From Model-Driven Software Development Processes to Problem Diagnoses at Runtime

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

[1] Department of Computing and Communications, The Open University, 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. 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. Therefore, there is a gap between templates and runtime adapted models. Generalising from the concrete problematic examples in MDSD processes to a model-based problem diagnosis, the chapter presents a procedure to separate the automated fixes from those runtime gaps that require human judgments.

Keywords: Model-Driven Software Development, Problem Frames.

1 Introduction

Decades after Alan Turing introduced the computing machine that uses a tape of infinitely long '0' and '1' binary numbers to store data and programs [22], abstraction levels of programs have become closer to human understanding of the physical world [13]. High-level programming languages can be automatically translated and optimised into Turing machines by compilers, freeing programmers from having to think in terms of machine instructions [2]. Naturally, one would like to model the physical world, and generate the code for implementing the machine from the model, in the same automated way as compiling source program into binary code. This vision motivates model-driven software development methods (MDSD) [10], using an input model much more abstract than the binary code of Turing machines.

For example, our graphical modeling tool to support the Problem Frames approach (PF) [12] 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 developing the PF modeling tool is a straightforward application of MDSD methods.

N. Bencomo et al. (Eds.): Models@run.time, LNCS 8378, pp. 188–207, 2014.

However, this assumption needs to be checked, both from a requirements engineering (RE) perspective and from a practical, 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 us whether MDSD directly meets the requirement of "supporting a graphical modeling language"; second, it tells us 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 about any generic concerns 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, the practitioners need a new methodology for diagnosing them. In this chapter, we will show that runtime diagnosis of the gap between models in two minds (of a developer and of a user) must be reconciled. We will also demonstrate the feasibility through a new runtime model diagnosis framework summarised at the end of the chapter.

Background and Terminology of MDSD

To demonstrate the problems, a chain of automated tool support from the Eclipse Modeling project[1] and the terminology used in this chapter will be discussed. 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 one concrete syntax to represent EMF, and one may 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. Since 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

[1] `www.eclipse.org/modeling`
[2] `www.eclipse.org/modeling/emf`
[3] `http://www.eclipse.org/Xtext/`

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.

Example: 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 that "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 [12] 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: the PF Graphical Modeling Tool (see Figure 1) is no exception.

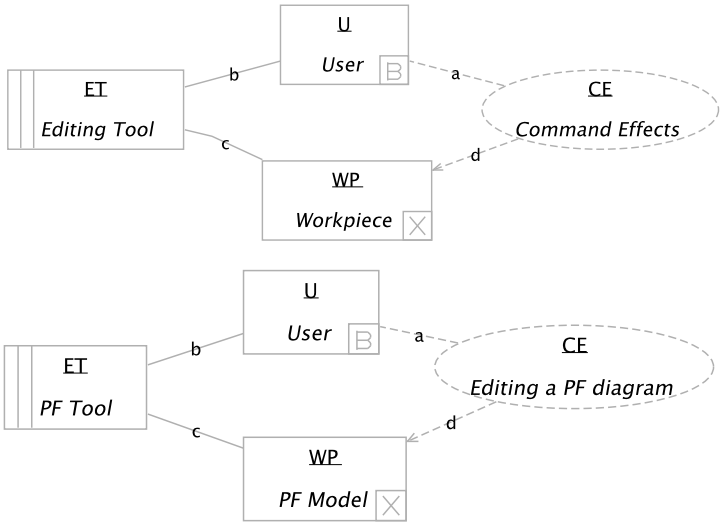


Fig. 1. A Work Piece frame and its instantiation for the PF editing problem

[4] www.eclipse.org/modeling/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 in the literature so far were drawn using either a text editing tool such as LaTeX, or a diagramming tool such as Dia[5]. This raises many interesting questions: "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 role – "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 these productivity and maintainability requirements in mind, there is still one alternative solution to 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 in a problem diagram? 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 checking aside, syntax highlighting, syntax-driven editing, auto completion, pattern matching,

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

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, in addition to the graphical editing features such as drag-and-drop, zooming, panning, layouting, and printing. Instead of asking "why bother with MDSD", one would ask "why bother with implementing all these nice features yourself" instead.

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

2 Problems and Concerns in the MDSD Process

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

Now we now look at the darker corners of the MDSD approach, explaining some issues experienced when applying it. A possibly shocking concern we documented here resembles the experience in several non-trivial instances. 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* [24]. 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 i*, making it much easier to perform new kinds of analysis such as programmatic scoping of the contexts for alternatively refined subgoals [5].

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

Figure 2 summarises the alphabet concern in the "convention over configuration" MDSD paradigm. By "convention", the template code is generated by

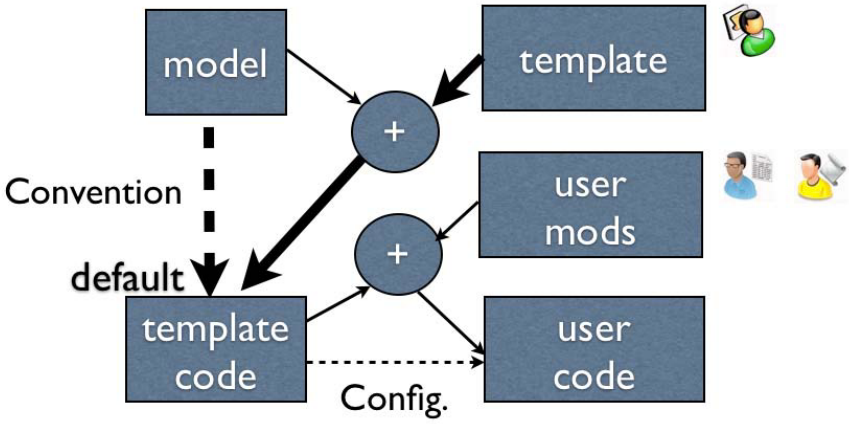


Fig. 2. The additional alphabet (or tacit knowledge) concern in the "convention over configuration" MDSD paradigm. Developers of the templates may not know the requirements of individual programmers, and the individual programmers may not fully understand the rationale behind the default behaviours in the generated template code.

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 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 as the "additional alphabet" or "tacit knowledge" concern of MDSD.

The "Detached" Requirement Phenomena. 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 horizontally or vertically with the node on the other side of the link, the end of the link would not be connected to the requirement node, appearing as if they were 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 were not solved. By tracing 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 loaded a `org.eclipse.gef.NodeEditPart` class in the runtime class library of GEF framework. When such an "import" statement in the customised class was removed, the GMF editor loaded 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 3 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 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 our shared "common sense".

On second thought, this incident could have revealed an interesting type of pitfall in MDSD, which we called "model feature interaction" [21]. 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, and a sophisticated mechanism is needed to prevent this from happening again. For example, a developer may want to avoid 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 advantages gained must be revisited. First one needs to maintain the traceability between the abstract description and the concrete implementations, and second, one must be aware that the designer of the MDSD tools could have introduced some alphabets that may lead to unwanted behaviours when they are composed with the generated code.

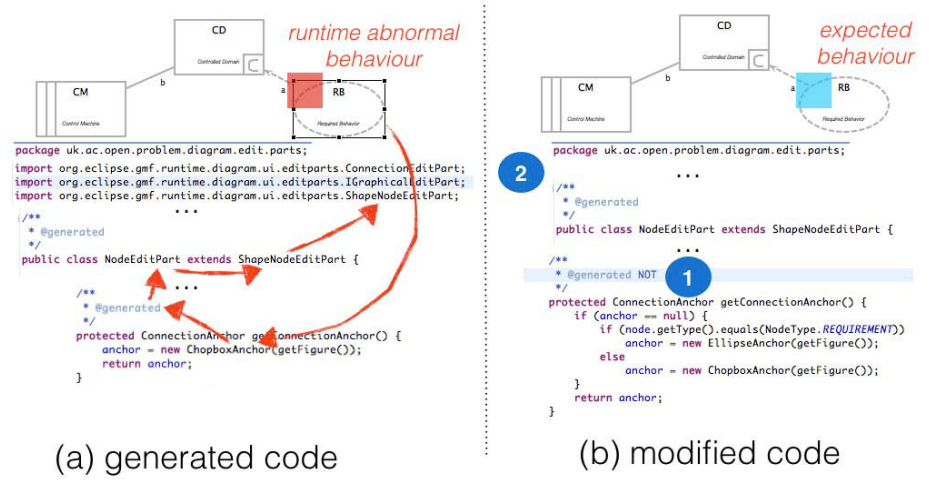


Fig. 3. Contrasting the observable problems and the code implementations respectively for the abnormal and correct behaviours

The developer's interpretations of the additional alphabet may not be the same as the original designer's. This might have a serious implication to security problems, adding further difficulty in maintaining and checking the traceability [27].

The Manual Refactoring Phenomena. 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 a problem node is accessed, it could refer to an i* model element specified in the none-prefixed classes. A subtle but annoying behaviour 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 the University of Toronto have made substantial improvements to 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 [28], 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.

The Dependency Injection Phenomena. Instead of Aspect-Oriented Programming (AOP) [14], 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 were "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 complained that the IDLink resolution class was 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 obtained a package named "uk.ac.open.problem.problem", in accordance with the particular naming convention adopted by the developer of the MDSD tool (i.e., `xtext`).

The Template-User Synchronisation Phenomena. 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, 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. 4).

Fig. 5 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.

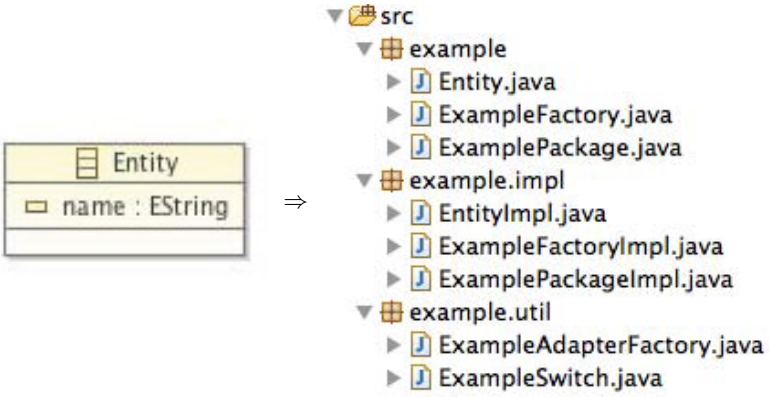


Fig. 4. Default code generated from the EMF meta-model

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.

As 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 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. 6. 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. 7: 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

```
1 package example;
2 import org.eclipse.emf.ecore.EObject;
3 /** @model */
4 public interface Entity extends EObject {
5   /** @model */ public String getName();
6   /** @generated */ void setName(String value);
7 }
```

```
1  package example.impl;
2  import example.Entity;
3  ...
4  /** @generated */
5  public class EntityImpl extends EObjectImpl implements Entity {
6      ...
7      /** @generated */
8      protected String name = NAME_EDEFAULT;
9      ...
10     /** @generated */
11     public String getName() { return name; }
12     /** @generated */
13     public void setName(String newName) { ...  }
14     ...
15     /** @generated */
16     @Override
17     public String toString() {
18         if (eIsProxy()) return super.toString();
19         StringBuffer result = new StringBuffer(super.toString());
20         result.append(" (name: ");
21         result.append(name);
22         result.append(')');
23         return result.toString();
24     }
25 } //EntityImpl
```

Fig. 5. Parts of the generated code in Fig. 4

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()` 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 [25], we have developed a two-layered synchronisation framework, blinkit, to address this problem.

Figure 8 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 [25] 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 phenomena at the development time. They are all related to the "alphabet mismatch" problem that reveals a gap between the understanding of a model developer and a model user. Since the model developer and user are guessing

```
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
```

⇓

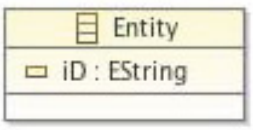


Fig. 6. User modifications to the generated code: insertions are underlined and the deletions are ~~stroked out~~; the changes are reflected

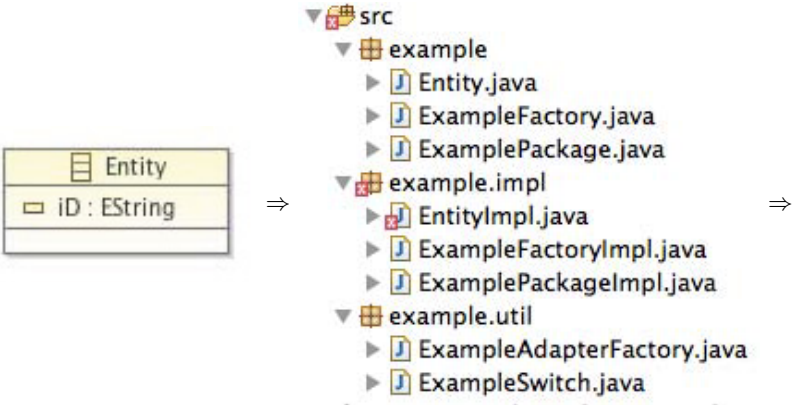
each other's model in mind, ultimately only runtime reconcilation could resolve their differences.

3 Generalised Problem and Related Work

With the advent of self-adaptive systems, according to Baresi and Ghezzi [6], the boundary between development time and runtime is disappearing. What is typically regarded as development time activities in a MDSD process may now be regarded as runtime activities.

Using the examples presented so far, we identify three gaps in the current research on the runtime problem diagnosis.

Monitoring mismatching requirements. If one would be able to know requirements that are implemented by the default template code, as well as specific requirements customised by 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. More generally, developers and users are often inconsistent in terms of their understanding of requirements. Related to this, Requirements Awareness [18] is a key issue. Without runtime awareness of the




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

requirements of individuals, it is harder for developers and users to agree on the current status of the system with respect to the requirements satisfaction.

In general, the MDSD process would require an additional step to regenerate the solution from the modified model. However, without explicit modeling of the generated code and the template code, it is not possible to automate every change through code generation. Due to the lazy binding of problems and solutions, at runtime such mismatches become even more severe. Current requirements

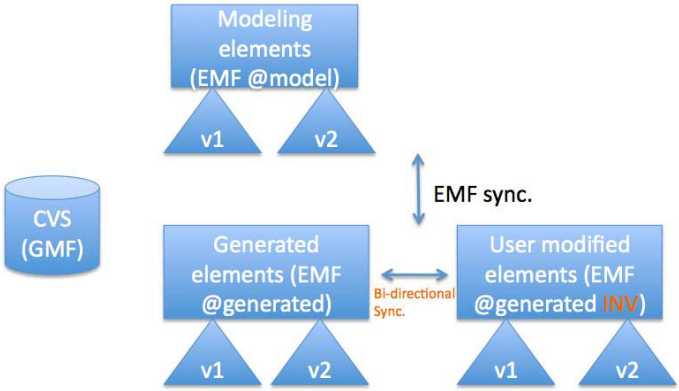


Fig. 8. An overview of the horizontal and vertical traceability links in the bidirectional invariant traceability framework: `blinkit`. V1 and V2 are two revisions of model, template or user codes extracted from the CVS repository of a software development project using EMF code generation.

conflict detection techniques require both models to have similar structures (e.g., mergeable) [15]. If the two models do not have similar structures, the question is how to model them so that they are still verifiable. Another research question is of course to have an explicit encoding of requirements in the templates to prepare for such verifications.

Recently, Akiki et al. [3] proposed the use of interpreted runtime models instead of static models or generative runtime models. Although it is limited in the GUI domain, the proposed solution seems to be promising to bring adaptivity to the runtime systems. A prototype and architecture to support adaptive UI has been developed and demonstrated [4] for adaptive UI of enterprise software applications through service-oriented adaptations.

Runtime traceability. Unlike the use of traceability at development time, runtime traceability of MDSD systems has to listen to the chain of events at the runtime. One example of such mechanisms is the event handler in Java runtime virtual machines. By cascading the listeners to the events, the call traces at the point of failure can give 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 could 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, for example, by studying the historical events in stack traces [11]. 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 [23]. The challenges we are facing here is that the MDSD processes use more complicated models than goal refinements.

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 [9]. 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 [20] 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.

Recently, Bencome et al. [7] proposed a framework to support on-the-fly interoperability at runtime by generating emergent middleware that can synthesise multiple runtime behaviour models in labelled transition systems. In order to avoid feature interactions at runtime, uncertainty handling is still regarded as one of the future work. Tun et al [19] addressed the runtime feature interaction problem by encoding the composition frames using the Event Calculus and resolving the conflicts through a composition controller. However, identifying the composition requirement remains a challenge.

4 Problems Diagnosis at Runtime

This section generalises these runtime concerns into a runtime diagnosis procedure which may become a key component of a self-adaptive problem analysis framework.

On basis of our earlier work on runtime adaptive model interpretation middleware [3, 4], runtime requirements failure diagnoses [19, 23], and invariant traceability [25], we propose a new framework in Figure 9 to consider tacit knowledge for runtime problem diagnosis (PD@runtime). Various sources of information are brought to the attention at the runtime, these include the template development hidden from the users, and the assumptions about the environment hidden from the developers.

The information monitored at runtime includes a context model about the environment [5] and a self-awareness model about the working of the system. A mismatch between the system implementation and the environment expectation is regarded as a system failure or error. To determine what to be included in the system model, at least two kinds of models owned by different stakeholders need to be considered. Generalising from the MDSD process, the template system model captures the knowledge of a developer, whilst the user system model captures the knowledge or at least the perception of a user. Both template and user system models need to be monitored to tell whether any change could lead to a mismatch between the requirements in the way they are understood by the stakeholder. Given that both models are complex, it is usually hard to let either the developer or the user to construct them alone. Instead, the

PD@runtime framework uses a procedure to filter out the changes that can be handled by the underlying automated fixing mechanisms (such as compilers and bidirectional model-transformations [25], such that only information relevant to the requirements mismatch will be passed on to the human stakeholders. Overall, the requirements awareness problem can be defined as the combination of the awareness of system failure and the awareness of the requirements mismatch among stakeholders.

4.1 Meaningful Changes Detection and Propagation

While the template and user models 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 for identifying changes that are meaningful to different stakeholders [29], as well as the bidirectional transformation framework to propagate the meaningful changes between the template code and the user modifications [25]. The meaningful change detection tool can detect any mismatch between two normalised structures, which covers typically all models that can be described by a computer language. Although the tool is powerful, it requires guidance [26] to learn what kind of information is regarded as meaningful from large datasets. Presumably what is meaningful for one stakeholder may not be meaningful to another. Therefore, we have started a new research agenda to refine the viewpoints of different stakeholders into concrete rules in order to judge the relevance to the other stakeholders.

Once the relevant and meaningful information is defined, the tool generates predefined runtime monitors which can already collect information in such a way that when the abnormal execution traces are obtained, one can trace backwards to track the location of faults. If the faults involve any wrong trust assumption about the environmental contexts, an appropriate adaptation alternative will be switched to at runtime [16, 17].

4.2 Feedback Loops

Debugging programs written in a high-level programming language typically requires 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 gets even more difficult.

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 [8], 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.

Data: E: environment context model, S_D, S_U: developer's and user's system models, $\mu(\Delta)$: meaningful change, $\not\cong$: mismatching judgment, T: system execution traces, C: program code
Result: Traces
while *true* **do**
  $(\Delta E, \Delta S, \Delta T, \Delta C) = (E' - E, S' - S, T' - T, C' - C)$;
  $\mu(\Delta E, \Delta S, \Delta T, \Delta C) = (\mu\Delta E, \mu\Delta S, \mu\Delta T, \mu\Delta C)$;
  if $\mu\Delta T \not\cong \mu\Delta C$ **then**
    program_fixed = Abnormal trace fault location and fixing;
    if *! program_fixed* $\wedge \mu\Delta S \not\cong \Delta E$ **then**
      failure_fixed = System failure detected and fixing;
      if *! failure_fixed* $\wedge \mu\Delta S_D \not\cong \Delta S_U$ **then**
        bidirection_transformation = Reconcile requirements mismatch;
        if *! bidirection_transformation* **then**
          inform developer and user about the problem;
          $(S'_D, S'_U) = update(S'_D, S'_D)$;
        end
        else if *bidirectional_transformed* **then**
          $(S'_D, S'_U) = bidirectional\_transformed(S'_D, S'_D)$;
        end
      end
      else if *failure_fixed* **then**
        $(E', S') = failure_fixed(E', S')$;
      end
    end
    else if *program_fixed* **then**
      $(T', C') = program_fixed(T', C')$;
    end
  end
  $(E, S, T, C) = (E', S', T', C')$;
end

Algorithm 1. An illustration of the PD@runtime procedure

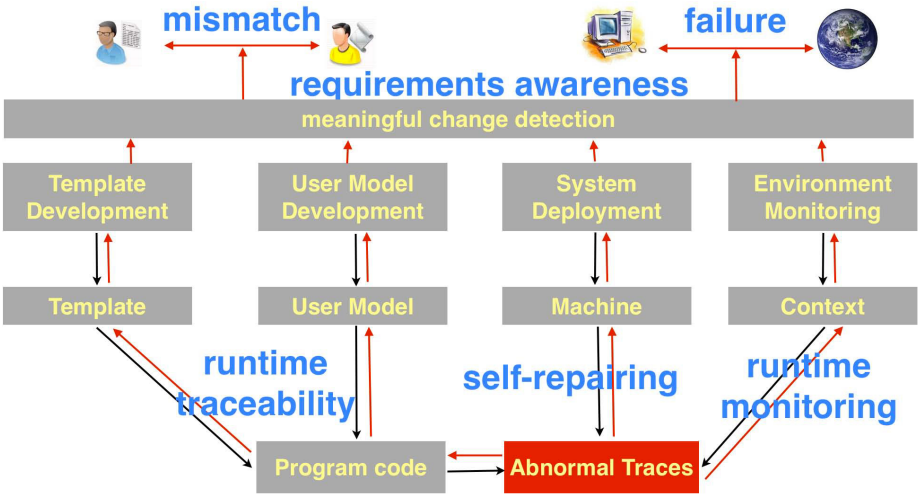


Fig. 9. PD@runtime: tracing the causal chain of events backwards for runtime problem diagnosis for MDSD processes

The following pseudo code describes and summarises the framework into a problem diagnosis at runtime procedure. As one can see, a large part of the procedure can be automated (self-repairing), while some steps still require human on top of the feedback loop to control the overall diagnosis direction.

5 Conclusion

Following the MDSD process blindly at runtime will create more problems in the development than it solves. In summary to the three reported problems, we propose an *additional alphabet* or *tacit knowledge* concern to the MDSD process. The concern can be expressed as follows: "When the MDSD process 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 (unwittingly) introduced by the developer of the modeling language". To avoid such problems at runtime, it is required to have additional tools to check any violation of the concern.

A more general problem of requirements awareness is derived from the problems we identified from the MDSD process. To tackle it, we show a systematic procedure that uses meaningful changes detection to differentiate the changes in environment contexts (execution traces) and in program implementations (of developer and of user), or in more abstract terms, the mismatches between models of different stakeholders. The procedures is controlled by a feedback loop where automatic fixes are employed with compiler and bidirectional transformations. However, when the automatic fixes are not available, human must be informed to handle the more difficult cases.

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).

References

1. 15th IEEE International Requirements Engineering Conference, RE 2007, New Delhi, India, October 15-19. IEEE (2007)
2. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Prentice Hall, 2 edn. (August 2006)
3. Akiki, P.A., Bandara, A.K., Yu, Y.: Using interpreted runtime models for devising adaptive user interfaces of enterprise applications. In: Maciaszek, L.A., Cuzzocrea, A., Cordeiro, J. (eds.) ICEIS (3), pp. 72–77. SciTePress (2012)
4. Akiki, P.A., Bandara, A.K., Yu, Y.: RBUIS: Simplifying enterprise application user interfaces through engineering role-based adaptive behaviour. In: The 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS 2013, pp. 3–17 (2013)
5. Ali, R., Dalpiaz, F., Giorgini, P.: A goal-based framework for contextual requirements modeling and analysis. Requir. Eng. 15(4), 439–458 (2010)
6. 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 FoSER 2010, pp. 17–22. ACM, New York (2010), http://doi.acm.org/10.1145/1882362.1882367
7. Bencomo, N., Bennaceur, A., Grace, P., Blair, G.S., Issarny, V.: The role of models@run.time in supporting on-the-fly interoperability. Computing 95(3), 167–190 (2013)
8. Chen, B., Peng, X., Yu, Y., Zhao, W.: Are your sites down? requirements-driven self-tuning for the survivability of web systems. In: RE, pp. 219–228. IEEE (2011)
9. Djuric, D., Gasevic, D., Devedzic, V.: The tao of modeling spaces. Journal of Object Technology 5(8), 125–147 (2006)
10. Hailpern, B., Tarr, P.: Model-driven development: The good, the bad, and the ugly. IBM Systems Journal 45(3), 451–461 (2006)
11. 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), http://www.csc.ncsu.edu/faculty/xie/publications/icse12-stackmine.pdf (June 2012)
12. Jackson, M.: Problem Frames: Analyzing and structuring software development problems. Addison Wesley (2001)
13. Jackson, M.: Some notes on models and modelling. In: Borgida, A.T., Chaudhri, V.K., Giorgini, P., Yu, E.S. (eds.) Mylopoulos Festschrift. LNCS, vol. 5600, pp. 68–81. Springer, Heidelberg (2009)
14. Kiczales, G.: Aspect-oriented programming. ACM Comput. Surv. 28(4es), 154 (1996)

15. Sabetzadeh, M., Nejati, S., Liaskos, S., Easterbrook, S.M., Chechik, M.: Consistency checking of conceptual models via model merging. In: RE [1], pp. 221–230
16. Salifu, M., Yu, Y., Bandara, A.K., Nuseibeh, B.: Analysing monitoring and switching problems for adaptive systems. Journal of Systems and Software 85(12), 2829–2839 (2012)
17. Salifu, M., Yu, Y., Nuseibeh, B.: Specifying monitoring and switching problems in context. In: RE [1], pp. 211–220
18. Souza, V.E.S., Lapouchnian, A., Robinson, W.N., Mylopoulos, J.: Awareness requirements for adaptive systems. In: Giese, H., Cheng, B.H.C. (eds.) SEAMS, pp. 60–69. ACM (2011)
19. Tun, T., Laney, R., Yu, Y., Nuseibeh, B.: Specifying software features for composition: A tool-supported approach. Computer Networks 57(12), 2454–2464 (2013), http://oro.open.ac.uk/37468/
20. 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)
21. Tun, T.T., Yu, Y., Laney, R., Nuseibeh, B.: Early identification of problem interactions: A tool-supported approach. In: Glinz, M., Heymans, P. (eds.) REFSQ 2009. LNCS, vol. 5512, pp. 74–88. Springer, Heidelberg (2009)
22. Turing, A.M.: Computability and lambda-definability. J. Symb. Log. 2(4), 153–163 (1937)
23. Wang, Y., McIlraith, S.A., Yu, Y., Mylopoulos, J.: Monitoring and diagnosing software requirements. Autom. Softw. Eng. 16(1), 3–35 (2009)
24. Yu, E.: Modelling strategic relationships for process reengineering. University of Toronto Toronto, Ont., Canada (1995)
25. 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 (June 2012)
26. 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, MALETS 2011, pp. 51–54. ACM, New York (2011), http://doi.acm.org/10.1145/2070821.2070828
27. Yu, Y., Jürjens, J., Mylopoulos, J.: Traceability for the maintenance of secure software. In: ICSM, pp. 297–306. IEEE (2008)
28. Yu, Y., Jürjens, J., Schreck, J.: Tools for traceability in secure software development. In: ASE, pp. 503–504. IEEE (2008)
29. Yu, Y., Tun, T.T., Nuseibeh, B.: Specifying and detecting meaningful changes in programs. In: 26th IEEE/ACM International Conference on Automated Software Engineering, pp. 273–282 (November 2011), http://oro.open.ac.uk/29450/