ModelDoc: A Model-Driven Framework for the Automated Generation of Modelling Language Documentation

Jevon M. Wright

School of Engineering and Advanced Technology,
Massey University,
Palmerston North,
New Zealand
Email: `j.m.wright@massey.ac.nz`

Abstract

An important step of developing a modelling language is in its documentation, in order to assist end-user developers in creating valid model instances of the language. However, documentation written separately from the actual implementation is prone to becoming out-of-date, and often introduces a long turn-around time between development and documentation. In this paper, we present *ModelDoc* – a model-driven framework that extracts documentation automatically from existing sources and combines it with manually-specified documentation. This repository of facts may then be used in the domain-specific generation of authoritative and up-to-date documentation for modelling languages.

Keywords: model documentation, domain-specific languages, model-driven engineering, documentation by example

1 Introduction

As discussed by Meservy and Fenstermacher (2005), models are commonly used to represent complex systems, and have long been advocated to improve the reliability and efficiency of software development. Model-driven engineering is a specific area of software modelling which advocates the use of models as first-class citizens in the development process; its intent is to allow software systems to be generated automatically from the model itself, ensuring that the model does not go out-of-date (Koch 2007).

In order to benefit from model-driven engineering, a model needs to have a well-defined specification; this represents the *modelling language* or *metamodel* across all valid model instances of that language. A modelling language may be represented in many ways, such as a formal representation of the syntax through grammars such as EBNF, or the informal semantic description of the language through plain text (Wright 2011). As Kelly and Pohjonen (2009) discuss, the success of a modelling language greatly depends on many factors, such as choosing the correct paradigm, specifying the correct level of abstractness, and providing an appealing language syntax. They also find that the provision of adequate documentation is critical to ensure the success of a modelling language.

For all but the most trivial of languages, the language definition itself cannot be used as the only documentation source, because a language implementation involves many other concepts – such as platform-independent concerns, design concepts, rationale for visual representations, and specific implementation methods – which currently cannot be adequately described within any existing metamodelling technology.

In this paper, we will investigate a variety of existing approaches of keeping documentation for a particular modelling language consistent with its implementation, as well as perform an investigation into the types of features that most documentation sources provide to their users. This feature investigation will be used to inspire the design for a repository of documentation facts, which may be implemented in a model-driven fashion. This will culminate in a proposal for *ModelDoc*, a model-driven framework that can extract documentation from a variety of existing sources, and compose these facts with manually-defined documentation in order to support the automatic generation of consistent documentation sources for a modelling language.

The rest of this paper is as follows. In Section 2, a brief overview of some of the concepts behind good documentation sources will be discussed, followed by a discussion on existing documentation features in Section 3. The implementation of the ModelDoc framework will be discussed in Section 4, and this paper will conclude with a discussion of related and future work in Section 5.

2 Background

The most widely-known modelling language used in software engineering is the Unified Modeling Language (UML, Object Management Group (2007)); this general-purpose modelling language aims to support the analysis, design and implementation of software and processes. The UML specification defines the syntax and semantics of the language through a mixture of formal constraints, visual rules and structured English language; this specification is vital to ensure the consistency and meaning of UML model instances.

Within model-driven engineering, a modelling language needs to be well-documented to ensure that model developers – the end-users that use model instances of that language – can efficiently create, modify and utilise the resulting model instances. This modelling language documentation represents the *documentation source* of the language, and a key concern is in keeping this documentation source consistent with the implementation of the language itself.

As van Deursen and Kuipers (1999) discuss, a good documentation source is consistent with the implementation of the system, and provides a flexible way in which to navigate through the different levels of abstraction necessary for a user to understand the system. A model-driven approach satisfies this second constraint well; if a repository of documentation facts is provided within a model-driven environment, it is possible to generate many different versions of documentation for each group of stakeholders. For example, the documentation for a modelling language may focus on the visual representation for business

users, or focus on the implementation aspects for a technical implementation guide.

2.1 Deriving Documentation Automatically from Source Code

The overall framework for the *ModelDoc* process is adapted from the documentation generation process discussed by van Deursen and Kuipers (1999). They propose the combination of manual and automatic (re)documentation as a method for providing documentation about a system. In particular, van Deursen and Kuipers discuss the process of translating system sources into repositories of *facts*, which may then be *formatted* into documentation.

The documentation of a software system generally cannot be fully derived from its source; as Massoni et al. (2005) argue, it is not usually possible to extract the structural intent of the original design from source code. Similarly, one can argue that it would not usually be possible to fully extract the design intent of a modelling language from its metamodel definition. It may also be difficult to translate derived facts between different layers of abstraction without human intervention, and it is therefore generally necessary to provide additional manual facts.

2.2 Documentation by Example

Learning by example is a universally accepted principle of good teaching and good learning; yet Wu et al. (2010) and Lieberman (1986) argue that it rarely finds its way into the design of programming environments. As Lieberman argue, "a good teacher presents examples of how to solve problems, and points out what is important about the examples."

Since a modelling language environment is a particular implementation of a programming environment, the learning by example concept must similarly apply. A model developer may use a repository of example models, properly documented, as part of the language learning process. This paper will therefore pay special attention to providing examples as part of a modelling language documentation source.

Documentation sources including examples are already present in many existing software frameworks, such as *SWT Snippets* for the Eclipse SWT framework (Eclipse Foundation 2010), and within many of the individual packages of the Java API (Oracle Corporation 2010). Within the model-driven engineering ecosystem, examples are also an important part when introducing design patterns (Gamma et al. 1995), metamodelling architectures (Kleppe et al. 2003), or new modelling languages (Ceri et al. 2002).

3 Common Documentation Features

To understand the types of documentation features that a model developer will expect of a documentation source, a variety of existing popular modelling language and programming languages will be investigated. The results of this investigation will be used to propose the structure of a repository for documentation facts. This repository of documentation facts can then be populated through the combination of automatic fact extraction and manual fact definitions, and translated into a usable documentation format through standard model-driven transformation frameworks.

A range of existing modelling languages and programming languages, within the domains of software engineering and web engineering, were therefore selected as inspirational documentation sources. Only languages implemented using an object-oriented approach were considered. Each language can then be evaluated in order to identify the most common documentation features of existing documentation sources. In this paper, the eight languages selected were:

HTML 5: Hypertext Markup Language A working draft of the popular language for describing hypertext, which are then interpreted by web browsers (W3C Group 2008).

UML: Unified Modeling Language A unified modelling approach to describing and documenting software, business processes and other related processes (Object Management Group 2007).

OWL: Web Ontology Language A family of languages for representing ontologies as part of the semantic web, which can then be interpreted by ontology engines (W3C Group 2004*a*).

PHP: Hypertext Processor A programming language for writing interactive web applications, which is then interpreted by a virtual machine or compiled natively (Lerdorf et al. 2006).

WebML A language for designing data-intensive web applications which may then be generated into code (Ceri et al. 2002).

Java Virtual Machine A programming language for designing object-oriented software applications, which is then interpreted with a virtual machine (Gosling et al. 2005).

SLAng: SLA Definition Language An approach for formally and informally documenting the service level agreements between customers and providers, which may then be verified at runtime through JMI (Skene and Emmerich 2006).

EMF: Eclipse Modeling Framework A metamodelling framework which can generate Java scaffolding for a metamodel implementation, along with Javadoc documentation describing the model (Steinberg et al. 2009).

The major documentation sources for each of these eight languages were individually evaluated in order to identify the common documentation features of each documentation source. The language-specific evaluations of these features is provided here in Table 1; a description of each of these particular features will now be discussed.

3.1 Structural Documentation

These documentation features relate to the *structure*-based documentation of the language, as discussed by Steinberg et al. (2009). More is not necessarily better; a language that describes a language in terms of all of these documentation features may become too large and hinder end-user understandability. As discussed by van Deursen and Kuipers (1999), it should be possible to provide different versions of the same documentation for different stakeholders.

- **Packages:** For large languages, common concepts are often split into separate packages.
- **Classes:** The different primitive types, elements or classes defined in the language.
- **Attributes:** Classes may define specific or shared class attributes.
- **References:** Classes may also define references to other element instances.

- **Subclasses:** For languages that use subtyped classes, the documentation should provide navigation over this subtype hierarchy.
- **Abstract Classes:** For languages that use the concept of abstract classes, the documentation should highlight that these classes cannot be directly instantiated.
- **Rationale:** When developing a modelling language, it is important to discuss the rationale behind the inclusion of each element of the language (Moody 2009).

3.2 Semantics

While a documentation source for a modelling language may easily reproduce the structure of the language, it is more difficult to define the behaviour or meaning of the language itself. As discussed by Harel and Rumpe (2000), the notation of a language is represented as its *syntax*, and the meaning of the language is represented as its *semantics*. As a language focused on the representation of knowledge, the OWL family of languages is well-defined through both direct semantics and RDF-based semantics (W3C OWL Working Group 2009).

- **Formal:** Formally defined semantics; that is, semantics defined according to a formal syntax (Skene and Emmerich 2006).
- **Informal:** Informally defined semantics; that is, semantics not defined to any formal syntax, such as the English language.
- **Syntax:** A formally defined syntax or language structure (Harel and Rumpe 2000); for visual languages, this may include the visual representation of each modelling language element.
- **Invariants:** Invariants or constraints that cannot be defined using syntax or structure, defined separately from informal semantics, but using a formal syntax.

3.3 Documentation Accessibility

These documentation features do not refer to the content of the documentation source, but rather to particular accessibility and usability features that can improve the efficiency of the documentation for end-users.

- **Index:** An alphabetised list of key concepts and phrases used within the documentation, along with a navigable interface such as hyperlinking for each result.
- **FAQ:** An informal section describing frequently asked questions and answers for using the language itself.
- **Examples:** Documentation by example allows new users to quickly grasp onto language concepts, and provides a repository of common design patterns, as discussed earlier in Section 2.2.
- **Pseudocode:** For executable modelling languages, pseudocode provides a platform-independent way of describing how operations work, and pseudocode documentation allows users to understand the effects of an operation without having to investigate the source.
- **Link to Source:** For open-source projects, it may be possible to directly link from the documentation of a model element to the underlying implementation of that element.
- **User Contributions:** By allowing users to contribute new or missing knowledge directly to the documentation source, the language documentation becomes a richer and more collaborative environment. However, this means that the documentation source must continually be re-released, or made available online.

The suite of documentation for each of the selected eight programming languages may now be evaluated against each of these documentation features, and the results from this investigation are presented here in Table 1.

3.4 Discussion

The language-specific evaluation of documentation sources provided in Table 1 can be used to infer some conclusions on the documentation approaches of each language.

For example, we can see that while informal semantics are often documented, it is less likely that formal semantics are also documented. Similarly one can also see that syntax is often specified, yet invariants are not supplied as often. This is important, because invariants allow the syntax of the language to be further refined. In particular, neither PHP or WebML has documented formal semantics, and this may highlight an area of future work for the developers of these languages – in the case of PHP, the runtime semantics of their large suite of test cases (PHP Group 2011) could be integrated directly into their documentation.

Out of the eight documentation sources evaluated, PHP was the only language that provided a list of FAQs, and similarly the only language that provided a mechanism for including user contributions to its major documentation source. It may be the case that languages with a formal definition – such as UML and OWL – provide a specification strong enough that their authors consider FAQs and user contributions unnecessary. However these documentation elements improve the accessibility of the language, and it may be beneficial to include FAQs or user contributions in future documentation sources.

4 Implementation

In this section, the implementation of the *ModelDoc* framework will be described. ModelDoc aims to combine automatically derived documentation from source code with manually provided facts, in order to keep the documentation source consistent with the implementation itself.

4.1 Eclipse Modeling Framework

If a model designer is using the Eclipse Modeling Framework (Steinberg et al. 2009), then much of the structure and syntax of a language may be derived automatically from the Ecore metamodel. EMF also supports the annotation of model elements with additional documentation, as in Listing 1; these annotations can then be made available and accessible at runtime.

If the EMF annotation uses a particular *EAnnotation* as illustrated in Listing 1, then this annotation will be included in the Javadoc documentation for the generated model element. This annotation will then be available through standard Javadoc generated documentation. EMF annotations therefore provide an easy way to attach documentation to model elements, however the standard EMF code generation

Language	HTML	UML	OWL	PHP	WebML	Java	SLAng	EMF	IAML
Structural Documentation									
Packages	-	✓	-	✓	✓	✓	✓	✓	-
Classes	✓	✓	✓	✓	✓	✓	✓	✓	✓
Attributes	✓	✓	✓	✓	✓	✓	✓	✓	✓
References	✓	✓	✓	✓	✓	✓	✓	✓	✓
Subclasses	-	✓	✓	✓	-	✓	✓	✓	✓
Abstract Classes	-	✓	-	✓	-	✓	✓	✓	✓
Rationale	✓	✓	✓	-	✓	-	-	-	✓
Semantics									
Formal	✓	✓	✓	-	-	✓	✓	-	-
Informal	✓	✓	✓	✓	✓	✓	✓	-	✓
Syntax	✓	✓	✓	✓	✓	✓	-	-	✓
Invariants	✓	✓	✓	-	-	-	✓	✓	✓
Documentation Accessibility									
Index	✓	✓	✓	✓	✓	✓	-	✓	✓
FAQ	-	-	-	✓	-	-	-	-	-
Examples	✓	✓	✓	✓	✓	✓	-	-	✓
Pseudocode	✓	-	✓	-	-	-	✓	-	-
Link to Source	-	-	-	-	-	-	-	-	✓
User Contributions	-	-	-	✓	-	-	-	-	-

Table 1: Feature comparison of existing documentation sources against the ModelDoc-generated documentation of IAML

```
<eClassifiers xsi:type="ecore:EClass" name="CastNode"
    eSuperTypes="#//ActivityNode">
  <eAnnotations source="http://www.eclipse.org/emf/2002/GenModel">
    <details key="documentation" value="Allows one {@model DataFlowEdgeSource datatype}
      to be cast to another {@model DataFlowEdgeDestination datatype}. Has an outgoing
      &quot;check&quot; {@model DataFlowEdge} which can be used to check for invalid
      casts (otherwise a failing conversion is silent)."/>
  </eAnnotations>
</eClassifiers>
```

Listing 1: An annotation in EMF for additional model element documentation

platform presently supports only the single annotation "`documentation`" (Steinberg et al. 2009).

Within EMF, the metamodels for a modelling language can be considered the "source code" for the modelled system, as discussed by van Deursen and Kuipers (1999). This means that many of the documentation features discussed earlier in Section 3 – such as packages, classes, attributes and enumerations – can be automatically derived from the instance of the Ecore metamodel.

4.2 The Internet Application Modelling Language

As a proof-of-concept, the ModelDoc framework will be applied to a platform-independent modelling language for for Rich Internet Applications, called the *Internet Application Modelling Language* (IAML) as described by Wright (2011). This language includes a visual notation within a generated graphical environment, allowing for RIAs to be defined visually and generated into executable web applications from within an Eclipse-based environment. The implementation of this language is available online under an open-source license, allowing for documentation facts to link directly to their implementations.

The metamodel for IAML is implemented as an Ecore metamodel through EMF as `iaml.ecore`. As described by Steinberg et al. (2009), this Ecore metamodel instance is then used to generate the Java source code scaffolding necessary for interaction with instances of this modelling language. As a model-driven language, IAML also defines a suite of code generation templates through the openArchitectureWare Xpand framework (Efftinge et al. 2008), which may be used to generate a PHP/Javascript-based web application from a given model instance.

Finally, IAML also implements the concept of *model completion*, where an intended model may be automatically inferred from a base model as per documented conventions. As discussed by Wright and Dietrich (2010), one of the major challenges for providing model completion is in providing adequate documentation for the process; a developer needs to understand these documented conventions without having to manually study the inference rules to understand their effects.

These three aspects of the IAML implementation illustrate three key sources of documentation that need to be provided to a model developer: the structure and syntax of the metamodel, including its visual representation; the intent behind the inference rules of model completion; and any platform-specific concerns that arise with the code generation templates.

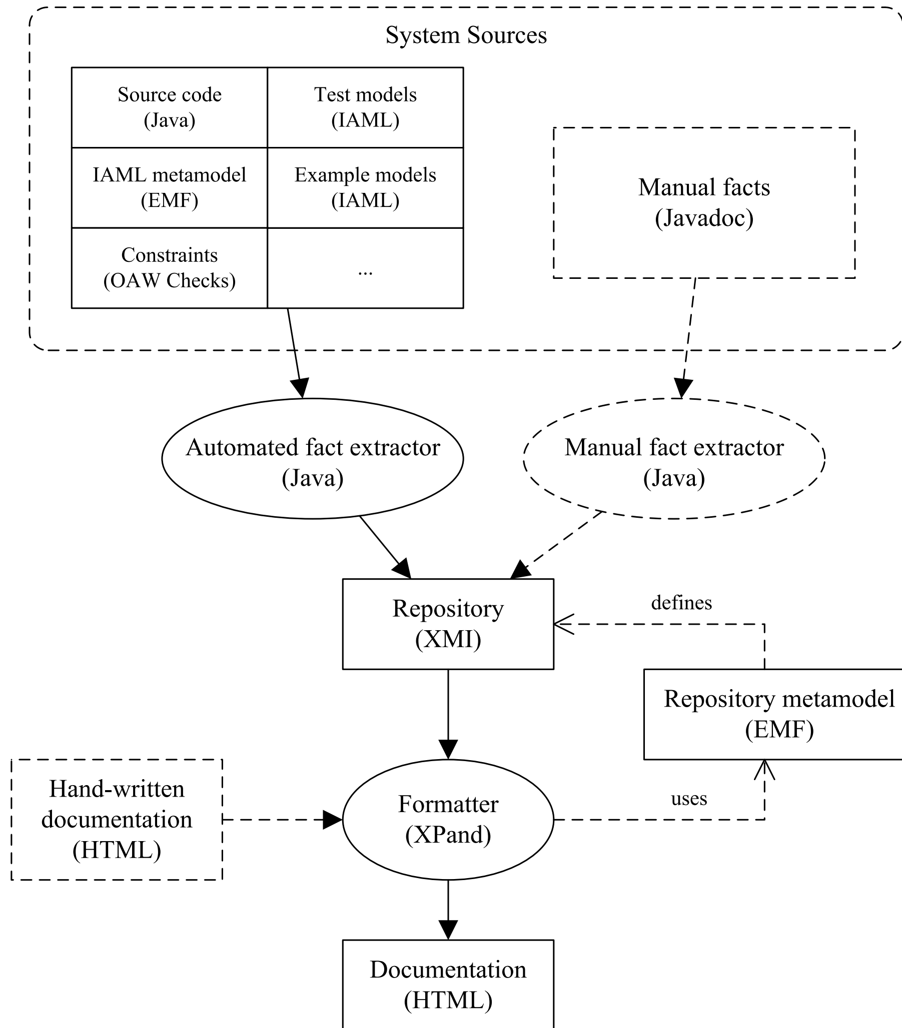

Figure 1: The model-driven process of ModelDoc in generating IAML system documentation, adapted from van Deursen and Kuipers (1999)

4.3 Fact Extraction

The process of generating the documentation in ModelDoc is illustrated here in Figure 1; in this figure, solid lines represent automated processes, and dashed lines represent the inclusion of information defined manually. Facts are derived from a variety of system sources, such as test models, constraints and code generation templates, using *documentation loaders*[1] written in Java.

To store these facts within a repository, a metamodel is defined using EMF (Steinberg et al. 2009), which is used as the definition of a model instance that stores facts for a given set of system sources. A full discussion on the structure of this metamodel is well outside the scope of this paper; however, the interested reader may obtain a copy of this metamodel online at `http://openiaml.org/modeldoc/`.

A ModelDoc model instance is then passed along to a model-to-text transformation implemented in openArchitectureWare, which can import additional documentation defined manually in HTML. Alternatively, the model instance may be serialised to the XML-based XMI format, and loaded by any number of other model-driven tools that support Ecore metamodels.

[1] *Documentation loaders* are Java classes which implement the `org.openiaml.docs.generation.ILoader` interface; in the current implementation of ModelDoc, thirteen such loaders are provided.

4.4 Documentation through Javadoc Tags

To support metamodel developers in the definition of the *manual facts* of the system – that is, documentation that cannot easily be derived with the appropriate level of abstraction from the system sources – ModelDoc reuses the existing concept of Javadoc tags (Kramer 1999).

Javadoc allows a developer to provide additional annotations for some component of a Java software system, such as a class or method. These annotations are provided as part of a source comment for a particular source element. For example, the `@version` and `@author` tags allow an author to annotate a Java source element with version and author metadata; similarly, the `@return` tag adds additional meaning to the return value of a method. These tag instances are then used in the generation of an API documentation using *doclets*.

Importantly, the design of Javadoc places the annotations of source elements very close to the actual source elements themselves. The intent behind this process is that it encourages the use of documentation as an integral part of the specification, and also tries to remove inconsistencies between different documentation sources that may not be synchronised (Kramer 1999). However, the structure of the documentation produced is restricted to the code structure, as discussed by Aguiar and David (2005).

In the case of the documentation of IAML, the standard Javadoc tags – such as `@version` and

Javadoc Tag	Intended Usage
`@model`	An inline tag which provides a hyperlink to another model element according to its simple name; similar to the builtin `@link` Javadoc tag.
`@implementation`	The annotated source code element describes the behaviour or platform-specific concerns of the implementation of a particular modelling element, such as its underlying architecture, target languages, or operation pseudocode.
`@inference`	The annotated source code element describes an inference rule used in *model completion*, as described by Wright and Dietrich (2010); the tag describes the intent of the rule in plain English, with regards to particular modelling language elements.
`@example`	The annotated source code element describes an *example model*; the tag describes the example and its rationale for inclusion in the documentation in plain English, with regards to particular modelling language elements.

Table 2: Custom Javadoc tags defined by ModelDoc, used to provide manual facts in the documentation of IAML

`@author` – are reused, and additional custom tags are also provided as illustrated in Table 2. This approach is not novel, and has been used in other scenarios for software engineering – as discussed later, this includes domains such as design patterns and API contracts.

Listing 2 illustrates an example use of these custom tags as applied to a Java class in order to document additional facts about the source system. In this example, we illustrate that the intended behaviour of a *ExitGate* model element is that it prevents access outside of a *Scope* until all incoming *Conditions* to that *Scope* are false.

These facts are attached to a test case that directly tests these particular facts, reducing the possibility of the documentation losing synchronisation with the actual implementation – if the behaviour of any of these model elements change in the future, then the metamodel developer will have to subsequently modify the failing test cases, and this erroneous documentation can be fixed simultaneously when fixing the test case.

4.5 Generation of Hypertext Documentation

In order to present the final documentation source of IAML, hypertext has been considered is the most natural fit, as it provides "a single interface to browse across large classes of information" (Berners-Lee and Cailliau 1990). All of the elements of the generated documentation are therefore accessible using a standard web browser, including all of the source code elements in the system. If the source code is also made publicly available – such as under an open source license – then derived facts can also be directly linked to the source code itself.

While using hypertext documentation is fine for describing textual and some graphical documentation, one difficult arises when trying to illustrate examples that are implemented in a graphical editor. Normally, an example model would need to be loaded by the graphical environment itself, which may incur a mental penalty in terms of switching between different interfaces.

ModelDoc solves this problem by automatically transforming the example models into hypertext graphical interfaces, which do not require the modelling environment to be loaded (or even installed). While this means that detail about the model may be lost – for example, it is not possible to drag shapes around, or to see all of the attributes of individual model elements – there is still enough information for a user to form a basic mental model, without switching out of the documentation source. For every example model exported this way, ModelDoc also provides a link to the original example model itself, so that the user may load up the model instance normally if necessary.

Since IAML models are designed in a hierarchical fashion, it is possible to split up the source model into many different views, linking these views together using hyperlinks. Figure 2 illustrates how an example model in IAML is navigable; a documentation user may click on almost any model element in order to view its contained elements, and this view is exported directly from the underlying GMF implementation.

5 Discussion

For the end-user model developer, the metamodel documentation for IAML generated by ModelDoc has been extremely valuable for the development of IAML model instances. An annotated screenshot of this generated documentation is provided here in Figure 3. This generated documentation is available online at `http://openiaml.org/model/`, and the interested reader is encouraged to view this documentation themselves and interact with the examples.

In the development of an implementation of the *Ticket 2.0* benchmarking application (Wright and Dietrich 2008), the generated language documentation was referred to frequently. In particular, the development of the IAML model instance only required the Eclipse modelling environment and a standard web browser to be open. Similarly, the range of documented examples provides a simple way to understand how certain modelling concepts work, and on common model instance design approaches.

With respect to the development process of the modelling language itself, ModelDoc has also been valuable in keeping the documentation of the language synchronised with the language itself. At the time of writing, the IAML metamodel has been modified 234 times over its 35-month development lifecycle[2]; without the support of a tool like ModelDoc, this could imply that an independent documentation source would have also required 234 revisions, which represents a significant amount of development effort.

ModelDoc can also assist in the modelling language design process, if the ModelDoc framework is introduced early into the development lifecycle. By translating a source metamodel into a platform-independent hypertext representation, interested third parties can contribute in the design of the modelling language without needing to install the metamodelling environment itself. Similarly, the fact

[2] This represents the number of times that the `iaml.ecore` metamodel definition file has been modified in its Subversion repository.

```
/**
 * Demonstrating the use of ExitGate to prevent access outside a
 * Scope without first viewing an advertisement page.
 *
 * @author jmwright
 * @example ExitGate
 *    Using an {@model ExitGate} to prevent access outside of a {@model Scope} without
 *    first viewing an advertisement {@model Frame}.
 * @implementation ExitGate
 *    If a {@model Session} contains a {@model ExitGate}, all incoming
 *    {@model Condition}s must be false before the {@model Scope} may be left.
 */
public class ExitGateAd extends CodegenTestCase {
   ...
```

Listing 2: A sample ModelDoc comment

extraction process may be integrated with a metrics generation tool to create metamodelling metrics, as illustrated by Wright (2011).

5.1 Related Work

There is a significant body of related work that also utilise the Javadoc documentation method to supply additional documentation, and a wide repository of proposed Javadoc tags for particular use cases. For example, Torchiano (2002) proposes tags such as `@pat.name` and `@pat.role` to describe instances of design patterns in Java software (Gamma et al. 1995). Similarly, Briand et al. (2003) propose tags such as `@pre`, `@post` and `@invariant` to describe the contract-based pre-conditions, post-conditions and invariants of an API.

Javadoc tags may also be used to extract a metamodel from an existing software structure; for example, the EMF framework can identify `@model` annotations on an existing API to define an Ecore metamodel instance (Steinberg et al. 2009), and uses the `@generated` tag to mark automatically-generated scaffolding.

Leslie (2002) reuse the Javadoc fact extraction process itself to propose an XML-based repository of both the structure of Java software and the Javadoc comments within its source. This serialisation process is similar to the ModelDoc serialisation of extracted facts, and the ModelDoc repository is automatically serialised into the XML-based XMI format. XML-based model transformations such as XSLT (W3C Group 2007) may be used to integrate the two repositories. However, the EMF-based ModelDoc metamodel provides a richer interface to third parties than an DTD-based XML representation of the same facts, and this is a benefit of using a model-driven approach.

Another related work is XSDoc as proposed by Aguiar and David (2005), which also advocates weaving of multiple documentation sources; in this case, from Wiki text. In this approach, snippets of Java software can be directly inserted into generated documentation. This approach is better suited to describe particular aspects of an existing software system, rather than in describing *all* aspects of the system, as it is focused on the source code level rather than the architectural level. Nevertheless, it may be desirable to use such an approach within ModelDoc for describing complex operations, if the alternative of describing pseudocode through a `@pseudocode` tag is not desirable.

While this paper has focused on the documentation of a modelling language, there is also extensive work on the documentation of *model instances* of modelling languages. The WebML web application modelling language includes a tool called *WebMLDoc* (Ceri et al. 2002), which generates a Javadoc-style API documentation for a given WebML model instance. As WebML is a visual modelling language, this documentation includes a navigable visual interface to the model instance, and this approach is similar to the hypertext navigation of example models approach used in ModelDoc. However, by design the ModelDoc extraction of example models can apply to *any* GMF-based diagram editor, whereas the WebMLDoc extraction process can only apply to WebML model instances.

5.2 Future Work

The fact extraction process of ModelDoc excels at extracting domain-independent facts, such as the metamodel structure, code generation templates, and inference rules. However, some modelling languages may introduce new first-class concepts – such as type systems or events – and these new concepts should similarly be treated as first-class elements within the documentation.

Currently, these first-class concepts are extracted by extending the ModelDoc code generation templates, and by defining additional language-specific Javadoc tags[3] which are identified by these templates. As the code generation templates of ModelDoc are defined by the language Xpand, these extensions could be provided through the concept of *dynamic templates* (Efftinge et al. 2008); here, a third party can provide aspect-oriented extensions to a suite of templates, which are merged at runtime.

It may be possible to describe these first-class concepts using a special mapping-based metamodel as input to the ModelDoc process; for example, specifying that all containment references to a type named *Event* should be identified as *events*, and similarly provide a new inline Javadoc tag `@event`. This may improve the usability and independence of the ModelDoc approach, and remains an interesting avenue of future work.

There are also a rich variety of existing documentation sources that can be integrated into the ModelDoc framework as future work. For example, code analysis tools such as GUERY (Dietrich 2011) can be used to identify architectural antipatterns in dependency graphs, and the output of these tools could augment the documentation repository. It is possible to use the metamodel extensibility support of EMF to define tool-specific extensions to the ModelDoc metamodel,

[3] For example, the IAML-based implementation of ModelDoc defines the additional inline Javadoc tags `@event` and `@type`, for *Events* and XSD datatypes (W3C Group 2004*b*) respectively.

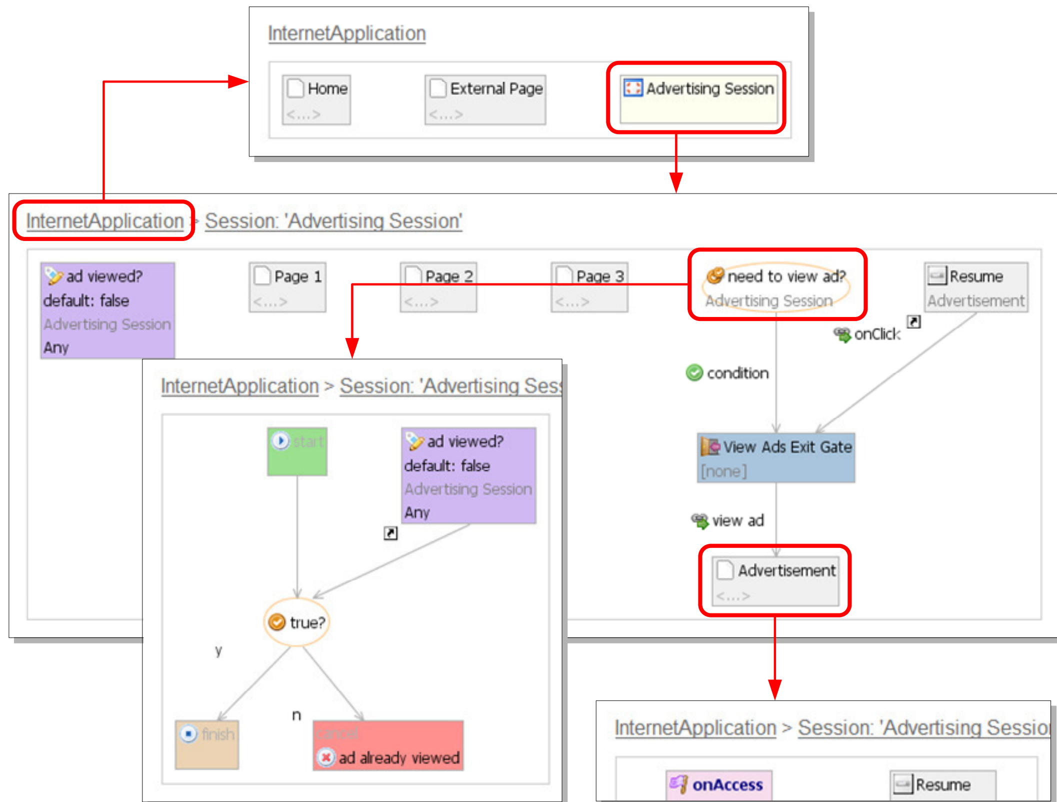

Figure 2: Hypertext navigation of example models

which can then be integrated into the documentation through the dynamic templates support of Xpand.

Some of the desired documentation source features – such as pseudocode, syntax, FAQs and user contributions – have not yet been implemented through the ModelDoc framework, as illustrated in Table 1. In some cases, these can be supported through the definition of new Javadoc tags. For example, a new `@psuedocode` tag could be used to define pseudocode on operations, and included as part of the annotation-based documentation of the EMF model itself; alternatively, it may be possible to derive the pseudocode from an annotated OCL definition (Steinberg et al. 2009). An implementation of user contributions would require a method of accepting, storing and parsing user feedback – such as a web service – and this similarly remains future work.

As the repository-to-documentation source process is performed in a platform-independent way, it is possible to transform the same repository into many different versions and formats. This has already been achieved in the printable version of the documentation for IAML, as illustrated by Wright (2011); here, the Xpand templates were used to generate LaTeX-formatted templates[4], which could be included into a standard LaTeX document.

Since the Eclipse environment supports inline web browsers[5], it may be possible to provide a specific documentation view within the modelling environment that shows the generated documentation for the type of a selected graphical model element. This may improve the performance and efficiency of developing model instances in a graphical model-driven environment, and would integrate well into a metamodelling environment as a whole.

[4]Since the hand-written documentation of IAML was implemented using HTML, it was necessary to implement an HTML-to-LaTeX converter as part of this extension.

[5]For example, web browsers may be supplied through the Eclipse `org.eclipse.ui.browser` package.

Finally, since the ModelDoc repository is represented as an EMF metamodel, it is possible to document the ModelDoc repository structure using ModelDoc itself. This would represent the ultimate in documentation dogfooding, and remains an interesting aspect of future work.

The work described in this paper has been implemented as free software under an open-source licence, as part of a model-driven graphical CASE tool for modelling RIAs. The interested reader is encouraged to learn more online at `http://openiaml.org/`.

References

Aguiar, A. & David, G. (2005), WikiWiki Weaving Heterogeneous Software Artifacts, *in* 'Proceedings of the 2005 International Symposium on Wikis (WikiSym '05)', ACM, New York, NY, USA, pp. 67–74.

Berners-Lee, T. & Cailliau, R. (1990), WorldWideWeb: Proposal for a HyperText Project, Technical report, CERN.

Briand, L. C., Labiche, Y. & Sun, H. (2003), 'Investigating the use of analysis contracts to improve the testability of object-oriented code', *Software: Practice and Experience* **33**, 637–672.

Ceri, S., Fraternali, P., Bongio, A., Brambilla, M., Comai, S. & Matera, M. (2002), *Designing Data-Intensive Web Applications*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Dietrich, J. (2011), 'The GUERY Framework'.
URL: *http://code.google.com/p/gueryframework/*

Eclipse Foundation (2010), *SWT Snippets*.
URL: *http://www.eclipse.org/swt/snippets/*

Efftinge, S., Friese, P., Haase, A., Hübner, D., Kadura, C., Kolb, B., Köhnlein, J., Moroff, D.,

Index

Supertypes

Abstract Classes

Rationale

Visual Syntax

< Back to list

DomainType extends *ContainsWires*, *ExtendsEdgeDestination*, *ExtendsEdgesSource*, *GeneratedElement*, *GeneratesElements*, *ParameterEdgesSource*, *RequiresEdgesSource*

Schema Name : DomainType

The composition of individual data elements into a domain-specific structure.

A DomainType can be composed of many DomainAttributes, each of which has a particular *data type*. Foreign keys are supported by declaring that a given DomainAttributes extends another through an ExtendsEdge. A DomainType can *extend* another DomainType through an ExtendsEdge to describe a simple form of inheritance. Multiple inheritance is supported, but circular inheritance is not allowed.
[DomainType.html]

Informal Semantics

Direct subtypes:

- Role

Subtypes

All subtypes:

- Role

Examples

Documentation by Example

- DomainInheritance.iaml (source)
  - DomainTypes can extend other DomainTypes to inherit their attributes. [DomainInheritance]
- DomainInheritanceEditing.iaml (source)
  - Creating a new instance of an inherited DomainType,accessible only within a Session. [DomainInheritanceEditing]
  - Creating a new instance of an inherited DomainType. [DomainInheritanceEditing]

Inference Rules

Inference Documentation

- A DomainType which does not contain a primary key DomainAttribute will contain one named 'generated primary k... [domain]
- A DomainSource of type `PROPERTIES_FILE` will populate the 'properties' DomainType with string- typed DomainAttri... ...es file. [file-domain-object]
- A primaryKey DomainAttribute in a DomainType, represented as a supertype of another DomainType, will be copi... ...ked as a foreign key. [domain]
- A non- primaryKey DomainAttribute in a DomainType, represented as a supertype of another DomainType, will be copied into the subtype schema and marked as a subattribute. [domain]

Constraints

Invariants

- "A DomainType should have at least one DomainSource specified" [Checks.chk]
- "You cannot have a DomainType called single_values (reserved word)" [Checks.chk]

Attributes

Link to Source

(none)

Inherited Attributes

Attributes

- generatedRule : EString (optional)
  To assist in debugging, this attribute may be populated with the name of the model completion rule that created this eleme... ...port may be provided in the future.
- id : EString (optional)
  A unique identifier for this model element. Should be globally unique.
- isGenerated : EBoolean (optional)
  Set to `true` if the current element has been generated.
- overridden : EBoolean (optional)
  If a GeneratesElements is *overridden*, then this element will not generate any elements directly.

. . .

- wires : [0...*] *Wire*
  A list of contained Wires.

References

References

- inSchemas : [0...*] SchemaEdge [opposite: to]
  A list of outgoing SchemaEdges to specify the potential instance sources of this type.

Inherited References

- generatedBy : [0...*] *GeneratesElements* [opposite: generatedElements]
  A list of elements that generated this element; conversely, overriding the source will prevent this element from being created.
- generatedElements : [0...*] *GeneratedElement* [opposite: generatedBy]
  A list of GeneratedElements that this element has generated through *model completion*.
- inExtendsEdges : [0...*] ExtendsEdge [opposite: to]
  A list of incoming ExtendsEdges.

Figure 3: Hypertext documentation for an IAML model element *DomainType* as generated by ModelDoc

Thoms, K., Völter, M., Schönbach, P., Eysholdt, M. & Reinisch, S. (2008), *openArchitectureWare Documentation*.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional.

Gosling, J., Joy, B., Steele, G. & Bracha, G. (2005), *The Java Language Specification*, 3rd edn.

Harel, D. & Rumpe, B. (2000), Modeling Languages: Syntax, Semantics and All That Stuff, Part I: The Basic Stuff, Technical report.

Kelly, S. & Pohjonen, R. (2009), 'Worst Practices for Domain-Specific Modeling', *IEEE Software* **26**, 22–29.

Kleppe, A. G., Warmer, J. & Bast, W. (2003), *MDA Explained: The Model Driven Architecture: Practice and Promise*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Koch, N. (2007), 'Classification of Model Transformation Techniques Used in UML-based Web Engineering', *Software, IET* **1**(3), 98–111.

Kramer, D. (1999), API Documentation from Source Code Comments: A Case Study of Javadoc, *in* 'Proceedings of the 17th Annual International Conference on Computer Documentation (SICDOC '99)', ACM, New York, NY, USA, pp. 147–153.

Lerdorf, R., Tatroe, K. & MacIntyre, P. B. (2006), *Programming PHP*, 2nd edn, O'Reilly.

Leslie, D. M. (2002), Using Javadoc and XML to Produce API Reference Documentation, *in* 'Proceedings of the 20th annual international conference on Computer documentation (SIGDOC '02)', ACM, New York, NY, USA, pp. 104–109.

Lieberman, H. (1986), 'An Example Based Environment for Beginning Programmers', *Instructional Science* **14**(3), 277–292.

Massoni, T., Gheyi, R. & Borba, P. (2005), A Model-Driven Approach to Formal Refactoring, *in* 'Companion to the 20th ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '05)', ACM, New York, NY, USA, pp. 124–125.

Meservy, T. O. & Fenstermacher, K. D. (2005), 'Transforming Software Development: An MDA Road Map', *IEEE Computer* **38**(9), 52–58.

Moody, D. L. (2009), 'The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering', *IEEE Transactions on Software Engineering* **35**, 756–779.

Object Management Group (2007), OMG Unified Modeling Language (OMG UML), Superstructure, v2.1.2, Technical report.
URL: *http://www.omg.org/spec/UML/2.1.2/*

Oracle Corporation (2010), *Java Platform SE 6*.
URL: *http://download.oracle.com/javase/6/docs/api/*

PHP Group (2011), 'PHP Quality Assurance Team'.
URL: *http://qa.php.net/*

Skene, J. & Emmerich, W. (2006), Specifications, not Meta-Models, *in* 'Proceedings of the 2006 International Workshop on Global Integrated Model Management (GaMMa '06)', ACM, New York, NY, USA, pp. 47–54.

Steinberg, D., Budinsky, F., Paternostro, M. & Merks, E. (2009), *EMF: Eclipse Modeling Framework*, 2nd edn, Addison-Wesley Longman, Amsterdam.

Torchiano, M. (2002), Documenting Pattern Use in Java Programs, *in* 'Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002)', Montreal, Canada.

van Deursen, A. & Kuipers, T. (1999), 'Building Documentation Generators', *IEEE International Conference on Software Maintenance* p. 40.

W3C Group (2004*a*), OWL Web Ontology Language Reference, Technical report, W3C Recommendation 10 February 2004.
URL: *http://www.w3.org/TR/owl-ref/*

W3C Group (2004*b*), XML Schema Part 2: Datatypes Second Edition, Technical report, W3C Recommendation 28 October 2004.
URL: *http://www.w3.org/TR/xmlschema-2/*

W3C Group (2007), XSL Transformations (XSLT) Version 2.0, Technical report, W3C Recommendation 23 January 2007.
URL: *http://www.w3.org/TR/xslt20/*

W3C Group (2008), HTML 5: A vocabulary and associated APIs for HTML and XHTML, Technical report, W3C Working Draft 26 February 2008.
URL: *http://www.w3.org/html/wg/html5/*

W3C OWL Working Group (2009), OWL 2 Web Ontology Language Document Overview, Technical report, W3C Recommendation 27 October 2009.
URL: *http://www.w3.org/TR/owl2-overview/*

Wright, J. (2011), A Modelling Language for Interactive Web Applications, PhD thesis, Massey University, Palmerston North, New Zealand.

Wright, J. & Dietrich, J. (2008), Requirements for Rich Internet Application Design Methodologies, *in* 'Proceedings of the 9th International Conference on Web Information Systems Engineering (WISE 2008)', Auckland, New Zealand.

Wright, J. & Dietrich, J. (2010), Non-Monotonic Model Completion in Web Application Engineering, *in* 'Proceedings of the Doctoral Symposium at the 21st Australian Software Engineering Conference (ASWEC 2010)', Auckland, New Zealand.

Wu, Y.-C., Mar, L. W. & Jiau, H. C. (2010), CoDocent: Support API Usage with Code Example and API Documentation, *in* 'Proceedings of the 5th International Conference on Software Engineering Advances', IEEE Computer Society, Los Alamitos, CA, USA, pp. 135–140.