behaviour. Furthermore, the parent class of the generated code is one of the predefined classes in the GMF runtime class library. Without changing that inheritance, the default behaviour cannot be overridden. Second, (b) is observed to behave normally, such that the connection to the requirement nodes are not clipped by the rectangle. The fixing changes required are (1) a customization of the method default implementation to switch the anchor class to ellipse shape if the node type is a requirement; (2) the generated import statements are removed manually, such that the ShapeEditPart class in the domain-specific package is to be used, overriding the default behaviour of the predefined GMF runtime class library.

We were wondering why a generated class name such as NodeEditPart clashes with the runtime library, only to realise that the MDSD tool itself had been developed using the MDSD approach. Their choice of using "Node" to name a

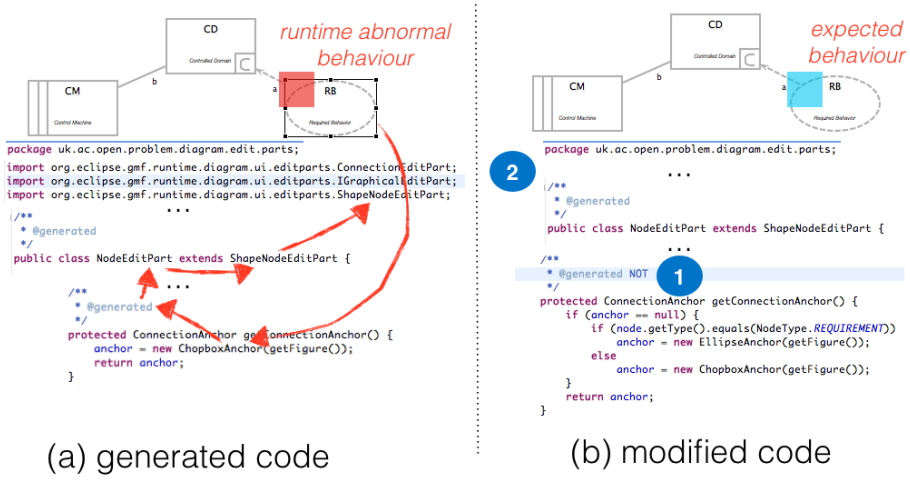



Fig. 6: Contrasting the observable problems and the code implementations respectively for the abnormal and correct behaviours.

class of nodes and using "Link" to name a class of links happened to be the same as ours. In other words, the clash was due to the common sense.

Thinking it twice, this incident could have revealed an interesting type of pitfalls in MDSD, which we called "model feature interaction" [13]. The design details abstracted away in the language specification could indeed be interacting with the generated code because they refer to the same name in different namespaces. The runtime class loader is not smart enough to distinguish them, thereby a sophisticated mechanism is needed to prevent this from happening again. For example, to avoid a developer using the names "Node/Link" when modeling the graphical language. If this is the case, the alphabet of the namespace must be restricted, leading to the following discussions.

In general, when abstracting away design details, the advantage must be revisited. First one needs to maintain the traceability between the abstract description and the concrete implementations, secondly one must be aware that the designer of the MDSD tools could have introduced some unwanted alphabets that may lead to unwanted behaviours when they are composed with the generated code. Their interpretations of the additional alphabet may not be the same as the original developers. This might have a strong implication to security problems, adding difficulty in maintaining and checking the traceability [19].

### 3.3 The manual refactoring problem

The second major problem we encountered could be a headache to other developers too. As we discussed earlier, it was fine when MDSD tool were applied separately to PF and i* languages. Each application generates a separate EMF metamodel in Ecore (Ecore is a self-defining metamodel). The PF ecore model

was newly "generated" from the concrete syntax in `xtext`, while the i* ecore model was imported from the existing release of OpenOME maintained at the University of Toronto. The generated classes for i* plugins were thereby prefixed by "edu.toronto.cs". The `xtext` tool could not know this, as a result of its code generation, no package prefix was added to the generated classes.

However, the combined metamodel needs to reference the i* classes in hundreds of places. For example, every time when a problem node is accessed, it could reference to an i* model element specified in the none-prefixed classes. A subtle but annoying behaviours was caused by this because the generated classes without prefixes were the skeleton code that should work if no customisation had been applied. However, developers at U of Toronto have made substantial improvements of almost every aspect in the graph editing tool. It is thereby necessary to switch to use the Toronto classes and keep their prefix. Instead of manually renaming all these places where the class names were referenced, we used automated refactoring for the name of generated plugin projects to reintroduce the missing prefix. After such refactorings, we still had to remove the refactored plugin projects such that at runtime the class loader would not get confused by the class paths to throw the ClassNotFound exceptions.

Automated refactoring on Eclipse project names using LTK could have been applied here [20], however, to accommodate every change in the PF language, one must specify which classes need to be renamed to which, and remember to manually change the references to the class names in the plugin specification too. Not a trivial task without further customising the automated refactoring tool.

### 3.4 The dependency injection problem

Instead of Aspect-Oriented Programming (AOP) [10], the designer of MDSD tool `xtext` uses the Dependency Injection pattern implemented by the Google Guice framework to inject functionalities at runtime. Similar to aspectJ, the new functionalities could be injected into the base system by specifying an adaptor class that uses the reflection mechanism of Java. Unlike aspectJ, the behaviour of the weaved system is somewhat controlled by the base system, in order to make the potential joinpoints explicit.

Ideally such technical details should be hidden from the developers who use MDSD because in principle one would not bother to know how it works if it works. However, one must be aware that the Guice framework assumes that the classes are singletons. If they share the same namespace, e.g., prefixed by the same package names while being located in different plugin projects scope, the dependency injection may still result in runtime conflicts.

As watchful observers for research problems, we are "lucky" enough to experience such a problem when developing the PF/i* integration tool. When we prefix our DSL language "Problem" and our adapted DSL language "Istar" with the same prefix "uk.ac.open", the generated code complains that the IDLink resolution class were not found even though it was present in the packages of the plugin component. After changing the prefix of one of these language into e.g.,

"uk.ac.open.problem", this conflict was resolved. A side effect was that we got package named by "uk.ac.open.problem.problem" due to the particular naming convention decided by the developer of the MDSD tool (i.e., `xtext`).

### 3.5 The synchronisation problems

When model and code co-evolve, they change concurrently. Since in MDSD, model and generated code are related by transformations, it is required to propagate changes from one end to the other.

To illustrate the problem concretely, we use a constructed example here. Suppose an EMF user initially specifies a simple model that consists of one `Entity` class with a single `name` attribute. Using the code generation feature of EMF, she will obtain a *default* implementation which consists of 8 compilation units in Java (Fig. 7).

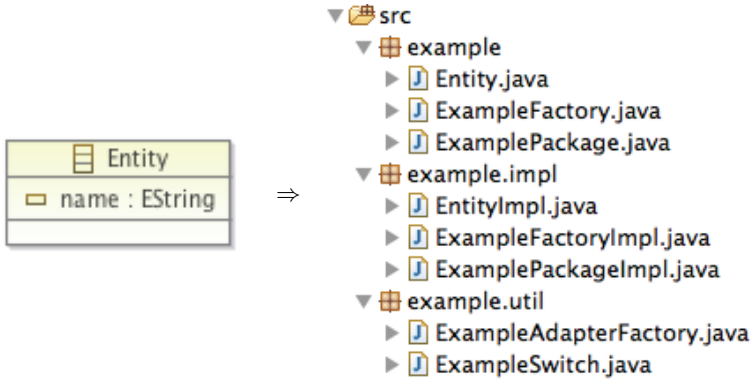



Fig. 7: Default code generated from the EMF meta-model

Fig. 8 lists parts of the generated code. The `Entity` Java interface has getter and setter methods for the `name` attribute. They are commented with `@generated` annotations which indicate that the methods are part of the default implementation. Similarly, such `@generated` annotations are added to every generated element in the code, e.g., shown in the skeleton of `EntityImpl` Java class.

The annotation `@generated` defines a *single-trip traceability contract* from the model to the annotated code element. A change in the model or a change in the modelling framework can be propagated to the generated code; however, a change in the generated code will not cause a change to the reflected model and will thus be discarded upon next code generation.

Because the default implementation is not always desired, the code generation shall keep user specified changes as long as they are not inside the range of generated traceability, the set of methods marked by `@generated` that keeps the changes of generated templates. This can be achieved by adapting the `@generated` annotation into `@generated NOT`, a non-binding traceability that

```java
package example;
import org.eclipse.emf.ecore.EObject;
/** @model */
public interface Entity extends EObject {
  /** @model */ public String getName();
  /** @generated */ void setName(String value);
}
```

```java
package example.impl;
import example.Entity;
...
/** @generated */
public class EntityImpl extends EObjectImpl implements Entity {
  ...
  /** @generated */
  protected String name = NAME_EDEFAULT;
  ...
  /** @generated */
  public String getName() { return name; }
  /** @generated */
  public void setName(String newName) { ... }
  ...
  /** @generated */
  @Override
  public String toString() {
    if (eIsProxy()) return super.toString();
    StringBuffer result = new StringBuffer(super.toString());
    result.append(" (name: ");
    result.append(name);
    result.append(')');
    return result.toString();
  }
} //EntityImpl
```

Fig. 8: Parts of the generated code in Fig. 7

reflects programmers' intention that it will not be changed when the implementation code is regenerated. Note that such non-binding traceability indicated by `@generated NOT` is still different from those without any annotation at all: Without such an annotation, EMF will generate new implementation of a method body following the templates.

This workaround does not work when a user parametrises the `toString()` method to append an additional `type` to the returned result. To guard the method from being overwritten by future code generations, the annotation `@generated NOT` is used. She also applies a *Rename Method* refactoring, changing the `getName` method into `getID`. The modified parts are shown in Fig. 9. Propagating these changes back to the model, the `name` attribute will be renamed into `iD` automatically, following the naming convention that attribute identifiers start with a lower case character.

Code regeneration results in the changes in Fig. 10: the setter methods and the implementations of both getter/setter methods are modified according to the default implementation of the new model. These are expected. However, two unexpected changes are not desirable. First, a compilation error results from the change in the default implementation, where the attribute `name` used in the user controlled code no longer exists. Second, the default implementation of the `toString()` method is generated with the original signature, which will of course become dead code since the user has already modified all call sites of `toString()` to reflect the insertion of the new type. Similarly, the user specified `toString()`

```
1  /** @model */
2  public interface Entity extends EObject {
3    /** @model */ public String getNameID();
4    /** @generated */ public void setName();
5  }
6  ...
7  /** @generated */
8  public class EntityImpl extends EObjectImpl implements Entity {
9    /** @generated */
10   public String getNameID() { return name; }
11   ...
12   /** @generated NOT*/
13   @Override
14   public String toString(String type) {
15     if (eIsProxy()) return super.toString();
16     StringBuffer result = new StringBuffer(super.toString());
17     result.append(" (name: ");
18     result.append(name);
19     result.append(')');
20     result.append(type);
21     return result.toString();
22   }
23 } //EntityImpl
```

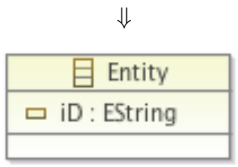



Fig. 9: User modifications to the generated code: insertions are underlined and the deletions are ~~stroked out~~; the changes are reflected

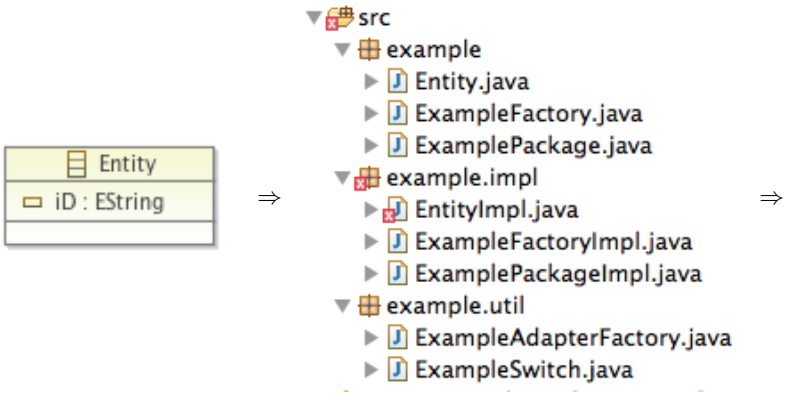
method can also become dead code, if it is no longer invoked by the new default implementation.

Compilation errors are relatively easy to spot by the programmer with the aid of the Eclipse IDE, but the dead code problems are more subtle because the IDE will not complain. Therefore, it will be more difficult for developers to notice the consequences.

In [17], we have developed a two-layered synchronisation framework, blinkit, to address this problem.

Figure 11 presents an overview of the framework when it is applied to the case study of EMF/GMF, where EMF is the synchronisation framework for vertical traceability and blinkit framework is the horizontal synchronisation counterpart. Examples in [17] indicate that when the complementary changes to templates and user-modified code are conflicting or redundant, our tool can avoid some dead code redundancies and raise some warnings as compilation errors.

So far we have enumerated, using the concrete example, several common MDSD problems at the development time. Next section expands it to runtime concerns, especially for systems that are self-adaptive.




```
1  /** @model */
2  public interface Entity extends EObject {
3    /** @model */ public String getID();
4    /** @generated */ public void setNameID();
5  }
6  ...
7  /** @generated */
8  public class EntityImpl extends EObjectImpl implements Entity {
9    /** @generated */
10   public String getID() { return nameiD; }
11   ...
12   /**@generated*/
13   public String toString() {
14     if (eIsProxy()) return super.toString();
15     StringBuffer result = new StringBuffer(super.toString());
16     result.append(" (iD: ");
17     result.append(iD);
18     result.append(')');
19     return result.toString();
20   }
21   /** @generated NOT */
22   public String toString(String type) {
23     if (eIsProxy()) return super.toString();
24     StringBuffer result = new StringBuffer(super.toString());
25     result.append(" (name: ");
26     result.append(name);
27     result.append(')');
28     result.append(type);
29     return result.toString();
30   }
31 } //EntityImpl
```

Fig. 10: Regenerated code from the model: insertions are underlined and the deletions are ~~stroked out~~, the compilation error is doubly underlined.

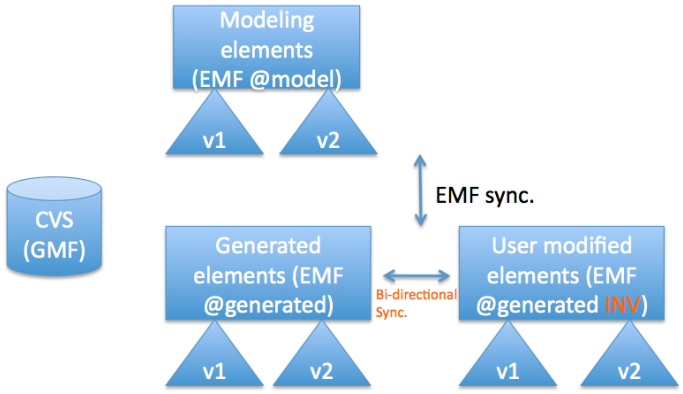



Fig. 11: An overview of the horizontal and vertical traceability links in the bidirectional invariant traceability framework: `blinkit`. V1 and V2 are two revisions of model, template or user codes extracted from the CVS repository of a software development project using EMF code generation.

## 4 Runtime diagnosis of MDSD problems

Debugging high-level programming language programs requires typically traceability between the location where error is spotted and the corresponding location in the source code. Because compiler translations add a layer of indirection, if the optimisation option such as `-O` has been turned on, diagnosing runtime errors become much harder. Programmers would typically *trust* that the optimising transformations do not change the execution behaviour, while debugging the machine code with as few optimizations as possible, e.g., facilitated by the option `-g`. Since MDSD is motivated by the success of compilers, and the models are at a higher level of abstraction than the high-level programming languages, trust needs to be established by a solid understanding of what to diagnose and where to fix problems. However, the template code that addresses most users' requirements may not be exactly what the individual user wanted. Therefore, whenever such diagnoses trace back into the template code, the problem is getting harder.

With the advent of self-adaptive systems, according to Baresi and Ghezzi [3], the boundary between development time and runtime is disappearing. What's typically regarded as development time activities in MDSD may be regarded as runtime ones. This imposes a number of interesting research questions that we do not know an answer yet. Here we try to articulate, using the examples presented so far, an incomplete list of such questions.

– **Runtime traceability.** Unlike the use of traceability at development time, runtime traceability of MDSD systems requires a chain of relaying events to listen to at the runtime. One example of such mechanisms is the event handling in Java runtime virtual machnes. By cascading the listeners to the events, the call traces at the point of failure can present the user a clue

about the fault location. However, such a mechanism require developers to be cooperative: explicit exceptions must be thrown or caught in the try-catch blocks. Otherwise, it can be too late to tell where the exception were generated in the first place. Several machine learning approaches have been proposed to address this issue, by studying the historical events in stack traces [7]. However, runtime traceability requires responsive reactions on the mismatching template and user code which is still not well understood. Earlier work on monitoring and diagnosing software requirements may be helpful to make use of the goal models as a priori knowledge to diagnose problems in the event traces [15]. The challenges we are facing here is that the MDSD processes use more complicated models than goal refinements.

- **Monitoring mismatching requirements** In one would be able to know which requirements are implemented by the default template code, as well as which specific customisation requirements of individual users, then it can be promising to add runtime monitors to places where the mismatches between the two sets of requirements happen at runtime. Right now, the requirements conflict detection techniques require both models to have similar structures (e.g., mergeable). If they don't, the question is how to model them to make it verifiable. Another research question is of course to have an explicit encoding of requirements in the templates to prepare for such verifications.
- **Model interactions problems.** As we described earlier, MDSD is a complicated process which may involve more than one metamodel. The "Tao" is to have a megamodel to unify the different metamodel code generation processes [5]. However, different metamodels may be created by different people and thereby inherently embed interaction *bombs* between the tacit knowledge. They are not necessarily compatible to each other, yet may not be notified by the developers and users at the runtime. A mechanism to protect the different MDSD generated code from feature interaction problems [12] will be very useful. One possible direction of research is to investigate the use of AOP technique to detect and resolve undesired interactions between models at runtime. For instance, dynamic aspect weaving techniques provide a mechanism to inject code to resolve runtime conflicts between models.
- **Meaningful change propagation.** While the models and code co-evolve, a systematic approach is required to propagate the changes from one end to the other. Earlier we have developed the meaningful change detection tools from the specifications what changes are meaningful to different stakeholders [21], as well as the bidirectional transformation framework to propagate the meaningful changes between the template code and the user modifications [17]. One future direction to consider is how to guide the users to specify such meaningful changes or to learn from examples [18].
- **Feedback loops** Since boundary between development-time and runtime is disappearing, the distinction between adaptation and evolution in such systems is also getting blurred. Depending on whether requirements change at runtime, one may separate evolution from adaptation. Yet, the blurring boundary in practice makes it necessary to address MDSD concerns at runtime too. Runtime self-adaptive systems require some form of feedback loops,

e.g., using the PID controller [4], to be able to react to quality requirements changes accordingly. It is our hope that the tacit knowledge concern of MDSD can be addressed such that one can also apply the feedback loop mechanisms to the runtime MDSD problems.

## 5 Summary

There can be more feature interactions in the development if we were to follow the MDSD blindly. In summary to the three reported problems, we propose an *additional alphabet* or *tacit knowledge* concern to the MDSD frame. The concern can be expressed as follows: "When MDSD tool generates code with additional alphabet introduced (in the form of plugin names, package names, class names, or method names), one must ensure these names are not conflicting with the names (subconsciously) introduced by the developer for the modeling language". To avoid such feature interaction problems at runtime, it is required to have additional tools to check any violation of the concern.

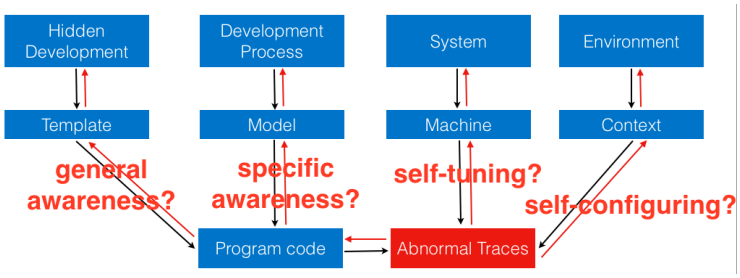



Fig. 12: Runtime problem diagnosis for MDSD systems: tracing the causal chain of events backwards

Figure 12 presents a promising framework to combine tacit knowledge into the consideration of runtime problem diagnosis. The various sources of information including the template in the hidden development, the assumptions of the environmental contexts, are brought together to the attention of users at the runtime. Through predefined runtime monitors, information is already collected in such a way that when the abnormal execution traces are obtained, one can trace backwards to understand the location of fault. If the faults reside in the wrong trust assumptions to the environmental contexts, appropriate runtime adaptation to alternatives in the machine will be switched to at runtime [11].

## 6 Acknowledgement

This work is supported by the ERC Advanced Grant no. 291652 "ASAP: Adaptive Security and Privacy" (2012-2017) - http://www.asap-project.eu. We thank

our colleague Michael Jackson for useful discussions on earlier drafts. Most of the work could not have be done without easy-to-use MDSD tools. We have also benefited from fruitful discussions with Jan Koehnlein and anonymous developers through open-source fora (e.g., https://bugs.eclipse.org/bugs/show_bug.cgi?id=326220).

## Appendix

The concrete syntax of PF:

```
grammar uk.ac.open.Problem with org.eclipse.xtext.common.Terminals
generate problem "http://open.ac.uk/problem"

ProblemDiagram: "problem" ':' name=ID ("for" highlight=[Node])?
((nodes+=Node|links+=Link))*;

Node:
  name=ID (type=NodeType)?
  (':' description=STRING)?
  (""
     (hiddenPhenomena+=Phenomenon
       (',' hiddenPhenomena+=Phenomenon)*)?
     (subproblem+=ProblemDiagram
      | "see" "domain" problemNodeRef+=[Node]
      | "see" "problem" problemRef+=[ProblemDiagram]
      | "see" href+=STRING)*
    "")?;

enum NodeType:
    REQUIREMENT="R" | MACHINE="M" | BIDDABLE="B" | LEXICAL="X"
  | CAUSAL="C" | DESIGNED="D" | PHYSICAL="P";

Phenomenon:
  (type=PhenomenonType)? (isControlled?="!")? name=ID
     (':' description=STRING)?;

enum PhenomenonType:
  UNSPECIFIED="phenomenon" | EVENT="event" | STATE="state";

Link:
  from=[Node] (type=LinkType) to=[Node] ('' phenomena+=Phenomenon
   (',' phenomena+=Phenomenon)* '')? (':' description=STRING)?;

enum LinkType:
  INTERFACE="--" | REFERENCE="~~" | CONSTRAINT="~>";

terminal ID: ('#' (!('#'))+ '#') |
  ('^'? ('a'..'z' | 'A'..'Z' | '_')
        ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*);
```

The concrete syntax of i*:

```
grammar uk.ac.open.istar.Istar with org.eclipse.xtext.common.Terminals
import "platform:/resource/openome_model/model/openome_model.ecore"

Model:
  "istar" (name=ID)? ":"
    (containers+=Container
   | intentions+=Intention
   | decompositions+=Decomposition
   | dependencies+=Dependency
   | contributions+=Contribution
   | associations+=Association)*
  ;

Container: Actor | Agent | Role | Position;

Actor: "actor" name=ID
   (is_a+=[Actor] | is_part_of+=[Actor])*
   (""
     (intentions+=Intention)*
    "")?
;
Agent: "agent" name=ID
   (""
     (intentions+=Intention)*
    "")?
  ;
Role: "role" name=ID
   (""
     (intentions+=Intention)*
    "")?
  ;
Position: "position" name=ID
   (""
     (intentions+=Intention)*
    "")?
  ;

Intention: Goal | Softgoal | Task | Resource | Belief;

Goal: "goal" name=ID
 (""
   (decompositions+=[Decomposition])*
  ""
 )?;
Softgoal: "soft" name=ID;
Task: "task" name=ID;
Resource: "resource" name=ID;
Belief: "belief" name=ID;

Link: Association | Dependency | Decomposition | Contribution;
```

```
Dependable: Intention | Container;
Association: source=[Container] "~~>" target=[Container];
Dependency: dependencyFrom=[Dependable] "~>"
dependencyTo=[Dependable];
Decomposition: AndDecomposition | OrDecomposition;
AndDecomposition: target=[Intention] "<-(and)-" source=[Intention];
OrDecomposition: target=[Intention] "<-(or)-" source=[Intention];
Contribution: AndContribution | OrContribution
  | HelpContribution | HurtContribution | MakeContribution | BreakContribution;
AndContribution: source=[Intention] "-(and)->" target=[Intention];
OrContribution: source=[Intention] "-(or)->" target=[Intention];
HelpContribution: source=[Intention] "-(+)->" target=[Intention];
MakeContribution: source=[Intention] "-(++)->" target=[Intention];
HurtContribution: source=[Intention] "-(-)->" target=[Intention];
BreakContribution: source=[Intention] "-(--)->" target=[Intention];

terminal ID: ('#' (!('#'))+ '#') |
  ('^'? ('a'..'z' | 'A'..'Z' | '_') ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*);
```

An example instance of the problem diagram description.

```
problem: #Workpiece#
CE R :"Command Effects"
ET M :"Editing Tool"
U B :"User"
WP X : "Workpiece"
ET -- U : "b"
ET -- WP : "c"
CE ~> WP : "d"
CE ~~ U : "a"
```

An example instance of the i* description.

```
istar:
  role Developer
  goal #create and maintain modeling tool#
  soft #maintainability#
  soft #productivity#
  soft #compliance by developer#

  role Researcher
  goal #define PF language#
     soft #compliant extent#

  role User
     goal #create/edit PF models#
     soft #compliant intent#

  soft #compliance#
  #create and maintain modeling tool# -(++)-> #maintainability#
  #create and maintain modeling tool# -(++)-> #productivity#
```

```
#create and maintain modeling tool# -(++)-> #compliance by developer#
#define PF language# -(++)-> #compliant extent#
#create/edit PF models# -(++)-> #compliant intent#
#compliance# <-(and)- #compliant intent#
#compliance# ~> #compliance by developer#
#compliance# <-(and)- #compliant extent#
```

## References


1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Prentice Hall, 2 edn. (Aug 2006)
2. Ali, R., Dalpiaz, F., Giorgini, P.: A goal-based framework for contextual requirements modeling and analysis. Requir. Eng. 15(4), 439–458 (2010)
3. Baresi, L., Ghezzi, C.: The disappearing boundary between development-time and run-time. In: Proceedings of the FSE/SDP workshop on Future of software engineering research. p. 1722. FoSER '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1882362.1882367
4. Chen, B., Peng, X., Yu, Y., Zhao, W.: Are your sites down? requirements-driven self-tuning for the survivability of web systems. In: Requirements Engineering Conference (RE), 2011 19th IEEE International. pp. 219–228. IEEE (Sep 2011)
5. Djuric, D., Gasevic, D., Devedzic, V.: The tao of modeling spaces. Journal of Object Technology 5(8), 125–147 (2006)
6. Hailpern, B., Tarr, P.: Model-driven development: The good, the bad, and the ugly. IBM Systems Journal 45(3), 451–461 (2006)
7. Han, S., Dang, Y., Ge, S., Zhang, D., Xie, T.: Performance debugging in the large via mining millions of stack traces. In: Proc. 34th International Conference on Software Engineering (ICSE 2012) (June 2012), http://www.csc.ncsu.edu/faculty/xie/publications/icse12-stackmine.pdf
8. Jackson, M.: Problem Frames: Analyzing and structuring software development problems. Addison Wesley (2001)
9. Jackson, M.: Some notes on models and modelling. In: Conceptual Modeling: Foundations and Applications. pp. 68–81 (2009)
10. Kiczales, G.: Aspect-oriented programming. ACM Comput. Surv. 28(4es), 154 (1996)
11. Salifu, M., Yu, Y., Nuseibeh, B.: Specifying monitoring and switching problems in context. In: Requirements Engineering Conference, 2007. RE '07. 15th IEEE International. pp. 211 –220 (Oct 2007)
12. Tun, T.T., Trew, T., Jackson, M., Laney, R.C., Nuseibeh, B.: Specifying features of an evolving software system. Softw., Pract. Exper. 39(11), 973–1002 (2009)
13. Tun, T.T., Yu, Y., Laney, R.C., Nuseibeh, B.: Early identification of problem interactions: A tool-supported approach. In: Glinz, M., Heymans, P. (eds.) REFSQ. Lecture Notes in Computer Science, vol. 5512, pp. 74–88. Springer (2009)
14. Turing, A.M.: Computability and lambda-definability. J. Symb. Log. 2(4), 153–163 (1937)
15. Wang, Y., McIlraith, S.A., Yu, Y., Mylopoulos, J.: Monitoring and diagnosing software requirements. Autom. Softw. Eng. 16(1), 3–35 (2009)
16. Yu, E.: Modelling strategic relationships for process reengineering. University of Toronto Toronto, Ont., Canada, Canada (1995)

17. Yu, Y., Lin, Y., Hu, Z., Hidaka, S., Kato, H., Montrieux, L.: Maintaining invariant traceability through bidirectional transformations. In: Proc. 34th International Conference on Software Engineering (ICSE 2012). ACM/IEEE, Zurich, Switzerland (Jun 2012)
18. Yu, Y., Bandara, A., Tun, T.T., Nuseibeh, B.: Towards learning to detect meaningful changes in software. In: Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering. p. 5154. MALETS '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2070821.2070828
19. Yu, Y., Jürjens, J., Mylopoulos, J.: Traceability for the maintenance of secure software. In: ICSM. pp. 297–306. IEEE (2008)
20. Yu, Y., Jürjens, J., Schreck, J.: Tools for traceability in secure software development. In: ASE. pp. 503–504. IEEE (2008)
21. Yu, Y., Tun, T.T., Nuseibeh, B.: Specifying and detecting meaningful changes in programs. In: 26th IEEE/ACM International Conference On Automated Software Engineering (Nov 2011), http://oro.open.ac.uk/29450/, to appear