A Modelling Language for Interactive Web Applications

A thesis presented in partial fulfillment of the requirements for the degree of

Ph.D. in Computer Science

Jevon Wright

Massey University, Palmerston North, New Zealand

September 30, 2011

Abstract

This thesis presents the Internet Application Modelling Language (IAML), a modelling language to support the model-driven development of Rich Internet Applications (RIAs). This definition includes a visual syntax to support the graphical development of IAML model instances, and the underlying metamodel satisfies the metamodelling and viewpoint architectures of the Model-Driven Architecture.

While there are many existing modelling languages for web applications, none of these languages were found to be expressive enough to describe fundamental RIA concepts such as client-side events and user interaction. This thesis therefore presents IAML as a new language, but one which reuses existing standards where appropriate. IAML is supported by the development of a proof-of-concept reference implementation within the Eclipse framework, released under an open source license to encourage industry use, that successfully integrates a number of different model-driven technologies to demonstrate the expressiveness of the metamodel.

The IAML metamodel supports many features not found in other web application modelling languages, such as ECA rules, the expression of reusable patterns through Wires, and a metamodel core based on first-order logic. Through the implementation of the RIA benchmarking application *Ticket 2.0*, the concepts behind the design of IAML have been shown to simplify the development of real-world RIAs when compared to conventional web application frameworks.

Acknowledgments

This thesis would not have been possible without the inspiration and support of many people.

In particular, to my supervisor Jens Dietrich – thank you for all of your advice, encouragement and inspiration throughout my undergrad degree and my thesis. I am gracious to all of the patience and perseverance that you have given me throughout my time here.

To my office mates at Massey that have helped me throughout these years – Graham Jenson, Fahim Abbasi, Yuliya Bozhko and others – thank you for your feedback, your critiquing, and the innumerable breaks for caffeine, whether in coffee, chocolate or cola form.

To all of my other friends at Massey – including Patrick Rhyhart, Dilantha Punchiwewa, and Michele Wagner – thank you for helping me with the stress and time involved in the administrative side of research.

To all my bandmates – Aaron Badman, Aaron Shirriffs, Ian Luxmoore, Josh Williamson, Matt Carkeek, Steve Starr, Wayne Bryant – thank you for supporting me throughout my busy schedules, and for giving me a noisy outlet to express my creativity far away from the thesis.

To all of my family, especially my mum and dad – thank you for your ... etc

And finally to my best friend and fiancé, Krystle Chester, you have helped me and supported me through more than I could have imagined, and this thesis would not have been possible without your love, patience and mews. <3

Contents

List of Tables

TODO Remove list of tables from final thesis.

List of Figures

List of Listings

Chapter 1

Introduction

The invention and introduction of the Internet in 1969, and more significantly the World Wide Web in 1990, has had a massive and permanent influence on our lives [153]. While originally designed as a purely informational medium, the web is increasingly evolving into an application medium supporting complex software systems accessible through different devices. This evolution culminated in the development of *Rich Internet Applications* (RIAs) to unify traditional desktop applications with the distributed nature of the web [33].

1.1 Web Applications

As discussed by Berners-Lee and Fischetti [25] and illustrated in Table 1.1, the World Wide Web was originally envisaged as a medium to share documents, but the dynamic publication of content soon followed. These dynamic server-side applications became the first *web applications*, which reversed the conventional architecture of software applications at the time.

Conventional software applications were generally designed to be installed or downloaded onto a single machine, and executed locally on the *client*. Web applications are instead designed to executed remotely on a *web server*, and the client simply renders the results within a *web browser*. This approach had significant advantages: processing power could be centralised onto a single server, reducing the performance requirements of clients; software updates could be applied automatically, lowering the total cost of ownership of the software [188]; and software could now be accessed by a wider audience across a wide variety of devices.

As the popularity of the World Wide Web increased, new concepts and technologies soon followed [217]. The foundation of the World Wide Web Consortium (W3C) in 1994 standardised many of these technologies, such as HTML [290], XML [298] and CSS [300]. Over time, web applications became more distributed and independent, and *service-enabled web applications* [189] could be developed using technologies such as WSDL [**?**], REST [83] and RSS [255].

However, these conventional web applications suffered from poor responsiveness; could not be used offline; and their interfaces were not as effective as desktop interfaces [33]. This was because all interaction had to adhere to request-response cycles over relatively slow network connections. Furthermore, once an application has been implemented with a particular selection of technologies, its interpretation by the client's *web browser* may differ, exacerbated by the continuing *browser wars* between competing developers [**?**].

In the early 2000s, the use of client-side scripting to dynamically modify the web application

Year	Era	Scenarios	Standards	Platforms
1990				WorldWideWeb
1991	World Wide Web	Document sharing	HTML Tags, HTTP 0.9	
1992				
1993		Server-side scripting	CGI	Mosaic
1994				Netscape Navigator
1995		Interactivity	Java Applets, HTML 2.0	Internet Explorer
1996	Web Applications	Client-side scripting	CSS, ActiveX, Javascript	Opera
1997			HTML 4.0	
1998			ASP, CSS 2, XML	Mozilla
1999			RSS, HTML 4.01, JSP	
2000			PHP 4, XHTML, REST	
2001	Service-Enabled Web Applications	Web services	WSDL 1.1	
2002				Mozilla Firefox
2003			SOAP 1.2	Safari
2004		Semantic web	RDF, OWL	
2005		Mashups	OpenID, Javascript 1.6	Flock
2006			XMLHttpRequest	
2007	Rich Internet Applications	Push events	WSDL 2.0	iPhone, Prism
2008				Google Chrome
2009		Cloud computing	OWL 2, OCCI	Android 2.0
2010			OAuth	
2011	...	...	HTML 5, CSS 3, ...	...

Table 1.1: A Brief History of Web Application Technologies

(known as *dynamic HTML*, or DHTML) became popular, and in 2005 the term AJAX – standing for Asynchronous Javascript and XML – was introduced by Garrett [106]. By embracing this additional functionality, these richer applications attempted to reduce the impact of slow network connections and improve the interactivity of their interfaces. For example, the input validation of a text field could now occur asynchronously through background network requests, rather than waiting for validation to only occur through the request-response cycle to the server.

The term "Rich Internet Application" was first introduced in a Macromedia whitepaper in 2002, describing the unification of traditional desktop and Web applications in an attempt to leverage the advantages and overcome the drawbacks of these architectures [33]. Rather than simply providing new types of user interface elements, RIAs aim to improve all of the aspects of interactivity, accessibility and reliability in a web application.

The range of technologies to implement RIAs continued to expand, and the development of an RIA now requires the integration of dozens of different technologies and concepts [217]. Mikkonen and Taivalsaari [205] have argued that the complex and tangled structure of RIAs resembles the spaghetti programs of the 1960s and 1970s. Disciplines such as *Web Engineering* have been established to improve the quality of Web applications [153, pg. 3], but the development effort necessary to combine all of these technologies together continues to be an issue.

1.2 Modelling

By simplifying a system into an abstraction, *models* are already used extensively in software development, improving developer productivity [89]; for example, domain-specific models, relational database schemas, and programming languages are all different forms of models. System models may also be evaluated against verification tools and processes, to ensure that the model satisfies the requirements of the system, and to identify potential design or security flaws.

The application of models to web application development has shown promising results [264], with languages such as WebML [43], UWE [168] and UML [224]. In one case study, Acerbis et al. found that applications modelled in WebML only needed a third of the development effort compared to using conventional development techniques [1]. However, these existing languages (as discussed later in Section 2.4) focus on conventional web applications, and cannot model essential RIA concepts such as client-side events and user interaction [319].

In order to obtain the benefits of applying model-driven techniques to web application development – such as improved productivity, security and documentation – this thesis will investigate the development of a modelling language for RIAs. It is unclear whether the fundamental concepts of RIAs could be "bolted on" to an existing language, or whether a language will need to be designed from scratch.

1.3 Research Questions

The absence of a suitable modelling language for Rich Internet Applications, along with the increasing complexities offered today by user-oriented interactivity in rich web applications, leads to the thesis statement:

> *The development of a modelling language for interactive web applications will address many of the challenges currently faced in web development; in particular, improving the reliability, usability and security of the modelled applications, and simplifying their development and maintenance.*

This thesis statement is naturally broken down into six key research questions, which will each be addressed and highlighted throughout the course of this thesis:

1. What interactive web application concepts and technologies introduce unique challenges for the modelling of these web applications?
2. What are the shortcomings of modelling interactive web applications using existing modelling approaches?
3. Can a modelling language be developed, either from the extension of an existing language or the development of a new language, to address these challenges?
4. What techniques are available to improve the maintainability of a modelled interactive web application?
5. Can it be shown that an interactive web application modelling language increases the reliability, usability and security of the underlying system?

6. Similarly, can it be shown that such a modelling language improves the development process, in terms of speed, simplicity and consistency?

1.4 Research Design

The design of a piece of given research is often described in terms of qualitative or quantitative research, as summarised by Creswell [55]. The research design is a major consideration in the process of designing the research method and selecting the types of research questions which will be asked. Importantly, Creswell argues that these designs are not mutually exclusive, but rather that particular research questions *tend* to be more of one method than the other.

Quantitative research focuses examining relationships between well-defined variables, often using statistics, in order to investigate a research problem; these research questions are usually framed in terms of numbers and open-ended questions. *Qualitative research*, on the other hand, focuses on exploring and understanding the meaning of a problem creatively, with answers proposed through induction. These questions are usually framed in terms of words and closed-ended questions.

There appears to be little or no previous research or theory explicitly into the design of RIAs. While two very similar research areas – web applications, and – have been the focus of extensive research, there is little work in unifying these two areas. In this research, a qualitative research design is used, as the research problem is closely characterised as a qualitative research problem, as discussed by Creswell [55] and Morse [211]:

1. *The concept is "immature" due to a conspicuous lack of theory and previous research.* While there is a large body of research, and to a lesser extent theory, into the related topics of Internet applications and hypermedia, the body of work for RIAs is significantly less.

2. *A notation that the available theory may be inaccurate, inappropriate, incorrect, or biased.* The only available theory on RIAs are derived from the component topics of web applications and hypermedia, and each of these topics fail to consider fundamental concepts from the other.

3. *A need exists to explore and describe the phenomena and to develop theory.* By proposing a modelling language for RIAs, we will be attempting to identify the core concepts and constructs of RIAs. These proposals will undoubtedly help in developing a theory for RIAs as a whole, although this is outside the scope of this thesis.

4. *The nature of the phenomenon may not be suited to quantitative measures.* The development of modelling languages is difficult and expensive, and their success is strongly linked to the success of its implementation [89]. Evaluating languages are also difficult and expensive, and it would be difficult to provide a large enough sample size for conclusive evaluations.

However, it is also important to note that in the process of addressing the thesis statement qualitatively, a number of experiments were also performed quantitatively. The results from these experiments – such as evaluating the performance of model completion discussed by Wright and Dietrich [320], and the implementation metrics discussed later in Section 2.6 – can then be used to support the thesis statement.

1.5 Research Method

While keeping the selected research design of qualitative research in mind, a detailed research method can be proposed. For this research, this method consisted of the following steps, which are each discussed in detail in the body of this thesis. As part of the research process, the outcomes of a number of these steps have been published and peer-reviewed externally.

1. Identify the requirements and features of Rich Internet Applications; identify the requirements and existing work into developing other modelling languages, including strengths and weaknesses and existing evaluations; and outline existing future work in this area. From this research, the decision whether to extend an existing language, or develop a new language from scratch, can be made. These requirements have been published earlier by Wright and Dietrich [319].

2. Investigate and propose a number of methods in which to evaluate the proposed modelling language; this could include methods such as quality criteria, language metrics, expressibility checklists, user evaluations, and so on. In this thesis, one such method resulted in the development of the benchmarking application *Ticket 2.0*, which has been published earlier by Wright and Dietrich [318].

3. Investigate methods in simplifying the effort necessary for a model developer to develop and maintain a model-driven implementation, possibly through the use of frameworks, inference or conventions. Some of these results have been published earlier by Wright and Dietrich [320].

4. Perform an extensive amount of prototyping and exploratory design work to adapt existing metamodels, propose new modelling constructs, and experiment with visual syntax. The knowledge gained in this prototyping stage will be used to inform the intended design.

5. From this top-down design, develop a proof-of-concept CASE tool and incrementally implement this software following an evolutionary software process model. Follow agile development processes to improve feedback and evaluation, and to adapt to technology changes in the RIA industry. This tool is available online under the terms of the Eclipse Public License [70] at `http://code.google.com/p/iaml/`.

6. During this implementation stage, frequently evaluate the implementation against the final requirements and intended design. Refactor or remove elements from the implementation as appropriate.

7. Once this implementation is complete, evaluate the design implementation against the previously defined evaluation criteria, to understand the strengths and weaknesses of this particular approach.

8. Investigate a wide range of formal verification methods, and prototype each method with the proof-of-concept implementation of the proposed modelling language. The subsequent evaluations may be used to discuss the suitability of supporting formal verification directly with the modelling language itself.

1.6 Thesis Outline

The remainder of this thesis follows the sequential order of the proposed research method in the previous section, and will be summarised here.

Chapter 2 discusses the state of the art and related work in the field of modelling RIAs. Two suites of modelling language requirements are discussed, and a benchmarking application called *Ticket 2.0* is introduced. The chapter concludes with a short discussion on software engineering methods, one which will be used in the proof-of-concept implementation.

Chapter 3 provides an in-depth investigation into the many aspects and concepts involved in system modelling, such as domain-specific languages, model-driven development, visual modelling and model verification. The technology of *model completion* is introduced and defined, which is used later to simplify the end-user development effort of the proposed language.

Chapter 4 provides an in-depth discussion into all of the fundamental RIA concepts that will be considered as part of the proposed modelling language. For each concept, a variety of technologies used to implement the concept are investigated, and their relevant benefits and drawbacks are discussed. The chapter concludes with a discussion on visual modelling techniques and best practices.

Chapter 5 details the design, approach and rationale behind the proposed modelling language, named the *Internet Application Modelling Language* (IAML). For each fundamental RIA concept, a set of modelling constructs are proposed, along with their rationale and semantics. The semantics of the inference rules and platform-specific implementation notes are instead provided in Appendix **??**.

Chapter 6 details the evaluation process used to select the implementation technologies that form the basis of the proof-of-concept implementation in the Eclipse framework.

Chapter 7 then provides a detailed decomposition of the proof-of-concept implementation in terms of the components necessary to implement the final system.

Chapter 8 evaluates the proof-of-concept implementation of IAML against the evaluation criteria discussed in the Background chapter, along with the evaluation of the implementation of the *Ticket 2.0* benchmarking application.

Chapter 9 concludes the thesis with a discussion on the overall research contributions and outlines areas of future research.

This thesis also consists of a number of appendices, including:

Appendix ?? lists the existing Rich Internet Applications used to identify the overall categories of RIA requirements in Chapter 2.

Appendix ?? details the suite of 69 use cases developed as part of the process to identify the detailed requirements of RIAs in Chapter 2.

Appendix ?? illustrates the architecture and complexity of implementing the *infinitely redirects* model verification constraint using the NuSMV model verifier, as discussed in Section 6.6.8.

Appendix ?? presents the full metamodel documentation of the IAML language, inference rules and platform-specific implementation notes, as generated by *ModelDoc* from the source code of the language.

Appendix ?? illustrates the incomplete implementation of *Ticket 2.0* using IAML, and is provided in both a graphical format, and the underlying XMI representation of the model.

Appendix ?? describes the content of the media attached to this thesis. This media includes a redistributable copy of the proof-of-concept IAML modelling environment, along with its full source code, and copies of the XMI model instances used throughout this thesis.

Chapter 2

Background

In this chapter, the key concepts and topics behind this research project will be discussed in further detail. The current state of the art in web application modelling research will be presented, along with a suite of requirements and evaluation techniques that have been developed to evaluate these modelling languages.

2.1 Rich Internet Applications

In the previous chapter, a brief history of web applications and RIAs were introduced. In this section, some specific details of RIAs will be discussed, including a summary of technologies that may be used to implement an RIA; implementation classifications; a definition of what *interactivity* actually means; and on the impact of enterprise-level software.

2.1.1 Technologies of Rich Internet Applications

Kappel et al. defines a *web application* as "a software system based on technologies and standards of the World Wide Web Consortium (W3C) that provides Web specific resources such as content and services through a user interface, the Web browser" [153, pg. 2], and this definition is used in this thesis. The underlying architecture of the web application considerably influences its quality, and many potential architectures exist as discussed by Eichinger [76]; for the nature of this thesis, the web application architecture illustrated in Figure 2.1 will be assumed.

As discussed by Nussbaumer and Gaedke [217], the development of a web application involves the particular combination of a number of components and technologies within a chosen architecture, and may include:

1. *Presentation technologies* such as various HTML dialects [290, **?**, 299] or CSS [300];
2. *Server-side languages* such as C/C++, PHP, Java, Ruby, Perl, Python, or Haskell;
3. *Client/server communication protocols* such as SMTP, HTTP, cookies, or sessions;
4. *Configuration file formats* such as XML [298], YML [240], JSON [56] or flat text files;
5. *Application platforms* such as Flash [60], Silverlight [313], J2ME [282] or Java Applets [61];
6. *Web service languages* such as SOAP [296], REST [83], XML-RPC [314], or WSDL [**?**]; and

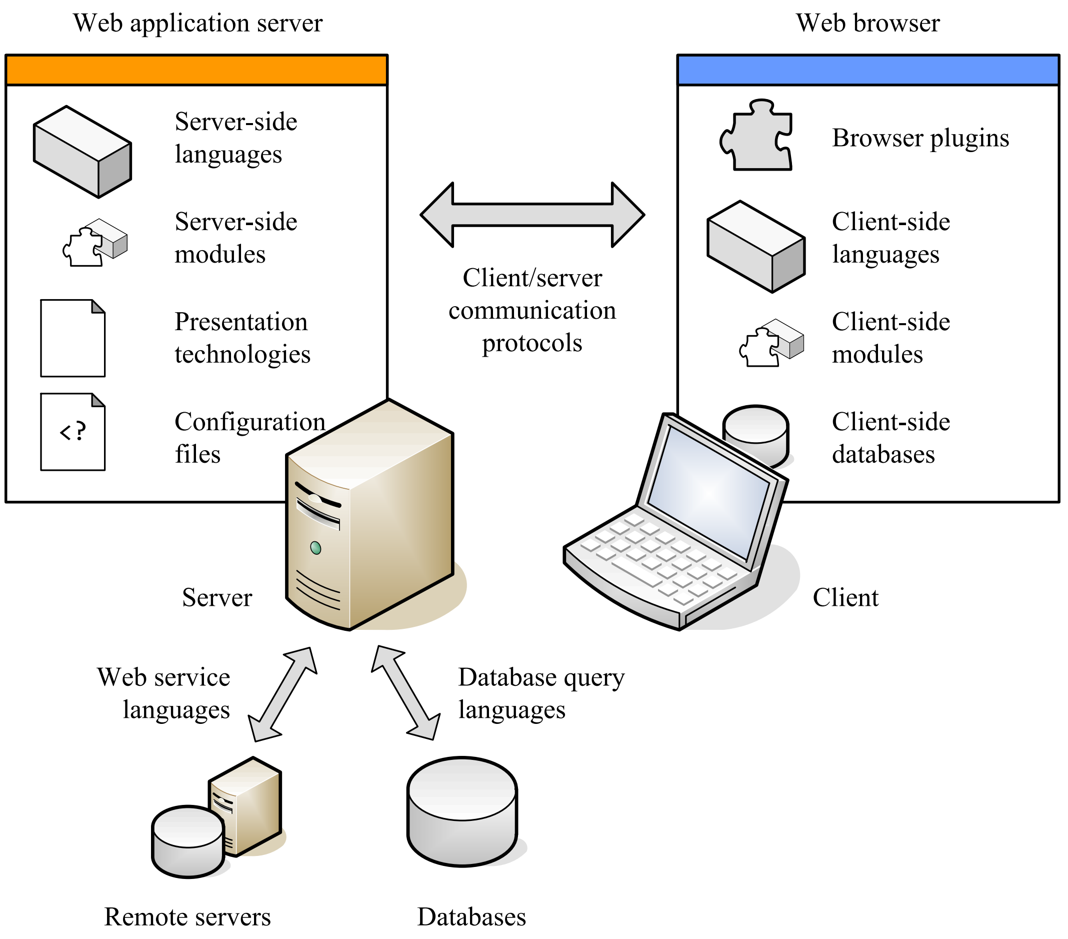


Figure 2.1: A reasonable architecture for Rich Internet Applications

7. *Database query languages* such as versions of SQL (MySQL, PostgreSQL, NoSQL and others), OQL, or LINQ.

One of the reasons that web application development is so complex is simply dealing with the combinatorial nature of these technologies. For example, the choice of a *content markup language* – between HTML, XHTML and HTML 5 – will dictate the results of applying an instance of a CSS stylesheet, and some web browsers do not implement these content markup languages correctly.

The *Acid* series of web browser tests were created in an effort to improve the compliance of web browsers to standards [35, **?**], but many still fail to fully implement these standards correctly, hindering web development [**?**]. The choice of architecture also has a significant impact on the scalability and reliability of deploying the application [40].

Rich Internet Applications introduce three new categories of technologies and platforms – client-side scripting languages, modules and databases – to this already significant repository of potential technologies, as illustrated in Figure 2.1. In particular, a RIA will be implemented[1] using some of the conventional web application technologies discussed earlier, along with any number of the following additional technologies:

1. *Client-side scripting languages* such as Javascript [74], or VBScript [161];

[1]As an example, Section **??** discusses the implementation of a standard RIA which required the use of 16 different technologies.

2. *Client-side modules* such as CommonJS [173]; and

3. *Client-side databases* such as Google Gears [112] or HTML 5 [299].

This further increases the complexity of interactions between all of the components. For example, Flash is commonly used to play video content, but is not available on the iPhone platform [309]; however, older browsers cannot support the new video features of HTML 5, so some form of backwards compatibility must be supported by the RIA.

2.1.2 Classifications of Rich Internet Applications

While the individual technologies and platforms selected for the implementation of a particular RIA may differ, the architectures of these applications may be classified into four categories, as described by Bozzon et al. [33]:

1. *Scripting-based*, in which client-side logic is implemented via scripting languages, and interfaces are based on presentation technologies such as HTML and CSS;

2. *Plugin-based*, where browser plugins integrate application platforms into the browser itself, such as Flash, Silverlight or Java Applets;

3. *Browser-based*, where client-side interactivity is natively supported by the browser in a browser-specific way, such as ActiveX [105] and XUL [32]; and

4. *Web-based desktop technologies*, where software applications are downloaded and executed over the Internet, but executed using a different platform, such as Java Web Start [191] and Adobe Air [2].

Since this work by Bozzon et al. was published, a fifth category – *Desktop-based web applications* – has emerged. These are RIAs that appear to be native applications to the end-user and are executed away from the web browser, but are actually standard RIAs within a browser wrapper [**?**]. These technologies include Mozilla Prism [213] and Fluid [67].

However, these categories are not mutually exclusive; it is possible (and in many cases necessary) to have a RIA that covers multiple categories. For example, a client-side application (*scripting-based*) could also need to interact with the Google Gears plugin [112] (*plugin-based*) in order to emulate offline functionality.

Some related research in describing interactive web applications adapts research from *hypermedia* applications, which refers to the mixture of hypertext and multimedia [241, 242], although hypermedia usually refers to standalone proprietary user interfaces [17]. Web applications can be considered low-level hypermedia applications [175]; however web applications adds unique challenges to system requirements – such as security, performance, usability, navigability and internationalisation [108] – that existing surveys tend to neglect.

2.1.3 Interactivity

The term "interactivity" can be a controversial term, as discussed by Gane and Beer [104]. Interactivity does not refer to a list of requirements of an interface, but rather is a variable measure of the ways in which a system supports or requires interaction. Almost all types of media are interactive; Manovich

[190] argues that even sculpture and architecture are interactive media, because the demand the viewer to move to appreciate the structure.

Consequently, it is not possible to compare the interactivity of two different media types, but the term rather refers to *differences* in the way a system is presented. The consensus seems to be that increased forms of interactivity are beneficial, and provide more usable and effective user interfaces [243]. However, it is widely accepted that conventional web applications have *less* interactivity than Rich Internet Applications [33, 242, 319].

2.1.4 Enterprise Software

Instances of software can be categorised according to their challenges, development complexities and intended domain [153, pg. 5–7], and certain applications may be categorised as *enterprise software*. There is no precise definition of an "enterprise application", however they often possess common characteristics and design goals.

Some of the characteristics of enterprise software, in terms of functional and non-functional requirements, are discussed by Fowler [88]. These characteristics include needing to persistently store a lot of data; supporting concurrent data access; handling many user interface screens; expressing complex business logic; integrating with other enterprise applications; and supporting the reuse of data in semantically diverse ways.

Enterprise applications must also possess "reasonable performance characteristics". This non-functional requirement is often measured using metrics such as scalability, latency and throughput. To improve the success rate of developing enterprise applications, certain application design patterns should be used [88].

2.2 Frameworks

A practical solution to bridging the conceptual gap in web development is through the use of a framework, which is "a reusable, 'semi-complete' application that can be specialised to produce custom applications" [80]. By taking a pragmatic approach, frameworks are popular in industry due to rapid development time and stability. The distinctions between frameworks, programming languages and modelling languages are discussed later in Section 3.1.11.

The Ruby on Rails framework [280] has arguably been one of the most influential frameworks for the development of web applications, which inspired the Symfony for PHP [240] framework. Other popular server-side frameworks include Struts for JSP [7]; Seaside for Smalltalk [78]; Spring for Java [150]; and Spring.NET for ASP.Net [274].

The common tasks faced by the client-side scripting aspect of RIAs has also been addressed through a number of client-side frameworks, such as JQuery [181] and the Prototype Javascript Framework [246]. Recently, some web application frameworks such as Google Web Toolkit [129] and Symfony [240] have combined both server-side and client-side functionality, to provide the first *Rich Internet Application frameworks*.

Frameworks solve a number of problems in complex software development. They often provide or generate common infrastructural source code[2] specific to a particular domain; for example, Symfony can automatically generate object-relational mapping source code from a defined Propel schema [240].

[2]This is often called *boilerplate code*, referring to how the code can often be included in new contexts without requiring significant change.

They can also provide an abstraction layer upon a number of different technologies; for example, PDO for PHP [180] provides database abstraction to ten different relational database managers.

Finally, client-side frameworks can abstract away differences in browser implementations; for example, the Prototype Javascript Framework provides a common *Ajax* component, which performs identically across different browsers [246]. In these ways, frameworks can adapt to the fragmentation of different technologies, which is especially pronounced in the field of RIA development.

Extending a framework to work in different problem domains is a significant issue; in many cases, additional complexity must be introduced into the framework, or the framework may need to be re-architectured. The runtime overhead of using a framework as opposed to a native application is another concern, due to the large libraries that they require as illustrated later in Section **??**.

The distinction between a framework and a programming language is generally well-defined. While both aspects provide a level of abstraction, a programming language usually has its own syntax, semantics and implementation into a lower target language (e.g. through compilation) [262]; whereas a framework reuses (or extends) the syntax and implementation of existing programming languages. Similarly, the distinction between a framework and a modelling language is also generally defined in the same way; that is, a modelling language has its own syntax, semantics and optional implementation, whereas a framework reuses an existing programming language.

A framework is therefore ideal for providing a domain-specific abstraction within an existing general-purpose software development environment, whereas a modelling language is ideal for providing the same abstraction in a platform-independent manner. In both cases, however, a framework can be used as part of the implementation of either a programming language or a modelling language; although Kelly and Pohjonen note that many modelling languages suffer the problem of adhering too closely to a particular framework [157].

2.3 Features of Rich Internet Applications

In order to effectively evaluate existing approaches in modelling RIAs, comprehensive sets of evaluation criteria need to be defined. In this research, two sets of evaluation criteria were defined. The first set of criteria define a general list of feature categories that can be evaluated subjectively against a modelling approach, as defined by Wright and Dietrich in 2008 [319]. The second set of criteria focuses solely on the functionality of modelling approaches to define a comprehensive list of modelling requirements, as defined by Wright and Dietrich in 2008 [318].

2.3.1 Feature Categories

An informal evaluation process was used to identify the general feature categories of modelling RIAs [319]. This process involved the general overview of some of the first popular Rich Internet Applications of the time, to identify general categories. The results of this evaluation was the identification of thirteen feature categories, which will now be briefly summarised in this section.

Events

Web applications are inherently event-driven, as web servers respond to incoming request events from clients. Events are particularly important to RIAs as they are more sensitive to interactivity and asynchronous events, and events can either occur locally (on the client) or remotely (on the server). Remote events permit a client to communicate with the server and with other clients, and the server can communicate directly to the client (e.g. through a pushlet).

Since events are such a fundamental part of the web, it may be useful to promote events as *first-class citizens*[3] in a RIA modelling language. A RIA modelling language should be able to describe events, which may in turn conditionally execute defined actions.

Browser Interaction

The web browser is the client-side user interface of a web application [153], and consequently must be considered when modelling RIAs. A RIA modelling language should be able to model navigation concepts, such as the "back" button or out-of-order navigation; storing data on the client through cookies or offline databases; opening additional windows; identifying available scripting and plugins support; and being able to identify the user agent, and being able to respond appropriately [319].

Lifecycle Management

The events and conditions surrounding the creation, termination, and state changes of a software element can be expressed as its *lifecycle*. Lifecycles are more than just firing events, and can add contextual information to fired events, or group similar events in a well-defined way. While lifecycles are not necessary to implement an RIA, the concept can simplify the development of certain types of web application. For example, a JSP servet may trigger events when the servlet is initialised or destroyed (the `init` and `destroy` methods) [139]; likewise, a browser window may trigger events when it is loaded or unloaded (the `onload` and `onunload` handlers) [290].

Users

It is important to identify different web application users, identified using particular credentials with respect to a certain access policy. *Users* are therefore an important part of web applications, with the term *user* referring to both people and non-human entities[4]. Most web applications deal with the creation, management and authentication of user accounts, so a RIA modelling language should also promote users to first-class citizens. RIAs may also support collaborative interaction between different users.

Security

The security of a web application should be an inherent property of a designed application. Sensitive or private data should not be unnecessarily disclosed, and should be unavailable to those without appropriate permissions; likewise, it should not be possible for a malicious user to corrupt or otherwise circumvent the intentions of the application developer.

Security concerns include both authentication and authorisation, two distinct processes that are sometimes mistaken [201]. Authentication involves validating the identity of a certain party, often in order to permit them access. Authorisation, on the other hand, is the definition of which rights and permissions a certain party will have once access is granted. Open standards for distributed

[3]A *first-class citizen* represents an element that can be represented indirectly by a variable or expression, as opposed to a second-class citizen, which can only be used directly [277].

[4]For example, software that utilises a web service can also be considered a *user*.

authentication and authorisation on the web have recently been announced, such as the OpenID [247] and OAuth [127] standards, respectively.

Databases

The vast majority of web applications are concerned with the integration with databases, in essence becoming *data-intensive web applications* [44, 43]. Abstraction of data access to a persistent domain model can increase the database-independence, reliability and development productivity of the application [240], and a wide variety of existing domain models – such as UML class diagrams [224] and ER diagrams [46, 244] – already exist. A RIA modelling approach should also support the common use cases of users uploading content, and describing offline functionality provided by offline toolkits, discussed earlier in Section **??**.

Messaging

Messaging encompasses the entire concept of sending contextual data from one device to another. In web applications, e-mail was once the most popular and accessible form of messaging, but a rapid increase in communication mediums have introduced many new ways of communicating. In RIAs, this includes domains such as sending e-mails; sending short text messages to mobile phones; invoking and responding to web services; using RSS feeds and public APIs; and using OpenID authentication.

User Interfaces

RIAs provide an extended suite of user interface elements to web browsers, such as maps, calendars and autocompletion [318]. A modelling language for RIAs should support a wide suite of existing and new user interface elements. The integration of the presentation and underlying models of a RIA tend to remain strongly correlated [8], often following best practice architectures such as the Model-View-Controller (MVC) design pattern [170]. A RIA modelling approach should therefore support these best practices, yet simultaneously not be restricted to a single architecture.

Standards Support

Innovations on the web occur very quickly and frequently [217, pg. 111], and the most popular innovations codified and shared through standards, such as the W3C standardisation of HTML [290] and DOM [292, 291], and the ECMA standardisation of Javascript [86]. To improve acceptance by the web development community, a RIA modelling approach should embrace new innovations where possible, but overwhelmingly adhere to existing standards.

Platform Independence

As discussed earlier, a web application may be implemented on any number and combination of technologies. Since a model is an abstraction, a RIA modelling approach should not be limited to a single combination; the model should be *platform-independent* to permit deployment on any number of platforms. A platform-independent model also allows for incorrectly implemented standards on certain platforms to be ignored, as workarounds can be deployed automatically. Platform independence also adheres to the viewpoint modelling architecture advocated by the OMG, as discussed later in Section 3.1.5.

Use of Metamodels

A modelling approach should consider itself as part of a larger model-driven development community. The Model-Driven Architecture (MDA), for example, discusses the interaction between models, meta-models and the real world [69]. These model-driven standards aim to improve the reliability and interchangeability of different model instances. A full discussion on modelling is provided in Chapter 3.

Verification

It is possible to describe a model instance that is valid according to the syntax of a modelling language, but unusable according to informal standards outside of the language definition. For example, deadlocks may be expressed in most procedural programming languages, even though this is generally not desirable in software applications [131]. It is preferable to identify these errors as early as possible, as *defect amplification* dramatically increases the relative cost of correcting errors once they have been made [244, pg. 197–205].

Verification allows the analysis of such model instances to identify these problems, and this is discussed in greater detail in Chapter **??**. Verification is already used against conventional web applications for scenarios such as detecting dead links or unreachable code [276], but there is little research in using verification against RIAs. For example, some RIAs allow multiple users to work concurrently on a single document; a verification tool to locate potential deadlock and resource starvation code may be beneficial. Consequently, a RIA modelling language should investigate integrating these checks into its implementation.

Software Support

Finally, a RIA modelling language should be supported by software tools, such as a dedicated CASE tool for designing model instances, code generators, or analysis tools [263]. Proof-of-concept implementations of a language can be used as a reference implementation for further work, and can increase language acceptance by the development community [89]. For model-driven approaches, Kent [158] argues that a software implementation is essential in order to maximise the benefits of using models. However, a single software implementation can be at odds with keeping the language platform-independent; to an extent, platform- or technology-dependent implementation concerns should be prevented from modifying the design of the modelling language [157].

2.3.2 Detailed Modelling Requirements

While these feature categories are useful for quickly evaluating a RIA modelling approach, their subjective nature is not useful in performing a detailed evaluation, or to specifically identify missing functionality. Subsequently, the functionality of seven popular Rich Internet Applications were investigated in detail, as discussed in Appendix **??**. Each application was investigated in order to identify the individual features that are a result of the RIA domain, rather than from business or design decisions. For each requirement, a *use case* was filled out, describing the actors, sequence, pre- and post-conditions, exceptions, related use cases and other interesting comments [244].

From this survey, 69 use cases of RIA features were identified; this suite of use cases are reproduced here in Appendix **??**. These use cases were then summarised into a suite of 59 individual

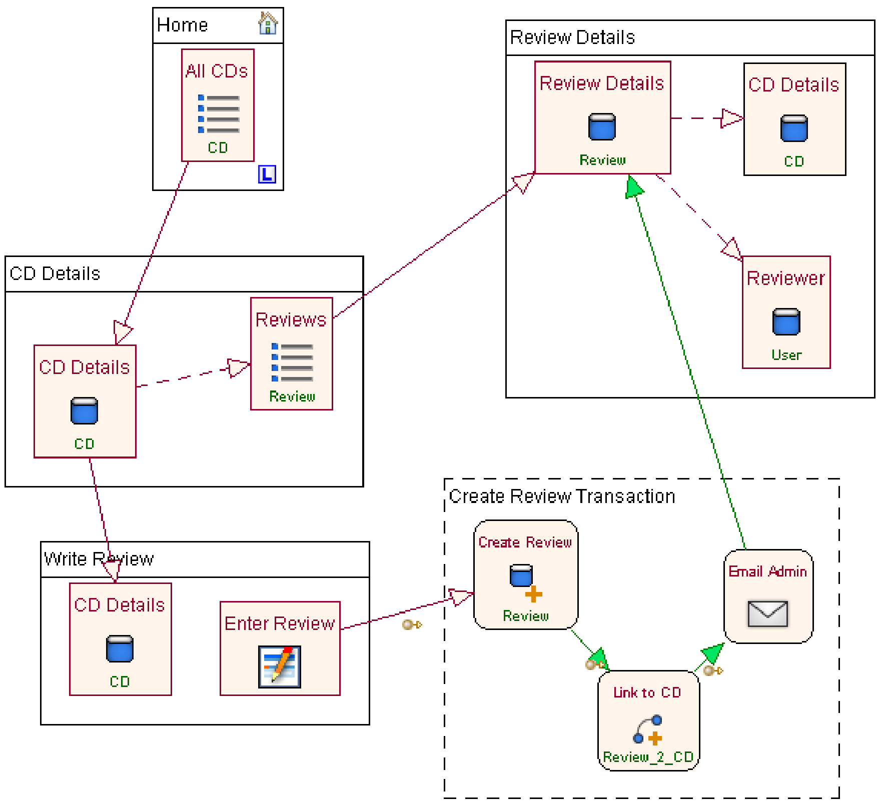


Figure 2.2: Hypertext Model of a simple WebML application

functionality requirements, as described in Wright and Dietrich [318] and republished here in Appendix **??**. This suite of detailed requirements may then be used to perform a detailed evaluation of a modelling language in terms of its modelling functionality, without having to rely on subjective measurements.

2.4 Existing Web Application Modelling Languages

Web application modelling has been extensively researched, and many reviews and surveys of existing approaches exist. Selmi et al. [269] provides an excellent survey of some of the fundamental problems with existing approaches. Other reviews concern themselves with evaluating the functional requirements of languages [123, 8], and consistently find that existing languages are inadequate due to deficiencies in expressibility, usability or implementation support. A discussion on the suitability of model-driven approaches is discussed later in Section 3.1.

In this section, a range of existing RIA modelling approaches will be briefly discussed. This evaluation had been performed earlier by Wright and Dietrich [319], which identified that these existing languages tend to either be abandoned or poorly implemented. This section will instead focus on new research that has emerged since this evaluation was published, with regards to RIA modelling techniques.

2.4.1 WebML

WebML is a well-researched and commercialised approach to modelling data-intensive web applications [43]. Web applications are described using five models, of which the *hypertext model* (the combination of the *composition* and *navigation* models) is the most important. In Figure 2.2, a simple web application for publishing CD reviews is illustrated. These models are combined together using code generators and custom XSLT templates [297] to generate the final Java-based application, which can then be published onto a web server.

The language is well-supported with a CASE tool called *WebRatio*, which is still actively developed but at the time of writing, has not been extended to implement new modelling approaches, such as recent client/server extensions to the model [91]. This CASE tool is commercialised and closed-source which hinders extensibility, although there is some support for plugins. With regards to supporting RIAs, it appears that WebML is focused on supporting conventional data-driven web applications, and instead attempts to "bolt on" RIA support to the WebML model, instead of using RIA concepts fundamentally.

Wright and Dietrich [319] found that WebML was the most complete modelling language for web applications, yet lacked support for many of the fundamental concepts of RIAs, such as object lifecycles, user interface modelling, and controlling the browser. Following the publication of this review, the WebML authors agreed that WebML was lacking these important features [91].

2.4.2 UML

UML is a general-purpose modelling language for the analysis, design and implementation of software systems and other similar processes [224], which can include web applications and RIAs. However, this flexibility impacts on its suitability as a RIA modelling language. UML model instances are often verbose and extremely informal, with a lot of domain knowledge assumed. Common web concepts such as sessions, timed events and e-mails are difficult to describe with standard UML, and extensions are usually required.

As UML does not have a defined set of formal semantics and implementation rules [99], it is not possible to take an arbitrary UML model instance and translate it into a functional piece of software. This negatively impacts the suitability of UML as a modelling language for RIAs, as UML therefore needs to be extended – which is likely to spawn platform-dependent UML model instances that cannot integrate with other UML model instances. Some authors argue that UML should not capture code-level semantics, such as Fu et al. [99]; however, this viewpoint suggests that UML is therefore strictly for documentation-purposes only, negating many of the benefits of model-driven approaches.

The OMG recommends extending UML through the use of three key extension mechanisms; stereotypes, tagged values, and constraints [36]. In particular, a *stereotype* is defined as "a kind of *Class* that extends *Classes* through *Extensions*" which are applied to an existing metaclass[5] [224, pg. 670], allowing domain-specific terminology to be provided against a base UML model. The formal combination of a particular set of extensions can be published as a single *UML Profile* [100]. In practice, only the stereotypes of a UML Profile are supported by existing CASE tools [261], and constraints defined by UML Profiles are poorly supported.

UML's extensibility is entirely additive; you cannot remove existing UML notations or create new operations, without creating an entirely new UML derivative language [36]. Bruck and Hussey argue that DSLs provide more precision and less complexity, and recommend extending UML only if the majority of your concepts easily map onto existing UML concepts [36].

Nevertheless, one UML extension, UWE – discussed in the next section – has emerged as a

[5]The UML term *metaclass* may also be known as the *defining metamodel class for a given instance of a metamodel element*.

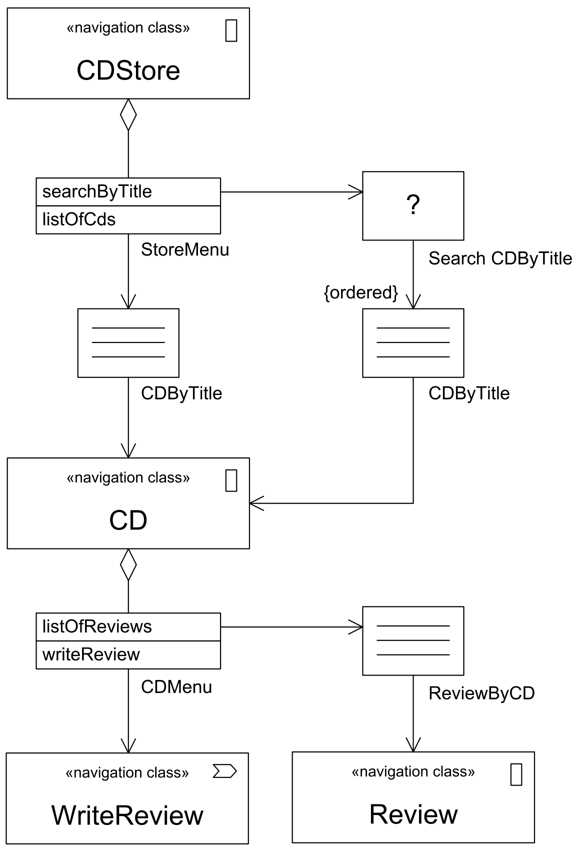


Figure 2.3: Navigation Structure model of a simple UWE application, adapted from Koch [168]

promising candidate for modelling web applications. Another extension of UML by Conallen [51] also attempts to extend UML to describe web applications, and can translate the model into a functional web application; however this approach is very limited and dated, compared to UWE. A third extension, WUML [154], attempts to add models to describe web concepts, such as events and user profiles; these models appear to be intended for documentation and extensions, rather than describing applications themselves.

2.4.3 UWE

UML-based Web Engineering [169] is a web application modelling language extension to UML, following the UML extensibility requirements [36], which has a heavy focus on using modelling standards. UWE uses more types of models than WebML[6], but advocates using automatic and semi-automatic tools to assist the developer in constructing these models[7].

Compared to WebML, the model instances are often more verbose but the individual model elements are simpler; this means the models are easier to understand, but less efficient to develop large applications. In Figure 2.3, the *navigation structure model* of the same CD reviewing application is illustrated. UWE definitely has potential to be a powerful web application modelling language, however its existing focus is on conventional web applications.

[6]UWE uses at least eight models [168]: requirements, content, architecture, navigation, process, business logic, architecture and integration models; whereas WebML uses at least three models [43, **?**]: data, hypertext, and presentation models.

[7]Using model transformation technology such as QVT and ATL, as discussed later in Section **??**.

```
define page editUser (u : User) {
  title { "Edit User: " output(u.name) }
  section {
    header { "Edit User: " output(u.name) }
    form {
      par { "Name: " input(u.name) }
      par { "Password: " input(u.password) }
      par { action("Save Changes", saveUser()) }
    }
    action saveUser() {
      u.persist(); return viewUser(u);
    }
    navigate(home()) { "return to home page" }
  }
}
```

Listing 1: Part of a web application implemented in WebDSL, adapted from Groenewegen et al. [117]

Wright and Dietrich [319] found that UWE had no support for common web concepts such as e-mails and browser identification. A significant extension to UWE is therefore necessary to support basic web concepts, which would then need to be extended again to support RIA concepts; but these extension would be vastly simplified due to its adherence to model-driven standards. The language's openness, automatic developer processes, clean models, and standards-based approach is promising for future extensibility.

2.4.4 WebDSL

WebDSL is a DSL for implementing data-driven web applications with a rich data model, with limited support for user interface modelling [117]. It can support a wide range of security models – including most of the security models discussed later in Section 4.8 – although these policies have to be individually implemented [118]. No visual model has been developed, so instances are represented textually, as in Listing 1.

Web applications defined in WebDSL may then be generated into source code through the model transformation language *Stratego/XT*. Language extensions also extend the code generator by providing plugins upon the base language. At the time of writing, WebDSL is still fairly recent and lacks supports for RIA concepts such as client-side scripts and browser control, making it unsuitable in modelling RIAs.

2.4.5 Web Information Systems

Rather than providing a visual model, work has been done on describing web applications from a more formal perspective, with roots in algebra and graph theory. Schewe [260] describes *Web Information Systems* (WIS) as a triplet of issues: content, navigation and presentation, with each issue described using strict notation. Like most existing approaches, WIS lacks modelling support for most web concepts, and cannot handle the interactivity of RIAs.

The argued benefit for using formal models is that it simplifies formal verification of the model; but the line between formal and informal models continues to blur, as modelling languages (and instances)

Feature Category	WebML	UWE	W2000	OOWS	OOHDM	Araneus
Events	Some	-	Poor	-	-	-
Browser Interaction	Poor	-	-	-	-	-
Lifecycle Management	Poor	Good	Poor	-	-	-
Users	Good	Poor	Poor	Poor	-	-
Security	Some	Some	Poor	-	-	-
Database Support	Good	Some	Poor	Poor	Poor	Poor
Messaging	Good	Poor	Some	-	-	-
UI Modelling	Poor	Some	Some	Poor	Some	Poor
Platform Independence	Excellent	Excellent	Good	Excellent	Good	Some
Standards Support	Poor	Excellent	Excellent	Some	Poor	-
Use of Metamodels	Poor	Excellent	Excellent	Poor	-	-
Verification	Some	Some	-	-	-	Poor
Software Support	Good	Some	Poor	Some	-	Some

Table 2.1: Existing modelling language support for the general feature categories of modelling Rich Internet Applications, adapted from Wright and Dietrich [319]

Rating	Concept Support
-	No support at all
Poor	Very limited support, difficult to implement
Some	Some support, some aspects cannot be implemented
Good	Most aspects can be implemented with ease
Excellent	Ideal implementation of the concept

Table 2.2: Subjective measurements legend, as used in Table 2.1

are increasingly using formal syntax and semantics in their own definition [166]. By themselves, formal models are difficult for developers to understand or utilise, so any formal modelling approach requires tool support in order to make any impact [41].

2.4.6 Older Languages

There is a vast selection of other modelling approaches to web applications, and each of these have been similarly investigated as to the feature requirements of modelling RIAs. Wright and Dietrich [319] investigates the languages W2000 [13], OOWS [238], OOHDM [253] and Araneus [198] and finds that these languages are particularly unsuitable for modelling RIAs. All of these older languages have been found to lack support for many of the fundamental concepts in web applications [242, 269], and many are currently out-of-date or poorly maintained. These approaches will therefore not be considered in any further detail in this thesis.

2.4.7 Evaluation

Wright and Dietrich [319] critically evaluated many of these languages against the general feature categories of RIAs introduced earlier in Section 2.3.1, with respect to a subjective ranking. The results of these evaluations are summarised here in Table 2.1, as per the subjective ranking legend of Table 2.2.

This review found that the majority of approaches have limited existing support for standard web application concepts that should be supported already, such as sessions[8], events, messaging and security. These languages are therefore unsuitable as candidates for an extension to support RIAs, as significant effort must first be expended to address *conventional* web applications. Existing approaches do not directly support events or interactive functionality, but instead provide this functionality through the use of user interface libraries. Fundamental web concepts such as users and security were often ignored, and very few approaches directly investigated the use of verification techniques.

Along with the existing reviews discussed earlier at the beginning of this section, one can safely state that no existing modelling language satisfies the requirements for a RIA modelling language as defined by Wright and Dietrich [319]. In particular, there is an obvious conceptual gap between the high-level web system concepts and the low-level technologies required to support its implementation. This gap is highlighted by the arrival of new technologies, which add additional complexity that needs to be simplified.

2.5 Benchmarking Application

While the basic feature evaluation discussed in the previous section is useful to obtain a brief overview of a RIA modelling language, a more detailed functionality-based evaluation is necessary in order to evaluate the expressiveness of these different approaches with greater accuracy. Importantly, these evaluations may be compared objectively, and will also highlight any specific functionality deficiencies of a particular approach.

Functional *benchmarking applications* may be useful in identifying and validating the expressiveness of different technologies, and have been used in many computing domains such as business rules and server-side performance [110, 310]. These applications allow different technologies to implement an agreed-upon concept, and the final implementation in each technology can be evaluated against metrics such as size, performance or development cost.

Wright and Dietrich [318] propose a new benchmarking application named *Ticket 2.0* as a functional benchmarking application for the evaluation of RIA modelling languages. The 59 detailed functionality requirements introduced earlier in Section 2.3.2 were each included as functionality within the requirements for a RIA, within the domain of a social networking-enabled, event ticketing application. As described in the paper:

> "Its business goal is to provide a rich interface for users to browse upcoming events and book tickets using a credit card. They may interact with other users on the site through friends lists and chat rooms on the event detail pages themselves, permitting open discussions and user interaction. It also aims to provide a unified interface for event managers, allowing them to schedule upcoming events and track their progress." [318]

A conceptual overview of the intended application is provided in Appendix **??**. For the full description of the requirements, and a mapping from the defined requirements to the features of the application itself, the interested reader is referred to the published paper. *Ticket 2.0* may also be used to compare other forms of RIAs, such as conventional web application frameworks and general-purpose languages.

[8] A *session* is "a sequence of related HTTP requests between a specific user and a server within a specific time window" [217, pg. 114].

2.6 Metrics

Measurement is as essential part of software engineering, as discussed by Pressman [244]. Park et al. [237] argue that software measurement allows the characterisation, evaluation, prediction or improvement of the process. This concept of measurement is defined formally by the IEEE through a *metric* [141], which is "a quantitative measure of the degree to which a system, component, or process possesses a given attribute."

In the software engineering domain, these measurements may therefore be used to get some sense of whether the requirements are consistent, testable and implemented; whether the design is of high quality; to understand the complexity of the system; and whether the development activity can be improved [81, pg. 3–21]. Software metrics, in particular, can be combined in a well-defined way in order to aid analysis and assessment of a software system through a *software quality model* [81, pg. 338–344], such as the ISO 9126 model [145].

Individual metrics are defined for a particular domain, and often cannot be evaluated across different domains. For example, web application metrics such as "number of pages" cannot be directly compared to implementation metrics as "lines of code" or "number of comments". In this thesis, four domains of metrics will be discussed: domain-specific metrics for a particular web application (*web application metrics*); metrics for the modelling approach used (*metamodelling metrics*); and metrics for the development process itself (*system metrics*).

A number of other domain-specific metrics were considered, but have not been included in this thesis. For example, Selic [267] introduces the model-driven metrics of *compilation time* for translating a model, and the *turnaround time* for applying incremental changes. Other metrics, such as code coverage [**?**] and execution time were also considered, however these metrics are more appropriate for highlighting deficiencies in a single implementation, and not for evaluating different implementations of the same system.

2.6.1 Web Application Metrics

Web application metrics are metrics evaluated against a particular implementation of a web application, and there is a significant body of existing research into defining and using these metrics. These metrics are often used to estimate the effort required to develop different applications, but they can also often be used to compare different web applications for complexity. They cannot be used to compare different implementations of the *same* design, but are instead used to compare the complexity of different web application designs.

By reviewing a wide selection of research into web application metrics, Mendes et al. [196] found that most existing metrics are concerned with estimating the development effort for a given web application (*cost estimation*). As an example, Cowderoy [53] defines size metrics such as number of web pages; amount of text; number of images; number of features off-the-shelf; and include metrics dedicated to reuse or outsourcing.

Many metrics have been proposed for object-oriented software systems; for example, the MOOD, MOOSE, EMOOSE and QMOOD metrics are all different collections of object-oriented software metrics, as discussed by Baroni and Abreu [15]. Baroni [14] describes a comprehensive collection of software metrics, including these object-oriented metrics, that can be applied to a system. As a web application may be implemented using any conceptual methodology – including object-oriented, procedural and functional – these metrics cannot be consistently applied to web applications.

Metric [207]	[288]	**Description**
	TNP	Number of packages, i.e. EPackages.
NoC	*TNC*	Number of classes, i.e. EClasses.
NoAC		Number of abstract classes, i.e. EClasses that are specified as abstract.
NoD		Number of primitive datatypes, i.e. EDataTypes.
TNoR		Total number of references, i.e. EReferences.
TNoA	*TNA*	Total number of attributes, i.e. EAttributes.
NoE		Number of enumerations, i.e. EEnums.
Nav		Navigability: The proportion of references with a defined eOpposite.
Cont		Containment: The proportion of containment references to all references.
Dat		Data quantity: The proportion of attributes to overall structural features.

Table 2.3: Selected metamodelling metrics within the Eclipse Modeling Framework, adapted from Monperrus et al. [207] and Vépa et al. [288]

2.6.2 Metamodelling Metrics

Metamodelling metrics can be useful to compare different metamodelling approaches intending to model the same domain. These metrics can either be concerned with individual elements of the metamodel, or the metamodel as a whole. For example, Rossi and Brinkkemper [254] proposed a set of seventeen non-empirical – i.e., not derived from usability studies – complexity measurements for modelling languages, such as a metric for average complexity ($\overline{C}(M_T)$), and a metric for total conceptual complexity ($C'(M_T)$). Siau and Cao [271] applied these metrics to UML [219], and found that UML is large and complex compared to other object-oriented modelling techniques with respect to these metrics.

Monperrus et al. [207] investigated applying similar metrics to modelling languages implemented in the Eclipse Modeling Framework[9], and some of these proposed metrics are listed here in Table 2.3. Vépa et al. [288] also look at a selection of metamodel metrics, in order to measure and compare different model repositories, and derive their metrics through a series of ATL transformations[10]; these metrics are also listed here in Table 2.3.

Another metric for evaluating a particular modelling approach, or domain-specific language, would be through measuring the productivity of the language in terms of function points per staff month; this was illustrated by Mernik et al. in order to measure programming language productivity [199].

Since in the Model-Driven Architecture[11] all metamodels are also models, we could evaluate metamodels through standard model metrics. For example, the *degree* of a given class would refer to the number of attributes and references directly by the class, and *distinct types* would refer to the distinct number of metamodel types used (e.g. classes, attributes, references, or enumerations). A suite of some model metrics have already been proposed by Wright and Dietrich [320] to evaluate the complexity of EMF model instances.

[9]The Eclipse Modeling Framework (EMF) is described in further detail later in this thesis, in Section 6.2.1.
[10]ATL transformations are described in further detail later in this thesis, in Section **??**.
[11]The Model-Driven Architecture (MDA) is described in further detail later in this thesis, in Section 3.1.5.

Metric	Description
Tasks	Tasks implemented
Time	Development time, in whole weeks
NDev	Number of developers [267]
FC	Number of file changes
Rev	Number of SVN revisions
DTech	Number of different programming technologies used [229]
DMedia	Number of different media types used [229]

Table 2.4: Overall system metrics, adated from Selic [267] and Olsina et al. [229]

Metric	Description
Files	Number of files
Size	Size of all files, in bytes
NCLOC	Physical lines of code [236, 81]
ALOC	Average NCLOC per file

Table 2.5: Language system metrics, adapted from Park [236]

2.6.3 System Metrics

There are a limited number of *system metrics* that can be used to compare two diverse implementations of the same project, possibly because the development artefacts depend heavily on the staff and technologies selected for a given implementation. Metrics that are useful within one approach (for example, "number of plugins" in a Symfony project) may have no reciprocal metrics in another approach (for example, "number of visual models" in a MDD project).

In this research, it seems that only two common domains exist between different implementations of a given RIA. *Overall system metrics* may be used to evaluate the system size and development effort, as illustrated in Table 2.4; a number of these metrics are derived from existing work by Olsina et al. [229] and Selic [267]. Similarly, *language system metrics* may be used to evaluate the physical representations of the development artefacts, as discussed by Park [236].

With regards to both of these domains of metrics, frameworks allow three different types of development effort to be independently evaluated. As discussed earlier in Section 2.2, these approaches can automatically generate *boilerplate code*, or encapsulate common functionality in runtime libraries. Consequently, system metrics should be applied to each of these different versions of a system:

- The *manual effort*, representing only content that has been developed independently of any boilerplate code or external libraries;
- The *generated application*, representing both the custom code and the generated boilerplate code provided by the framework; and
- The *complete application*, representing all custom code, generated code, and included libraries.

By splitting up system metrics into these three different versions, the level of complexity hidden or managed by the framework can be quantified. For example, external libraries can be expected to have a large number of files (*Files*); and the size of the generated application (*Size*, *NCLOC*[12]) should be significantly greater than the manual effort.

2.7 Development Approaches

There are a number of approaches in which a modelling language may be developed – for example, through the extension of an existing language, or the development of a new language. In this section, a range of different development approaches with regards to existing models will be discussed, along with a brief discussion on some development process models that may be used in such a development.

2.7.1 Modelling Language Approaches

If a modelling language is being developed with respect to an existing modelling language, three methods in developing a new modelling language are commonly used: restricting the language; extending the language; or abstracting the language into a new level of abstraction. Each of these approaches have a resulting impact on the complexity or expressivity of the language, and on the complexity of modelling a system using the language.

Extension

Extension is the officially-supported extension approach of UML , through the use of stereotypes, tagged values, and constraints [36]. The development of a framework for a programming language falls under this category as well, because the existing libraries are essentially being extended, adding complexity in the implementation. The extension of a common industry-accepted language, such as UML or Java, improves the acceptance of the new approach because the concepts are already defined and widely understood in industry [157]. The improved expressibility simplifies development of the resulting model instances, but the complexity of the language itself increases. Kelly and Pohjonen [157] also argue that it is difficult to adapt a general-purpose language to a particular domain.

Restriction

Restriction is the approach of removing language elements and concepts that are deemed unnecessary, thereby reducing the complexity of the language. For example, the removal of all but UML class diagrams in UML would likely lead to a simpler language similar to ER diagrams. However, a well-designed language is one that cannot be restricted any further; the restriction of languages necessitates a reduction in functionality.

Abstraction

The *abstraction* of a modelled system into a higher-level model is the most popular approach in model-driven approaches, and the development of a DSL follows this approach [89]. Model instances at the higher level of abstraction are intended to be later transformed into lower-level model instances of the abstracted existing language [28]. By abstracting a model, the resulting language can become more expressive, less complex, and easier to use; however, these gains only occur if the abstracted language

[12]In this thesis, the *NCLOC* metric represents the physical lines of code that are not comments, as defined by Fenton and Pfleeger [81, pg. 247].

can be translated into the lower-level language. Developers of new abstractions must also be careful not to create an abstraction that is too generic or too specific [157].

UML Profiles

As discussed earlier in Section 2.4.2, UML's extensibility is entirely through extension; restriction is not supported. By defining new stereotypes, tagged values and visual stereotypes, UML Profiles also permits limited abstraction. However, since UML does not specify translation semantics, the translation from these abstracted levels into lower-level models must be provided separately. Bruck and Hussey argue that UML should only be extended if the majority of your concepts easily map onto existing UML concepts [36].

Despite its popularity in industry, there seems to be no reason why WebML was developed as a separate DSL, rather than an extension of UML (like UWE). Perhaps it was because UML and WebML were being developed at roughly the same time; or perhaps the WebML developers thought that web application concepts could not easily map onto existing UML concepts. Ceri et al. discuss how to translate WebML models into UML models [43], but this integration appears to be one-way.

Discussion

The actual approach used for a model-driven project should depend on the project's requirements. If a system is already fully described in an existing language, then restriction should be used; or if an existing language almost describes the system, then extension may be the best approach. If the system does not easily map onto any existing language, then abstraction may be the best approach.

Other than the overall architecture used in the development of a new language, there are also practical concerns. Kelly and Pohjonen [157] investigate 76 cases of modelling language development, in which they identify fifteen common problems in the development of modelling languages. For example, they find that ignoring or avoiding prototyping and language evolution is a common mistake; evolution is inevitable, especially when the language is intended for real-world usage. The interested reader is referred to Kelly and Pohjonen [157] for a detailed discussion on these real-world problems.

2.7.2 Software Process Models

To satisfy good software engineering principles as discussed by Pressman [244], software development should be performed according to a process model adapted to the individual project. In this section, two development models will be discussed: the linear sequential model, as the classic software development lifecycle [244, pg. 30]; and the evolutionary model, which may be better suited towards the implementation of modelling languages due to their natural evolutionary properties [157]. Many other process models have been discussed in literature – such as prototyping, rapid application development (RAD), component-based and formal methods [244] – but a full discussion is outside the scope of this thesis.

Linear Sequential Model

The *linear sequential model* is sometimes called the *classic life cycle* or the *waterfall model*, and suggests a systematic, sequential approach to software development [244, pg. 28–30]. The classical model will theoretically resolve into a complete solution, but assumes that all requirements can be captured at the beginning of the process, and the final architecture can be completed before any development begins.

However, it is difficult to capture all of the requirements with such detail at the start of a project; the model cannot adapt to changing requirements; and a working version cannot be produced until the very end of the process, delaying any sort of feedback cycle. These disadvantages of the linear sequential model generally outweigh the benefits for all but the simplest of software projects [244, pg. 30].

Evolutionary Model

The *evolutionary model* covers an ever-expanding list of different models, such as the agile model, SCRUM, the spiral model, and others [244, pg. 34–42]. They refer to an approach in which developers produce increasingly more complex versions of the final system, with each version a deployable artifact. Evolutionary models encourage the rapid production of working, but incomplete, solutions, that evolve over time into the final version. An evolutionary approach guarantees that a working copy of an early solution is always available, allowing for rapid and early feedback. It also ensures that even if the project exhausts its available resources, it still has at least partially-implemented working deliverables.

It is important to note that the terms *iterative* and *evolutionary*, while similar, have two distinct meanings; that is, all evolutionary models are iterative, but not all iterative models are evolutionary. The iterative model refers to a process where a series of smaller sequential tasks are done repeatedly. For example, the linear sequential model can be executed iteratively, but only on the final iteration can the software be deployed [244].

Discussion

Evolutionary development is particularly suited to model-driven approaches: Kelly and Pohjonen argue that modelling language evolution is inevitable, and should be embraced rather than avoided [157]; similarly, Fowler argues that the development of domain-specific languages are well suited towards evolutionary development [89]. UML has itself been developed in an evolutionary fashion, and at the time of writing is continuing to evolve [166].

It is important to also consider the impact of a particular software process model in a research environment such as this thesis. Purely evolutionary approaches are not suitable for research environments, as the design and analysis of a research problem is often more valuable than providing an implementation. However, purely sequential approaches are also not suitable for research environments, as the validation of a partial implementation is often more valuable than providing a complete implementation that cannot be validated. Within a research environment, it seems that a balanced approach between the sequential and evolutionary approaches may be beneficial.

2.7.3 Test Driven Development

Test-driven development (TDD) is not a software process model; rather, it is a development style that aims to improve the quality of software. As Beck [19] discusses, the goal of TDD is to develop “clean code that works”: in particular, it is a predictable way to develop, encourages frequent releases and can support changing requirements. TDD is therefore not very useful for designing metamodels, but would rather be a good development style to use to produce the proof-of-concept implementation of the metamodel, and would mesh well with the evolutionary software process model.

Outside of an implementation of a modelling language, TDD may also be independently used by the model developer to improve the quality of their developed model instances. Kent [158] argues that tooling to support *model-driven testing* is essential to maximise the benefits of model-driven development. Model-driven verification, as described earlier in Section 2.3.1, may also be introduced as part of a model-driven testing framework.

2.7.4 Open Source

In the software development world, *open source* refers to software where the source code is available and distributed under a particular *open source license* [203]. Mockus et al. [206] argues that open source software can be extremely successful and of a high quality, as access to the source code of a software project can be used to identify bugs and develop third-party extensions. In particular, open-source projects have been found to reduce defects compared to similar proprietary software [206]. Development under an open-source license can also improve the functionality of the developed system, as software projects under compatible licenses can be integrated together.

In this thesis, the open source license will only be briefly mentioned, and a full discussion on the preferable range and combinations of licenses will not be discussed. There may be legal issues in a model-driven architecture similar to those in a code compilation scenario. For example, a GPL-licensed compiler [92] cannot enforce the GPL on the code that it compiles, *except* in the case that portions of the compiler – for example, runtime libraries – are also included in the compiled output[13].

In particular, the implementation of a model development environment for web applications may consist of a number of components, and each may be under a separate open source license. A full legal discussion legal repercussions of combining different open source licenses is well outside the scope of this thesis, but it is important to keep licensing restrictions in mind.

2.8 Conclusion

This chapter has discussed the new challenges that Rich Internet Applications have imposed on conventional web development processes, and that no existing modelling language for web applications can support these challenges. The extension of an existing general-purpose programming language in terms of a framework is not desirable, in order to gain the benefits of a platform-independent abstraction to RIAs.

Consequently this thesis proposes the development of a new language for modelling *Basic RIAs* from scratch, but reusing concepts and semantics from established languages; and one that can be manipulated using a visual representation. This language will be used to implement the benchmarking application *Ticket 2.0* according to some of the development approaches discussed in this chapter, and will be evaluated using a selection of metrics, feature comparisons, and evaluating the initial language requirements.

[13]The interested reader is referred to the *GPL FAQ* [94] – “Can I use GPL-covered editors [...] to develop non-free programs?” – and the *GCC Runtime Library Exception* [93].

Chapter 3

Modelling

The general area of modelling within software engineering is a rich area of existing research. This chapter will investigate and briefly summarise many of the techniques, approaches and standards in the model-driven area. This is necessary to clarify the scope of this research, to understand its interactions and relationships within the larger scope of model-driven approaches, and to improve later interoperability with other modelling languages.

3.1 Modelling

A model of a system can represent almost anything, and be represented by almost anything; for example, source code can be considered a model, yet the grammar of the source code can also be considered a model. The most consistent and agreed-upon definition is that *a model is a simplified abstraction of reality* [125]. Models may coexist in different modelling spaces and themselves be abstractions, or instances, of other models [69].

However, the idea that a model is an abstraction of reality does not necessarily mean that a given model is useful. Selic [267] argues that for a model to be useful and effective, it must sufficiently possess the following five key characteristics:

1. **Abstraction:** "A model is always a reduced representation of the system that it represents." If the model is actually more detailed than the original concept, then it is not a model.

2. **Understandability:** "It isn't sufficient just to abstract away detail." A model must still contain concepts that are understandable.

3. **Accuracy:** "A model must be a true-to-life representation of the modeled system's features of interest."

4. **Predictiveness:** "You should be able to use a model to correctly predict the modelled system's interesting but non-obvious properties, either through experimentation or some type of formal analysis."

5. **Inexpensive:** "It must be significantly cheaper to construct and analyze [a model] than the modelled system."

Discussions on the suitability of model-driven approaches are found widely in existing literature, with good overviews provided by Meservy and Fenstermacher [200] and Djurić et al. [69]. Gitzel et al.

[108] discusses the use of MDA and it's natural advantages to address the unique problems introduced by web applications, such as improving usability and future maintenance.

Within the world of modelling, there are many different techniques and standards, and the relationships between them are often misunderstood or ill-defined. This section will describe each of these approaches and their relationships to other model-driven approaches.

3.1.1 Metamodels

If a model is a simplified abstraction of reality, a model developer still needs to define what the abstraction means, and how the reality may be mapped to a model. This represents the *modelling language* for a collection of models, and this language describes the syntax and semantics of the abstracted *model instances*[1]. These model instances are said to be *valid* if their representation conforms to the syntax, and their meaning conforms to the semantics defined in the modelling language. The syntax of a modelling language is often represented formally using grammars such as EBNF, while the semantics of a modelling language is usually represented informally using plain text [130].

The most widely-known modelling language used in software engineering is the Unified Modeling Language (UML) [224], as described earlier in Section 2.4.2. This *general-purpose* modelling language aims to support the analysis, design and implementation of systems and processes, although it is mainly used in software engineering. For this language, both its syntax and semantics are defined using a mixture of formal constraints, visual rules and structured English language.

A modelling language is also known as a *metamodel*, and the two terms may be used interchangeably [69]. In this thesis, the term *model instance* is preferred over the term *model* to reduce confusion; likewise, the term *metamodel* is preferred over *modelling language*, to distinguish the approach from a *domain-specific language*, as discussed in the following section.

3.1.2 Domain-Specific Languages

A domain-specific language (DSL) takes the abstraction concept of modelling and represents it in terms of a human-editable language. Fowler defines a domain-specific language as "a computer programming language of limited expressiveness focused on a particular domain" [89, pg. 27], and this definition is used in this thesis. He argues that the boundaries between a DSL and a general-purpose language are blurred, but certain elements of a language may be used to categorise the language.

Fowler puts a heavy emphasis on the limited expressiveness of a language as an indicator of a DSL. For example, he argues that the powerful statistical language R [**?**] has enough general-purpose functionality to be categorised as a general-purpose language, and not a DSL. In particular, a domain-specific language should not exhibit Turing completeness. Similarly, Fowler argues that a DSL should be designed to be human-editable; for example, if the language is serialised using XML, and is intended to be edited in this representation, then the language cannot be considered domain-specific.

The Benefits and Drawbacks of Domain-Specific Languages

Fowler [89, pg. 33–39] discusses some of the potential benefits and drawbacks of using a domain-specific language, in order to help inform the design choices of a system architect. Many of these

[1]One logic-based definition of the relationships between *modelling languages* and *model instances* is discussed later in Section 3.1.10.

mirror those achieved when using models as abstractions, as discussed in the previous section; the differences in these benefits will be briefly discussed here.

1. **Improving development productivity:** As discussed earlier, abstractions can simplify the development of complex software applications. A DSL provides an expressive way to manipulate instances of these abstractions, reducing defects and improving productivity.

2. **Communication with domain experts:** The abstraction of a DSL allows domain experts to interact directly with a tailored abstraction for their domain.

3. **Change in execution context:** As discussed later in Section 3.1.7, general-purpose languages often can only be evaluated once the instances have been compiled into a lower-level platform. DSLs on the other hand can permit evaluation either at compile time or at runtime, allowing a shift in the execution context of a particular approach.

4. **Alternative computational model:** A domain-specific language may allow a developer to use alternative paradigm models. For example, DSLs can be used to introduce functional or object-oriented code into a procedural language.

There are also a number of drawbacks that need to be considered when using a domain-specific language.

1. **Language cacophony:** This is the concern that "languages are hard to learn, so using many languages will be much more complicated than using a single one" [89, pg. 37]. This viewpoint is particularly noteworthy in the field of RIA development as discussed earlier in Section 2.1.1, where a RIA must be implemented across many different languages resulting in the conceptual gap of development.

2. **Cost of building:** The effort necessary to design, implement, test and maintain a DSL cannot be ignored, and not every abstraction will improve productivity to the degree necessary to justify the investment.

3. **Ghetto language:** Similarly to *feature creep* as discussed by McConnell [194, pg. 319–344], an incrementally-developed DSL may gradually gain features and functionality until it reaches a point of general-purposeness, at which point the development investment necessary to maintain the system outweighs the value of the DSL. Fowler argues that the only defense to feature creep is to firmly define the scope of the DSL.

4. **Blinkered abstraction:** A danger when using a certain abstraction is that of trying to implement concepts outside of the original abstraction scope, and not permitting changes to the original abstraction. This manifests itself when the time spent trying to implement a certain piece of functionality exceeds the effort that would have been necessary to adapt the abstraction to absorb the new behaviour.

Metamodels and Domain-Specific Languages

Modelling languages and domain-specific languages are two similar concepts; however, the former represents a method of defining an instance of an abstraction, and the latter represents a restriction on the types of abstractions used. The terms are not mutually exclusive, and can be used simultaneously; adapting the definition of a DSL provided by Fowler [89, pg. 27], a *domain-specific modelling language* therefore represents a modelling language of limited expressiveness focused on a particular domain. As discussed earlier, this distinction is made clearer by only referring to modelling languages as metamodels.

3.1.3 Model-Driven Development

Model-Driven Development (MDD) does not refer to a defined approach, but rather the idea that models should be embraced as part of systems development. Selic [267] asserts that the key of model-driven development is not the use of particular technologies or standards, but rather the essentials of model automation (including model transformation and model verification), and modelling standards; using these essentials, a model-driven process can be tailored for a specific system development.

System development can benefit from model-driven development in a number of ways, as the models can be used in a number of potential applications as summarised by Czarnecki and Helsen [57]:

- Generating lower-level models, and eventually code, from higher-level models;
- Mapping and synchronising among models at the same level, or different levels, of abstraction;
- Creating query-based views of a system;
- Model evolution tasks such as model refactoring; and
- Reverse engineering of higher-level models from lower-level models or code.

Model-driven development also encourages a system to integrate into the ecosystem of existing models, to increase interoperability and model reuse. Finally, another important application of model-driven development is the ability to verify models against consistency and correctness properties; this has become increasingly important in integrating models with the semantic web and related reasoning applications [267, pg. 21][2].

3.1.4 Model-Driven Engineering

Model-Driven Engineering (MDE) is closely related to MDD, and some authors consider the terms equivalent to the point of interchangeability; for example, Koch [168] and Koch [167] appear to use the terms MDE and MDD interchangeably in their definitions of UWE. In this thesis, the distinction between the two terms will be taken from the definition by Schmidt [263]: MDE follows the same concepts as MDD, but proposes using specific technical approaches for the applications of MDD, whereas MDD is technology-agnostic.

These technical approaches are still platform-independent however, and are not dependent on any particular implementations of each technology. For example, MDE advocates using DSLs as opposed to general-purpose modelling languages, but these DSLs can be implemented in any model-driven environment. Consequently, while all MDE approaches are also MDD approaches, not all MDD approaches are MDE; it is perfectly acceptable to use general-purpose languages in a MDD approach, or to ignore model transformations completely.

[2]The specific details of verifying and validating model instances against certain properties is discussed in further detail in Chapter **??**.

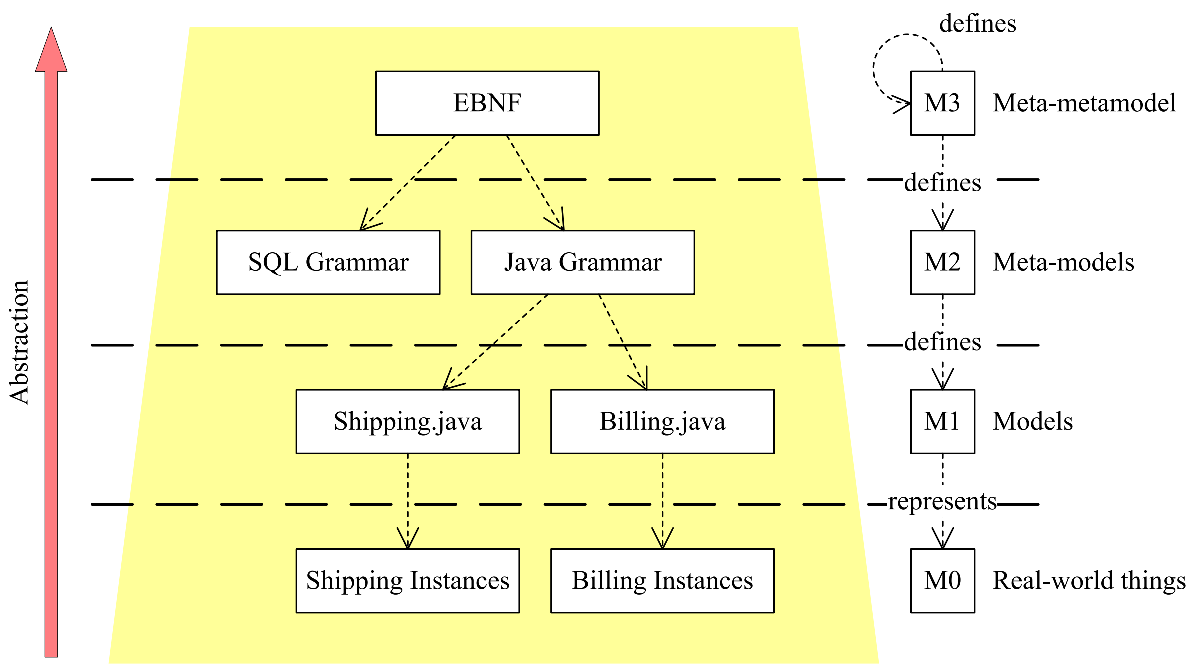


Figure 3.1: The Metamodelling Architecture of MDA, as discussed by Djurić et al. [69]

3.1.5 Model-Driven Architecture

Model-Driven Architecture (MDA) is an ongoing software engineering effort to standardise model-driven approaches, and is advocated and trademarked by the OMG [214]. MDA focuses on developing *architectural* standards for the development and interchange of modelling approaches; it is both an MDE initiative, and an MDD approach. At the time of writing, only a draft outline[3] of the proposed architectures has been published [214]. In particular, MDA is currently based on two conceptual architectures: a four-layer *metamodelling architecture*, and a four-layer *viewpoint architecture*. These architectures are not mutually distinct, but considering both of these architectures simultaneously is useful.

Metamodelling Architecture

The first architecture behind MDA is based on a four-layer metamodelling architecture [214], as illustrated in Figure 3.1; each layer has a practical purpose, as described by Djurić et al. [69]. Each layer provides a metamodel for the layer below with an increased layer of abstraction, and the OMG has supplied a relevant modelling standard for each layer. This metamodelling architecture is used for all OMG modelling standards [163].

1. The *Reality layer* (M0) represents the real-world objects that are being modelled, which can include everything, including non-tangible things such as modelling languages and other abstract concepts. For example, real-world entities such as rocks or trees can be part of M0, as could a running Java application, or its source code.

2. The *Model layer* (M1) represents the model of the real-world object; these models are called *analysis models* in UML [166]. Individual models are known as *model instances*. For example,

[3]The technique of publishing a number of drafts before standardisation of a language was also used during the development of UML [166] and the HTML standards [290, 299].

Metamodel	Meta-metamodel
Java	EBNF [113]
SGML	ISO 8879 [143]
HTML 4.01	DTD [290]
HTML 5	English Language [299]
XML	EBNF [298]
DTD	EBNF [298]
Ecore	Ecore [275]
EBNF	EBNF [144]
MOF	MOF [221]
UML	MOF [223, 163]
English Language	English Language

Table 3.1: List of Popular Metamodels and their Meta-metamodels

a Java application in M0 can be modelled with its Java source code in M1.

3. The *Metamodel layer* (M2) represents the model of the models in M1, also known as metamodels or *modelling languages*. For example, all Java source code in M1 must adhere to the Java language specification [113], which would be defined in M2. One OMG standard for metamodels is the *Unified Modeling Language* (UML) [224].

4. Finally, the *Meta-metamodel layer* (M3) represents the model of the metamodels in M2. For example, the definition of the Java language is provided in terms of *Extended Backus-Naur Form* EBNF [113]. EBNF would therefore be defined in M3. Most importantly, all M3 meta-metamodels are instances of themselves; that is, EBNF is defined formally as an instance of its own syntax [144], and there is no such thing as a *meta-meta-metamodel*. As a standard for meta-metamodelling, the OMG proposes the *Meta-Object Facility* (MOF) [226].

Importantly, this architecture satisfies the problem on where increasing levels of metamodelling abstraction will end. Since models can themselves be defined in terms of metamodels, it is a realistic question to ask whether this redefinition could continue forever. By defining M3 models as defined by themselves, the layers can be restricted to four: "The OMG defined that all elements of layer M3 must be defined as instances of concepts of the M3 layer itself" [163, pg. 88].

For comparison, a selection of common metamodels and their associated meta-metamodels are listed in Table 3.1. A full description of each of these models will not be provided here. If the meta-metamodel for a given metamodel is identical to the metamodel, then this meta-metamodel is *M3-compliant* according to the MDA, and can be used in the development of an MDA-compliant modelling approach. There are two particularly interesting aspects of this list: firstly, HTML 4.01 is defined in terms of DTD, which itself is not M3-compliant; and secondly, at the time of writing HTML 5 had only been defined using the English language [299].

A model instance is not limited to conforming to a single metamodel, and multiple metamodels are usually combined in a logically conjunctive way. For example, HTML 5 is currently defined only in terms of the English language, but in the future may also be defined using XML Schema or DTD [299], with each metamodel further restricting the valid range of HTML 5 instances to improve precision. Similarly, the Web Ontology Language (OWL) [304] is defined in three metamodels: EBNF abstract

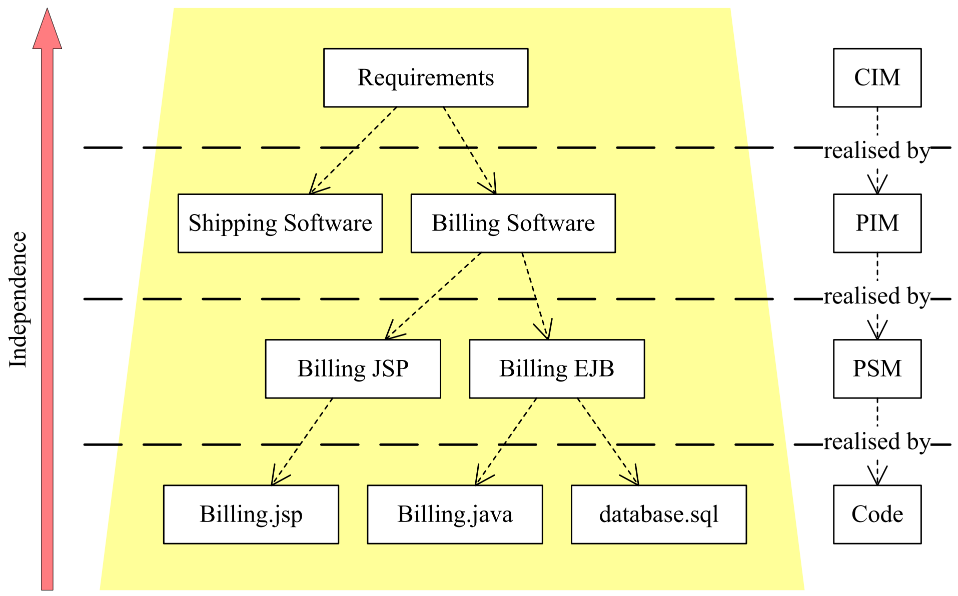


Figure 3.2: The Viewpoint Architecture of MDA, adapted from Mukerji and Miller [214]

syntax, direct model-theoretic semantics and model mappings [293].

Viewpoint Architecture

A second architecture proposed by the MDA is the four-layer viewpoint architecture [214, 163], illustrated in Figure 3.2[4]. In this architecture, each layer tries to be an independent abstraction of the layer below it, with the intention of simplifying the development of complex model-driven approaches across an entire business, rather than just a software product. Kleppe et al. [163] discuss the implementation of this architecture with a real-world example.

The OMG does not define any standards for any layer, as each layer is instead advocating a particular architecture, although many of these layers could be implemented using existing OMG standards. For example, both a PIM and a PSM can be represented using UML [163].

1. The *Computation Independent Model* (CIM) is a software-independent model or business model used to describe a business system.

2. The *Platform-Independent Model* (PIM) is the model of a software system that is independent of any implementation technology, and these models have a very high level of abstraction[5]. It is not possible to automatically translate a CIM into a PIM, because the decision of which parts of the business will be supported by software must be made by a human [163].

3. The *Platform Specific Model* (PSM) is the specification of a software system using a particular implementation technology. In some cases, PSMs can be produced from the automatic translation of PIMs [163]. Importantly, PSMs are only understandable by people who are experienced in the specific technology.

[4]**TODO:** Get out Kleppe et al. [163] again and see if they also provide this same architecture; if so, place an additional reference in the caption of the figure.

[5]**TODO:** Jens feedback: Mention how the PIM is similar to virtual machines. But, virtual machines have nothing to do with modelling, and unless I can find a reference saying that the viewpoint architecture was inspired by virtual machines, I probably can't mention this in the thesis. Consider reading this in Kleppe et al. [163].

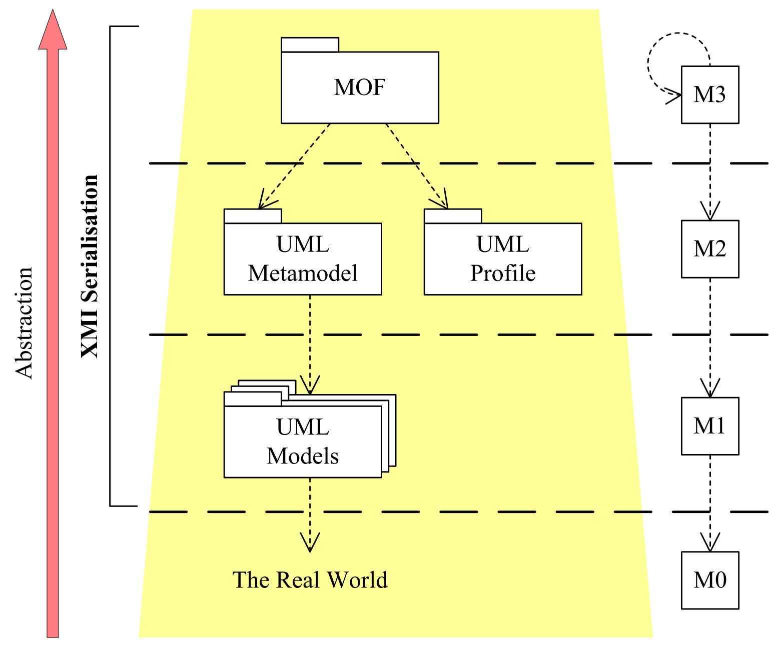


Figure 3.3: The layers of UML under the MDA, adapted from Djurić et al. [69]

4. Finally, the *Code* layer is the actual implementation of the PSMs. The transformation from PSM to code is relatively straightforward.

Kleppe et al. [163] argue that there are four main benefits from using this MDA architecture. Firstly, by shifting developer focus from platform-specific models to platform-independent models, the abstraction improves the efficiency of development. Secondly, everything specified at the PIM level is completely portable, depending on the automated transformation tools that are available; this also increases the interoperability of models. Finally, Kleppe et al. argue that since the PIM is at a higher level of abstraction than the PSM, the PIM also fulfills the function of high-level documentation.

Meta Object Facility

The *Meta Object Facility* (MOF) is an OMG standard that was designed and published as the standard M3 language for all OMG models, and is used to define modelling languages [163]. As a standard at the level of M3, the MOF is defined in terms of MOF, and the overall design was heavily derived from UML. The integration of the MOF with other OMG standards like UML is illustrated in Figure 3.3. The ambition of MOF – codified as the design goals of the standard – was to simplify the interoperability and communication between different modelling languages, and to clear up some of the definitions used in UML; the full list of design goals is published as part of the MOF specification [226].

The MOF model is also known as the *Complete MOF* (CMOF) package, which is built upon a subset of the UML 2.0 Infrastructure [223]. MOF also defines a smaller *Essential MOF* (EMOF) package, which simplifies the complexity needed for compliance[6] by providing a model focused solely on *kernel metamodelling* [226]. The Ecore metamodel of the Eclipse Modeling Framework, which also resides at the M3 layer, was built upon EMOF [**?**] and model instances can be serialised directly to EMOF [275, pg. 40]. A full discussion on EMF is provided later in Section 6.2.1.

[6] An implementation of MOF must adhere to at least one of the two *compilance points* of CMOF or EMOF.

XMI

In order to share models, the XML Metadata Interchange (XMI) standard [220] was proposed by the OMG, which is an XML-based standard for sharing metadata. This standard integrated XML, UML and MOF together into one standard, which is now an international ISO/IEC standard [146]. Many model-driven CASE tools, especially those used in UML modelling such as ArgoUML [251] and Eclipse UML [36], support exporting and important model instances formatted according to XMI, and a sample XMI model is illustrated in Section **??**. In the same way that XML has improved integration between different technologies, XMI can be used to improve integrations between different model-driven technologies, and is an important technology within MOF.

The Big Picture

As the viewpoint and metamodelling architectures in the MDA are not distinct, it can be useful to consider a model-driven approach that uses *both* of these architectures at the same time, when trying to define the scope of different interacting models. Figure 3.4 illustrates a range of model-driven technologies that may be used to implement a RIA, and highlights their interaction with the two architectures of the MDA. For example, a RIA may be described in a platform-independent manner, which may then be translated into platform-specific EJB [150] and RIA user interface model instances. These may, in turn, be translated into instances of HTML [299], SQL [164] and other related technologies.

By considering both of these architectures simultaneously, it is easier to understand the interaction between each of these MDA architectures. In the domain of RIA development, it is simple to see how a single model described by a single metamodel (in the PIM) can be implemented using many different RIA technologies, and how every metamodel has an independent meta-metamodel. This figure also illustrates how the PIM model and metamodel should not have any reference to the specific technologies in the PSM or code, highlighting the platform-independence of this layer.

In this example matrix, it is unclear what the PIM M0 for modelling RIAs would represent (marked as '??' in Figure 3.4), and this remains an unanswered question. Would this represent the real-world mental model of our intentions? Would it represent the analysis or design of the modelling language in this thesis? Both of these appear to be valid answers, as the M0 layer can represent any concept.

3.1.6 Model Spaces

As discussed earlier, if you take something from the real world (M0) and then model it, you obtain a model instance on M1. However, this model instance can itself be considered a real world object on M0, and the model instance can be modelled (or abstracted away) again; that is, models can exist on different layers at the same time. In order to further understand model-driven development approaches, a method is necessary to understand how models can exist in multiple domains concurrently.

Djurić et al. [69] propose the concept of a *modelling space*, which is a particular modelling architecture within a particular domain. A modelling space consists of all four layers from the MDA, with the models defined within this space representing the real world from one point of view. Conversely, each layer of the MDA and its contents are also part of M0, so it is always possible to abstract away anything into a separate model space.

For example, consider a developer that is trying to use UML to model a Java program, as in Figure 3.5. UML can model both the Java program and the Java grammar itself [69], and without modelling spaces it can be unclear where this distinction lies. If we place UML models into a *MOF*

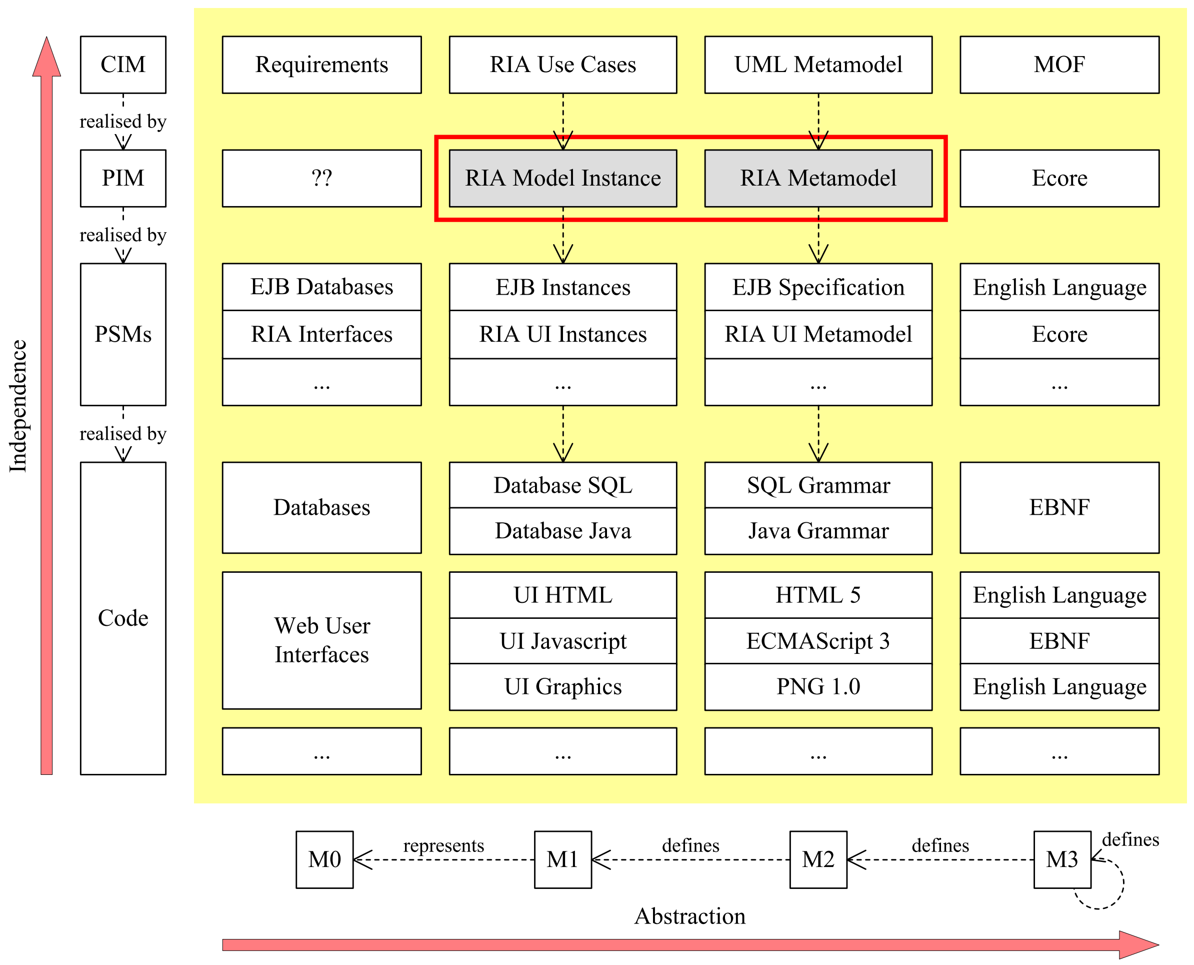


Figure 3.4: Using both architectures of the MDA to describe the model-driven development of RIAs

modelling space, we can introduce a separate *EBNF modelling space* which holds models for the Java aspect of the system. This full process is described in detail by Djurić et al. [69].

These separate modelling spaces for Java software and the MOF metamodelling approaches can make it much clearer to understand where modelling concepts should reside. For example, the syntax of Java reflection would be modelled in the UML Java *grammar model*; but the source code of Java reflection actually being used would be modelled in the UML Java *program model*.

3.1.7 Model Transformations

As discussed by Czarnecki and Helsen [57], one of the important applications of model-driven development is in the generation of lower-level models, and eventually code, from one or more higher-level models. This process can be generalised as a *model transformation*, which encompasses all aspects of transforming one model instance into another; a formal description of a model transformation is discussed later in Section 3.1.10. This is particularly important within MDA, where many models within many layers of abstraction need to interact with each other [133]. For external domain-specific languages, this transformation step is essential if the language is to be used [89, pg. 46].

Model transformations can be implemented with many technologies and approaches, and a detailed evaluation of four existing technologies is provided later in Section 6.4. For example, if the

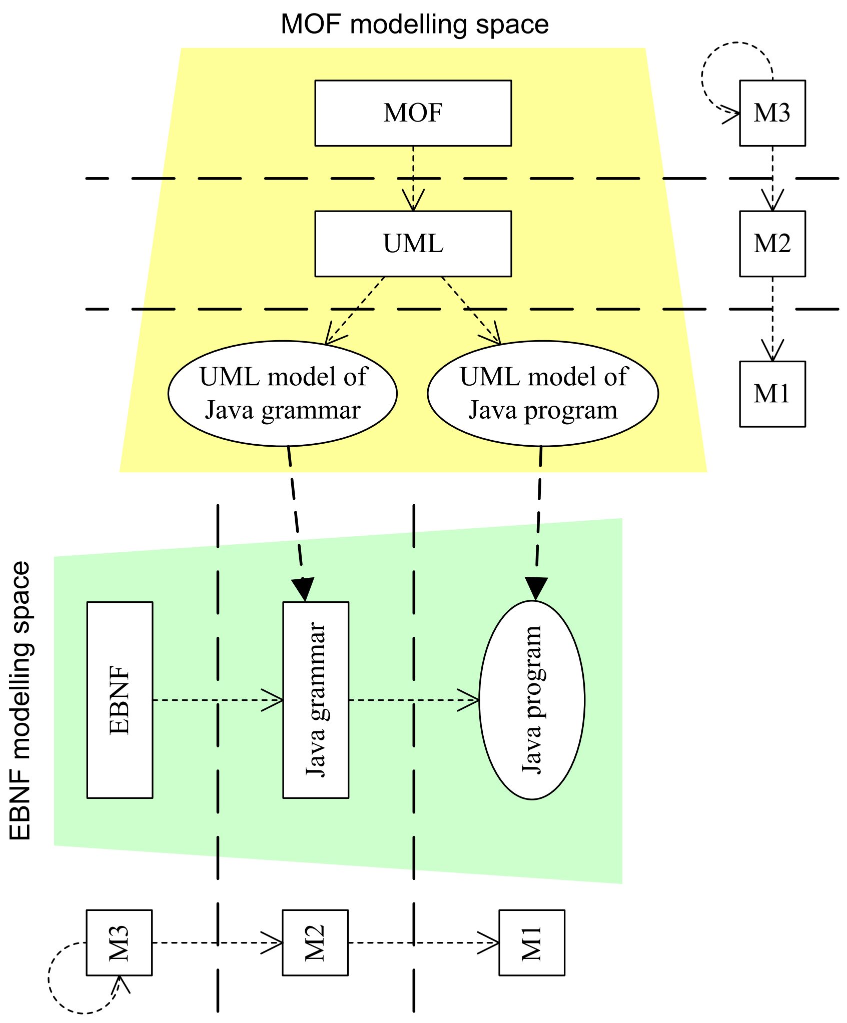


Figure 3.5: Dealing with different Modelling Spaces, adapted from Djurić et al. [69]

source models are represented using XML, then *Extensible Stylesheet Language Transformations* (XSLT) [297] can be used to express simple translation rules. For MDA-oriented approaches in particular, the QVT and ATL languages are two proposed standards for representing model transformations in a platform-independent way, using both declarative and imperative definitions [152].

It is possible to describe the architecture of model transformations using the metamodelling architecture of the MDA, where different model instances and metamodels span the M1, M2 and M3 metamodelling layers. Bézivin [28] proposes one common transformation pattern, which is illustrated here in Figure 3.6. The implementation of UWE uses this pattern heavily, as UWE model instances need to be transformed repeatedly in order to produce deployable artefacts [167].

Code Generation

Code generation and *compilation* are two types of model transformation, and are especially common if programming languages are also considered as modelling languages. Both of these transformations take platform-independent models at a higher level of abstraction, and refine it into platform-specific models at a lower level of abstraction as *source code*. In particular, model transformation may be considered code generation if the target metamodel is on the *Code* layer of the MDA viewpoint architecture, or the model transformation can be considered a *model-to-text* transformation as discussed in

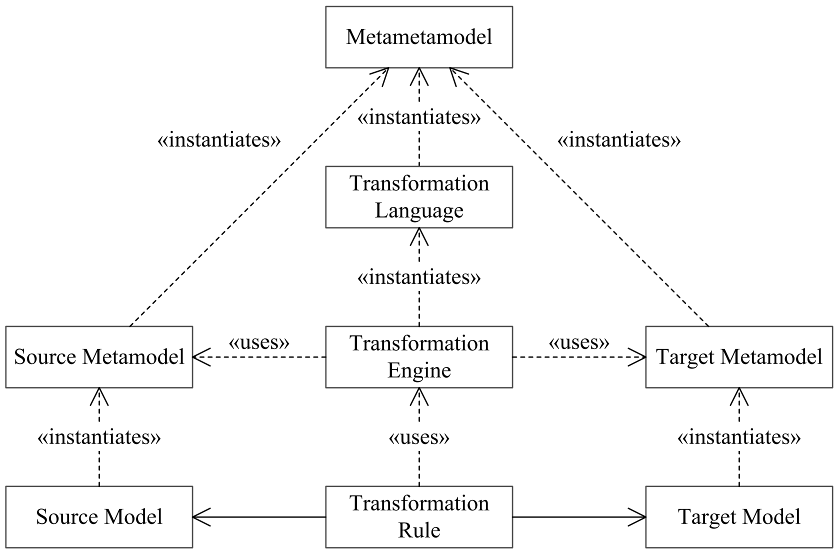


Figure 3.6: The Model Transformation Pattern in MDA, adapted from Bézivin [28]

the next section.

It is unclear whether there is a difference between these two terms, and there is no consensus between the two terms in the research community [57]. In this research, the following distinction is made: while both techniques generate executable instructions, a *code generator* generates models (or source code) that are expected to be edited by a developer, whereas a *compiler* generates models (or source code) that are not expected to be edited manually.

3.1.8 QVT

As part of the development of the MOF architecture, the OMG appreciated the need for a standard model-to-model transformation language. Consequently the OMG developed the Query/Views/Transformations (QVT) language to satisfy this need [228]. QVT consists of three separate languages:

1. A high-level declarative language, *QVT Relations*. Instances of this language are generally represented using a graphical notation [168, pg. 107].
2. A low-level declarative language, *QVT Core*. *QVT Relations* transformations may themselves be translated into equivalent programs within *QVT Core* [152], and are generally represented textually.
3. An imperative language, *QVT Operational Mappings* or *QVTo*. Instances of this language are also generally represented textually.

Each of these languages have a different scope and area of intended use. For example, QVTr is designed for providing high-level model transformations, whereas QVTo is designed for implementing the code generation of higher-level models into lower-level and platform-independent models. Since this research was started, the OMG published the *MOF Model to Text Transformation Language* (MOFM2T) to address model-to-text transformations [225].

The QVT standard relies heavily upon MOF, and all three QVT languages are defined according to a MOF-compilant metamodel. QVT only supports model-to-model transformations between models that adhere to the MOF specification [228]. Unless a textual target language is first represented using

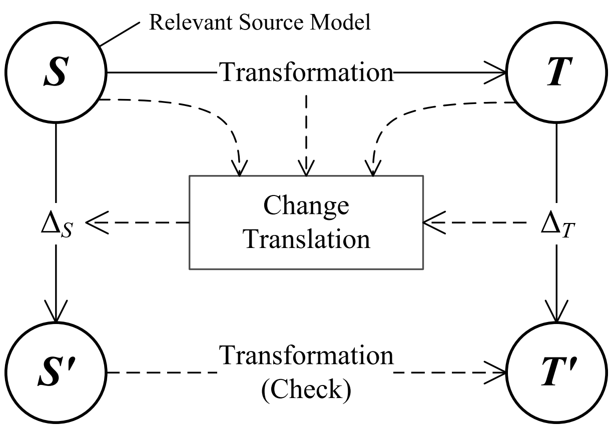


Figure 3.7: Using change models for round-trip engineering, adapted from Hettel et al. [133]

an intermediary modelling language, QVT-based approaches are not suitable for direct code generation. For example, the JavaML project [9] provides a metamodel for Java programs, similar to an abstract syntax tree, which can easily be utilised in model-to-model transformations [29].

The Atlas Transformation Language (ATL) [29] was one of the first implementations of the OMG's proposal for QVT, and also provides an Eclipse-based implementation [151]. Jouault and Kurtev investigate and discuss the architectural alignment of ATL and QVT [152], and find that "it is possible to have a reasonable interoperability between [the two languages]" [152, pg. 1194]. In particular, *QVT Core* and *QVTo* are very closely aligned to the design of ATL.

Round-Trip Engineering

When using a model transformation that follows code generation concepts, it is often desirable to modify the translated model manually. These target models may be altered or extended, due to maintenance or changing requirements. However, the modified target model may no longer be an accurate result of the original source model transformation. *Round-trip engineering* (RTE) is the process used to ensure that the source model remains consistent when the target model is modified [133].

Hettel et al. [133] provide an excellent overview into the challenges in round-trip engineering, and also discuss some of the approaches used to solve this problem. One of these approaches, common in the model-driven development community, is on the use of *change models*, as illustrated in Figure 3.7. In change models, a change Δ_T to the target model T is translated into a change Δ_S for the source model S.

Round-trip engineering is very similar to *reverse engineering*, however, RTE is more concerned with keeping source and target models consistent by supporting both forward and reverse engineering. Chikofsky and Cross II define reverse engineering as a process of extracting *abstractions* from target systems, and these abstractions do not necessarily need to be identical to the abstractions of the original source models [47].

In general, the reverse engineering of model transformations within the *same* metamodels is very difficult. As Hettel et al. [133] discuss, this can only be achieved if the model transformation can be defined entirely using *injective* functions, each with an *injective inverse* function defined. However, even primitive arithmetic functions such as addition are not injective due to information loss[7], making

[7]For example, $2+2$ may be transformed to 4, but it is impossible to reverse engineer 4 back into $2+2$ without providing additional knowledge or making assumptions.

reverse engineering of arbitrary model transformations very difficult [133, pg. 33].

Traceability

When translating requirements into the analysis, design or implementation from a set of requirements, it is often desirable for the relationships between these source and target artifacts to persist and be documented explicitly. This is known as *tracability*, and is often desirable for model-driven approaches to improve understanding and maintainability [126, 163].

Traceability is particularly important when a PIM to PSM translation is not complete, and the user must fill in gaps in the PSM. If there is the possibility that the user may overwrite parts of the automatically translated code, and the environment cannot update the source model, then the user should at least be warned about any potential problems [163]. This is also closely related to round-trip engineering, where changes in the PSM need to be synchronised with the original PIM. Traceability is not a concern with frameworks, as frameworks are extensions of a source model within the same modelling layer as discussed earlier in Section 2.2.

Some Agile development proponents advocate skipping some or all traceability in favour of "travelling light" to reduce development overhead, as discussed by Hailpern and Tarr [126]. This approach is likely to be more successful if most of the translation is automated, and if model developers are supported by rich model-driven tools to deal with changing requirements. However, such an approach makes reverse engineering much more difficult, and this needs to be considered when following such an approach.

3.1.9 The Syntax and Semantics of Metamodels

While a metamodel is still being developed, it is acceptable for the meaning of the elements and structure within the metamodel to be contained within informal design specifications, use cases and test cases – or even within the metamodel designers' head[8]. As the language matures and is published however, this knowledge should be expressed formally, and possibly into a format that aids the computation and evaluation of the metamodel.

This specification process is also known as providing *semantics* to the modelling language, and there are many different ways in which this may be achieved. Harel and Rumpe [130] provide an excellent overview into the different types of semantics that a modelling language may be defined with. In particular, the *syntax* of a language represents its notation, and the *semantics* represents its meaning. In this section, four related concepts will be briefly discussed.

Syntax

In terms of modelling languages, the *syntax* or *structure* of the language defines how a valid model instance of that language may be constructed, and is often provided in terms of a formal language. For example, XML Schema [294] may be used to restrict instances of XML to a particular structure; similarly, EBNF is used to define the valid grammar of Java source code [113]. Syntax may be defined informally (such as in the English language), but this approach is often undesirable as informal definitions can introduce ambiguity or hinder machine evaluation. Syntax can also only provide a limited amount of meaning to a model instance; for example, a containment relationship [275] can imply some sort of ownership or parent-child relationship.

[8]**TODO:** I think I may have taken this quote from somewhere else – but now I can't find the reference.

```
context Classifier::allParents() : Set(Classifier) body:
  allParents = self.parents()->union(
    self.parents()->collect(p | p.allParents())

context Classifier inv:
  not self.allParents()->includes(self)
```

Listing 2: Implementation of the *acyclical class inheritance* constraint of UML class diagrams in OCL [224, pg. 53–54]

Invariants

Invariants are a way to provide additional restrictions in a metamodel in situations where the restriction cannot be defined in the syntax of the metamodel. For example, the "acyclical class inheritance" restriction on UML class diagrams cannot be expressed in terms of MOF, but can be expressed as an invariant using the syntax of the OCL language [222], as illustrated in Listing 2.

Formal Semantics

Formal semantics refer to the meaning that is provided through instances of a formally-defined language; that is, one that "has been endowed with precise and unambiguous definitions" [130], and are unambiguous enough that their assertions may be conclusively proven. A wide range of approaches provide frameworks for such formal semantic definitions, such as the operational frameworks of *structural operational semantics* [239], *Hoare logic* [135] and *abstract state machines* [124]. A full discussion on the functionality of each of these frameworks is well outside the scope of this thesis, however the interested reader is referred to a brief overview by Zhang and Xu [323].

Informal Semantics

Informal semantics attempt to try and provide the same meaning to a modelling language as *formal semantics*, but within a language that permits ambiguity. Evolving modelling languages will often have their semantics initially defined informally – such as the definition of HTML 5 using the English language [299] – but informally-defined semantics cannot be proven without translation into some formal representation.

3.1.10 A Formal Definition of Metamodels

In order to precisely understand the relationships between models, metamodels and model transformations, it is necessary to define the semantics of these terms and their relationships formally. In this thesis, the definition published earlier by Wright and Dietrich [320] will be used as a basis for discussing the formal semantics of metamodels, and this section will briefly summarise these definitions. In terms of the metamodelling architecture of the MDA, the meta-metamodel (M3) of these definitions is first-order logic [85].

One can consider a model to consist of a set of artefacts M which occur within the universe of all possible artefacts, i.e. $model \in 2^M$. The types of artefacts in this model are not restricted, but in order for a model to have any meaning, a particular model needs to be restricted against certain semantics in the modelled domain. To achieve this restriction, the metamodel $\mathscr{S}$ is defined as the valid range of all possible models, $\mathscr{S} \subseteq 2^M$, and this metamodel represents a modelling language.

For example, consider a model universe 2^M which contains all possible UML classes and UML class inheritance relationships; any instance of a UML class diagram *model* would then be a member of this model universe $model \in 2^M$. Since the UML specification defines a number of restrictions – for example, class inheritance must be acyclical [224, pg. 69] – this universe needs to be restricted according to all of these restrictions in the UML metamodel $\mathscr{S}_{UML}$. Any set of artefacts within the model universe 2^M that is also within the metamodel $\mathscr{S}_{UML}$ would therefore be a valid UML model instance and UML class diagram instance.

This restriction can be defined using any number of mechanisms or technologies, including English language. Most metamodelling technologies, such as UML and EMF (as discussed later in Section 6.2), support the restriction of model instances using a variety of structural techniques, such as cardinality constraints and containment references. Restrictions that cannot be defined within the metamodelling environment may be defined using specialised constraint languages such as OCL [224]. A full suite of possible restrictions is discussed later in Chapter **??**.

A Formal Definition of Model Transformations

As discussed earlier in Section 3.1.7 by Czarnecki and Helsen [57], a model transformation is the process of converting one or more source model instances into one or more target model instances; however, in this thesis, the formal definition of a model transformation will be restricted to only those transformations that process *one* source model instance into *one* target model instance.

With respect to two meta-models $\mathscr{S}_1, \mathscr{S}_2$, a model transformation can be portrayed as a function $T : \mathscr{S}_1 \to \mathscr{S}_2$. If the target meta-model is also the source meta-model, then the transformation can be simplified as a single-model transformation $T_1 : \mathscr{S} \to \mathscr{S}$.

3.1.11 The Bigger Picture of Modelling Approaches

The modelling community has not yet arrived at a consensus on how to consolidate and compare these different modelling approaches. While all of these approaches advocate the use of models in one way or another, and many can be used simultaneously without conflict, many do not describe their relationships with other modelling approaches.

In this thesis, these missing relationships are resolved through the definitions proposed in Table 3.2, and is necessary in order to understand the bigger picture of modelling approaches. Many of these relationships are straightforward, and have been discussed in greater detail throughout this chapter.

The distinction between a modelling language (that is, a metamodel) and a programming language is hard to define, and needs to be discussed here. They are both abstractions of a system with well-defined syntax and semantics, and usually both have an implementation in a target language. The representation of language instances also does not affect the distinction, as modelling languages may be textual, and programming languages may be visual.

The main distinction between a metamodel and a programming language appears to be the difference in their scope. A programming language is often a general-purpose metamodel on the *PSM* layer of the MDA viewpoint architecture, used to define model instances of the *Code* layer; whereas a metamodel is often a domain-specific metamodel on the *PIM* layer.

Term	Definition
Model	A set of artefacts representing a simplified abstraction of reality, that are valid to the constraints of a modelling language.
Metamodel	Defines a valid type of model according to a set of defined syntax and semantics.
Programming Language	A general-purpose modelling language with a specific syntax, semantics, and compilation into a lower-level language.
Framework	The extension and abstraction of a programming language, contained within the same language, to simplify common tasks.
Domain-Specific Language (DSL)	A domain-specific modelling language with a specific syntax, semantics, and transformations into a lower-level language.
Model Transformations	The translation of one model instance into another model instance, within the same, or across different, metamodels.
Code Generation	A model transformation with a target metamodel on the *Code* level of the MDA viewpoint architecture.
Model-Driven Development (MDD)	A development approach that considers models as first-class development citizens.
Model-Driven Engineering (MDE)	A development approach that encourages domain-specific languages and automated model transformations.
Model-Driven Architecture (MDA)	Two four-layer architectures illustrating the relationships between different models in an integrated model-driven environment.

Table 3.2: The relationships between different modelling approaches

However, domain-specific programming languages such as IA-64 assembly language [137], and general-purpose modelling languages such as UML [224], blur these distinctions. Clements [49] argues that there is a definite overlap between programming languages and modelling languages. In this thesis, a language is defined as a *programming language* if its intent is to be general-purpose, and as a *metamodel* otherwise.

3.1.12 Metamodelling Environments

As discussed earlier in Section 2.3.1, a modelling environment should be supported with software tools in order to maximise the benefits of using models [158]. Such tools can include tools to define model instances, code generators, and analysis tools [263].

While the abstraction of complex systems has always been a part of software development, it is only recently with the development of new modelling software platforms that general-purpose model-driven software has become common. New model-driven technologies such as the Eclipse Modeling Framework [275] and the Graphical Modeling Framework [119] can provide a platform which encourages the cheap and quick development of model-driven approaches. A comprehensive evaluation of some existing frameworks is provided later in Section 6.2.

3.2 Model Completion

When designing a modelling language, a key challenge is balancing the level of detail in its design [320]. A language that is too simple will result in a rigid approach that cannot adapt to many situations; conversely, too much flexibility will force the developer to create and maintain large monolithic models.

TODO Jens feedback: Discuss OWL. (Motivation of OWL: Semantically ... to reason about constraints, to make new assertions.)

Software frameworks, discussed earlier in Section 2.2, have to deal with this problem as well; the frameworks must be flexible, but also simple to develop with. Web application frameworks such as Ruby on Rails [280] and Symfony for PHP [240] have proposed to resolve this balance through the use of *documented conventions*.

Model completion, as proposed by Wright and Dietrich [320], adapt the concepts behind software frameworks to the model-driven domain in order to provide this flexibility and simplicity to the model-driven development of systems. The remainder of this section will briefly discuss the semantics behind this process, which are necessary to ensure the consistency of the approach; for more detail, the interested reader is referred to the paper [320].

3.2.1 Documented Conventions

Within these frameworks, *documented conventions* are an approach to automatically complete missing parts of the application. The intent behind this process is to improve the productivity and efficiency of using the framework in common situations. For example, as web applications often require database integration, the Symfony web application framework supports the definition of an abstract database schema, which is then translated automatically into code for a particular database platform.

Three important aspects must be considered when using this completion process. Firstly, the conventions must be well-documented, so that the developer can anticipate changes without having to inspect source code. Secondly, the intent of the developer must always override the intent of the framework, and these intentions must never be overwritten; the developer must always be able to override these conventions, otherwise these conventions would become restrictions. Finally, if the framework has too many "magic" conventions, then it is likely that the complexity of the framework will become a mental burden for the developer.

3.2.2 Model Completion Concepts

Model completion is a single model transformation operating on a *base model* defined by a single metamodel, illustrated in Figure 3.8. Following documented conventions, a CASE tool takes the initial model and transforms it into an *intended model* complete with all necessary detail. As discussed earlier in Section 2.3.1, implementing this process within a CASE tool is essential in order to maximize the benefits of MDE [158].

This approach also maps well onto evolutionary development processes, as once the base model is transformed into an intended model, it may be used as input to other tools such as code generators or analysis tools [263], and evaluated in order to obtain feedback. This feedback can then influence refinements to the base model as part of a new development iteration [320].

Importantly, model completion needs to complete the intended system based on incomplete knowledge, a process known as *non-monotonic reasoning* [101]. Applied to our domain, this states that when we add additional information to the system, any information inferred may be retracted in the presence of new knowledge. This reasoning philosophy provides a great deal of flexibility.

To illustrate this point, consider a user interface element representing a boolean property within the system. This interface element is part of a platform-independent model, allowing us to ignore the

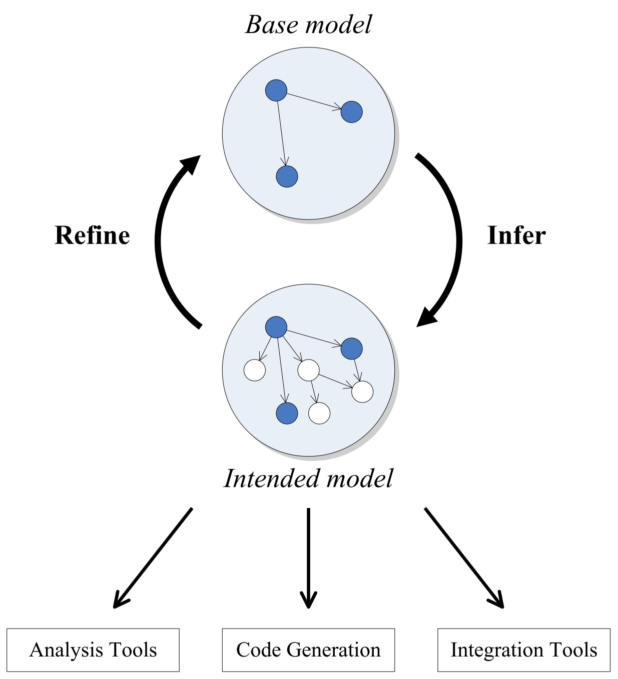


Figure 3.8: Model completion within model development, adapted from Wright and Dietrich [320]

technical details of its implementation. It is reasonable to assume that by default, this property should be rendered by a checkbox. However, the developer may instead wish to represent this property with a drop-down list containing the values *yes* and *no*. In this case, the default checkbox should be removed; it should no longer be generated through model completion, nor override this new knowledge.

If negative existentials are used in defining part of a convention, non-monotonicity may be a consequence. For example, this *default checkbox rule* can be expressed non-monotonically as:

$$\begin{array}{rl} \text{IF} & (\textit{there exists a boolean property}) \\ \text{AND} & (\textit{there does not exist an editor for it}) \\ \hline \text{THEN} & (\textit{create a checkbox editor for it}) \end{array}$$

Model completion restricts retraction within non-monotonic reasoning to only facts inferred by the reasoner itself; that is, the reasoning process can never retract any information in the base model. This is important to ensure developer effort is never inadvertently discarded.

3.2.3 Model Completion Semantics

In order to prove the consistency and correctness of this inference process within a model-driven implementation, we need to investigate the formal semantics of models and the model completion operation. This definition is essential to ensure that model completion preserves *consistency* with the original model [176]. Using the metamodel definitions provided earlier in Section 3.1.10, the model completion process itself will now be defined formally; for more detail, the interested reader is referred to Wright and Dietrich [320].

Given a meta-model $\mathscr{S}$ such that $model \in 2^M$ and $\mathscr{S} \subseteq 2^M$, model completion is defined as a function $C : \mathscr{S} \rightarrow \mathscr{S}$ operating within the same meta-model. That is, all completed models will also be valid models in our domain. For a model completion $C(X)$ operating on a model $X \subseteq \mathscr{S}$, two

$$\begin{aligned}
\Phi_1 = \quad & \mathit{property}(\mathtt{a}) \\
\Phi_2 = \quad & \mathit{property}(x) \land \neg\exists y : \mathit{editor}(y) \land \mathit{editorFor}(x,y) \\
& \quad \to \mathit{checkbox}(\mathit{newCheckbox}(x)) \land \mathit{editorFor}(x, \mathit{newCheckbox}(x)) \\
\Phi_3 = \quad & \mathit{checkbox}(x) \to \mathit{editor}(x) \\
\Phi_4 = \quad & \mathit{dropdown}(x) \to \mathit{editor}(x)
\end{aligned}$$

Figure 3.9: Definition of the rule program C for the *default checkbox rule* convention, adapted from Wright and Dietrich [320]

$$\begin{aligned}
A = \quad & \{\mathit{property}(\mathtt{a})\} \\
C(A) = \quad & \{\mathit{property}(\mathtt{a}), \mathit{checkbox}(\mathit{newCheckbox}(\mathtt{a})), \\
& \quad \mathit{editorFor}(\mathtt{a}, \mathit{newCheckbox}(\mathtt{a}))\}
\end{aligned}$$

Figure 3.10: Model completion $C(A)$, adapted from Wright and Dietrich [320]

conditions need to be imposed:

1. *Extensive*: Model completion must not retract any existing information in the base model, i.e. $X \subseteq C(X)$.
2. *Idempotent*: Once the intended model has been completed from a base model, applying model completion on this intended model will not change the model, i.e. $C(X) = C(C(X))$.

These two conditions are part of Tarski's classical axioms for inference operators [278]. Monotonicity is not a necessary condition; in the face of new information within a base model, previously inferred knowledge may need to be retracted, i.e. $X \subseteq Y \not\Rightarrow C(X) \subseteq C(Y)$.

In order to implement these conditions on the model completion function, new elements must be created according to a *factory function*, and introduced into the intended model using the concept of *stratification* [284]. This allows for the documented conventions to be represented as a set of rule formulae in a *rule program*, which may be evaluated against a model instance to complete the intended model. By using an *insertion queue* in the implementation of the factory function, the model completion process has been demonstrated to satisfy the extensive and idempotent conditions on $C(X)$ [320].

TODO If I described the insertion queue/stratification/rank in more detail, I could include the graph showing (rank) vs (number of elements) as per Wright and Dietrich [320].

Within this definition of model completion, the *default checkbox rule* may be expressed using the rule program C illustrated in Figure 3.9. This can then execute on an initial model A to create the completed model $C(A)$, as in Figure 3.10. The full expansion of the steps necessary to complete the intended model in this example through model completion is not provided here; the interested reader is instead referred to Wright and Dietrich [320].

TODO Discuss how an *L-Model* is defined as a language $L(R,F,C,V,=)$? See Wright and Dietrich [320].

3.3 Visual Modelling

As discussed by Moody [208], visual representations of models can be more effective than textual representations, because they can tap into the capabilities of the powerful human visual system. In particular, he argues that diagrams are believed to convey information more effectively, concisely and precisely than textual language, and the information is more likely to be remembered. Visual notations are also particularly important when communicating with end-users and customers.

Depending on the requirements of the artifacts, visual modelling can range from strictly formal approaches – visual models can be exactly as formal and structured as textual models [208] – to informal approaches as used by individual designers. UML models are intended to be drawn by hand, as communication was thought to be more important than automation (and therefore precision). However, recent revisions of UML have focused on improving its architecture and formalism in order to support software integration and reasoning [166].

Visual representations can be particularly powerful when applied to the domain of model-driven development, as visual representations are themselves models. The combination of a domain-specific language and an accompanying visual representation can be termed as a *domain-specific visual language*, as defined by Grundy et al. [120]. This visual language can be defined from a metamodel specification with a certain level of automation [270]. However, when developing a visual language for the representation of a model, it is important to document the rationale and design decisions used [208, 120].

There are situations where visual modelling may not be appropriate, and a textual representation would be better. Vlissides and Linton [286] argue that graphical models are best for domain-specific languages, rather than general-purpose languages. Graphical languages generally lack *efficiency of expression*; that is, it is generally very difficult to visually design a complex algorithm [270]. Finally, without the support of a graphical modelling environment or framework, the implementation of a visual modelling language is often more difficult than the implementation of a textual modelling language.

3.3.1 Visual Metaphors

A *metaphor* is a linguistic device where one or more words for a concept are used outside their conventional meaning, to express a *similar* concept [174]. Metaphors are very popular in software development, in particular *interface metaphors* for user interface design; *system metaphors* for the development of the software system; and the Rational Unified Process notion of *metaphor* as a simple system architecture [**?**][9]. These metaphors reduce the mental load for developers, and improve the accessibility of the system [244]. Metaphors can also apply to visual modelling as a *visual metaphor*; that is, a metaphor used to relate a visual representation of a model instance with a foreign concept.

Barr et al. [16] expand on the work of Lakoff and Johnson [174] to propose a taxonomy of interface metaphors. This taxonomy includes *orientational*, *ontological*, and *structural* metaphors. Structural metaphors – those that deal more directly with physical objects in the real world [16] – are particularly prominent in user interface design [16]. For example, the "desktop metaphor", which represents files within a file system as objects on an actual desk, is an instance of a structural metaphor.

3.3.2 Visual Metaphors in Existing Models

Existing visual modelling languages already use visual metaphors in their design and implementation, and this thesis will discuss some of the metaphors identified in WebML and UWE. Each of the WebML models represented visually has a particular visual metaphor, although the WebML specification does

[9]**TODO:** Consider removing this, it might not be necessary.

not define these metaphors directly. The WebML hypertext model, for example, appears to use a visual metaphor of *"a web application is the flow of data"*, because the relationships between content units are expressed as a flow.

Some papers on WebML claim that each *content unit* in the model uses a visual metaphor; for example, Di Martino et al. describe that "the *MultiMap* Unit is a visual metaphor for a map viewer able to render both vector and raster data arranged in layers" [65]. According to the definition of *visual metaphor* in this thesis, this is not a visual metaphor, because their description only defines the visual representation of the *rendered* unit, and *not* the visual representation of the unit itself within a model instance.

Each model of UWE also has a particular visual metaphor, although these metaphors are also not explicitly discussed. Many UWE models are extensions of existing UML models – such as class diagrams and activity diagrams – so these models reuse their existing metaphors. On the other hand, some UWE models use entirely new metaphors; for example, the *navigation structure* model represents the web application using the metaphor of *"web application navigation using states and menus"* [169]. Since UWE reuses common visual metaphors (from UML), it is easier for new users to pick up and understand these models, and it also promotes sharing these models with users who already understand the metaphors of UML. It would therefore be beneficial for a modelling language to reuse existing visual metaphors where appropriate.

3.3.3 Visual Modelling Software

While visual models can be drawn by hand and exchanged manually, software can be used as part of the process. Visual modelling software is a strictly optional component of model-driven approaches, but it can improve the efficiency of model design and development, and the quality of communication with other end-users. These editors, sometimes called *graphical object editors*, allow a user to manipulate the graphical representations of model instances directly [286]. Visual modelling software has already been discussed in depth by existing literature [286, 208, 251]; a range of existing visual modelling frameworks is discussed later in Section 6.3.

Since these visual editors provide interactive representations of an underlying model instance, these editors map naturally to the model-driven development approach advocated in this thesis. Since graphical editors often have very similar functionality – such as printing, layout and shape types – metamodels may be used to simplify the implementation of a model-driven graphical editor. These *model-driven graphical modelling environments* are represented as a model instance, and translated using model transformations into the source code of the final graphical editor.

3.4 Model Instance Verification

As discussed earlier in Section 2.3.1, an important feature requirement of a modelling language for RIAs is to support integration with formal verification tools. Software verification of an implemented application is the standard method for verifying correctness properties on that application [202], and verifiable web application properties include non-functional requirements such as broken links, syntax validation, load testing, page response time [22].

However, in many cases it is preferable to verify a model instance instead of an implementation, as the relative cost of fixing an error dramatically increases over time [244, pg. 197]. This also allows properties of web applications such as concurrency and user interaction to be evaluated directly, and often with less performance impact than testing an implementation [**?**]. There is a significant body of existing work on verifying model instances *reverse-engineered* from an implementation [21, 64, 265], keeping in mind that it is usually unfeasible to extract structural design intent from code [192].

3.4.1 Validation vs. Verification

Validation and verification are two closely related concepts, and are often combined together into the general term *verification & validation* (V&V). The differences between these two terms are fairly subtle, as illustrated by the wide range of definitions published by existing work.

For example, Adrion et al. [3] defines *validation* as "determination of the correctness of the final program or software produced from a development project with respect to the users needs and requirements" by verifying each stage of the development life cycle, and *verification* as "in general, the demonstration of consistency, completeness, and correctness of the software at each stage and between each stage of the development life cycle." Adrion et al. also distinguishes *valid input* separately from validation, as data that lies within a particular domain.

Conversely, Sargent [259] defines *verification* as "ensuring that the computer program of the computerized model and its implementation are correct", but it is unclear what "correct" is referring to; and *validation* as "substantiation that a computerized model within its domain of applicability possesses a satisfactory range of accuracy consistent with the intended application of the model". These definitions appear to be at odds with those defined by Adrion et al.

In this thesis, the definitions by Pressman [244, pg. 479] and Boehm [31, pg. 37] are adapted to the model-driven domain in order to distinguish the differences between these two terms. In particular, the following definitions of V&V are used in this thesis:

Model verification refers to the set of activities that ensure that the model correctly implements a specific function; and

Model validation refers to a different set of activities that ensure that the built model is traceable to customer requirements.

In practical terms, this means that a requirements-independent check of a model instance – such as checking that all element IDs are unique, or that all private data is secure – is an instance of a *model verification* activity. Conversely, *model validation* is the process in checking that the model instance implements the requirements of the model – such as making sure that the "View Product" page is present, or that a particular key is secure.

These definitions are not consistent with some published research which erroneously use the terms interchangeably. For example, Ricca and Tonella [250] propose that there are two types of web application testing techniques, *static verification* and *dynamic validation*; but in this thesis, these tests would both be treated as instances of verification, as they do not test the application requirements themselves.

3.4.2 A Formal Definitions of Constraints

One may consider model instance verification as a process operating on syntactically correct models that identifies invalid models through constraint violation. Earlier in Section 3.1.10, a formal definition of metamodels and their model instances was provided based on Wright and Dietrich [320]; this definition may be extended to describe the process of model instance verification.

In particular, a model X consists of a set of artefacts from M constrained by a particular metamodel $\mathscr{S}$, i.e. $X \in \mathscr{S} \subseteq 2^M$, with $\mathscr{S}$ defined by a language $L(R,F,C,V,=)$. A set of constraints may be used to identify valid models by defining a verification function $V : 2^M \rightarrow \{0,1\}$; a particular model X is valid according to this function if $V(X) = 1$.

A constraint is defined in terms of a verification language L_{ver}, consisting of type predicates[10], n-ary relations, variables and standard predicate logic connectives and quantifiers. A constraint has the the following form[11]:

$$\forall_{x_1,\ldots,x_n} T_1(x_1) \wedge \ldots \wedge T_n(x_n) \rightarrow P(x_1,\ldots,x_n)$$

where n represents the *arity* of the constraint; $x_1,\ldots,x_n$ are variables; T_i represent unary type predicates within the relations set L_R; and P a constraint formula containing only the free variables $x_1,\ldots,x_n$. A model is therefore valid with respect to this constraint if this formula is satisfied. The unary type predicates T_i ensure that the constraint is always satisfied if evaluated against instances of non-matching types.

The expressiveness of the verification language L_{ver} may impact on the performance and decidability of its constraints; for example, OWL 2 Full [304] is the most expressive language within the OWL family, but its expressiveness has been proven to be undecidable [212]. Similarly, relations expressed in CrocoPat may be queried by identifying graph patterns, but this subgraph isomorphism problem is NP-complete [27, pg. 143]. Different verification engines may therefore be classified on expressiveness restrictions, and on their maximum constraint tuple size.

Expressiveness Criteria

In this thesis, the expressiveness of a verification language is defined in terms of the permitted syntax of constraint formulae. In particular, three non-exclusive categories of language expressiveness are considered:

1. **Functions:** Within verification languages such as OCL, constraint predicates are defined in terms of functions defined within the model instances, and additional functions may be defined within the language. It is not possible to define additional relations within the language.

   For example, a constraint *"all Named Elements must define a name"* may be specified using the predicate $P_{named}(x) = \exists n : name(x) = n$. The resulting constraint may then be defined as the $\forall_x T_{\text{Named Element}}(x) : P_{named}(x)$.

2. **Relations:** Within verification languages such as OWL/RDF, constraint predicates are defined in terms of relations, and additional relations may be defined within the language using first-order logic. Relations may use self-reference to support recursion.

   A constraint *"a Frame cannot redirect to itself"* may first be specified through the relation $R_{redirects}(x,y) = \exists e,w : T_{\text{Event}}(e) \wedge onAccess(x,e) \wedge T_{\text{ECA Rule}}(w) \wedge from(w,e) \wedge to(w,y)$.

   An additional relation may then define the transitive closure of this relation through $R_{redirects}(x,y) = \exists z : R_{redirects}(x,z) \wedge R_{redirects}(z,y)$.

   The resulting constraint may then be defined as $\forall_x T_{\text{Frame}}(x) : \neg R_{redirects}(x,x)$.

[10] In this thesis, types are represented as unary predicates; for example, the type Visible Thing is represented as the type predicate $T_{\text{Visible Thing}}(x)$ on a model artefact x of that type, as illustrated in Wright and Dietrich [320].
[11] **TODO:** Reformat $\forall$ to use matrix style.

3. **Higher-order Logic:** Within verification languages such as OCL and OWL, the verification language supports the definition of additional relations or predicates using higher-order logic.

   For example, a relations-based verification language may provide the higher-order operator **TC** to represent transitive closure for a particular relation. This would permit the transitive closure definition of the previous constraint to be defined instead as $R_{redirects}(x,y) = \mathbf{TC}(R_{redirects}(x,y))$.

   Similarly within a functions-based verification language, the higher-order aggregation function **fold** may be used to evaluate a function over a collection of artefacts and reduce it into a single value.

If the level of expressiveness of each language impacts on the performance of a model instance verification process, it may be desirable to support many verification languages simultaneously. This would allow simple constraints – implemented using a functions-based verification language – to be evaluated quickly and regularly, while more complex constraints – implemented using a relations-based language – may be evaluated less frequently, due to their resource requirements.

Model Checking

Model checking is defined by Baier and Katoen [10, pg. 11] as "an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model." In particular, these constraints are checked against the entire state space of the system [131].

For a system to be evaluated using model checking, it must first be translated into a list of potential states and the permitted transitions between these states; this *system model* may then be evaluated against a property using a language such as LTL [10]. These constraints cannot be described using the general constraint form defined earlier in L_{ver}, and a formal definition of model checking is well outside the scope of this thesis[12].

However, since a model checker must navigate through *all* of the possible states of a modelled system, there is a significant performance penalty in evaluating this process. Consequently model checking may only be evaluated infrequently – for example, when a model instance is used to generate a production system, or as part of a long-running automated process to catch bugs – and it is often preferable to express a constraint using one of the verification languages discussed earlier.

3.4.3 Discussion

As each of these verification approaches have different performance characteristics, model instance verification will be implemented with regards to three categories of language expressiveness. This will allow a model developer to evaluate a model instance against increasingly precise definitions of correctness, dependent on the resources available to the developer. In this thesis, these three categories are defined as follows:

1. *Function Language:* Functions-based constraint languages will be used to quickly verify constraints, and will therefore be evaluated frequently against the most common design problems.

2. *Relation Language with Higher-order Logic:* Relations-based constraint languages with support for higher-order logic constructs will be used to verify more complex constraints, and will therefore be evaluated less frequently against more complex design problems.

[12]**TODO:** Is there a formal definition in Baier and Katoen [10]?

3. *Model Checking Language:* Finally, a model checking approach will be used to evaluate behavioural correctness of the modelled systems. As this approach requires significant amounts of resources, it is intended that this approach will be evaluated fairly infrequently.

3.5 Conclusion

In this chapter, different approaches and technologies to model-driven development have been investigated to assist in the design and implementation of a new modelling languages for RIAs. In this proposed modelling language, both the viewpoint and metamodelling architectures of the MDA will be considered; model transformations will be used to translate models into executable code; existing visual metaphors will be used for the visual representation of the modelling language where appropriate; and the technique of *model completion* will be introduced to improve the flexibility of model instance development.

Chapter 4

Rich Internet Application Modelling Concepts

As discussed earlier in the Background chapter, a modelling language for RIAs may need to support the expression of many web concepts, such as events, users or security. This chapter will investigate and discuss a range of existing approaches and implementations of each concept, but will not explicitly discuss the suitability of each approach with respect to RIA modelling languages. These discussions will instead be used to inspire the definition of the modelling language in the following chapter.

4.1 Managing Complexity through Decomposition

When developing a modelling language, it is important to consider the complexity of the resulting model instances, which can be strongly influenced by the design of the language. It is often more desirable to work with many simple model instances than to work with a single monolithic model instance; this is particularly true when working with a visual modelling language [208].

This section will investigate different techniques for dealing with the complexity of model instances[1]. These techniques are not mutually exclusive, but rather represent a variable measure of the extent that each technique is used; for example, different systems can have differing degrees of modularity [12].

4.1.1 Modularity

The most common way of reducing complexity of large systems is to divide them into smaller parts or *modules*, a technique called *modularity* [**?**, pg. 39-66]. Modules are units in a larger system that are structurally independent, but work together as illustrated in Figure 4.1, with different levels of interconnectivity representing differing *degrees* of modularity [12].

Modularity can encourage system reuse as long as the system has been implemented correctly [**?**, pg. 39], as modules with few dependencies on other modules may be more easily reused in other contexts. Modularity also encourages the use of small units, rather than larger ones which may be more difficult to understand. However, a poorly-implemented modular system can be confusing and difficult to maintain, if modules are made too small or too large, or there are too many dependencies

[1]The complexity of a modelling language, rather than its instances, has been discussed earlier with metamodelling metrics in Section 2.6.2.

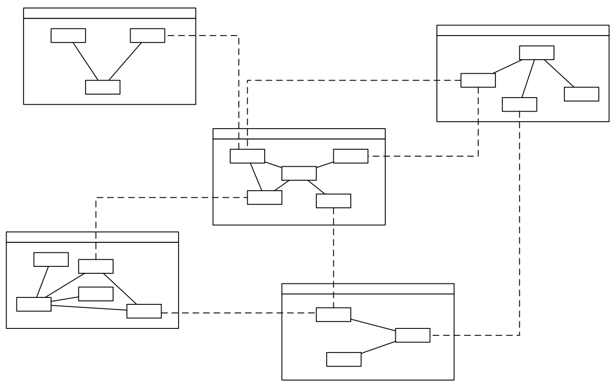


Figure 4.1: Modular Modelling

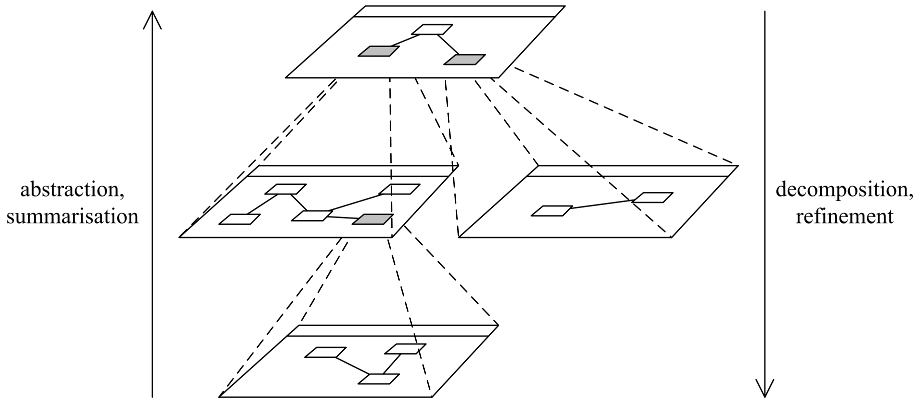


Figure 4.2: Hierarchical Modelling, adapted from Moody [208]

between modules. Without adequate documentation it can also be difficult to understand the bigger picture of a modular system. The effectiveness of a module is often expressed in terms of *cohesion* and *coupling*, as discussed by Pressman [244, pg. 353-355].

It appears that WebML [43], as discussed earlier in Section 2.4.1, has been designed to support modularity in its resulting model instances. In particular, an entire web application is expressed in a single monolithic model, but certain areas – such as pages, and operations – may be grouped together, in a manner similar to a module.

4.1.2 Hierarchical Modelling

As discussed by Moody [208], *hierarchy* is "one of the most effective ways of organizing complexity for human comprehension as it allows systems to be represented at different levels of detail, with complexity manageable at each level." It supports the decomposition of a complex system into separate models with each model becoming a part of a higher-level model, as illustrated in Figure 4.2.

Hierarchy is different from modularity in terms of the relationships between modelled elements; in modularity, the relationships connecting modules are those between *individual* model elements (and not the modules themselves), whereas the relationships in hierarchy are those between the higher-level parts. That is, modularity is focused on the encapsulation of elements as a single module, whereas hierarchy is focused on the decompositional layers of encapsulated elements. This means that a modular system may also be hierarchical if many of the modules may be decomposed, as discussed by [**?**, pg. 40].

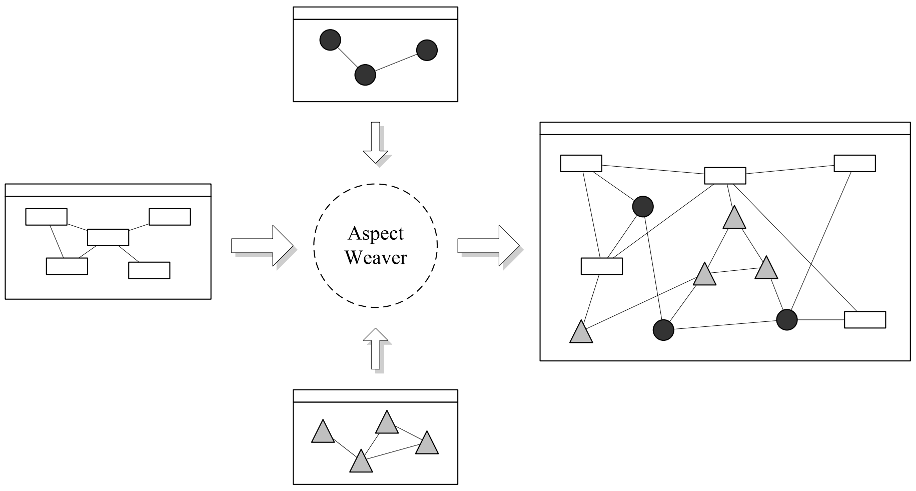


Figure 4.3: Aspect-Oriented Modelling, adapted from Kiczales et al. [160]

Hierarchical modelling encourages both top-down design and bottom-up design, in the sense that a model developer may abstract away complexity into many layers. It is important to provide appropriate context in a hierarchical environment, to orient the current view with the rest of the modelled system [208][2]. One example of a modelling language designed with hierarchy in mind is UML [224], where a system may be progressively decomposed through deployment diagrams, component diagrams, class diagrams, and activity diagrams.

4.1.3 Aspects

Aspect-oriented modelling adopts aspect-oriented programming [84, 160] to a model-driven environment, supporting the modelling of separate concerns of a model instance in separate models, which are then integrated together into a target model [90]. Concerns that span over the entire model are known as *cross-cutting concerns*, which are integrated as *aspects*. These aspects are integrated together using an *aspect weaver*, as in Figure 4.3. The intention behind aspects is to decompose cross-cutting concerns in a complex model into simpler aspects, which can also simplify the original system model.

Using aspect-oriented models has already been done in the domain of modelling web applications. For example, UWE uses aspect-oriented modelling to support the modelling of web application adaptivity [18]. Other web application concerns such as security, logging and user authentication can also be modelled using aspects with various degrees of independence.

As a relatively recent innovation, aspect-oriented approaches still have a number of important issues that need to be resolved. Aspect-oriented programming always depends on the underlying code that they weave, hindering their testability and reuse [5]. Understanding the interaction between many diverse aspects in a system, and how these interactions evolve over time during maintenance – known as *cognitive distance* – remains a critical issue [4]. The success of an aspect-oriented modelling approach is therefore very dependent on its implementation.

[2]In the proof-of-concept implementation of IAML, this is achieved by using breadcrumbing and shortcuts, as discussed later in Section 7.4.4.

4.1.4 Discussion

Each of these techniques approach complexity decomposition in a different way, but they are not mutually exclusive, and may be combined in almost any way. As discussed earlier in Section 2.7.2, software development should adapt existing software process models to the individual needs of the project; similarly, complexity decomposition approaches should be adapted to meet the needs of a modelling language.

For example, it would be possible to design a system using aspect-oriented modelling, with the modelled aspects following a hierarchical design; conversely, it would be possible to design a system using hierarchical modelling, and then apply aspect-oriented modelling to the different levels of hierarchy. Each of these systems would have different benefits and drawbacks; in the case of the former, it would be difficult for lower levels of the aspect hierarchy to communicate, whereas with the latter the aspects could not consider the higher levels of the hierarchy.

4.2 Events

As discussed earlier in Section 2.3.1, RIAs may be considered event-driven applications, and an RIA modelling language should have strong support for modelling web application events. In this thesis, the definition of an *event* is adapted from the UML specification; that is, an event is defined as "the specification of some occurrence that may potentially trigger effects" [224, pg. 440].

An event may trigger a particular action, and this invocation relationship may be restricted with a condition; this definition forms an *Event-Condition-Action* (ECA) rule, forming the basis of event definitions in this thesis. As discussed by Papamarkos et al. [235], ECA rules have been used in many settings, including active databases, publication/subscription technology, and in the implementation and specification of business processes. An ECA rule has the general syntax [68, 235]:

`on` *event* `if` *condition* `do` *action*

In order to select a conceptual approach for the modelling of events, the following sections will briefly illustrate and informally evaluate a range of existing approaches in terms of expressibility and simplicity. A similar approach was performed during the development of the R2ML rule modelling language [109]; Wagner et al. [307] evaluated seven existing approaches for visual rule modelling as the basis of event modelling in R2ML. For each modelling approach, the representation of a generic ECA rule will be illustrated in order to highlight the differences in syntax.

4.2.1 UML

UML permits the modelling of event handlers using *AcceptEventActions* [224, pg. 235], which can be used in UML activity diagrams to trigger behaviours. In UML, both signals and operation calls can be considered as events, and consequently *SendSignalActions* can be used to model a triggered event, with incoming parameters modelled using *InputPins*. This specification allows for an ECA rule to be modelled as illustrated in Figure 4.4.

Depending on the complexity of the ECA rule, a number of visual optimisations may be performed; these optimisations are not illustrated in Figure 4.4. If the condition is simple or the action is not restricted by a condition, the use of the *DecisionNode* with attached *decision behaviour* is not necessary. If the condition can be reduced into an instance of *ValueSpecification*, this specification can be used as a guard condition on a single *ActivityEdge*.

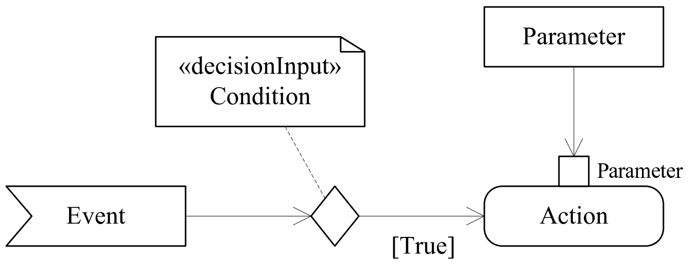


Figure 4.4: Event-Condition-Actions in UML activity diagrams [224]

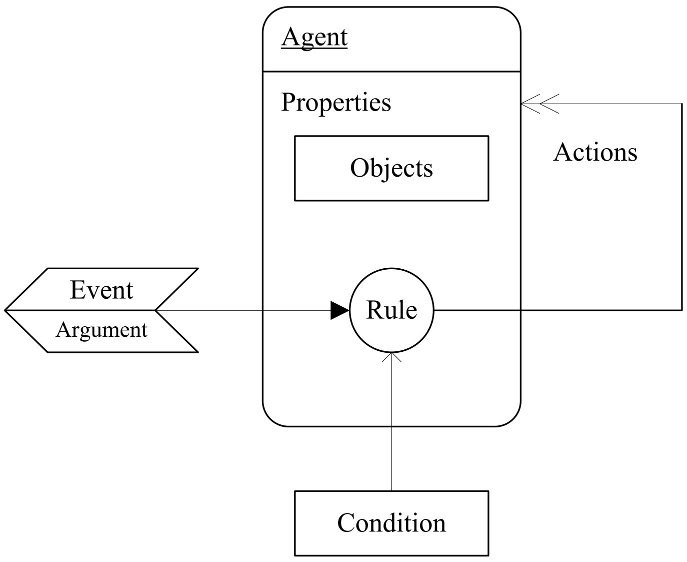


Figure 4.5: An AORML Interation Pattern Diagram, adapted from Wagner et al. [307]

UML also defines a number of fundamental events, such as *ChangeEvent* and *MessageEvent*, which are all instances of the *Event* abstract class. In order to define new events on the UML level, it appears that a UML Profile must be defined; conversely, to define new events on the model instance level, it appears that a *SignalEvent* may be used to handle instances of *Signals* as events.

4.2.2 AORML

The *Agent-Object-Relationship Modelling Language* (AORML) [305] is an agent-based modelling language that can describe the interactions between multiple agents in a system. As the name implies, AORML is intended to model agents rather than software components, with ECA rules owned by a parent agent; this may hinder the use of AORML in systems which are not agent-based. This language supports a wide range of visual notations for event modelling; for example, AORML can be used to describe ECA rules in terms of an agent as in Figure 4.5 [307][3].

4.2.3 URML

The REWERSE Working Group I1 developed the *UML-based Rule Modelling Language* (URML) [308] to allow visual rule modelling, and this metamodel largely overlaps with the metamodel of R2ML. In particular, URML can be "considered as a language that is derived from R2ML in order to provide UML-based rule modelling" [308].

[3]**TODO:** Need better reference

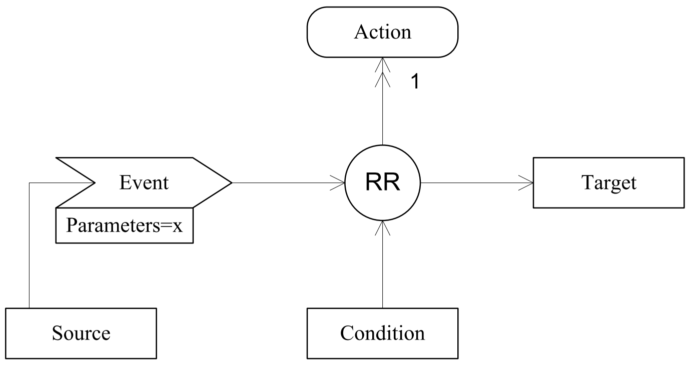


Figure 4.6: Modelling Reaction Rules in URML, adapted from Wagner et al. [308]

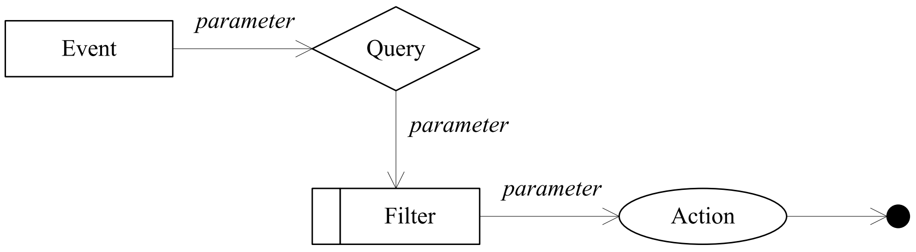


Figure 4.7: Visual Event Handler Definition in Kaitiaki, adapted from Grundy et al. [122]

URML supports three types of rule modelling – derivation rules, production rules, and reaction rules – but only reaction rules will be considered in this section, as their design is the most similar concept to ECA rules. The visual modelling of reaction rules in URML is illustrated in Figure 4.6, as defined by Wagner et al. [308].

4.2.4 Kaitiaki

As described by Liu et al. [186], the *Kaitiaki* project uses the *Event-Query-Filter-Action* metaphor in order to describe and define event handlers, which is very similar to ECA rules. This events metaphor has been integrated with the web service composition specification language *ViTABaL-WS* Li et al. [182]. This approach is developed with a proof-of-concept implementation in the Eclipse-based Marama framework [122].

4.2.5 Event Algebra

Events can also be modelled in a more formal way using *event algebra*, especially when modelling the timeline of the interaction between different event instances. As discussed earlier in Section 3.1.9, the definition of the formal semantics of a modelling language may be used to identify undesirable properties of a modelled system, and these benefits similarly apply in the case of event algebra. Similarly, event algebra may be used to improve the expressiveness of model verification constraints.

A full discussion on the different types of event algebras is well outside the scope of this thesis, but one of these event algebras will be briefly discussed to illustrate the concept. Zimmer and Unland [325] propose a metamodel for modelling complex events in active database systems, and some of the operators defined by their metamodel are listed in Table 4.1. These operators indicate some of the potential interactions that may occur between two distinct event instances.

Operator	Summary
$ei_1 ; ei_2$	*Sequence:* instances have to occur in the given order.
$ei_1 == ei_2$	*Simultaneous:* instances have to occur simultaneously.
$ei_1 \wedge ei_2$	*Conjunction:* instances have to occur, in any order.
$ei_1 \vee ei_2$	*Disjunction:* at least one of the instances have to occur, in any order.
$ei_1 \neg ei_2$	*Negation:* the specified events cannot occur within the specified instance period.

Table 4.1: Selected event algebra operators for composing complex events, adapted from an event metamodel proposed by Zimmer and Unland [325]

```
// Introduced in DOM Level 2:
interface MouseEvent : UIEvent {
  readonly attribute long            screenX;
  readonly attribute long            screenY;
  readonly attribute long            clientX;
  readonly attribute long            clientY;
  readonly attribute boolean         ctrlKey;
  readonly attribute boolean         shiftKey;
  readonly attribute boolean         altKey;
  readonly attribute boolean         metaKey;
  readonly attribute unsigned short  button;
  readonly attribute EventTarget     relatedTarget;
  void               initMouseEvent(...);
};
```

Listing 3: The IDL definition of the MouseEvent interface, adapted from the DOM Events specification [291]

4.2.6 DOM Events

As part of the definition of HTML, the W3C defined an events model known as DOM Events[4] which specifies a range of event interfaces on many different web browser elements. An event may be *bubbleable*, in that the event will "bubble" up the element hierarchy and can be intercepted by any of its parents; and an event may be *cancelable*, in that the default behaviour of the event can be overridden [291].

For example, the DOM event *click* is defined for "most elements" in the HTML 4.01 definition, and occurs "when the pointing device button is clicked over an element" [290]. This event is bubbleable and canceable, and occurs after the *mousedown* and *mouseup* events have been fired. This event can therefore be used to intercept a user clicking on a hyperlink, and cancelled to prevent the web browser navigating to the hyperlink destination.

When a DOM event is triggered, an event object instance is also created and provided to the event handler with specific information about the event. For example, the *click* event creates an instance of the MouseEvent interface, which provides attributes such as the position of the mouse on the screen when the click occurred; the state of keyboard modifier keys, such as `shift` and `ctrl`; and the button that triggered the click. The IDL definition [218] of this event interface is provided here in Listing 3.

[4]DOM events are also named *DOM Level 2 Events*, to distinguish the event model from a deprecated event model not formally defined by the W3C known as *DOM Level 0* [291].

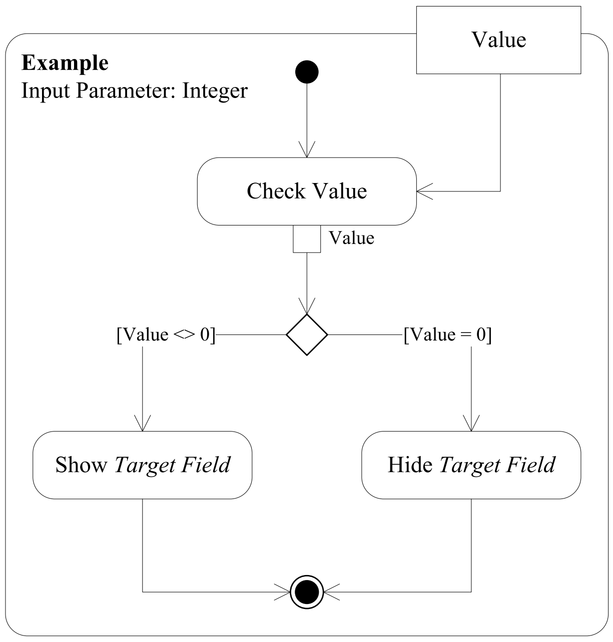


Figure 4.8: Operation modelling using UML activity diagrams [224]

4.3 Operation Modelling

Whilst events and actions refer to different types of activities, *operation modelling* refers to the modelling of the lower-level behaviours that make up these activities. This type of modelling emphasises the execution flow, data flow and conditions in these behaviours. A well-defined model-driven development approach can allow a modelling language to support multiple forms of operation modelling; for example, an activity could be modelled in a visual language, or textually in a general-purpose programming language. In this section, two visual operation modelling languages will be briefly discussed.

4.3.1 UML Activity Diagrams

UML activity diagrams [224] model behaviours of different aspects of a system at a lower level than other UML diagrams [23], and "emphasizes the sequence and conditions for coordinating lower-level behaviors, rather than which classifiers own those behaviours. These are commonly called control and object flow models" [224, pg. 295]. A sample UML activity diagram is included in Figure **??**; this activity implements the operation of changing the visibility of a particular user interface element, based on the value of an input parameter.

The UML activity diagram notation does not support primitive data operations, such as arithmetic[5]; this type of behaviour must either be defined informally, or by instantiating an extension within a UML Profile. UML activity diagrams can model a significant range of behaviours, but it is not possible to consistently compile these diagrams into code, because UML does not have a defined execution semantics (as discussed earlier in Section 2.4.2).

The specification of UML proposes that formal execution semantics may be provided to the UML activity diagram by translating diagrams into UML statecharts [224]; however, such a translation is

[5]The UML specification does not discuss why arithmetic is not supported. Perhaps this is due to the language design goal of staying platform-independent, or perhaps it is intended that expression languages such as OCL should be used.

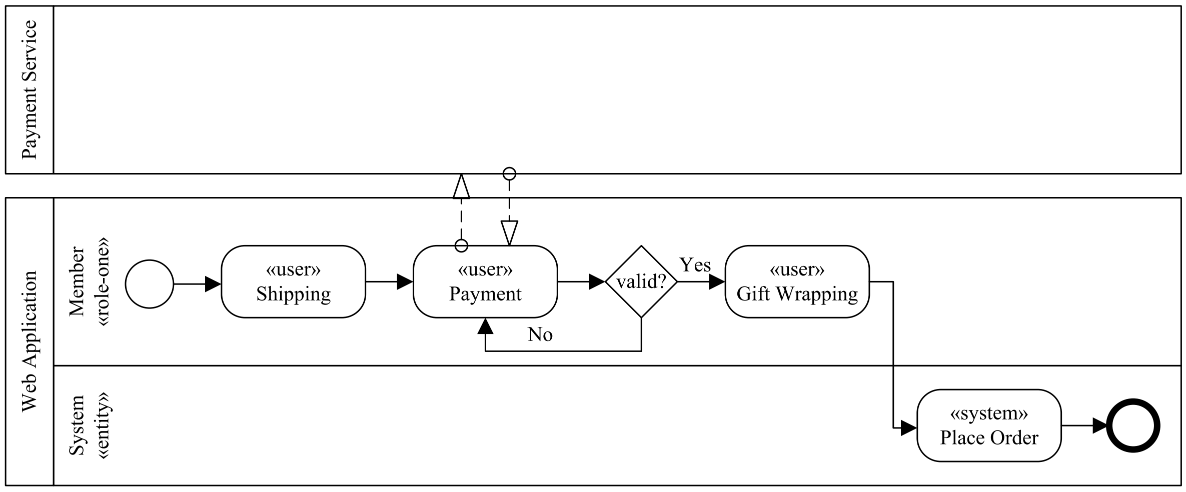


Figure 4.9: Operation modelling of a online shopping business process using BPMN, adapted from Torres et al. [283]

not provided in the latest edition of the UML specification, and Eshuis and Wieringa [77] argue that this process is inherently flawed as activity diagrams are too expressive. Eshuis and Wieringa instead translate activity diagrams into *workflows* in order to provide execution semantics to UML activity diagrams [77].

4.3.2 Business Process Modelling Notation

The *Business Process Modelling Notation* (BPMN), defined and specified by the OMG [**?**], is a visual modelling language for graphically representing the higher-level behaviours involved in business processes, with a notation that is intended to be immediately understandable by all business users. In terms of common business workflow events, the expressibility of BPMN has been found to be similar to UML activity diagrams by [**?**], however it is not clear whether the two notations are fully equivalent in terms of expressibility.

Similarly to UML activity diagrams, the formal semantics of the behaviour of BPMN model instances have not yet been defined. Wong and Gibbons [316] argue that the existing approach to translate BPMN instances into Petri nets in order to obtain execution semantics is incomplete. However, BPMN has already been used in the web application modelling domain to implement workflows, as discussed by Brambilla et al. [34].

A sample BPMN diagram of the business process behind a simple online shopping web application is illustrated here in Figure 4.9, adapted from Torres et al. [283]. This diagram illustrates the responsibilities of each entity within the process and the communications between them, and also highlights how similar the BPMN visual notation is to the UML activity diagram notation.

4.3.3 Predicate Modelling

In both UML and BPMN, predicates are often written textually, as illustrated earlier in Figures 4.8 and 4.9. While this simplifies the model instance, it also makes it difficult to translate the predicate into executable code. However, predicate modelling can be considered a form of operation modelling, on the restriction that the operations must return either `true` or `false`; a formal distinction between operations and predicates is provided later in Section 5.3.1.

This implies that the operation modelling techniques – such as UML activity diagrams, or BPMN – may be used to model conditions themselves. However in some situations, the additional complexity of designing a simple expression in a visual modelling language may not be desirable. Embeddable textual expression languages, such as OCL [222] and XPath [301], may be used instead in these situations to simplify the instantiation of simple conditional expressions. For example, the UML specification permits the definition of conditional constraints using both English language and OCL [224, pg. 58].

4.4 Lifecycle Modelling

As discussed earlier in Section 2.3.1, the development of certain types of RIAs may be simplified by supporting the concept of lifecycle management [319]. In this thesis, lifecycle modelling does not refer to a software engineering lifecycle, which is the more common usage[6]; but rather refers to the lifecycle of a particular component, scope or application in a software system.

In particular, lifecycle modelling can be considered as the modelling of the different *states* of a component; the *transitions* between these states; and integrating these transitions to fire *transition events* when a state transition has occurred. UML supports this modelling of lifecycle events explicitly through the UML state diagram. Every *Transition* from one state to another can specify a *Behavior* which will be performed when the transition fires, and domain-specific events may be described using *Signals* [224]. *Transitions* can also be specified to execute or capture particular events, including the sending or receiving of *signal actions*.

4.4.1 Implementation-Level Examples

To illustrate the power of modelling the lifecycle of software systems, three examples of lifecycle implementations will be briefly discussed.

OSGi

OSGi is a dynamic component platform for Java, where applications and components (known as *bundles*) may be added, updated, installed or remove at run-time [279]. The OSGi specification provides a component lifecycle for each bundle, represented here as a state diagram in Figure 4.10. The lifecycle events as part of a particular bundle can be listened[7] to by other bundles in order to perform complex bundle lifecycle management.

JUnit

JUnit is a popular framework for developing unit test cases for Java software [11], and is also discussed in further detail in Section 7.9. To assist writing tests, JUnit supports a number of lifecycle methods and annotations, as illustrated in Table 4.2. These lifecycle methods and annotations permit the unit test case developer to extract common functionality, reducing code duplication which would negatively impact the maintainability and understandability of test suites.

[6]For example, Pressman [244] describes the linear sequential and evolutionary models as *lifecycles*, which are instead defined in this thesis as *software process models* in Section 2.7.2.

[7]In OSGi, this is achieved through the BundleListener interface.

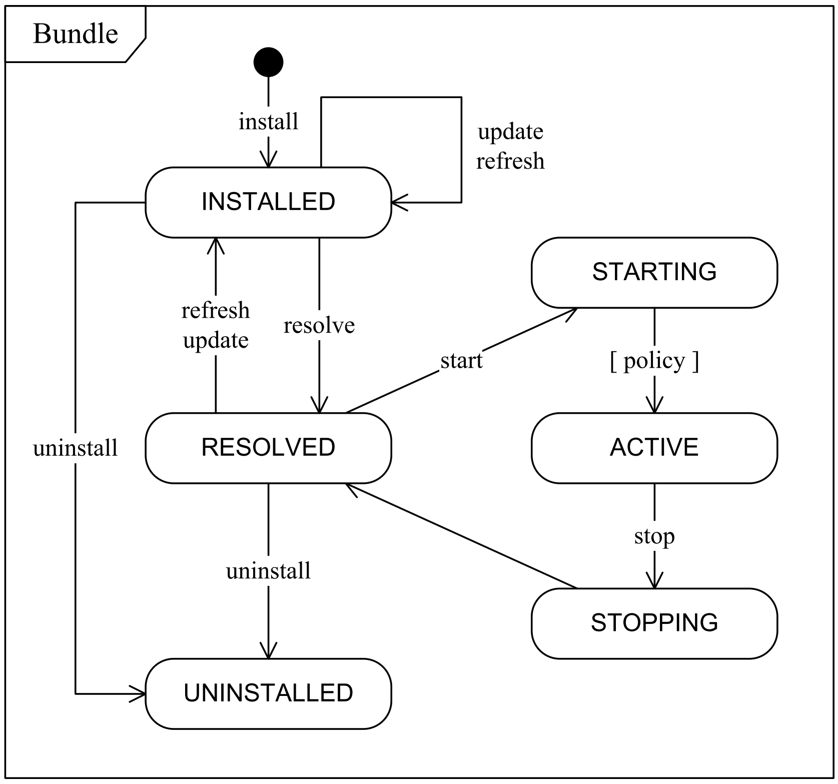


Figure 4.10: Component lifecycle events in OSGi [279], represented using a UML state diagram

Potential Lifecycles of Rich Internet Applications

As part of the survey of existing modelling langauges for RIAs, Wright and Dietrich [319] discuss a range of potential lifecycles as part of a Rich Internet Application. They do not specify a modelling approach or the actual events that may occur, but provide an overview of the types of lifecycles where modelling may be necessary. A summary of these potential lifecycles are reproduced here in Table 4.3.

4.5 Type Systems

Being able to model the definition, access and creation of data is essential for a web application modelling language, and is the basis of all web modelling languages (termed *data-intensive modelling languages* by Ceri et al. [43]) such as WebML [42]. It is still debated whether programming languages

Method	**Annotation**	**Summary**
n/a	@BeforeClass	Initialises the test state before a class containing unit tests are executed.
`setUp()`	@Before	Initialises the test state before a particular unit test.
`testXXX()`	@Test	Executes a particular unit test.
`tearDown()`	@After	Cleans up after a unit test has completed.
n/a	@AfterClass	Cleans up after a class containing unit tests has completed.

Table 4.2: JUnit lifecycle methods and annotations [20]

Layer	Scope	Example Use
Component	elements in the DOM	when a page widget is loaded, move focus to a text box
Request	a single HTTP request	transform XML to WML at end of request
Page	one page can contain many requests over AJAX	close an opened window when the parent page has closed
Session	a visitor has at least one session	delete shopping cart on session timeout
Login	can span multiple sessions, without manual re-authentication	save temporary shopping cart to database on logout
User	represents the user itself	when a user closes an account, delete all their blog entries
Application	represents the entire application	when application is created, create temporary administrator accounts

Table 4.3: Possible lifecycle layers of Rich Internet Applications and their potential use, adapted from Wright and Dietrich [319]

should be typed; while they potentially increase the workload of a developer, a well-designed type system can capture many routine programming errors before the system is deployed [39].

A particular type system can possess a variety of features and designs; for example, the system can be typed or untyped; provide static, or dynamic type checking; and support safe or unsafe typing. These type system definitions are discussed in detail by Cardelli [39], and will be briefly summarised:

- **Dynamically checked language**: "A language where good behaviour is enforced during execution." This often means that an object can have many different types over its lifetime.
- **Statically checked language**: "A language where good behaviour is determined before execution." This often means that an object can only ever have *one* type.
- **Strongly checked language**: "A language where no forbidden errors can occur at run time (depending on the definition of forbidden error)." Strongly checked languages prevent the implicit conversion of one type instance to another.
- **Weakly checked language**: "A language that is statically checked but provides no clear guarantee of absence of execution errors." Weakly checked languages permit the implicit conversion of one type instance to another.

4.5.1 Primitive Types

In a type system, a *primitive type* refers to the smallest datatype that can be expressed, and may also be termed an *atomic type* [295]. In other words, the type can not be expressed as the composition or restriction of any other types in the type system[8]. These primitive types are often then used in the composition of more complex types, and instances are often immutable. Because they refer to the smallest expressible type of data, all languages which possess a type system will have at least one primitive type.

[8]This constraint is discussed in the XML Schema specification: "primitive datatypes are those that are not defined in terms of other datatypes; they exist *ab initio*" [295].

4.5.2 Derived Types

Primitive types can be used to define *derived types* through a number of mechanisms, including composition, restriction, or extension. These mechanisms may be defined formally within the type system, allowing for *type reasoning* to ensure the validity of the typed system [39]. General-purpose programming languages often support restriction-based derived types, such as *subtyping* in object-oriented languages; composition-based derived types, such as *type unions*, are less commonly supported[9]. Derived types may be defined separately in type documents (for example, XML Schema) or directly in the system definition itself (for example, Java classes and interfaces).

One technology for representing type systems in XML documents is *XML Schema* [294]. XML Schema allows the formal definition of the syntax and semantics of a type system, which will be used to construct and verify instances of XML documents. This includes the definition of derived types based on existing types in XML Schema. For example, an *email* datatype can be naïvely defined as the composition of two strings, *identifier* and *host*; i.e., `identifer@host`[10].

XML Schema also defines the *XML Schema Datatypes* recommendation, which provides a hierarchy of built-in datatypes and derived datatypes through restrictions such as range bounding and regular expressions. This includes the definition of a *ur-type*, which is a built-in datatype with "an unconstrained lexical space, and a value space consisting of the union of the value spaces of all the built-in primitive datatypes" (and their subsequent lists) [294]. The hierarchy of these datatypes is illustrated here in Figure 4.11.

TODO Jens feedback: could also briefly discuss OWL here

4.5.3 Type Systems in the Model-Driven Architecture

Type systems can also be integrated into the metamodelling architecture of the MDA, as introduced earlier in Section 3.1.5. The integration of type systems into the MDA specifically is not discussed in any existing standard, but it is fairly straightforward to adapt the metamodelling architecture of XML Schema Structure [294] and XML Schema Datatypes [295] in this way. The resulting architecture is summarised in Figure 4.12.

In this architecture, a custom schema is placed in the metamodel layer (M2), and instances of this schema are placed in the model layer (M1). The real world objects that the schema instance represents remain in M0. Importantly, the XML Schema datatypes are also part of the metamodel layer (M2), because the custom schema can refer to these datatypes. Custom datatypes as defined by the schema are also placed in M2. This is because these datatypes can be considered the metamodel for the actual datatype instances in M1.

The concept of modelling spaces [69] as discussed earlier in Section 3.1.6 can be applied to simplify the complexity of this architecture, and to aid in its comprehension. This allows the metamodelling architecture to be split into the two modelling spaces of a *Structure Space* and a *Datatype Space*, as illustrated in Figure 4.12. It is important to note that the definition of XML Schema datatypes may simultaneously reside in *both* M2 and M3 at the same time. Djurić et al. [69] argue that this is acceptable according to the model-driven architecture.

[9]One notable exception is the Web Ontology Language (OWL).

[10]The definition of a valid e-mail address is much more complex according to RFC 5322 [248], as discussed later in Section 5.4.2.

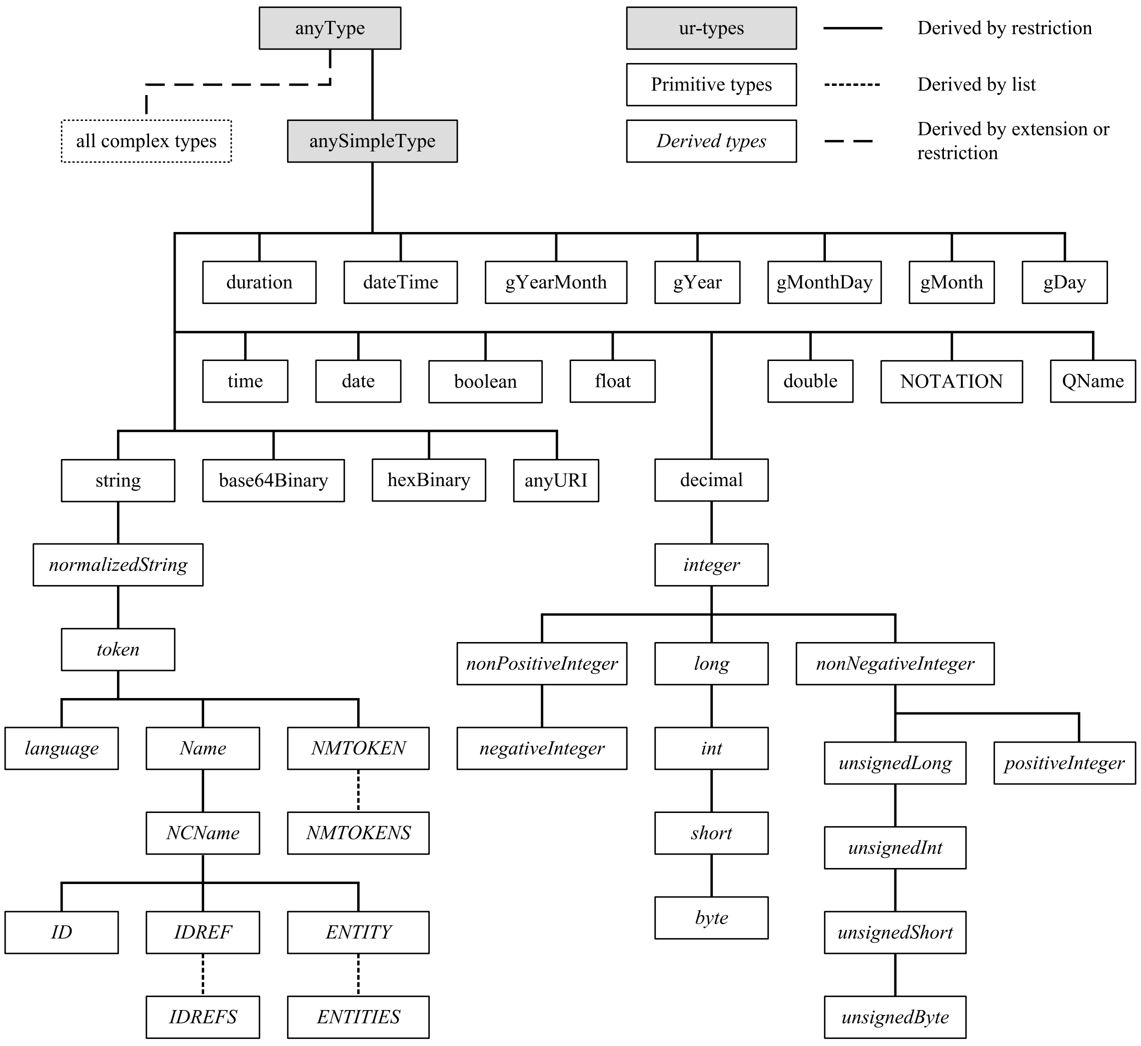


Figure 4.11: Built-in datatype hierarchy of XML Schema Datatypes, adapted from W3C Group [295]

4.6 Domain Modelling

The range of data and datatypes used in a web application is rarely limited to primitive types, such as strings and integers. Depending on the web application, the *domain* of the web application needs to be modelled in terms of the data necessary to model the application. For example, a shopping web application will likely need to model domain concepts such as products, taxes, and so on.

The modelling of these domain-specific data structures is known as *domain modelling*, which often involves the extension and composition of primitive types. The definition of a particular structure is known as a *schema* or *domain schema*, and is often conceptually separate from the method necessary to access or modify a particular instance of the schema. Instances of schemas can be stored using a variety of technologies, such as relational databases, object-oriented databases, or other structured storage methods[11]. In this section, a variety of techniques for modelling domain schemas will be briefly discussed.

[11]*Non-relational* databases, known as NoSQL, falls under the "structured storage" category [178].

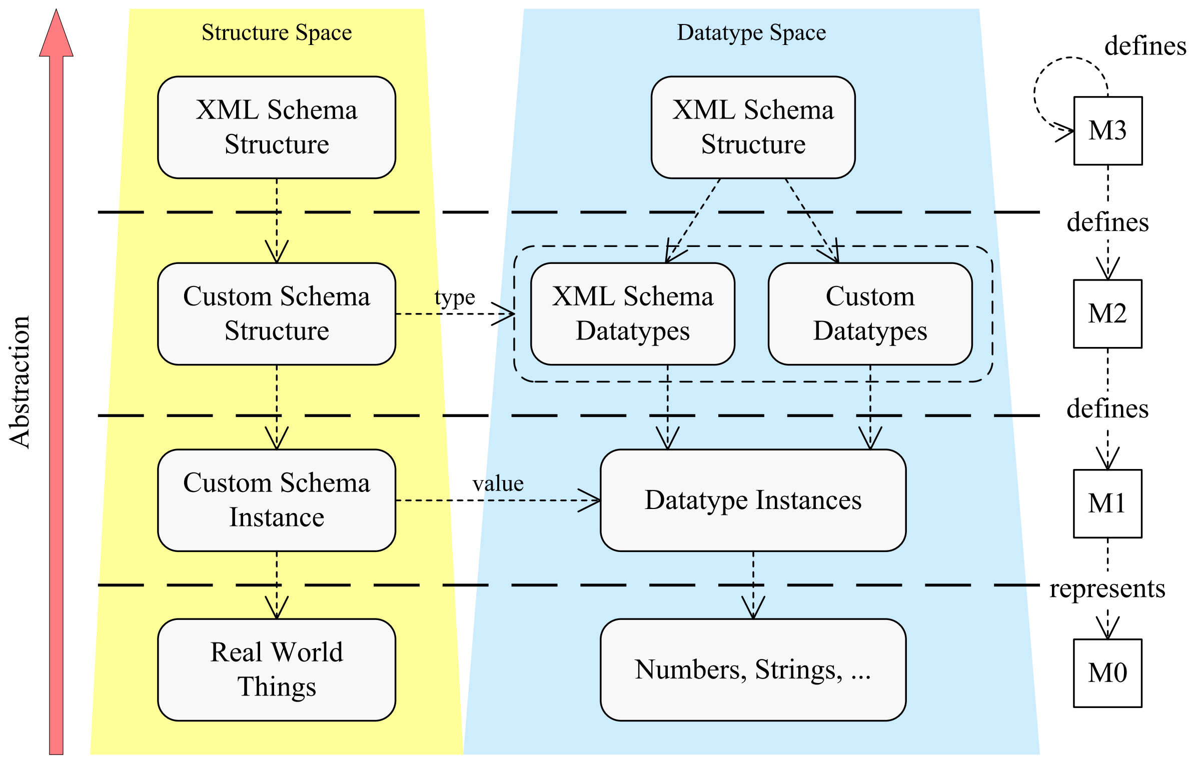


Figure 4.12: Representing XSD Schema Datatypes within the metamodelling architecture of MDA using modelling spaces

4.6.1 ER Diagrams

Entity-Relationship (ER) diagrams can be used to model relationships between entities; these diagrams were originally proposed by Chen [46] for the design of relational database systems, and have been extended by others [244, pg. 307]. For example, Figure 4.13 shows a system where a *Car* has a relationship to a *Manufacturer* through the relationship *builds*. The relationship edge arrowheads illustrate the modality and cardinality between the relationships.

ER diagrams cannot model the structure of the entities themselves; that is, ER diagrams cannot show that a *Car* has a *colour*. In this case, a *data object table* or schema is often also supplied to complete the model [244]. This data object table is also illustrated in Figure 4.13.

Because ER diagrams only illustrate the relationships between entities, they are good at giving an overall understanding of a system. However, the relationship edge arrowheads can be confusing to

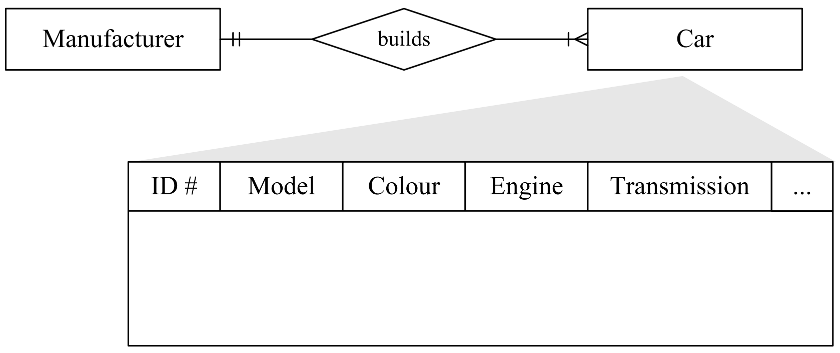


Figure 4.13: A sample ER Diagram and Data Object Table, adapted from Pressman [244]

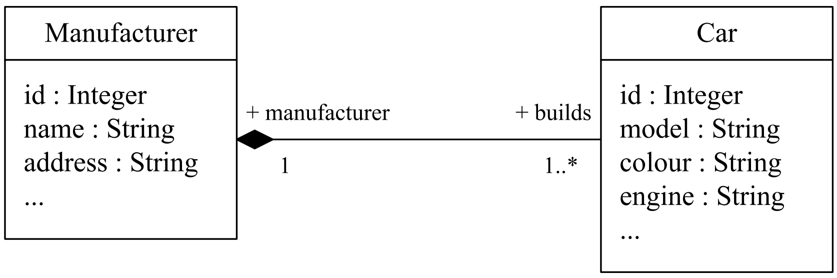


Figure 4.14: A sample UML class diagram [224]

people without experience in the ER modelling field, making ER diagrams unsuitable for communication with non-technical stakeholders. Finally, unlike the other three technologies discussed in this section, ER diagrams do not support the concept of subtyping or type inheritance.

4.6.2 UML Class Diagrams

UML class diagrams [224] include both the entity relationship modelling of ER diagrams and the schema definition of data object tables. For example, Figure 4.14 models the same Car/Manufacturer system as the combined ER/data object table example in Figure 4.13. Modelling the structure is optional, and this can simplify the diagram into a format similar to ER diagrams. Because UML class diagrams use text (e.g. “1..*”) instead of line edges to mark modality and cardinality, these diagrams can be easier for non-technical stakeholders to understand.

UML class diagrams are also much better suited for modelling object-oriented software systems than ER diagrams, as they support type system concepts such as abstract classes, interfaces and type inheritance. UML class diagrams can also support more advanced types of relationships, such as aggregation, association and composition.

4.6.3 XML Schema

As discussed earlier, XML Schema [294] can be used to define derived datatypes through the composition, restriction or extension of other datatypes. XML Schema can also be used for domain modelling by defining *complex types*, by defining each entity as an `element` associated with a `complexType` representing the entity’s schema.

However, XML Schema cannot directly model the constraints necessary to match the UML class diagram in Figure 4.14; in particular, it seems difficult (if not impossible) to enforce that both ends of the relationship are correct[12]. It is also difficult to enforce cardinality constraints based on primary or foreign keys; in some situations, this can be achieved by adding regular expressions to the attributes.

An XML Schema is therefore not suitable for domain modelling except in the simplest of circumstances due to its lack of expressiveness. However, it still remains suitable as a first step for validating model instances represented in XML, if the model instance is loaded using a *schema-validating XML parser* [294].

[12]That is, a car’s manufacturer must also have the same car listed in the list of cars it has manufactured; this constraint could not be expressed in Figure 4.14.

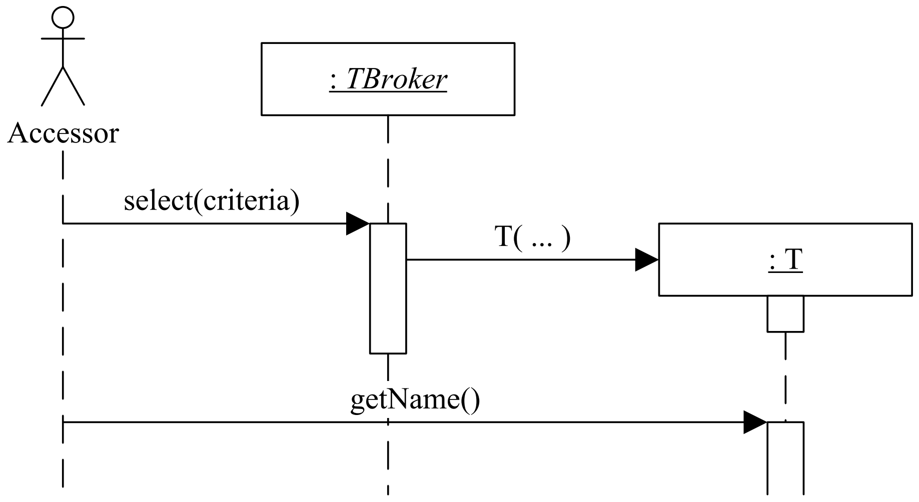


Figure 4.15: The Database Broker design pattern represented as a UML sequence diagram, adapted from Bennett et al. [23, pg. 469]

4.6.4 EMF Ecore

As discussed in greater detail later in Section 6.2.1, the Eclipse Modeling Framework (EMF) may be used to define metamodels for model-driven approaches [275]. The underlying meta-metamodel for EMF metamodels is the Ecore metamodel, which is defined in terms of itself. This self-definition allows for Ecore to reside at the M3 layer of the metamodelling architecture of MDA. As discussed earlier in Section 3.1.5, the Eclipse Modeling Framework was built on EMOF [?] and model instances can be serialised directly to EMOF [275, pg. 40].

One issue with the Ecore metamodel is that it only permits a single attribute on a given EClass to be the *ID* for the EClass; that is, it is not possible to define the concept of a multi-valued primary key, which is a common scenario for relational databases used in the web application domain. This problem is discussed in further detail later in Section 5.5.2.

4.7 Design Patterns for Data Access

In the previous section, three modelling technologies for describing domain schemas have been described; however, none of these technologies specify how domain data may be accessed as part of a modelled application. *Design patterns* formally document a solution to a common design problem in a technology-independent way [103]. In this section, the common design patterns of *Database Broker* and *Iterator* will be briefly discussed in terms of UML diagrams.

4.7.1 Database Broker

Bennett et al. [23, pg. 469–474] propose the *Database Broker* design pattern as one form of data access, using a TBroker intermediary class which retrieves and stores instances of a particular class T. This data access pattern is illustrated by the sequence diagram in Figure 4.15. Bennett et al. use class inheritance on the abstract supertype *TBroker* to define how the data instances are stored; for example, the broker may be defined by the subtype RelationalBroker of the in order to access a relational database.

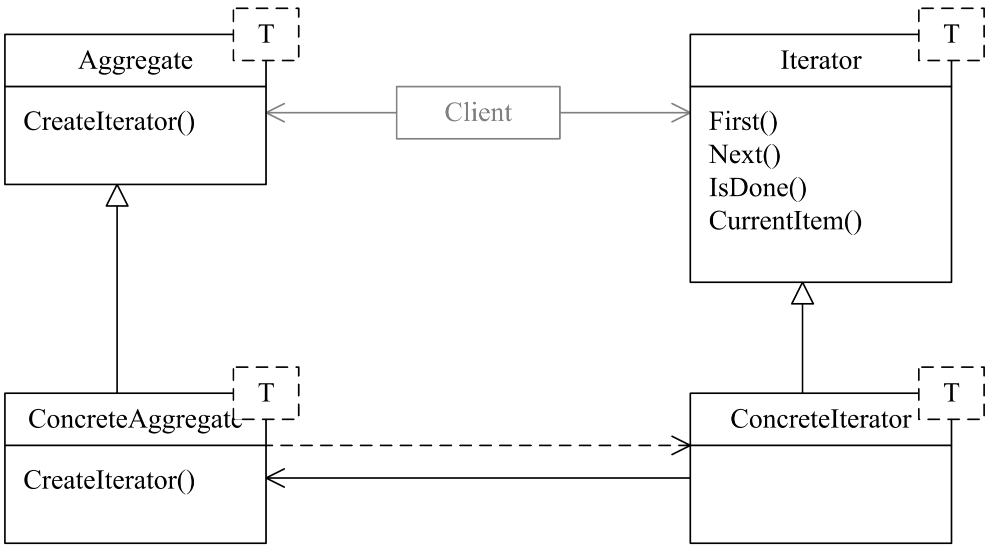


Figure 4.16: The Iterator design pattern adapted to specify parameterised types, adapted from Gamma et al. [103] and using a UML class diagram syntax [224]

Because the Database Broker can abstract away from the underlying representation of the data storage (e.g. relational database, in memory, etc.), a broker can simplify the maintainability of the system, because different storage representations can be used transparently. However, if a Database Broker can only return one instance from a select, it is necessary to execute many queries to retrieve multiple instances. If a broker can return a list of elements instead, it is easier to retrieve multiple instances, but simply returning a list of elements can be detrimental if there may be many results.

4.7.2 Iterator

The *Iterator* design pattern "provides a way to access the elements of an aggergate object sequentially without exposing its underlying representation" [103, pg. 257–271]. That is, an Iterator can be used to access a source of data without needing to understand its storage representation, whether it is a relational database or simply in memory. The Iterator design pattern is illustrated here in Figure 4.16.

The standard Iterator design pattern does not specify that a given Iterator can be applied to many different types, such as through parameterised types[13]. In Figure 4.16, these parameterised types have been added to the standard Iterator design pattern, represented here using the UML syntax of *template parameters* [224, pg. 633].

Both the Iterator and Database Broker design patterns include instances of the *Abstract Factory* design pattern [103, pg. 258], which can improve the maintainability of the designed system as the underlying source of the data can be abstracted away. The current state of the Iterator may be stored as a *cursor* to permit the same access algorithm to be used across different sources of data [103, pg. 261]. Importantly, an Iterator design pattern may also improve the scalability of the system; because an Iterator can fetch results "on-the-fly", computationally expensive methods can be deferred until the result is actually necessary [66].

One drawback of the Iterator design pattern is that each Iterator may keep a connection open to the original source of the data until the Iterator is finished or destroyed; in web applications, this is particularly important if the original source is a database. In this case, it is important to mark the

[13]In the Java language, this is known as *generic types* [37].

Iterator as *closable*, and the application can close the Iterator as necessary to release resources back to the system.

4.8 Users and Security

As discussed earlier in Section 2.3.1, a *user* represents both people and non-human entities that need to access a particular web application, and it would be beneficial to support user modelling in a RIA modelling language. User modelling is very closely related to security modelling, as user authentication often requires both authorisation and authentication.

There are many different techniques and methods for describing secure systems, and for describing the role of users in such a system. Depending on the scope and requirements of the modelled system, the security model can either be very simple, or very detailed. In this section, a number of existing access control methods will be discussed; however, a full discussion on the security properties of each of the following systems is well outside the scope of this thesis.

Because security is generally an overarching property of the system, the complexity management techniques discussed earlier in Section 4.1 may be applied to security modelling. For example, aspect-oriented modelling can be used to define system security separately from system functionality; UWE uses this approach to model access control in web applications by using UML state machines [322].

As Ceri et al. [44] discuss, although some user information is generic – for example, most web application users have an e-mail address and a password – other information is domain-specific. A modelling language for web applications therefore needs to support profile modelling for specific domains, whilst still supporting common user modelling scenarios. In many ways, user modelling can be considered a form of domain modelling, and this concept is discussed further in Section 5.9.1.

4.8.1 Access Control Lists

An *Access Control List* (ACL) is perhaps one of the simplest forms of user security in computing, and is extended in many of the approaches discussed in this chapter. ACLs allow an object to be associated with a list of permissions against particular users and groups. When the object needs to be accessed securely by a client, the client is evaluated against the list of permissions provided by the ACL in order to evaluate its current permissions.

Traditional Unix permissions are a simple form of ACLs applied to filesystems, where every filesystem object is provided with three permissions (read, write, and execute) for three types of users (owner, owner's group, and other users). Unix permissions are widely used in computing and in the deployment of web applications. However, Unix permissions can become too simple and limiting when different objects regularly need to be shared with specific people. Since permissions cannot be assigned per-user but only per-group, providing the same permission to two users involves the creation of an entirely new group.

4.8.2 Discretionary Access Control

Discretionary Access Control (DAC) policies are based on individual users, and on access rules stating what permissions each user possesses [257]. Every user in the system has a set of permissions against every object in the system, and this set may be empty. These rules are often represented using access

	File 1	**File 2**	**File 3**	**Program 1**
Ann	own read write	read write		execute
Bob	read		read write	
Carl		read		execute read

Table 4.4: Example of an Access Matrix, adapted from Samarati and di Vemercati [257]

matrices as in Table 4.4, but as the matrix is usually sparse, policies are often implemented using ACLs instead [118].

As secure systems often deal with large numbers of both users and objects, DAC policies can quickly become overwhelming. Without a concept of roles or groups, it can be difficult to accurately apply new permissions or objects to a common group of users. However since permissions are defined explicitly and per-user, it is not possible to override any of these permissions from elsewhere in the system, improving the inherent security of the approach.

4.8.3 Mandatory Access Control

Mandatory Access Control (MAC) policies are based on mandated regulations, determined by a central authority [257]. Groenewegen and Visser [118] explain how this policy can be implemented on assigning labels (e.g. *TopSecret*, *Secret*, *Unclassified*) to objects, and users are given a clearance label that indicates their access level. The relative importance between labels is defined by a partial order, and users can only create objects at a level they can access. The central authority regulations are then used to determine access permissions for a particular user against a particular object.

As Groenewegen and Visser argue [118], MAC policies are mainly aimed at preserving confidentiality of information, by preventing the unsafe transfer of information contained within objects to other security labels. The use of a MAC-based system is made easier if labels can override lower-level labels, but the resulting partial order between labels can cause problems if label order is not evaluated as a whole.

4.8.4 Role-based Access Control

The *Role-Based Access Control* model (RBAC), as described by Sandhu et al. [258] and proposed as a NIST standard [82], allows a great deal of flexibility in defining access and permissions. RBAC is essentially an extension of ACLs with the concepts of roles, in order to support the large-scale management of permissions and roles. A set of roles are defined, and each role provides particular permissions; these roles can then be globally applied to individual users, or temporarily applied within a session.

Compared with RBAC, MAC does not support individual permissions, however both supports a partial ordering on a hierarchy of roles. Samarati and di Vimercati [257] argue that as RBAC is more expressive, it is attracting increasing attention as an alternative security model to the traditional DAC and MAC models. Groenewegen and Visser [118] argue that the main benefits of the RBAC model include that role assignments are separated from users and permissions; many applications naturally consist of a hierarchy of roles; and a user can activate the minimum role able to perform a task, increasing security.

4.9 Describing Reusable Patterns within a Metamodel

When designing many different systems within a particular domain, there are often a number of common patterns of functionality that emerge. As an example in the RIA domain, it is a common requirement to keep the values of two objects synchronised; for example, keeping a form on the client synchronised with its stored representation on a server[14]. A RIA modelling language could therefore provide some way of reusing these common patterns, named *reusable patterns* in this thesis[15].

4.9.1 Independence of the Reusable Pattern Metamodel

One option for modelling reusable patterns is to integrate the patterns directly into the same metamodel; for example, a Text Field could have a reference "synchronised with", which references another Text Field in the model. This approach is useful if there is a predetermined set of potential patterns; however, as supported patterns are directly encoded into the metamodel, it would be difficult for a third party to create new patterns, as the metamodel must be modified directly.

Alternatively, these patterns could be supported using aspect-oriented modelling, discussed earlier in Section **??**. Here, pattern instances are defined as an instance of an entirely separate metamodel (a "patterns" metamodel), separating the additional functionality of the patterns from the base model itself. With this approach, it is easy for third parties to extend or create new patterns using this approach, and the set of available patterns does not need to be restricted.

4.9.2 Implementation of the Reusable Pattern Metamodel

Once a metamodel can support the instantiation of common reusable patterns as part of a model instance, it is also necessary to describe how the pattern will actually be implemented. With respect to the "synchronised with" reference discussed above, this would involve the creation of methods, events and any other related functionality necessary to implement the intent of the reusable pattern.

One option would be to implement the reusable pattern logic as part of the model transformation – or code generation – process of the metamodel. This would mean that the reusable pattern must be integrated into the model transformation, making it difficult for third parties to extend these patterns, or to create new patterns. This approach would also make it difficult for a model developer to override the functionality of a reusable pattern, as the pattern has essentially become an intent of the metamodel itself.

Alternatively, if the source metamodel already supports the manual definition of the reusable pattern, then *model completion* could be used to implement the reusable pattern within the same model instance. This approach leaves the model transformation stage intact, as it moves the pattern logic into the model completion stage, as discussed in Section 3.2. This can also be considered a form of aspect-oriented modelling as the reusable pattern logic is removed from both the metamodel and the implementation into a separate completion process.

[14]This design pattern is described as a Sync Wire later in Section 5.8.1.

[15]This nomenclature is provided to distinguish these types of patterns from the concept of architectural software *design patterns* as discussed by Gamma et al. [103].

Both of these approaches could be simplified by using additional abstraction layers in the language implementation. That is, an intermediary metamodel can be used to make the source and target metamodels smaller, at the expense of additional steps in the modelling environment. As discussed earlier in Section 2.4.3 this is the approach taken by UWE, where each metamodel instance represents a particular aspect of the intended design [168].

If both the design and implementation of reusable pattern modelling support is achieved in an aspect-oriented modelling way, then the reusable pattern could be removed entirely from the base metamodel and considered an independent extension. The benefit of this approach is that the metamodel and model transformations are not polluted by a range of reusable patterns, and would also improve third-party interaction. However, the drawbacks of aspect-oriented modelling as discussed earlier – such as understanding the interaction between two aspects – remains a concern.

4.10 Visual Notation

The development of visual representations for modelling languages is a very large research area, and is not the main focus of this research; a full discussion on the existing research into visual notations is therefore well outside the scope of this thesis. However, this research will perform a preliminary evaluation onto the efficiency and suitability of using a visual representation for a RIA modelling language. This section will summarise three evaluation concepts relating to the design of visual languages.

4.10.1 Notation Information Capacity

To evaluate the effectiveness of a visual notation, Moody [208] proposes measuring *cognitive effectiveness* through the systematic evaluation of a number of notation variables. By measuring these variables in a quantitative way, the effectiveness of two similar visual notations can be effectively compared, and this section will discuss two of these approaches.

The first approach – *ontological analysis* – measures the effectiveness of mapping the visual notation to the underlying model instance. In particular, four detrimental anomalies are identified, with respect to a desirable one-to-one mapping between constructs and notation [208, pg. 759]:

1. **Construct deficit**, a model concept does not have a corresponding visual notation;

2. **Construct redundancy**, multiple visual notations can represent a single model concept;

3. **Construct overload**, a visual notation can represent multiple model concepts; and

4. **Construct excess**, a visual notation does not represent any model concepts.

The second approach focuses on measuring the *information capacity* of a visual element with respect to a number of different visual variables, each of which have a particular information capacity. For example, Moody argues that we can distinguish between an unlimited number of element shapes, but we can only distinguish between a few element textures. The capacity of various visual variables as discussed by Moody are illustrated in Table 4.5; in particular, *power* represents the highest level of measurement that can be encoded, and *capacity* represents the number of perceptible steps.

Variable	Power	Capacity
Horizontal position (x)	Interval	10–15
Vertical position (y)	Interval	10–15
Size	Interval	20
Brightness	Ordinal	6–7
Colour	Nominal	7–10
Texture	Nominal	2–5
Shape	Nominal	Unlimited
Orientation	Nominal	4

Table 4.5: Information encoding capacity of different visual variables, adapted from Moody [208, pg. 770]

4.10.2 Cognitive Dimensions Framework

The *cognitive dimensions* framework proposed by Green and Petre [116] defines thirteen dimensions for the subjective evaluation of user interfaces. This framework may be used to evaluate the usability of visual modelling languages, as illustrated by Grundy et al. [121] in their evaluation of the View Mapping Language (VML).

A summary of these thirteen dimensions, along with the evaluation of these dimensions against a proposed visual representation for modelling RIAs, is provided in Appendix **??**. As discussed by Green and Petre [116], no given notation can satisfy all of these dimensions simultaneously; designers must understand the trade-offs between these dimensions in order to select the best compromise for a given visual notation.

4.10.3 Graphical Language Guidelines

Rumbaugh [256] proposes a list of twelve guidelines for the development of graphical modelling languages. A visual notation cannot satisfy these particular guidelines in a way that can be measured quantitatively, as earlier in this section. Nevertheless, some of these guidelines – such as "*must fax and copy well using monochrome images*" and "*easy to draw by hand*" – are still useful to consider when designing a visual notation.

Many of the guidelines proposed by Rumbaugh can be measured in terms of the guidelines proposed by Moody [208]. For example, Rumbaugh's "*no overloading of symbols*" guideline can be considered equivalent with Moody's guideline of *construct overload*; and "*uniform mapping*" equivalent to Moody's *ontological analysis* goal of having a one-to-one mapping between concepts and constructs [208].

4.11 Conclusion

The development of a modelling language for RIAs requires support for modelling many domain concepts – such as events, types and lifecycles – and each of these modelling concepts may be implemented using a number of existing approaches. In this chapter, a variety of approaches for each of these modelling concepts have been evaluated and discussed, along with comparisons to existing work where relevant. In the next chapter, these design concepts will be integrated into a new modelling language for RIAs.

Chapter 5

The Internet Application Modelling Language

This chapter will provide the definition of the *Internet Application Modelling Language* (IAML), a modelling language for Rich Internet Applications. It will informally discuss the design and behavioural aspects of the language, the rationale behind each design decision, and illustrate examples of the visual representation of the language. The complete definition of IAML, including the syntax, inference rules and implementation guidelines for each language element is provided in Appendix **??**.

Throughout this chapter excerpts of an example web application, *Ticketiaml*, will be provided to illustrate the use of certain IAML metamodel elements. As described later, this application is the IAML-based implementation of the *Ticket 2.0* benchmarking application, and its full definition is provided later in Appendix **??**.

5.1 Development Approach

Earlier in Chapter 2, we discussed the requirements of RIA modelling languages and how no existing language could completely model all of the features of RIAs. In this section, we will now fully define the scope for this research, and define the approach that will be taken in the development of our language, which will be called the *Internet Application Modelling Language* to reflect the intended final scope of the language.

5.1.1 Requirements Planning

Once the requirements for a RIA modelling language had been identified earlier in Section 2.3.2, it became clear that the resources necessary to implement such a language together with a proof-of-concept implementation far exceeded the resources available in a single Ph.D. The target set of requirements will therefore be simplified to a smaller set by omitting rarely-used features of RIAs.

After investigating our list of 59 modelling requirements, we found that some of these requirements could be characterised as describing the system complexity, distributed functionality, reliability and performance requirements of enterprise-level applications. These requirements were focused on describing large RIAs, rather than the features that made RIAs considerably different from other web applications. By removing these requirements, RIAs may be classified into two types based on their characteristics: *Full RIAs* and *Basic RIAs*.

Category	Requirements
Offline data	D7, D8
Server transactions	E9, E10
Multiple interfaces	U1, U9
Plugins and dynamic scripting	A6, A8, A9
Internationalisation	T4
Multiple domain support	T5
User collaboration	E7
Import external libraries	U11

Table 5.1: Additional Requirements for *Full RIAs*

The requirements set of a *Basic RIA* can be used to describe and construct the vast majority of Rich Internet Applications on the Internet, such as simple mashups, personal web sites, and small online shopping sites. Larger enterprise-level applications such as web-based e-mail and social networking platforms can instead be described as a *Full RIA*. A *Full RIA* has the modelling requirements of a *Basic RIA* with the additional thirteen requirements in Table 5.1; this reduces the feature scope for modelling Basic RIAs to a more reasonable 46 requirements (a 22% decrease).

This requirements restriction is not detrimental to the overall contributions of this thesis; as discussed earlier in Wright and Dietrich [319], no existing approach can model either Basic or Full RIAs, so the proposal and implementation of a modelling language for Basic RIAs will still be a significant research contribution. Nevertheless the language will be designed to support Full RIAs in the future where possible, to allow the implementation of these missing requirements as future work.

5.1.2 Modelling Language Development Approach

As discussed earlier in Section 2.7, there are three methods of developing a language: the extension, or restriction of an existing language; or an abstraction into a new language. The approach taken depends on how well existing languages match our intended domain. However, since no existing language maps well onto RIA concepts, and many RIA concepts do not easily map into existing languages such as UML, the extension or restriction of an existing language is not a desirable approach.

This research proposes abstracting existing RIA platforms into a *new* domain-specific language, but also reusing concepts and semantics from established languages. This approach is advocated by Kelly and Pohjonen [157], who argue that it is a good idea to reuse basic ideas and concepts such as data flow, control flow and type inheritance in the development of new modelling languages.

For example, UML defines UML class diagrams [224] which are generally accepted by industry and extensively used to model the schema and structure of object-oriented systems [23]. The concepts and semantics of UML class diagrams could therefore be reused and adapted to modelling schemas or objects within RIAs, without the language being burdened by having to implement the entire UML specification[1].

[1]**TODO:** Could mention how UML is trying to focus on developing a smaller kernel UML, simplifying implementation – or simply discuss other existing metamodel kernels.

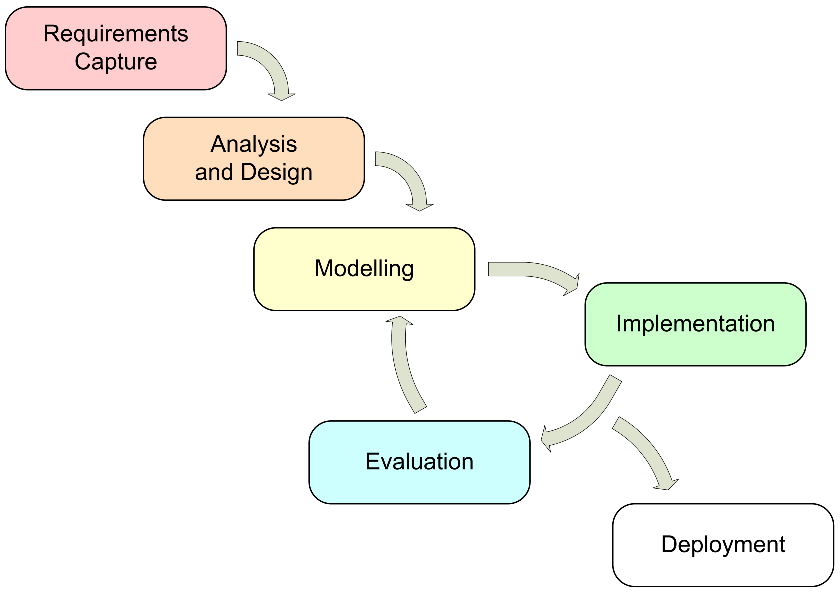


Figure 5.1: The hybrid modelling language process model used in the development of IAML

5.1.3 Software Process Model

This research will attempt to capture the benefits of both the linear sequential and evolutionary software process models, discussed earlier in Section 2.7.2. Since neither approach is purely suitable for research as discussed earlier, a hybrid approach may be beneficial, as illustrated in Figure 5.1. This approach is very similar to the *MDA software development process* advocated by Kleppe et al. [163]. The only difference is that as a research project, the analysis and design is done in a sequential manner.

In this hybrid approach, the *requirements capture* and *analysis and design* steps will be performed sequentially, to ensure that all of the desired requirements will be captured. This hybrid approach also includes the development of a proof-of-concept implementation of the language; as discussed earlier in Section 2.3.1, this implementation can be used as a reference implementation, and increase acceptance of the language within development communities [89].

The development, implementation and validation steps will be performed iteratively and in an evolutionary manner. This will ensure that this research will at the very least result in a fully validated partial language, rather than a complete language that cannot be fully validated. To improve the quality and reliability of the software implementation, some of the concepts of test-driven development, as discussed earlier in Section 2.7.3, will also be incorporated into the implementation process.

5.1.4 Evaluation

As discussed earlier in Section 1.3, the research method underlying this thesis proposes a number of methods in which to evaluate the proposed modelling language. In Chapter 8, five different evaluations will be performed:

1. *Feature Comparison Evaluation:* Earlier in Section 2.4.7, a feature comparison was performed to compare existing modelling languages for web applications against RIA features described in Wright and Dietrich [319]. These requirements will therefore be re-evaluated against the completed IAML metamodel, to compare the functionality of the proposed IAML metamodel with these existing languages.

2. *Modelling Requirements Evaluation:* The list of detailed modelling requirements for RIAs [318], discussed earlier in Section 2.3.2, will be evaluated against the proposed language in a requirement-matching evaluation. This comparison will identify any language requirements of *Basic RIAs* that are not yet supported.

3. *Benchmarking Application Evaluation:* The implementation of the proposed RIA benchmarking application, *Ticket 2.0* [318], will be used to illustrate that the language can be used to implement real-world applications, and that the modelling language is not excessively complex. In order to highlight any differences in development effort, the system metrics defined earlier in Section 2.6.3 will be used to compare the IAML-based implementation with the manually implemented benchmark.

4. *Metamodelling Metrics Evaluation:* The evaluation of the metamodelling metrics defined earlier in Section 2.6.2 will be used to compare the IAML metamodel with other similar metamodels, illustrating if the complexity of the metamodel could be considered unexpected.

5. *Visual Notation Evaluation:* The syntax and implementation of the visual notation of IAML model instances will be evaluated using two approaches discussed in Section 4.10. That is, the syntax will be compared by evaluating the information capacity of its notation, discussed by Moody [208]; and the implementation will be evaluated using the *cognitive dimensions* framework, discussed by Green and Petre [116].

5.1.5 Design Goals

In order to clarify the intent of the proposed modelling language and to improve understanding of the necessary decisions that will be made during its design, it will be helpful to summarise the *design goals* of this new language as follows:

1. The IAML metamodel will fully support the 46 requirements of *Basic RIAs*, with consideration also given to the additional thirteen requirements of *Full RIAs*, as discussed earlier in Section 5.1.1. This design goal is evaluated later in Section 8.2.

2. IAML will reuse concepts from existing modelling languages where appropriate, as discussed earlier in Section 5.1.2. This design goal is evaluated later in Section 8.1.

3. IAML will be supported with a visual notation to improve the end-user effectiveness of designing model instances, as discussed earlier in Section 3.3. The implementation of this design goal is covered later in Sections 5.14 and 7.4.3.

4. The proof-of-concept implementation of IAML will be implemented iteratively in an evolutionary manner, and the metamodel will be frequently refactored as appropriate, as discussed earlier in Section 5.1.3. This implementation is introduced in greater depth in Chapter 7.

5. IAML and its proof-of-concept implementation will be published under an open source license to encourage use within industry, seek community feedback and to reduce defects, as discussed earlier in Section 2.7.4. This design goal is discussed in greater depth later in Section 7.1.1.

6. During its development, IAML will be frequently evaluated according to the evaluation criteria discussed earlier in Section 5.1.4, in order to guide its development in an evolutionary manner. The results of these evaluations are provided later in Chapter 8.

7. The implementation of IAML will be supported by a suite of model instance verification tools, in order to aid in the development of correct web applications. This goal was discussed earlier in Section 1.5, and evaluated later in Section 7.7.

5.2 Metamodel Design Principles

Before the IAML metamodel is discussed, it is important to identify the principles used to answer design decisions that will arise during the development of the language, along with techniques for evaluating these principles.

5.2.1 Language Design Principles

The principles of design for a modelling language can be adapted from the principles of designing programming languages. For example, Hoare [136] introduces concepts such as simplicity and efficient object code as general design criteria. On a more practical level, Wirth [315] discusses some of the design decisions that need to be considered, such that mathematical formalisms of basic abstractions may identify inconsistencies[2], and a language is useless without an implementation and documentation.

Programming language design principles can be adapted to instruct the design of modelling languages; for example, Karsai et al. [155] proposes 26 guidelines on the design of domain-specific languages. While it is desirable for a modelling language to satisfy all of these principles and guidelines, many of these principles conflict with each other [155]. For example, adding language concepts to increase the functionality of a language directly conflicts with the principle of keeping a language simple.

This thesis will select four design principles as particularly important to guide the design of the IAML metamodel, with respect to the research questions proposed at the start of this thesis. In particular, IAML will be developed primarily according to the design principles of *scalability*, *simplicity*, *consistency* and *standards-compliance*.

Scalability

As discussed by Paige et al. [234], "the principle of *scalability* states that a modelling language should ideally be useful for both small and large systems." This can be measured by evaluating the modelling environment; for example, different views can be provided under the discretion of the model developer that hide unnecessary detail. The design of IAML to support hierarchical model instances is one way of satisfying this design principle, as discussed in Section 5.2.2.

Simplicity

Paige et al. describes a *simple* language as one that is "small and memorable" [234]. Karsai et al. suggests many guidelines to achieve simplicity, such as limiting the number of language elements; avoiding unnecessary generality; and avoiding conceptual redundancy [155]. One useful set of metrics to evaluate simplicity should therefore be the metamodelling metrics discussed earlier in Section 2.6.2. In general, a metamodel with a fewer number of model elements should be a simpler metamodel.

[2]Wirth also highlights how a balance must be achieved between formal and informal documentation; in particular, arguing that a "formal definition cannot be a substitute for an informal presentations and for tutorial material" [315, pg. 29].

Kelly and Pohjonen [157] also argue that developing a domain-specific approach is not about achieving perfection, but about developing an approach that works in practice: “It will always be possible to imagine a case that the language can’t handle.” They advocate concentrating on the core concepts in a domain, and initially building a prototype language for these features. Even languages such as UML are still undergoing evolution; at the time of writing, UML has been through 17 years of iterative development[3].

Consistency

Paige et al. [234] describes the principle of *consistency* within modelling languages as that “there is a purpose to the design of the language” and that all features of the language must support this purpose, and this definition is used in this thesis. As the purpose of IAML is to support the design and implementation of Rich Internet Applications, this means that every element within the language must have an analogous component in the web application domain.

For example, web applications should not support the execution of arbitrary commands on the server, as this presents security and scalability issues; IAML should therefore never include support for such primitives. Similarly, IAML is purposed to design RIA functionality, and not on the layout of user interfaces; IAML therefore delegates this responsibility to the CSS language [**?**].

The consistency of a modelling language must not be confused with the consistency of the model instances produced using the language [234], which may be evaluated using model instance verification as discussed earlier in Section 3.4. Similarly, modelling language consistency must not be confused with the consistency of the textual or visual *syntax* of the language, which is covered earlier as part of the cognitive dimensions framework in Section 4.10.2.

Finally, consistency can also be considered in terms of the high-level design of model instances, which is also a different concept to modelling language consistency. Depending on the development environment, design consistency can be addressed through naming patterns, coding conventions [**?**, pg. 875–902] and reusing architectural design patterns [103].

Standards-compliance

Standards-compliance refers to the cooperation of the language elements to existing specifications and standards, which should occur wherever possible either through composition or definition reuse [155]. This design principle is already reflected as a design goal of the language in Section 5.1.2. Importantly, standards-compliance may be at odds with simplicity and consistency principles, and may impact on proof-of-concept implementations; a metamodel which reused the *entire* UML metamodel would be much difficult to implement than one which only reused the UML state diagram metamodel.

5.2.2 Hierarchical Modelling Approach

In terms of the three complexity management techniques discussed earlier in Section 4.1, the IAML metamodel is designed to use a hierarchical modelling approach. Such an approach will allow model developers to see a high-level overview of a modelled system, but also zoom into the individual details of a particular model element. Each level of the hierarchy is designed to support a particular domain scope, as illustrated in Figure 5.2.

[3]As discussed by [**?**], the development of UML began in late 1994.

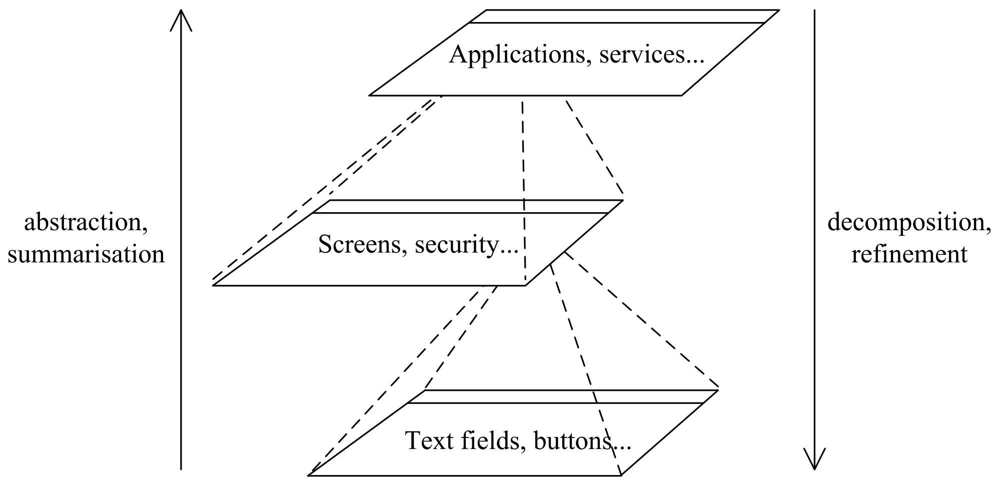

Applications, services...
Screens, security...
Text fields, buttons...
abstraction, summarisation
decomposition, refinement

Figure 5.2: Hierarchical Modelling in IAML

It is important for each viewed layer to provide appropriate context to the model developer, to orient the current view with the rest of the modelled system. In the IAML metamodel, it is envisaged that the proof-of-concept implementation can be used to provide this context. Hierarchical modelling is also very easy to integrate into the *model completion* framework, discussed earlier in Section 3.2.

A benefit of using a hierarchical modelling approach for modelling RIAs is that it becomes possible to use many visual metaphors at once. For example, the overview of a web application may use a deployment diagram metaphor; individual parts of the application may use a navigation metaphor; and the contents of methods may use an activity diagram metaphor[4]. This technique is used by UML; for example, an overview of a system can be visualised using a *deployment diagram*; this diagram can be decomposed into *class diagrams*; and individual methods of a given class can be decomposed into *activity diagrams*.

5.2.3 Guidelines for Metamodel Refactoring

During the development of a modelling language, a number of architectural decisions on the metamodel itself will arise. For example, a set of elements could be refactored into a single element with an enumerated property; conversely, a single element with an enumerated property could be split into a set of related elements.

During the development process of the metamodel for IAML, there were no known guidelines for when to apply such metamodel refactorings. This may be due to the fact that modelling language development is a relatively recent area, and such guidelines often come from experience in the development of modelling languages. However, two guidelines will be proposed that were obtained by adapting existing refactoring guidelines to the metamodelling domain.

Extracting Supertypes

If a set of related elements all have common functionality, then the common functionality can be extracted into a supertype for these elements. For example, Buttons and Input Text Fields both represent user interface elements that need to be rendered in some fashion; it may be preferable to let these metamodel elements inherit this "renderable" functionality, to improve component reuse

[4]**TODO:** How does UWE use different visual metaphors?

and simplify the metamodel development. However, it is important to note that such a metamodel refactoring will increase the number of classes in the metamodel, impacting the *simplicity* language design principle discussed earlier.

In the design of the IAML metamodel, this refactoring was encouraged according to the "three strikes and you refactor" guideline proposed by Fowler [87, pg. 57]; that is, once three metamodel elements share the same common functionality, this functionality can be extracted into a supertype. In terms of the previous example, this involved the creation of a Visible Thing supertype as discussed later in Section 5.12.2.

Merging Elements

This metamodel refactoring process is almost the converse of the *extracting supertypes* process. In this case, if a set of related elements all have the same supertype and there is very little different functionality between each type, then these subtypes can be merged together into a single type, and the differences in functionality represented using an attribute or a property. The number of elements in the metamodel can therefore be reduced, satisfying the language design principle of *simplicity*, and reducing the mental load on the developer.

However, a merged element is much more difficult for a third party to extend in commonly-available typed systems, as most object-oriented language supports object inheritance through third parties, yet few languages support direct manipulation of an object property. It is also important that the property-based distinction be easily visible, otherwise design-time errors may occur. Finally, any constraints on the original elements must be merged in a consistent way, impacting the simplicity of constraints.

This implies that related elements may be implemented in two conflicting ways. The following guidelines were consequently used in the refactoring process of the IAML metamodel:

1. Is it likely that the original supertype will be extended by a third party? If so, the supertypes should be kept.

2. Are there significant differences in the individual implementations of each supertype? If so, the supertypes should be kept, as many model transformation frameworks provide better support for exploring inheritance than exploring enumerations of properties, as discussed later in Section **??**.

3. Otherwise, the supertypes should be merged together into a parent type, and a new enumerated property should be provided to select between predefined behaviours, as previously permitted by the subtypes.

5.2.4 Package Overview

One design principle of the UML metamodel was to architect the metamodel using *packages*, in order to encourage modularity [223, pg. 11]; This approach has also been followed in the design of the IAML metamodel. An overview of the IAML metamodel packages and their dependencies is illustrated as a UML package diagram here in Figure 5.3.

In this figure, each package is annotated with the number of classes defined within that package

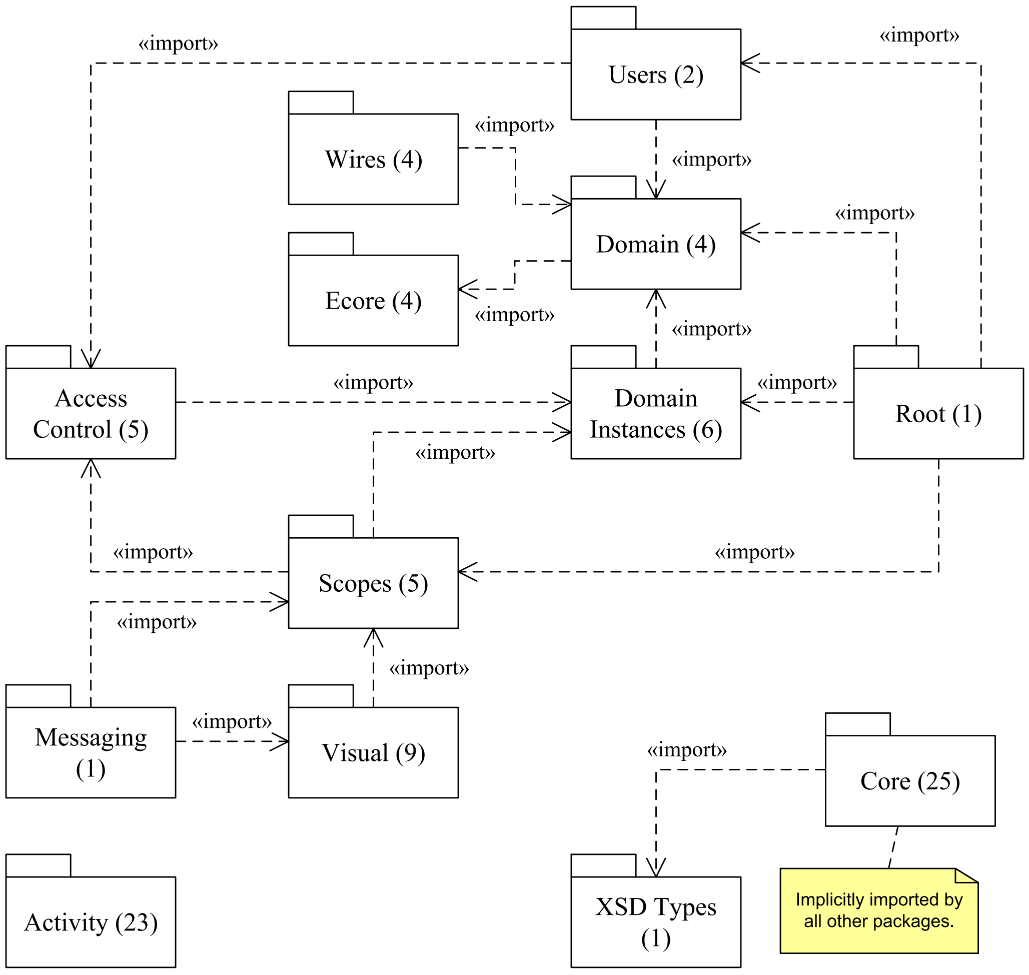


Figure 5.3: Dependencies between packages of related model elements within the IAML metamodel

as described in this chapter[5]. These packages are architectured in a independently layered fashion to improve the design of the language; this package diagram can be reinterpreted as a layered architecture diagram illustrated in Figure 5.4.

In this package diagram, the *Core* package is implicitly referenced by most of the other packages in the IAML metamodel, and forms the architectural *kernel* [223, pg. 12]. The remaining sections of this chapter will discuss the contents of each package in the IAML metamodel.

5.3 Metamodel Core

With the overall design principles of the modelling language discussed, the remainder of this chapter will focus on the metamodel structure of IAML, and on discussing the rationale behind the design decisions involved. In particular, each of the design concepts discussed earlier in Chapter 4 will be explored and implemented as part of the IAML metamodel.

The *Core* package forms the architectural kernel of the IAML metamodel in terms of its basic constructs – such as Predicates, Conditions, Events and Actions – and is therefore the most critical.

[5]While the proof-of-concept implementation of IAML uses EMF packages, these do not yet correspond to the packages described in this chapter. This package refactoring process represents future work, as discussed in Issue 281: *Refactor model elements into documented packages*.

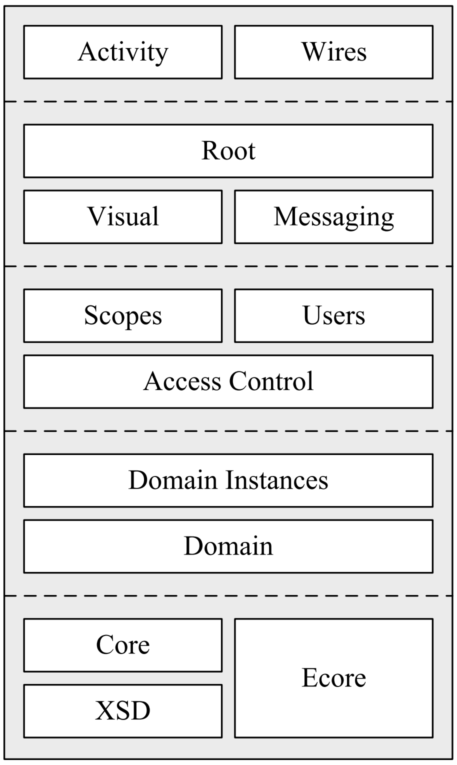


Figure 5.4: Layered architecture diagram illustrating the layered design of packages within the IAML metamodel

The description of this package will be split into five aspects: the underlying logic model; the function model; the event-condition-action model; the wires model; and a constructs model which unifies these separate models.

5.3.1 Logic Model

As discussed by Wirth [315], it is desirable to describe core concepts – such as conditions, operations, functions and values – on a rigorous mathematical definition, to provide a strong foundation for the rest of the language. The core of the IAML metamodel is therefore closely related to *first-order logic* [85], with a partial syntax provided here in Figure 5.5. The elements in this syntax are then mapped to IAML model elements according to the mapping in Table 5.2; the resulting structure of the core logic metamodel is introduced in Figure 5.6[6].

The fundamental building blocks of IAML logic elements are Functions and Parameter Values, which are associated through an instance of a Complex Term. A Function defines a function signature; a sequence of *slots*, each with a name and type, and the Function itself has a name and type. A Complex Term associates a Function with a set of Parameter Values, using the Parameter named association class[7].

When a Complex Term is referenced, each incoming Parameter Value to the Complex Term instance is bound according to the name of the associating Parameter, to the named slots of the Function. This creates variable bindings; that is, the value of a Parameter named x is bound to the

[6]OCL constraints defined throughout this chapter are implemented in the *Model Verification with OCL* component, discussed later in Section 7.7.2.

[7]A named association class is preferable over an ordered association relationship, as discussed later in Section 5.3.1.

$$\begin{array}{rcl}
\textit{Sentence} & \rightarrow & \textit{AtomicSentence} \mid \textit{ConnectiveSentence} \\
\textit{ConnectiveSentence} & \rightarrow & \textit{Sentence Connective Sentence} \\
\textit{AtomicSentence} & \rightarrow & \textit{Predicate}\ (\textit{Term+}) \\
\textit{Term} & \rightarrow & \textit{Constant} \mid \textit{Variable} \mid \textit{ComplexTerm} \\
\textit{ComplexTerm} & \rightarrow & \textit{Function}\ (\textit{Term+}) \\
\textit{Connective} & \rightarrow & \vee \mid \wedge \mid \oplus \\
\textit{Predicate} & \rightarrow & \neg \mid = \mid \ldots \\
\textit{Function} & \rightarrow & + \mid - \mid \textit{concat} \mid \ldots
\end{array}$$

Figure 5.5: A partial syntax of first-order logic, adapted from Fitting [85]

FOL Concept	**IAML Model Element**
Sentence	Condition
AtomicSentence	Simple Condition
ConnectiveSentence	Complex Condition
Connective	`AND` | `OR` | `XOR`
Predicate	Predicate
Term	Parameter Value
Constant	Value
Variable	*(not supported)*
ComplexTerm	Complex Term
Function	Function

Table 5.2: Associations between first-order logic elements and IAML elements

slot x of the Function of the Complex Term. An OCL constraint is provided to ensure that all slots of a Function are provided with exactly one matching named Parameter Value. The *evaluation* of a Function will *return* a new typed value according to the semantics of the given Function and the respective incoming bound variables.

The fundamentals of logic can then be built from these basic elements. A Predicate is a Function that always returns a `boolean`, as discussed earlier in Section 4.3.3. Similarly, a Condition is the instantiation of a Predicate with a set of Parameter Values, mirroring the Complex Term concept; consequently, a Condition can be defined as a Complex Term using a Predicate.

To support the definition of *ConnectiveSentences* – that is, a *Sentence* connected using *Connectives*, such as $\vee$, $\wedge$ and $\oplus$ – the concept of a Condition is split into Simple Conditions and Complex Conditions. Boolean functions such as negation and equality are implemented as Functions; for example, negation is provided by the XQuery function `fn:not` which accepts a single `boolean`-typed argument [303][8].

Variables are not currently supported in IAML – that is, these logic constructs only provide terms. Support for variables would require some method to define how to select elements within a model, and also some method to define how the selection process may access the model instance. This could be implemented using the XPath language [301] to select elements from the Document Object Model (DOM) of the model instance [292]. Because *Variables* are not supported in the logic metamodel of

[8]An example model illustrating the use of negation is provided later in Figure 5.23.

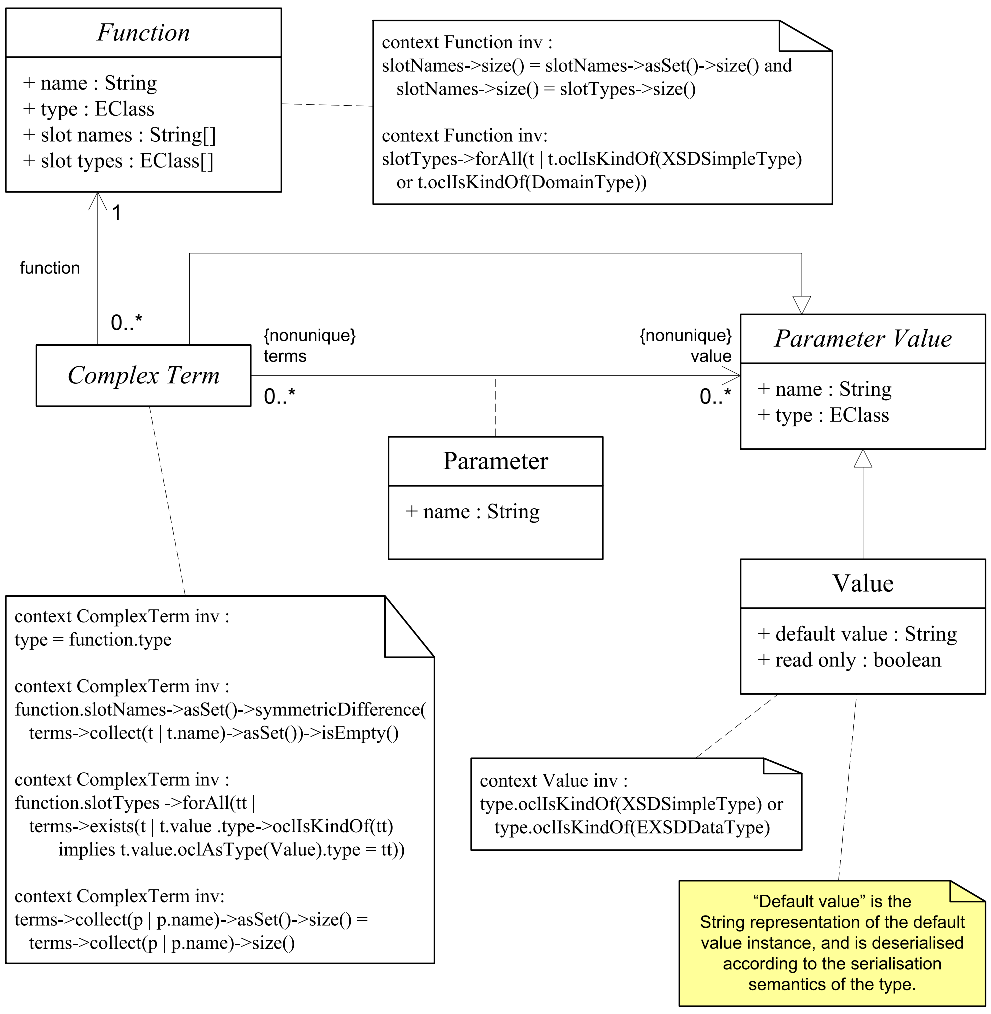

Function
+ name : String
+ type : EClass
+ slot names : String[]
+ slot types : EClass[]
context Function inv :
slotNames->size() = slotNames->asSet()->size() and
slotNames->size() = slotTypes->size()
context Function inv:
slotTypes->forAll(t | t.oclIsKindOf(XSDSimpleType)
or t.oclIsKindOf(DomainType))
1
function
0..*
Complex Term
{nonunique}
terms
0..*
{nonunique}
value
0..*
Parameter Value
+ name : String
+ type : EClass
Parameter
+ name : String
Value
+ default value : String
+ read only : boolean
context ComplexTerm inv :
type = function.type
context ComplexTerm inv :
function.slotNames->asSet()->symmetricDifference(
terms->collect(t | t.name)->asSet())->isEmpty()
context ComplexTerm inv :
function.slotTypes ->forAll(tt |
terms->exists(t | t.value .type->oclIsKindOf(tt)
implies t.value.oclAsType(Value).type = tt))
context ComplexTerm inv:
terms->collect(p | p.name)->asSet()->size() =
terms->collect(p | p.name)->size()
context Value inv :
type.oclIsKindOf(XSDSimpleType) or
type.oclIsKindOf(EXSDDataType)
"Default value" is the String representation of the default value instance, and is deserialised according to the serialisation semantics of the type.

Figure 5.6: UML class diagram for the *Core* Package: Logic Model

IAML, quantifiers such as ∀ and ∃ are also not supported[9] [10].

Named Parameters

One important question is on the design of parameters used in Complex Terms within the metamodel. One option is to create a simple association between the Complex Term and Parameter Value elements and label it as an "{ordered}" association [224, pg. 42]. That is, the relations are defined using *positional arguments* rather than *slotted arguments* [**?**]. However, such an approach directly impacts on the usability of the resulting model instance development environment.

In particular, this means that developers must always be mindful of the order of the arguments, and arranging the order of elements by accident may introduce significant problems that cannot be checked at design time. *Named parameters* rather ensure that serialisation order does not matter, and missing parameters can easily be detected.

Parameters are therefore designed as an explicit association class [224, pg. 46–48], with each parameter named; conversely, each Function defines a set of slot names. The association ends for the

[9] **TODO:** Jens feedback: This might also restrict event modelling. Events don't support properties; may be a consequence of not having variables. Restrictions in event handling.

[10] **TODO:** Giovanni feedback: Are variables needed? If not, a brief explanation.

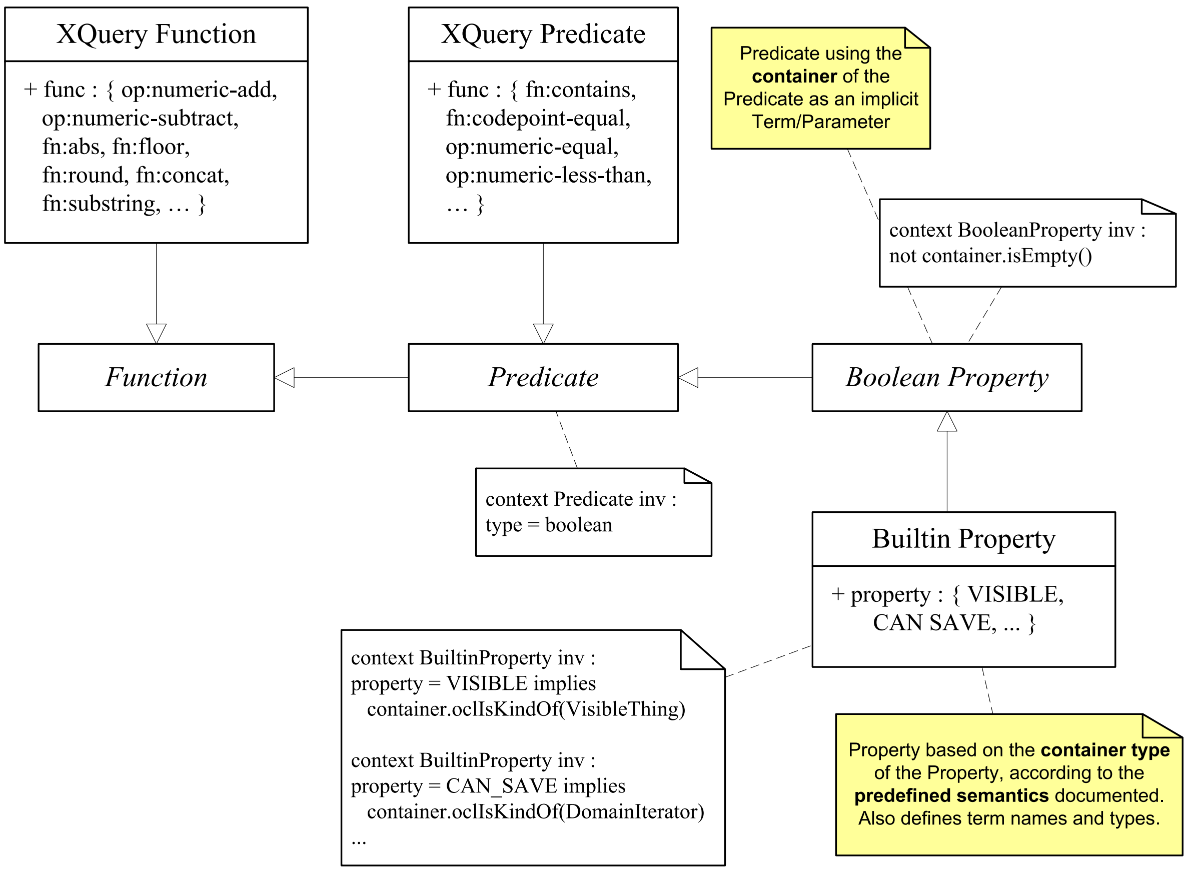

XQuery Function
+ func : { op:numeric-add, op:numeric-subtract, fn:abs, fn:floor, fn:round, fn:concat, fn:substring, … }
XQuery Predicate
+ func : { fn:contains, fn:codepoint-equal, op:numeric-equal, op:numeric-less-than, … }
Predicate using the container of the Predicate as an implicit Term/Parameter
context BooleanProperty inv : not container.isEmpty()
Function
Predicate
Boolean Property
context Predicate inv : type = boolean
Builtin Property
+ property : { VISIBLE, CAN SAVE, ... }
context BuiltinProperty inv : property = VISIBLE implies container.oclIsKindOf(VisibleThing)
context BuiltinProperty inv : property = CAN_SAVE implies container.oclIsKindOf(DomainIterator)
...
Property based on the container type of the Property, according to the predefined semantics documented. Also defines term names and types.

Figure 5.7: UML class diagram for the *Core* Package: Function Model

named parameters must also explicitly be non-unique, to allow for situations where a Complex Term uses the same Parameter Value for more than one slot; for example, the expression $5+5$ uses the same Value instance twice within a single Complex Term instance.

5.3.2 Function Model

The Function metamodel element is extended to introduce two new model elements – Boolean Property and Builtin Property – to simplify the development of the remainder of the IAML metamodel, as illustrated in Figure 5.7. The intent of these extensions is to permit model objects to contain Predicates specific to the instance of the model element. For example, a Domain Iterator contains the Predicate "empty", and the value of this predicate is unique to each iterator instance.

This may be achieved by providing the instance as an positional argument to the Predicate, such as $f(C, a_0, \ldots, a_n)$ where C represents the container. Within object-oriented languages, this syntax is deprecated in favour of $C.f(a_0, \ldots, a_n)$ to highlight that the container *owns* the particular function or method through its type[11]. This latter approach is used for IAML to define a Boolean Property, which is a Predicate using its *container* as an implicit *slot* with a predefined value.

Model element instances may be contained by other model element instances, and this relationship is defined as a *containment reference*. This reference may be defined explicitly, however to simplify the design of the metamodel, it is instead desirable to specify that *all* model elements have an implicit

[11] As discussed by [?, pg. 447], this syntax represents a *qualified procedure call* and rules that the function f must be available to the class C.

container relationship as an opposite to every containment reference. This is identical to the semantics of the eContainer method specified by the Eclipse Modeling Framework [275, pg. 31–32]. This *container* relationship is not explicitly modelled in the class diagrams throughout this chapter.

On the slot types of Functions

In the version of first-order logic used within this thesis, a *Function* has no restriction on the acceptable types of terms. A *Function* can therefore operate on anything within the defined universe[12].

However, in an EMF-based implementation of a modelling language, the definition of an EClass must explicitly state a type for each of its references, even if that type is to the generic EObject type. That is, the slot types must each be typed in the metamodel design. The question thus becomes one on the selection of the most appropriate type for Function terms.

With respect to the concepts of first-order logic, a Function can only be applied to two types of source data within the model – primitive types in the form of XSD Simple Types, and complex types in the form of Domain Types – as discussed later in Section **??**. A number of options for the type of slot types were considered:

1. The type of slot types could be defined as the union of XSD Simple Types and Domain Types. However, Ecore does not support the concept of *type unions* in references.

2. A reimplementation of XSD within EMF would allow the definition of a new common ancestor between XSD Simple Types and Domain Types i.e. the type of slot types would be a new interface IAML Type, with both XSD Simple Type and Domain Type implementing this interface. However, this represents additional implementation complexity and effort, and any updates to XSD would need to be propagated to the IAML implementation.

3. Remove the concept of Domain Types, and only allow XSD Types, which includes both XSD Simple Types and XSD Complex Types; i.e. the type of slot types would be the abstract XSD Type. The drawbacks of this approach are discussed later in Section 5.5.2; in particular, XML Schemas do not support the definition of operations, conditions or events on complex types.

4. Retain the concept of Domain Types, but specify that Domain Types are an extension of XSD Complex Types; i.e. the type of slot types would be the abstract XSD Type. This approach is also discussed later in Section 5.5.2; in particular, the complexity of XSD Complex Types would not easily map to relational databases – a fundamental target platform for this research[13].

While EMF does not support the concept of type unions, this design can be emulated using OCL constraints on the metamodel. The reference type of slot types is therefore defined to be EClass, which is the only common ancestor of IAML model elements and XSD types, as discussed later in Section 7.2. This specification is subsequently constrained to only be an instance of XSD Simple Type or Domain Type using OCL constraints, as illustrated in Figure 5.6. It is not desirable to leave the type open to only EClass, as this would support higher-order logic[14].

[12] *Functions* can also accept other *Functions* as terms, however this represents a higher level of logic [195] than first-order logic, which is outside the scope of this thesis.

[13] Use Case 1: *View Data*.

[14] For example, a Function with an instance of the slot types reference typed to Function.

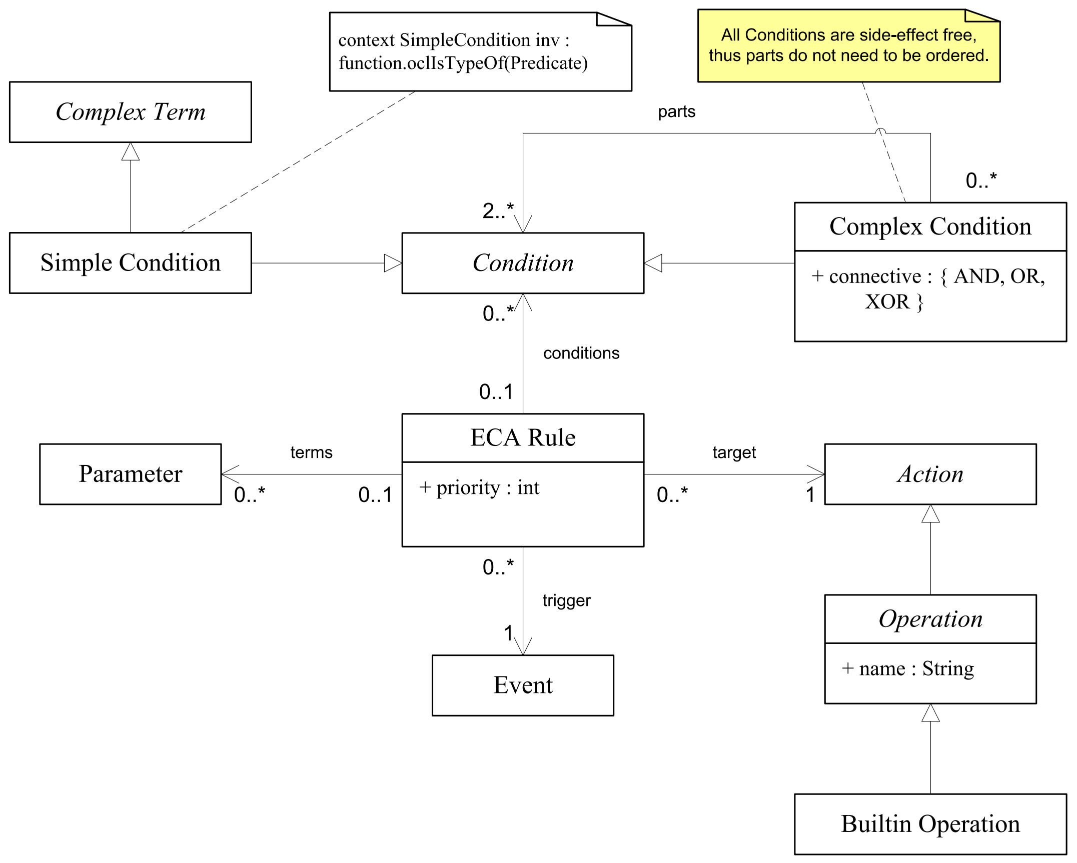


Figure 5.8: UML class diagram for the *Core* Package: Event-Condition-Action Model

5.3.3 Event-Condition-Action Model

Events are placed within the *Core* package of the metamodel as they are considered a fundamental aspect of RIAs within this thesis. While considering the evaluation of existing approaches to the visual modelling of events earlier in Section 4.2, the corresponding metamodel for ECA rules is illustrated in Figure 5.8. In particular, an ECA Rule states that an Event will execute a particular Action when the event is *triggered*, optionally constrained by the Conditions on the rule itself.

Corresponding instances of ECA rules may be represented visually as in Figure 5.9, illustrating an instance of an ECA Rule within the Ticketiaml application; here the visual notation for a UML *SendSignalAction* [224] is used to represent the Event. This rule is triggered by the *onAccess* Event of the containing Label (as defined later in Section 5.6.3), and a Parameter named "message" is provided to the target Operation.

It is not possible for a model developer to directly define their own domain-specific events in IAML, because the definition of an event also requires the definition of the semantics of when the event would be triggered. This would therefore require the definition of an *event modelling language*, which is outside the scope of this research.

Actions

In IAML, the UML concept of an *Activity* is reused as the Action abstract metamodel element; that is, a "specification of parameterized behaviour" which may contain actions of various kinds [224]. An ECA Rule may be provided a set of named Parameters, which will be passed along to the Action; for

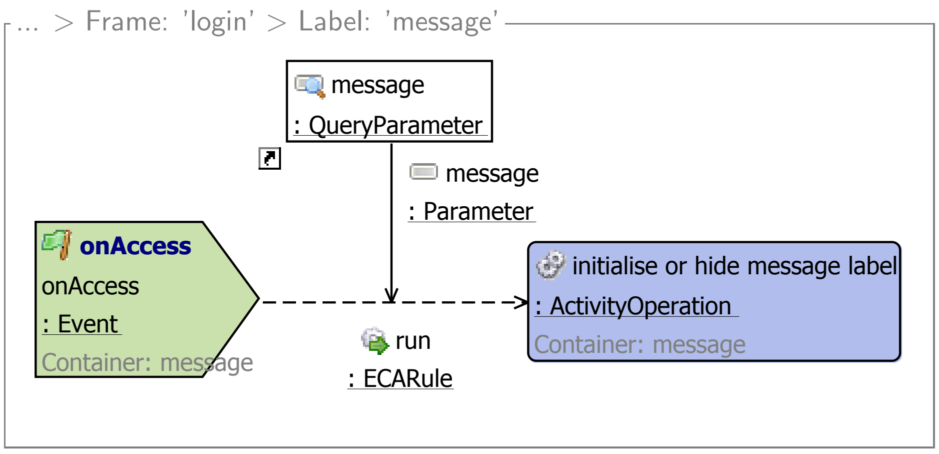


Figure 5.9: Ticketiaml: An ECA Rule connecting an Event to an Operation

example, as parameters to an Operation, as discussed later in Section **??**.

IAML does not use UML's *Action* element, as an *Action* can only be used and executed within a single *Activity*. It would be more desirable to be able to execute an action from many different contexts, which an *Activity* permits. However, IAML retains the name "Action", to match the *Event-Condition-Action* paradigm. All actions are contained directly by a target element, which can be referenced by the action itself through the implicit *container* relationship. Events are contained by a source element, and this is the element that triggers the event itself. More information on this technique is discussed in Section 7.4.4.

5.3.4 An example decomposition of a Simple Condition

To illustrate the mechanism by which a Simple Condition – a Complex Term – evaluates the instances of its provided Parameters, the following simple conditional expression will be decomposed as an example:

gig.minAge.value > textfield.value

This expression is given in a pseudo-textual format, and '>' represents the XQuery comparison function `op:numeric-greater-than` [303]. The subsequent decomposition of this expression is illustrated here in Figures 5.10 and 5.11. (The design rationale behind domain modelling types such as Domain Instances is discussed later in Section 5.5.)

The syntax used in Figure 5.10[15] is based on the UML syntax for defining instances of classes, as illustrated in the UML infrastructure specification [223, pg. 201]. Since the IAML metamodel is defined in terms of standard classes and not stereotypes [224, pg. 670], the notation for UML stereotypes should not be used. Generally, the metaclass of elements in UML model instances are distinguished based on their visual notation, and not with the metaclass name; however in this situation, explicitly displaying the metaclass will improve the understandability of the following decompositions.

[15]**TODO:** Should containment references be explicitly annotated as such?

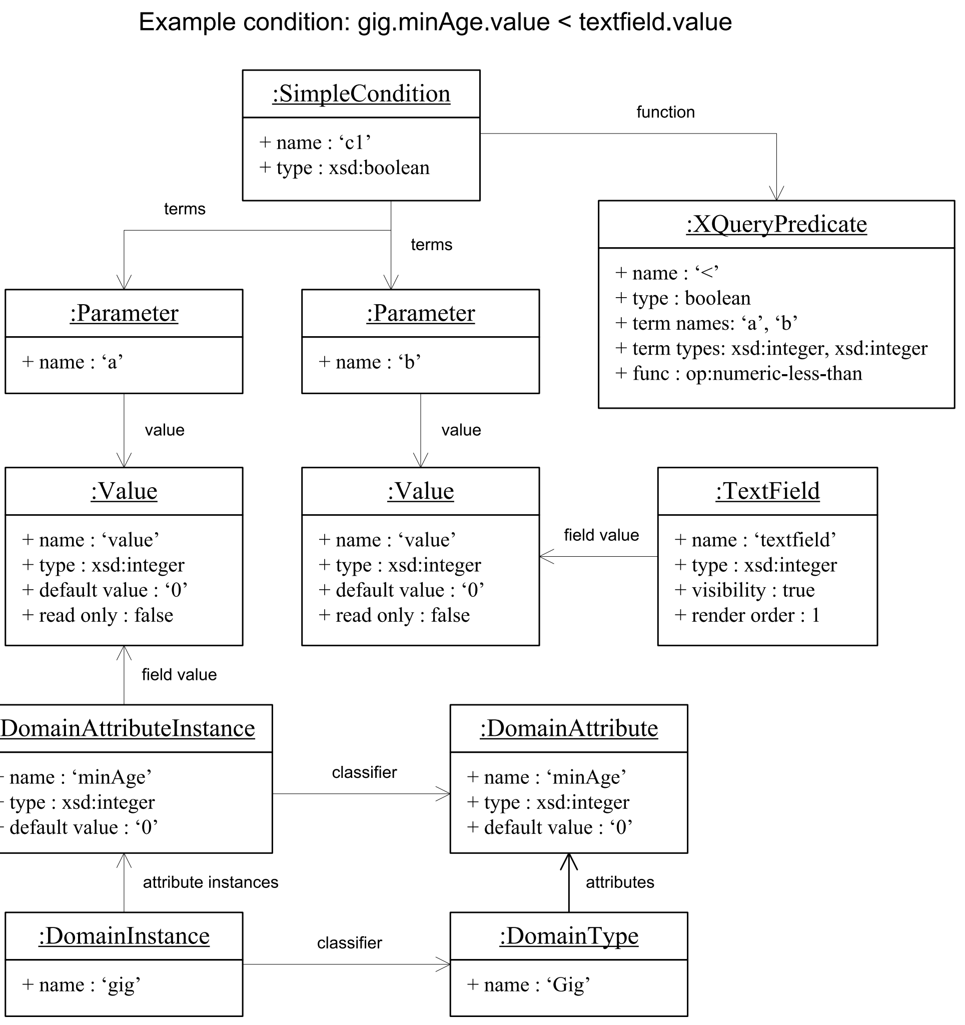


Figure 5.10: The Simple Condition decomposition of a single predicate using variables and domain types as terms, represented in terms of UML syntax

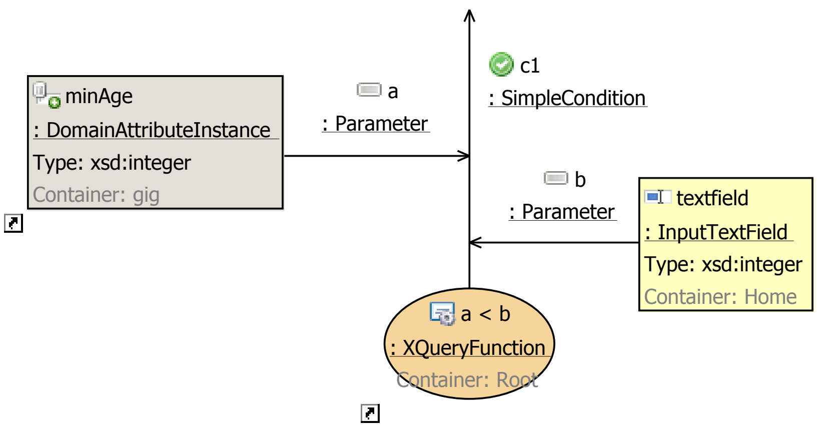


Figure 5.11: The model developer view of the Simple Condition decomposition from Figure 5.10, represented using IAML visual notation

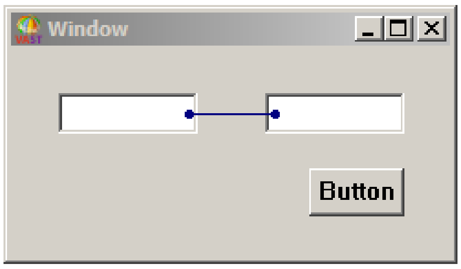


Figure 5.12: VisualAge for Smalltalk: Using Connections

The full decomposition as in Figure 5.10 is not normally provided to the developer, as this syntax is very complex. The complexity of this representation is expected, and would parallel the alternative decomposition of the textual expression into an abstract syntax tree. The representation provided to the model developer is illustrated instead in Figure 5.11.

In this model developer view, some model elements (such as Simple Conditions and Parameters) are reduced to edges; most of the model element attributes are moved into a separate *properties view*; and some referenced elements are removed entirely (such as the Domain Type and Domain Attribute of the first Parameter instance). A full discussion of the visual representation of IAML model instances is provided later in Section 7.4.

5.3.5 Wires Model

In order to support the inclusion of common reusable patterns in a modelling language for RIAs, IAML proposes the concept of a Wire as an abstract connection that represents additional functionality. Wires are inspired by the *connection* concept in VisualAge for Smalltalk [183]. VisualAge for Smalltalk supports a visual rapid application development approach, as in Figure 5.12; here, a simple application consisting of two text fields and a button are defined. The two text fields are associated with a *connection*, connecting the value property of each text field. When executed, the running application automatically keeps the two text fields synchronised.

Wires are implemented using *model completion* as discussed earlier in Section 4.9.1, which allows for the reusable pattern logic to be executed independent of the underlying metamodel implementation. Each reusable pattern instance is represented as a subtype of Wire, and the range of Wires subtypes within IAML are discussed later in Section 5.8. This infrastructure forms part of the *Core* package of the IAML metamodel as illustrated in Figure 5.13.

5.3.6 Constructs Model

The final aspect of the *Core* package of the IAML metamodel is the definition of the Changeable and Accessible interfaces, as illustrated in Figure 5.14. These interfaces are used to simplify the design of other IAML metamodel elements by providing a common infrastructure.

The Changeable interface is used to define a model element which has a *single* Value named field value. To simplify the development of model instances, Changeable is specified as a subtype of Value as a form of syntactic sugar. This means that Changeable elements may be used directly as Parameters in Complex Terms, and the element itself becomes an alias for the contained field value.

In order to support one of the use cases of model completion – that it should be possible for a

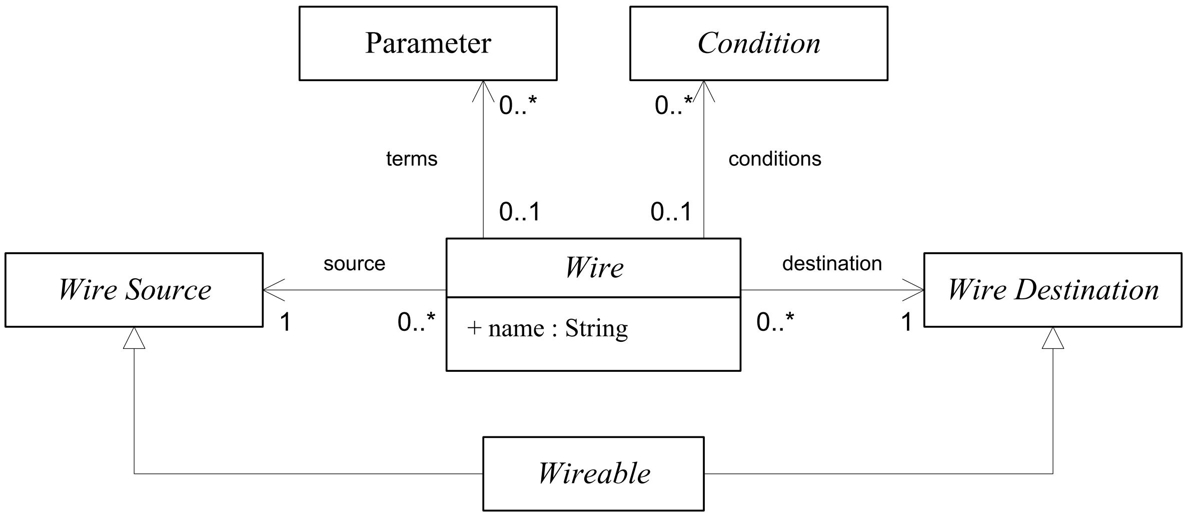


Figure 5.13: UML class diagram for the *Core* Package: Wires Model

model developer to complete an intended model, and then later remove these generated elements[16] – the abstract class Generated Element is also defined in Figure 5.14. All elements within the IAML metamodel should be subtypes of this abstract class to enable these use cases; the meaning of these attributes are discussed later in Section **??**.

Figure 5.14 also introduces the use of the «multiple» stereotype to simplify the UML class diagrams throughout this thesis. A «multiple» stereotype on an association is an alias for multiple identical associations on the two participating model elements, each with the same multiplicities for each association end, but with different association names as listed on the alias association. For model elements with multiple associations between the same element types – for example, the many Boolean Properties defined for a Domain Iterator later in Figure 5.20 – this stereotype significantly simplifies the resulting UML class diagram.

5.4 Type System

With respect to the type systems described earlier in Section 4.5, the IAML type system is designed to be *statically checked* [39]; Values can only ever have a single type[17]. Statically checked languages are often easier to verify using verification technologies, and this is one of the design goals of IAML.

However, the type system is also *weakly checked* [39]; a type instance can be implicitly translated into another type instance if necessary through a *cast*, as discussed later in this section. A weakly-checked language may simplify model instance development, as type instances do not need to be explicitly translated. However weak typing can hinder the verification of system correctness, unless these implicit casts are well-documented.

5.4.1 Primitive Types

As discussed earlier in Section 4.5.2, XML Schema [294] provides a framework of builtin datatypes and a framework for the definition of derived datatypes. Primitive datatypes in IAML are therefore

[16]The implementation of this use case is described in further detail later in Section 7.8.

[17]It is important to note that a statically checked type system does not prevent types from having many supertypes through a *subtype relation* [39].

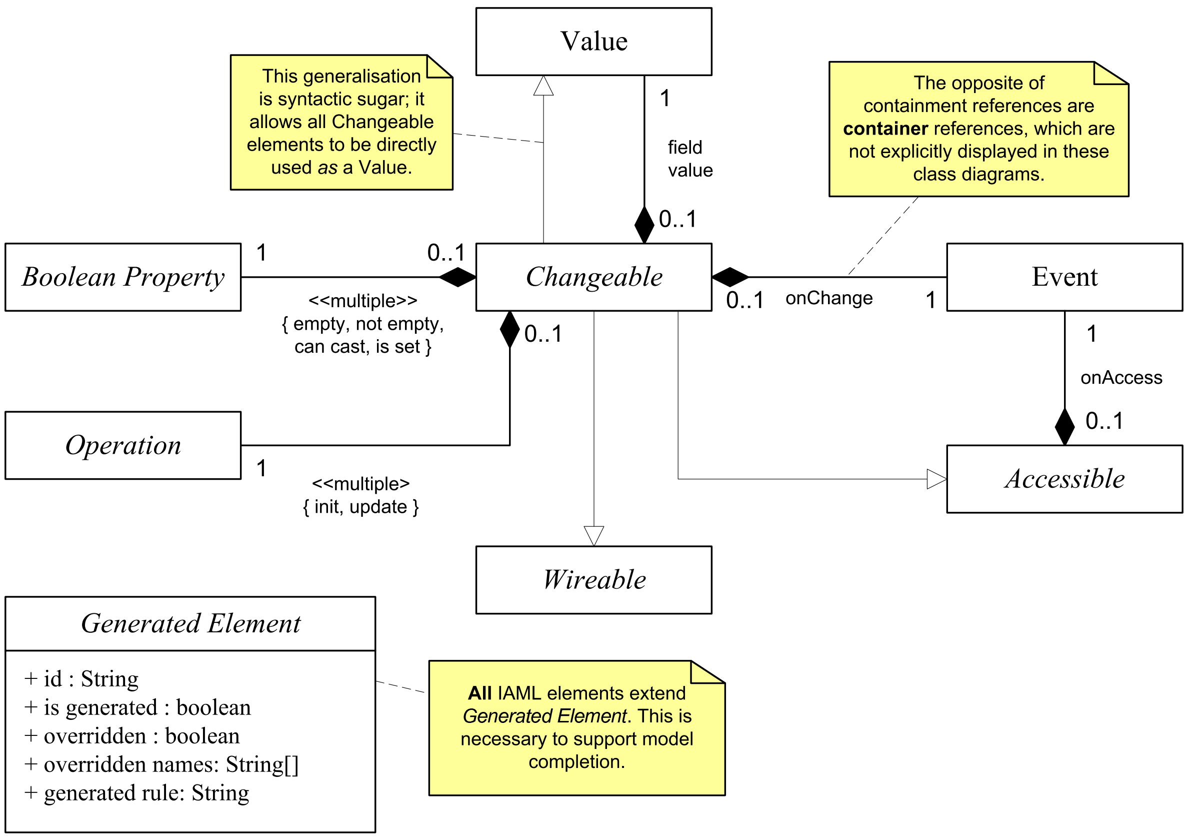

Value
This generalisation is syntactic sugar; it allows all Changeable elements to be directly used *as* a Value.
1
field value
0..1
The opposite of containment references are **container** references, which are not explicitly displayed in these class diagrams.
Boolean Property
1
0..1
Changeable
0..1
onChange
1
Event
<<multiple>> { empty, not empty, can cast, is set }
0..1
1
onAccess
0..1
Operation
1
<<multiple> { init, update }
Accessible
Wireable
Generated Element
+ id : String
+ is generated : boolean
+ overridden : boolean
+ overridden names: String[]
+ generated rule: String
All IAML elements extend *Generated Element*. This is necessary to support model completion.

Figure 5.14: UML class diagram for the *Core* Package: Constructs Model

based on the existing XML Schema datatypes [295], satisfying the design goal of reusing existing standards where possible. IAML can also support the definition of new simple types through the restriction or extension of existing datatypes, according to the XML Schema specifications. IAML does not explicitly support XML Schema complex types, as discussed later in Section 5.5.2.

The Eclipse Modeling Framework also provides its own datatype definitions – such as the datatypes EString and EInt – and reusing these definitions was considered. However, EMF datatypes must be implemented with Java methods such as `convertXXXToString()` and `createXXXFromString()` [275, pg. 390–391], whereas XSD datatypes are defined using a platform-independent formal semantics [295]. This would mean that any EMF-based custom types would need to be implemented in Java, reducing the platform-independence of IAML.

Primitive Type Systems in the Model-Driven Architecture

The primitive type systems architecture discussed earlier in Section 4.5.3 may be used to understand the role of these primitive types within a model-driven architecture. The primitive type system design of IAML is illustrated here in Figure 5.15; this architecture shows how predefined, built-in and custom datatypes (and their instances) interact with type definition references in the same model instances.

Within IAML, the suite of predefined datatypes are obtained directly from the XML Schema Datatypes definition [295], as supplied through the `org.eclipse.xsd` plugin developed by the Eclipse Foundation[18]. These predefined datatypes are extended to supply built-in datatypes such as `iamlE-mail`, as discussed in the next section. Finally, developers may introduce their own custom datatypes

[18]The EXSD Data Type metamodel element is necessary to bridge this gap between XSD and IAML.

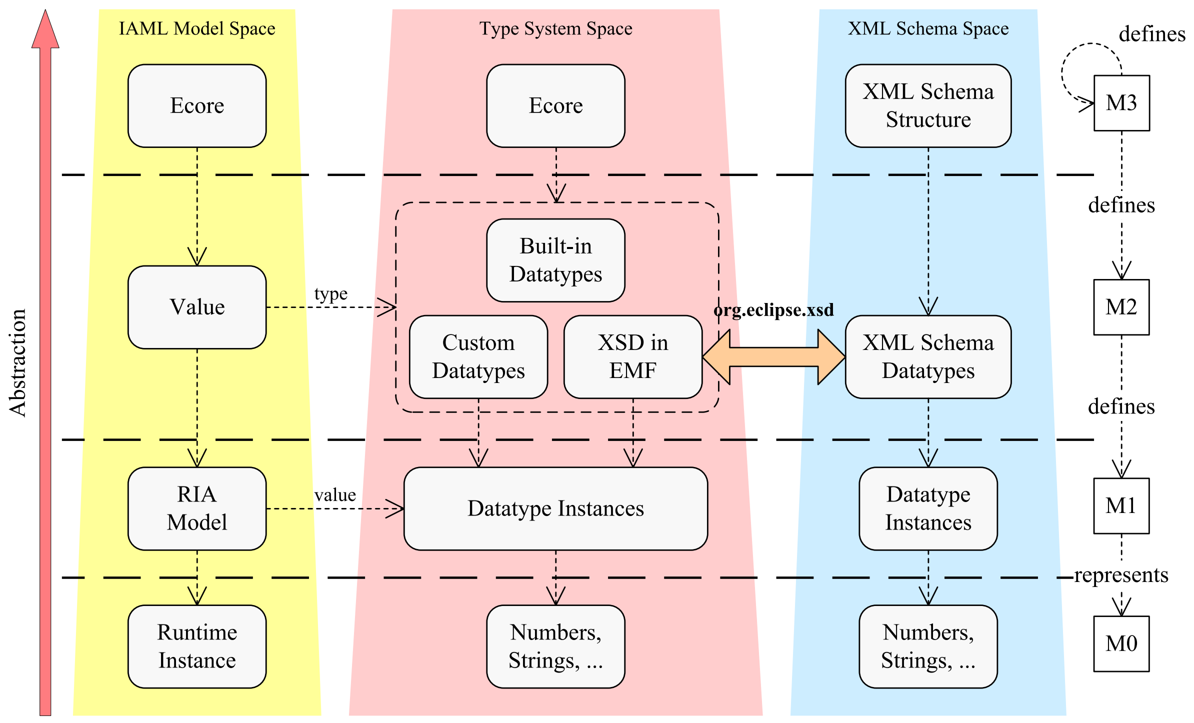


Figure 5.15: Adapting the metamodelling architecture of MDA to the use of primitive type systems

into the environment, which represents the definition of a new metamodel element within the M2 layer.

Comparison to MOF

The architecture of the IAML type system was partially inspired by the *semantic domain model for constructs* defined in the MOF [221, pg. 53–64]. This domain model was not used directly in the IAML metamodel as this domain model was deemed too complex for defining primitive types, with respect to the desire to reuse first-order logic within the metamodel core.

In particular, the IAML type system does not define metamodel elements for structural associations, instance slots[19], value specifications or structural features. However, the reuse of the EMOF-compatible Ecore metamodel [?], as discussed later in Section 5.5, does define associations and structural features within the scope of domain modelling.

Casting

Casting refers to the conversion of one instance of a datatype to another value of a different datatype. Ideally, a cast will not result in loss of precision (for example, an *integer* into a *real number*), but there are situations where a loss of precision cannot always be prevented (for example, a *real number* into an *integer*) In the latter case, the magnitude of precision loss is dependent on the original value; a cast without any loss of precision is deemed a *successful cast* within IAML.

[19]The MOF definition of a *Slot* is different from the definition of a *slot* in the IAML metamodel, as highlighted by its metamodel specification [221, pg. 54]. That is, a MOF *Slot* is owned by an *InstanceSpecification*, whereas an IAML *slot* is owned by a Function, as discussed earlier in Section 5.3.1.

	Source Type				
Target Type	`string`	`int`	`dateTime`	`email`	*default type*
`string`	✓	✓	✓	✓	✓
`int`	?	✓	?	✗	?
`dateTime`	?	✓	✓	✗	?
`email`	?	✗	✗	✓	?
default type	✓	✓	✓	✓	✓

Table 5.3: A selection of complete, conditional and unavailable datatype casts in IAML

The implementation of every primitive type in IAML must also implement a method to evaluate whether a particular instance of the type may be successfully cast into an instance of `string`, and whether a particular instance of type `string` may be successfully cast into an instance of that given type. A selection of these conditional casts with respect to certain XSD datatypes are illustrated here in Table 5.3[20]; these casts are separated into the three categories of *successful* (✓), where the cast is guaranteed to be successful; *conditional* (?), where the success of the cast is dependent on the runtime value of the instance; and *unavailable* (✗), where the cast is guaranteed to be unsuccessful for *all* values.

XML Schema defines how one instance of a given datatype may be converted into a different datatype, by defining the *lexical space* of a datatype [295]. Alternatively, when a datatype is being used with an XPath query [301], the conversion is defined as part of the *constructor function* of the particular target type[21] [303]. Within IAML, the casting semantics of the latter process are used to cast XSD datatype instances.

5.4.2 Additional Built-in Primitive Types

While XML Schema provides a wide range of built-in datatypes, these do not include datatypes common to the web application domain. For example, e-mail addresses and URLs are both common elements of data within Rich Internet Applications, but these datatypes are not supported natively in the XML Schema definitions. These common datatypes are therefore included as built-in primitive datatypes in the implementation of IAML, and some of these types will be briefly discussed in this section.

Default Type

IAML supports the concept of a "default type", which represents an untyped value; in some languages, this is also known as an *Any* type. Following the design of the type system as a weakly-checked language, such a value instance must first be cast to a specific value before it can be used in a typed scenario; this is in contrast to languages such as PHP, where the type of a value is determined at runtime depending on the context of its use [**?**].

All datatypes must also support the serialisation of instances into the default type (and this serialisation often follows the same semantics as casting to `string`). In particular, IAML requires that all

[20]In this table, `email` refers to the IAML-defined `iamlEmail` type introduced in the next section, as XML Schema does not define an "email" datatype.

[21]For example, the process of converting the `string` instance "3" into an `integer` instance would be expressed in XPath as `xs:unsignedInt("3")`.

type instances must be serialisable into a string-based format without a loss in precision.

iamlEmail

This datatype represents an e-mail address, such as the value “`jevon@openiaml.org`”. However, the complete definition of an e-mail address in RFC 5322 [248] is very difficult to implement in terms of a regular expression [**?**]. RFC 5322 allows arbitrary labels and whitespace throughout a valid e-mail address, vastly increasing the complexity of any regular expression.

iamlURL

This datatype[22] represents instances of Uniform Resource Locators (URLs) [26], such as the value “`http://openiaml.org/model#InternetApplication`”. XML Schema already defines a datatype for URIs (`anyURI`), but does not define URLs, which are a subset of URIs. A separate URL datatype is necessary in order to represent locations on the Internet; URIs, on the other hand, can represent content such as text strings or ISBN codes.

iamlOpenIDURL

The `iamlOpenIDURL` datatype represents a validated OpenID URL; that is, in order to be a valid instance, it must be both a valid URL, and has been validated using the OpenID protocol [247]. Consequently, the validity of its string representation can change over time; if a value is cast to a `string` instance and then back to an `iamlOpenIDURL` instance, the user may need to re-authenticate with their identity provider. The full rationale behind this datatype and its semantics, such as how a valid instance may lose its validity if cast, is discussed later in Section 5.9.4.

Proposed Datatypes

A number of other builtin datatypes have been considered, but yet have not been defined in IAML. The investigation and implementation of the following datatypes remains future work:

- `iamlPassword`, representing a password. By explicitly typing value instances as passwords, we can execute additional verification checks (e.g. never display a `iamlPassword` in clear text) and improve security (e.g. all `iamlPassword` instances must be stored through a hash function) as default properties of RIAs.

- `iamlFile`, representing a file. By providing a built-in datatype for files a developer would not need to conceptualise files differently from any other representation of data. This type could be further restricted, e.g. to only text files, binary files, and so on. A built-in datatype would also improve the platform-independence of such a RIA model, as files could be stored using a number of technologies depending on the scenario[23].

- `iamlImage`, representing an image, and could be a restriction on `iamlFile`. Such a datatype would therefore introduce similar advantages to those gained in defining the `iamlFile` datatype.

[22]Issue 269: *Implement iamlURL builtin datatype*.

[23]For example, files can be stored directly on the file system, or within relational databases, or distributed globally as part of a content distribution network; each of these approaches have various benefits and disadvantages, as discussed by Eichinger [76].

- `iamlIdentity`, representing a user identity along with the mechanism that may be used to verify a users' identity. For example, this could be an instance of an `iamlOpenIDURL` as verified through OpenID [247], or an e-mail address used by Microsoft Passport. This proposed datatype is discussed in further detail later in Section 5.9.4.

5.4.3 Value Instances

Now that the underlying type system of IAML has been defined, it is necessary to describe how instances of these types can be defined, created, modified and used. A Value represents an instance of a particular type within a particular scope, and the value of this instance may change over time.

If an instance of a Value model element is contained by a Scope (as discussed later in Section 5.10), then at runtime there will be many different instances of this single Value, and each instance will exclusively belong to a single instance of each unique Scope instance at runtime. The accessibility of a particular Value instance is dictated by the *containing scope* of the Value, a property calculated in the following manner:

- If the Value is contained within an Operation, then the containing scope of that value is the containing scope of the Operation's container.
- If the Value is contained within a Scope, then the value instance is unique to that Scope. For example, a Value contained within a Session can only have one instance of that Value per Session instance, and instances of that Value cannot be accessed by other instances of the same Session scope.
- Finally, if the Value is not stored within a Scope, then the value instance is available globally according to the root Internet Application. Only one instance of the Value can ever exist.

Values are used in the composition of complex objects. For example, a Input Text Field also possesses the values field value, default value and current input. These values can be referenced and modified directly, however it is often more desirable to use an associated Operation such as update (as discussed in Section **??**). Model instance developers can also define their own Values within almost any other model element instance.

While the default value of a Value only supports the string representation of the instance value, it may be desirable in the future to support more complex method of defining default values. Expression languages such as XQuery [302] or OCL [222] could be used to define values based on the DOM of the underlying model instance.

On merging model elements representing value instances

The four model elements of Value, Query Parameter (discussed later in Section 5.10.5), Activity Parameter and Temporary Variable (both discussed later in Section 5.7) are all concerned with the storage of value instances. As per the metamodelling design discussion of Section 5.2.3, it may be beneficial to merge these elements together into a single element to reduce the size and complexity of the IAML metamodel.

It would be undesirable to merge the Value and Query Parameter model elements together, as it is unlikely that a model developer would wish to change the storage semantics of a Value element to those of a Query Parameter. Allowing instances of Value to be set externally by the client via the request URL may also introduce a security problem. Similarly, a Parameter can only be set via an ECA Rule, whereas a Value can be set using any mechanism, which may similarly introduce a security problem.

The key difference between Value and Temporary Variable instances are the semantics of where the element may be accessed, and how long data persists within that instance. It may be desirable in the future to merge these two elements together, by introducing a new attribute for Value that defines the persistence of data; however, this attribute could conflict with the semantics of the *containing scope* property.

5.4.4 Client-side Input Validation

Because all client-side elements such as Input Text Fields have an associated type, IAML can automatically implement some client-side input validation. For example, consider an application model that specifies a Frame containing an `iamlEmail`-typed Input Text Field; if at runtime a user enters in an invalid e-mail address into this field, the application can automatically highlight the invalid field, and inform the user that the value is invalid. This free client-side input validation logic is provided through the model completion framework. For certain visual elements, the *onInput* Event may also permit validation to occur simultaneously while the user is entering in data, corresponding to the classical *input validation* use case[24] of RIAs [319].

5.5 Domain Modelling

As the primitive type system has now been defined, these definitions can be used in the composition of complex types to support *domain modelling*, discussed earlier in Section 4.6. In this thesis, domain modelling is split into two related concepts: the *definition* of domain modelling schemas, expressed as Domain Types; and the *access* of instances of these domain schemas, expressed as Domain Iterators.

5.5.1 Domain Types

The IAML domain metamodel, used to define the structure and attributes of these domain-specific complex types, is adapted from the Ecore metamodel [275]. As illustrated here in Figure 5.16, this metamodel replicates the structure of the Ecore metamodel, but this duplication is necessary in order for these metamodel elements to also subtype the Generated Element abstract class, necessary to support model completion.

Domain Type

A Domain Type may compose many primitively-typed instances of data into an overall complex type. Each primitively-typed instance of data is represented as a Domain Attribute contained by a particular Domain Type. Instances of a particular type may also reference other types; this reference is represented as a Domain Reference.

Domain Types may subtype others to support a simple implementation of multiple inheritance; multiple inheritance is necessary to support Roles as discussed later in Section 5.9. The IAML metamodel reuses the supertypes property of EClass in order to define these subtype relations, expressed as instances of Extends Edge.

[24]Use Case 19: *Asynchronous Form Validation*.

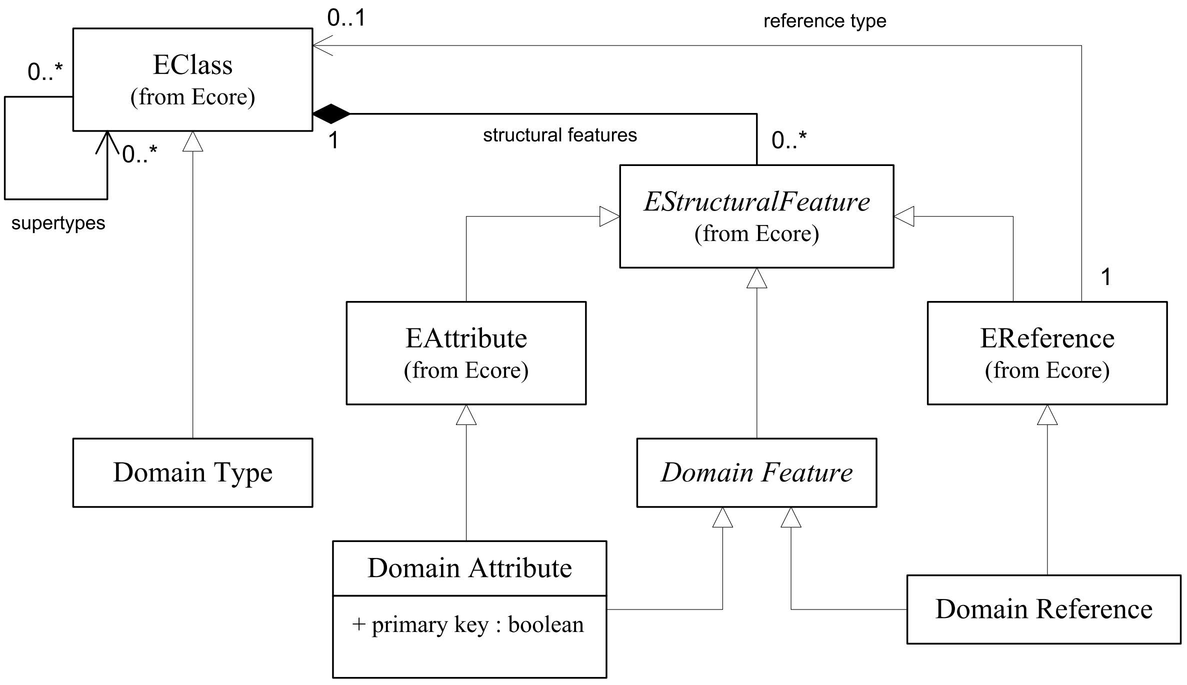


Figure 5.16: UML class diagram for the *Domain* package of IAML

Domain Feature

As discussed earlier, one of the fundamental target platforms[25] of IAML is relational databases, where complex types are represented as tables, and relationships between table instances represented as foreign keys. Relational tables also often have uniquely-identifying attributes collated into a *primary key* [23, pg. 456–457]; these primary keys are essential for association tables.

This is at odds with the Ecore definition of an EClass, where only a single EAttribute may be uniquely identifying, as specified by the ID of the EAttribute [275]. Consequently it is necessary to also define a boolean primary key attribute on Domain Feature, representing that this attribute or reference may be combined with other attributes into a primary key within a relational database. If a Domain Type does not define *any* primary key, then by default model completion will insert in a new `integer`-typed Domain Attribute as a primary key.

Domain Type Example

Since a Domain Type simply consists of a set of Domain Attributes, the corresponding visual representation is fairly simple. In Figure 5.17, the eight Domain Attributes used to define an *Event* Domain Type within Ticketiaml is represented; this includes the `integer`-typed ID attribute, `string`-typed title and description attributes, and `iamlDateType`-typed date attributes.

5.5.2 Design Decisions on Modelling Domain Schemas

Reusing Existing Type Systems

Similarly to the reasoning behind the design of the primitive type section, it is necessary to decide whether a new complex type system should be designed or if an existing complex type system can be

[25] Use Case 1: *View Data*.

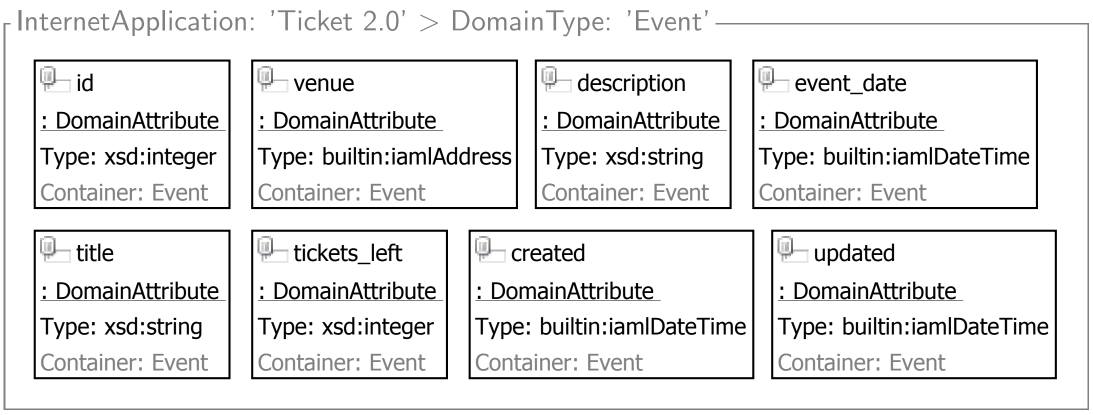


Figure 5.17: Ticketiaml: The defined Domain Attributes of the Domain Type *Event*

reused. As discussed in Section 4.6, four existing metamodels were considered: ER models, UML classes, XML Schemas, or EMF models.

As discussed earlier in Section 4.6.1, ER diagrams are not suitable for modelling complex types in IAML as the language does not support the concept of type inheritance, which is necessary to model users as discussed later in Section 5.9. The initial definition of ER diagrams [46] did not explicitly label relationship cardinality, object composition, or the direction of relationships; later extensions to ER diagrams included these concepts [244]. Finally, ER diagrams also require the use of the Data Object Tables model to define the inner attributes of classes, adding further complexity to a model instance.

Alternatively, a Domain Type could be defined as an extension of a XSD Complex Type, providing a complete implementation of XML Schemas within IAML. However, instances of XSD Complex Type can represent arbitrarily deep trees and sequences of complex data. Without arbitrarily restricting the depth of children permitted in complex types, implementing this functionality using relational database concepts – a fundamental target platform of this research[26] – would be difficult[27].

The UML class diagram supports complex type definitions through UML elements such as *Classifier* and *Property*. However, the infrastructure behind UML classes are very complex; the latest UML specification [224] comprised of 55 separate model element types for class diagrams, including elements such as *Slot*, *OpaqueExpression* and *BehavioredClassifier*, negatively impacting the simplicity language design principle of IAML. Furthermore, there is no standard implementation or defined semantics of UML class diagrams, and a model developer could not rely on any form of interoperability between existing models.

The Ecore metamodel of EMF represents the best choice for modelling complex types in IAML when considering these alternative approaches. At 17 elements, the Ecore metamodel uses much fewer model element types than UML classes; there is a well-defined sample implementation of Ecore metamodel instances [275]; and this implementation has been ported to other platforms such as C++ [111]. Nevertheless, some Ecore element types such as EDataType and EGenericType are not necessary for the IAML metamodel, so these element types will not be supported within IAML; as discussed earlier, the underlying primitive type system of IAML uses XML Schema datatypes [295].

[26]Use Case 1: *View Data*.
[27]For example, consider a single complex type which defines a *Company* to contains many *Persons*, each with their own *Address* complex type. At what point should these inner types be serialised to separate tables?

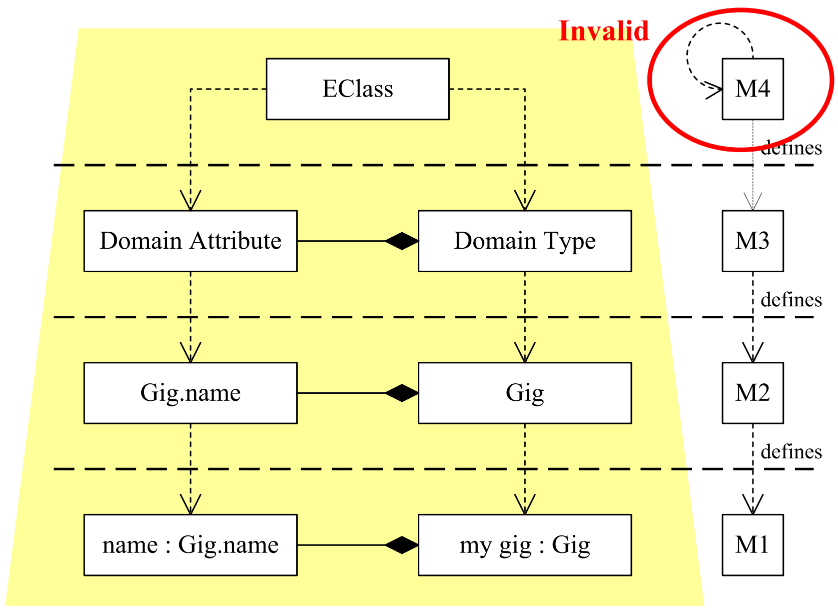


Figure 5.18: If domain types are defined using metamodelling, the resulting models are *incompatible* in terms of the MDA metamodelling architecture

Defining Domain Types using Metamodelling

Since the model of a domain type can be considered the definition of a metamodel for instances of that type, one can reasonably consider that the definition of domain types should reside within a separate metamodel, which is then used as the metamodel of a model instance. This scenario is illustrated in Figure 5.18, where the complex domain type *Gig* and an associated instance *my gig* is defined with a hypothetical domain modelling metamodel.

However, this approach does not satisfy the metamodelling architecture of the MDA discussed earlier in Section 3.1.5. If the metamodel layers are added to Figure 5.18, the IAML metamodel definition must reside on M3. Since IAML is defined in terms of the M3-compatible Ecore metamodel, and not in terms of itself, this situation would violate the conditions of an M3 layer – the metamodelling architecture of MDA does not support any M4 layer.

In UML, the approach taken is to define an *InstanceSpecification* type, which has a *classifier* reference to a *Classifier* instance, as illustrated in the UML infrastructure [223, pg. 54]. Both *InstanceSpecification* and *Classifier* reside within the same metamodel, meaning both domain types and instances of the domain types must reside within the same model instance. The IAML metamodel follows this approach as illustrated in Figure 5.19, and this design is compliant according to the metamodelling architecture of the MDA.

It is still possible to separate the concepts of domain types and domain type instances by separating the models into layers. This approach would be beneficial if the metamodel for domain modelling already exists as an independent metamodel, such as XML Schema or UML class diagrams; or if the domain type instances subsequently defined needed to be reused in model instances outside of RIAs.

Implementing Multiple Inheritance

As discussed by [**?**, pg. 519–568], multiple inheritance is theoretically favourable for object-oriented systems, as real-world objects can simultaneously be many types of common things. The main chal-

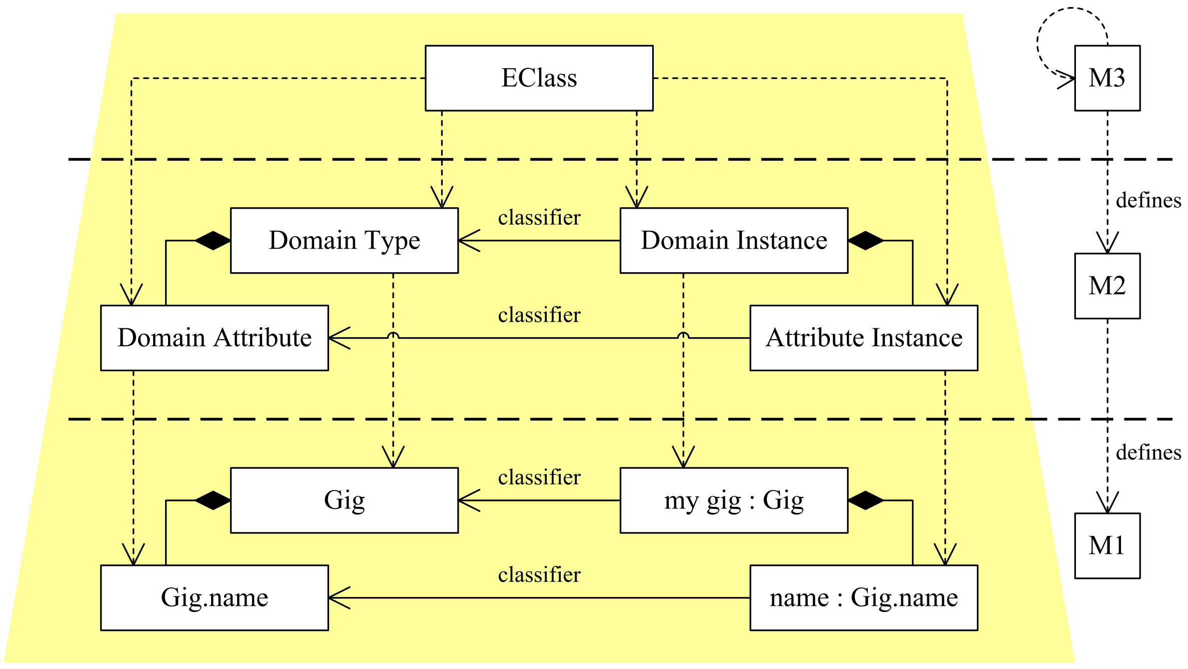


Figure 5.19: Defining instances using references instead of instantiation is MDA-compatible

lenge is in its implementation – for example, reliably permitting developer access to identically-named attributes derived through multiple inheritance. It is therefore important to consider how a multiple inheritance system within IAML may be implemented.

It is possible to decompose a multiple inheritance hierarchy into a system that only supports single inheritance, increasing system complexity. As discussed by Crespo et al. [54], these techniques include *emancipation* (all inheritance is flattened into a single monolithic class); *composition* (inheritance relationships are transformed into composition relationships); *expansion* (multiple inheritance relationships are expanded into many single inheritance relationships); and using a *variant type* (all inherited attributes are collated into a class which uses dispatch based on runtime type).

Both WebML, UML and UWE support single inheritance as part of their domain modelling approach; UML also supports multiple inheritance in UML class diagrams [23]. As UWE reuses much of the UML approach, it seems that UWE can also support multiple inheritance, although multiple inheritance is not considered in any existing work. WebML, on the other hand, only supports single inheritance in defining database models[28].

5.5.3 Domain Iterators

TODO Include a sample decomposition of domain attribute instances, similar to the composition decomposition in Figure 5.10?

In order to support access to instances of Domain Types, IAML uses the Iterator design pattern [103] discussed earlier in Section 4.7.2 as the structural basis for data access in the language. In particular, all data access is provided through a Domain Iterator, which has explicit references to a source of data (Domain Source) and a data schema (Domain Type). This structure is implemented according to the package diagram illustrated here in Figure 5.20[29].

[28]"Each entity is defined as the specialisation of at most one super-entity." [43, pg. 66]

[29]Within the implementation of IAML, the select reference of a Domain Iterator is implemented as the Select Edge model element. Similarly, the schema reference of a Domain Source is implemented as the Schema Edge model element.

In order to access data, the Domain Iterator contains a single Domain Instance, representing the current instance of the Domain Type of that iterator. This instance is subsequently composed of a number of Domain Feature Instances, each representing the current value instance of each Domain Feature within the specified Domain Type. As the iterator is evaluated, accessed and navigated, these instances are populated with the values from the current *instance pointer* of the current result set. The instance data within a particular Domain Iterator can be reloaded from the specified Domain Source by calling the reload operation within that iterator. A full semantic description of the behaviour of instances, iterators, their properties and operations is provided later in Section **??**.

Instances of Domain Iterators are stored and accessible according to the *containing scope* of the iterator itself in an identical fashion to the *containing scope* of Value instances, discussed earlier in Section 5.4.3. For example, multiple Sessions can access the same Domain Source using the same Domain Iterator, but as each session is unique, each session has a unique view over the domain source.

Domain Iterator Query

To access data from a particular data source, a query must be provided to select which data should be retrieved. If no query is provided, then the iterator will simply return all available instances of that type. An Domain Iterator query is provided as a single text string, but this does not prevent language extensions from using alternative query definition methods such as graphical queries or filters. For example, the query "`name = :name`" will select instances of a Domain Schema where the Domain Attribute named *name* matches the value of the incoming Parameter named *name*.

One important aspect of SQL is that it only provides a limited range of SQL functions, and these functions may not be provided by particular relational database implementations [164]. Each database vendor also independently defines their own functions, and many common functions – such as `MATCH`, which may be used to search against text patterns – are not defined in SQL99. Consequently, platform-independent database access must be done through database abstraction layers, such as Propel or PDO for PHP [240, 180].

IAML therefore defines its own query functions which are then translated into the function specific for a particular database. These query functions are discussed later in the documentation of Domain Iterator in Section **??**.

Domain Source

A Domain Source defines the source of a particular Domain Type. In particular, it defines where the data is stored; for example, if it is stored within a database, a file, or stored remotely (for example, an RSS feed). Clients using a Domain Iterator do not need to know where the source data is stored or accessed, or how it is saved.

Figure 5.21 illustrates a partial model used to define the *Edit Event* page of Ticketiaml. This Domain Iterator will select a single instance of an *Event* Domain Type as specified through the Domain Source connected to the iterator. This iterator uses a specific query, "`id = :id`" in order to select a specific *Event*, and this value is provided as a Query to the iterator from an existing Value.

RSS Feeds

By default, a Domain Source is assumed to be a local database on the same server as the web application. IAML also supports the use of an external RSS feed as the source of the Domain Type,

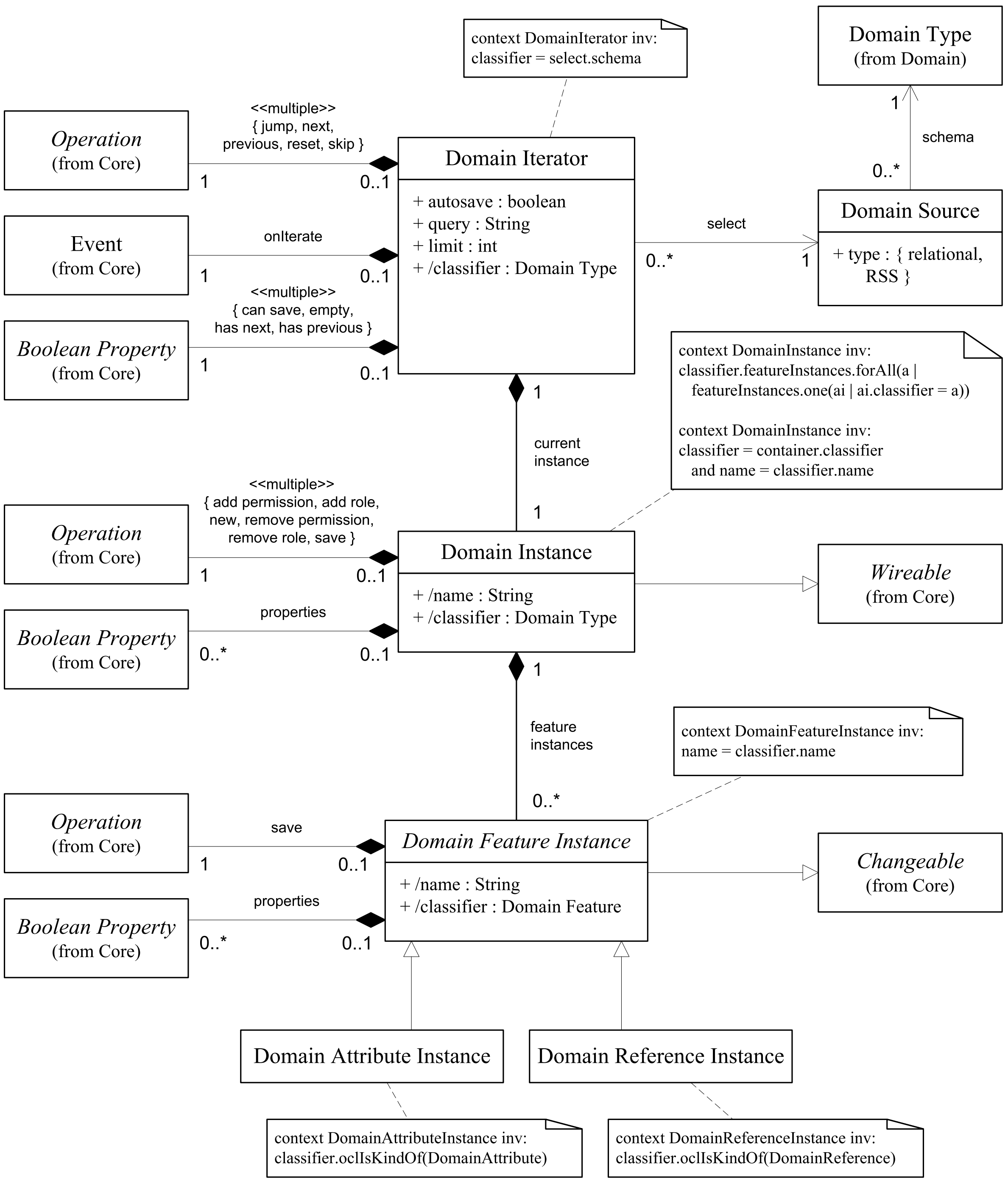

context DomainIterator inv:
classifier = select.schema
Domain Type (from Domain)
1
schema
0..*
<<multiple>> { jump, next, previous, reset, skip }
Operation (from Core)
1
0..1
Domain Iterator
+ autosave : boolean
+ query : String
+ limit : int
+ /classifier : Domain Type
select
0..*
1
Domain Source
+ type : { relational, RSS }
Event (from Core)
onIterate
1
0..1
<<multiple>> { can save, empty, has next, has previous }
Boolean Property (from Core)
1
0..1
1
current instance
1
context DomainInstance inv:
classifier.featureInstances.forAll(a | featureInstances.one(ai | ai.classifier = a))
context DomainInstance inv:
classifier = container.classifier and name = classifier.name
<<multiple>> { add permission, add role, new, remove permission, remove role, save }
Operation (from Core)
1
0..1
Domain Instance
+ /name : String
+ /classifier : Domain Type
Wireable (from Core)
properties
Boolean Property (from Core)
0..*
0..1
1
feature instances
0..*
context DomainFeatureInstance inv:
name = classifier.name
Operation (from Core)
save
1
0..1
Domain Feature Instance
+ /name : String
+ /classifier : Domain Feature
Changeable (from Core)
properties
Boolean Property (from Core)
0..*
0..1
Domain Attribute Instance
Domain Reference Instance
context DomainAttributeInstance inv:
classifier.oclIsKindOf(DomainAttribute)
context DomainReferenceInstance inv:
classifier.oclIsKindOf(DomainReference)

Figure 5.20: UML class diagram for the *Domain Instances* package of IAML

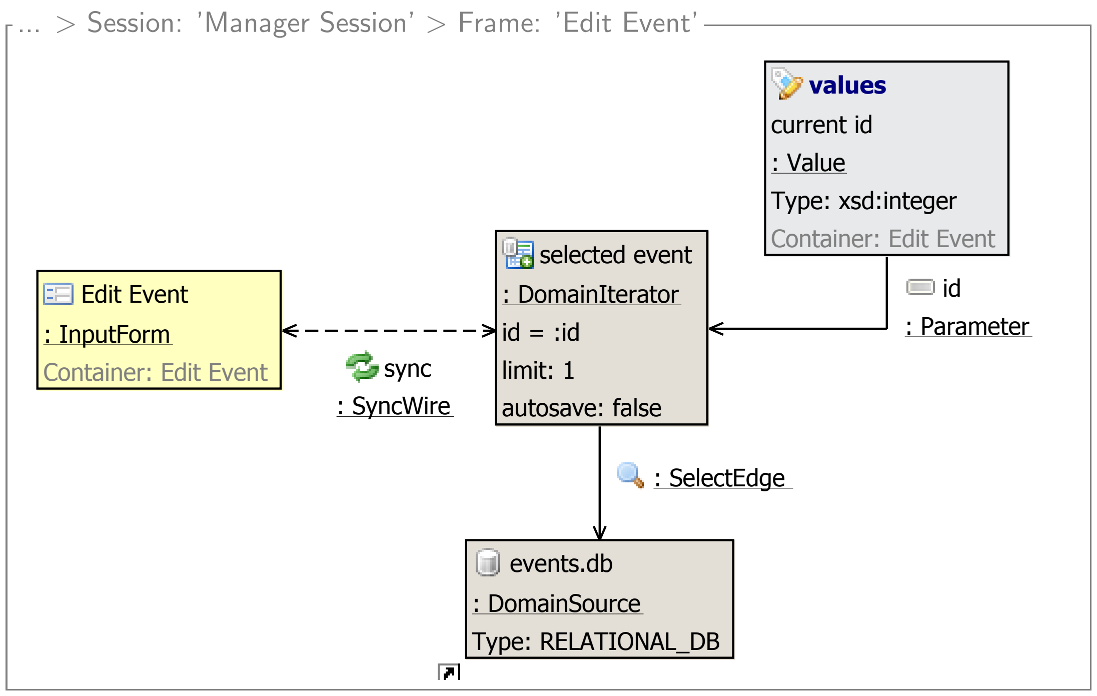


Figure 5.21: Ticketiaml: Selecting an instance of the *Event* Domain Type through a Domain Iterator connected to a Domain Source

as specified through the type attribute of the source. Such a feed will be will be cached locally and periodically updated[30]. The feed can be manually updated using the reload operation, or it will be updated automatically after a specified number of seconds (the cache attribute).

By default, a schema provided by an RSS domain source will automatically include the schema of RSS [255] as its Domain Type. For example, the Domain Type will have the attributes `title`, `link`, `description`, and so on; however, these attributes will only be provided at runtime if the particular RSS feed provides them. It is up to the developer to keep these attributes synchronised with other attributes in the schema, for example by using Sync Wires. This schema is included automatically using the model completion framework[31].

Failure Handler Rules

By associating an ECA Rule as a *failure handler* to a Domain Iterator, the failure handler will be used to capture exceptions or problems during the evaluation of the iterator, preventing the exception from otherwise crashing the application. This rule is specified as a failure handler by defining the name of the rule to "`fail`".

If an error occurs during the operation of a Domain Iterator – for example, the current *instance pointer* is requested to go out of bounds, or the Domain Source is no longer available – then the *failure handler* will be executed, according to the semantics of ECA Rule (Section 5.3.3). If the iterator does not have a failure handler defined, then the failure handler semantics of the containing Scope are used instead as discussed later in Section 5.10.3.

[30]The use of an RSS feed as a Domain Source of data must not be confused with providing an RSS feed through an RSS-typed Frame, as discussed later in Section 5.12.4.
[31]Issue 218: *Include RSS Type elements automatically in RSS Domain Sources*.

5.5.4 Domain Instances

A Domain Iterator contains a single Domain Instance representing the currently selected instance of the Domain Type. Through model completion, this instance is consequently populated with Domain Feature Instances corresponding to each Domain Feature defined within the selected Domain Type. Each Domain Iterator has an associated current *instance pointer*, indicating the position or *cursor* within the evaluated result set. This value can be accessed at runtime through the currentPointer Value of the iterator.

A Domain Iterator also provides a number of operations and predicates to control navigation over the result set, such as next, previous, hasNext, hasPrevious and so on. These methods may modify the *instance pointer*, and are discussed in more detail later in Section **??**.

This Domain Instance may then be connected to other model elements using concepts such as Sync Wires and Set Wires, introduced later in Section 5.8; since a Domain Iterator may only ever contain a single Domain Instance, these wires may also connect directly to the iterator itself, in order to simplify development. This is illustrated earlier in Figure 5.21, where the selected *Event* is kept synchronised with an Input Form, allowing the manager to edit the *Event* instance directly.

By default, a Domain Iterator only selects at most one result from the specified Domain Source. The limit attribute can be used to select many results, rather than just one, making it a iterator over multiple results. This resulting set of results can then be subsequently sorted using the orderBy reference. Modifying the limit of the iterator does not affect the Domain Instance within the iterator, but restricts the maximum number of results that may be iterated over.

Modifying Domain Instances

A Domain Iterator can be used not only for reading from a domain source, but also to modify and subsequently save changes to the domain data. Every Domain Attribute Instance contained within its Domain Instance can be modified, and once the iterator is saved these changes will be written to the original domain source. An iterator may also automatically save all changes by setting the autosave property of the Domain Iterator to `true`.

However, some domain sources such as RSS feeds are read-only, and trying to modify such a Domain Instance will result in an error. An application therefore should first evaluate the canSave predicate of the iterator to check that the iterator can be successfully saved. Alternatively, the failure handler discussed earlier can be used to handle these exceptions.

New Result Instances

A Domain Iterator may also be used to create *new* instances of a Domain Type within a particular Domain Source, by setting the query of the iterator to "`new`". If the autosave attribute of the iterator is `false`, then this new instance will not be committed to the database until the save operation is executed.

This approach is illustrated here in Figure 5.22, which is adapted from the Ticketiaml implementation of the *Signup* page. By connecting this Domain Iterator to a Signup Form via a Sync Wire, a new user can create a new profile directly. As this Domain Iterator has its autosave attribute set to `false`, the save operation must be called executed manually.

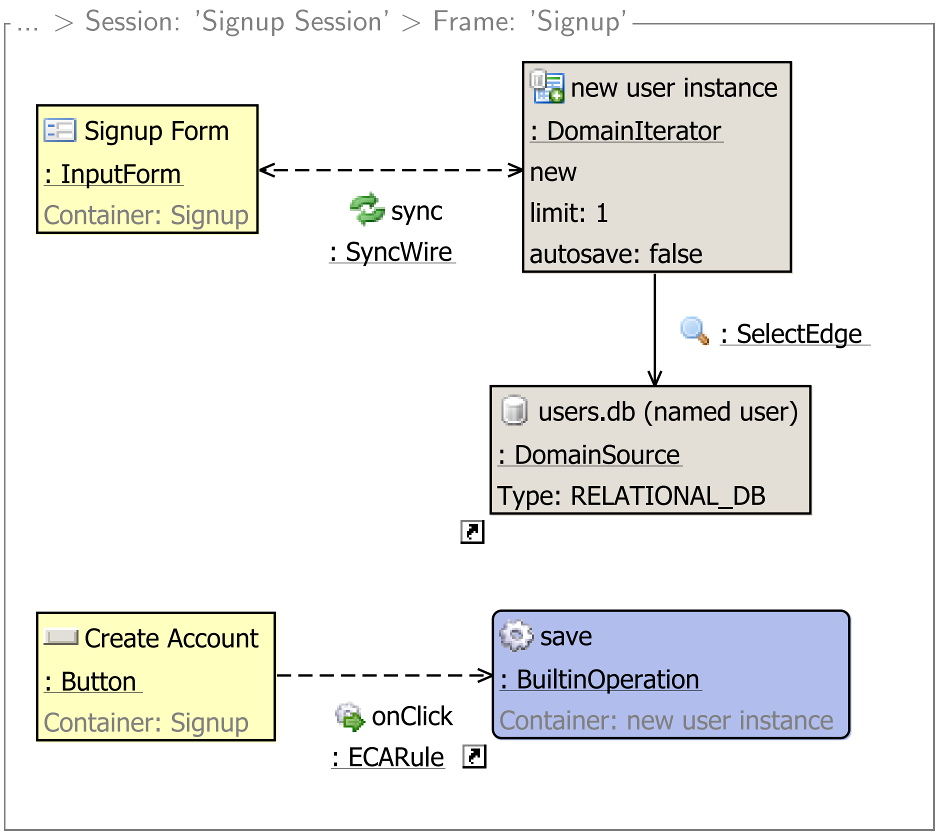


Figure 5.22: Ticketiaml: Creating a new instance of an *Event* using a Domain Iterator

5.5.5 Design Decisions on Modelling Domain Instances

Similarly to the design of Domain Types, a number of design decisions were considered during the development of the Domain Iterator model elements, in particular with respect to the underlying first-order logic model discussed earlier in Section 5.3.1.

Domain Attribute Instances should not be Complex Terms

A Domain Attribute Instance could be considered as a Complex Term; that is, a Function operating on a Domain Instance that returns the current value of the given attribute. However, this is only true if the value is read-only. Domain Attribute Instances can be modified, and can throw events when they are modified. This conflicts with the meaning of a Function, as Functions must not change the state of the system.

A Domain Instance may still provide both Domain Attribute Instances and corresponding Functions for each attribute instance – in essence, providing two methods of accessing the same data in two different contexts. However, this violates the principle of *uniform access* discussed by [?, pg. 57], where there should be no distinction between a module's services and its underlying implementation.

Attribute instances in Domain Instances are not directly Value instances

Domain Attribute Instances could also be considered as Values which is accessible and modifiable. However, it is not desirable for these attribute instances to directly be instances of a Value, according to the design philosophy of Value as described in Section 5.4.3; in particular, a Value cannot contain any operations, predicates or events, as a Value represents a *Variable* in first-order logic concepts. As discussed earlier, each Domain Attribute Instance contains a save operation, a can save? predicate and a *onChange* event, making it incompatible with the design philosophy of Value.

However, the IAML metamodel specifies that an Domain Attribute Instance is also an instance of a Changeable in order to provide the *onChange* event to attribute instances. Due to the syntactic sugar introduced in Section **??** which specifies that all Changeables can also be used as Values, this means that a Domain Attribute Instance eventually inherits the Value abstract type.

Navigation through foreign keys of Domain Type instances

Currently in the IAML metamodel there is no way to navigate between Domain Type instances through foreign keys, such as through instances of Domain References. This remains future work[32], and a number of potential modelling approaches have already been identified. For example, a Domain Attribute Instance may define a Domain Instance for each domain object instance referenced via a foreign key. Alternatively, a Domain Iterator could contain sub-iterators for each foreign key, which could then be navigated normally.

Nevertheless, it is generally not desirable to support extensive foreign key navigation, if domain modelling is considered in an object-oriented software development viewpoint. The *Law of Demeter* [184] provides a guideline or heuristic for developing high-quality object-oriented programs; in particular, Lieberherr and Holland [184] suggest that objects should only have limited knowledge about other objects, and knowledge should be restricted to “friends” of an object. Navigating over foreign keys should therefore be the exception in an IAML model instance.

5.6 Events

The rationale behind the use of events as part of Event-Condition-Action modelling has been discussed earlier in Section 5.3.3, as it forms an important part of the *Core* metamodel package. Events can also be used to implement lifecycle modelling, as discussed later in Section 5.10. In this section, each possible instance of an Event will be briefly introduced; for the full definition of their behaviours, the interested reader is referred to the definition of Event in Section **??**.

A triggered event does not have access to any information about the previous state of the triggering or containing objects; this may be a consequence of not supporting free variables within the core metamodel of IAML. For example, when an *onChange* event is triggered, it is not possible to access the previous value of the changed field value. Similarly, when an *onFailure* event is triggered for an Email, no failure message is available to the developer. This information has not proven necessary within the current proof-of-concept implementation, but has been noted as an avenue of further research[33].

5.6.1 onChange

The *onChange* event is intended to be fired when the field value of a modelled element changes. This is particularly important for visual elements such as Input Text Fields, discussed later in Section 5.12; in this case, the *onChange* event is fired when the control both loses the input focus *and* its contained

[32]Issue 228: *Permit navigation through foreign keys of instances of Domain Types.*
[33]Issue 198: *Populate Events with event information when events are fired.*

field value has been modified since it gained focus. The event is derived from the DOM Level 2 *change* event [291], and is also similar to JavaFX's *onReplace* event [311].

5.6.2 onInput

The *onInput* event is similar to the *onChange* event, but is defined in order to support the richer interfaces provided by RIAs. This event is fired whenever a *user* modifies the displayed value of the displayed element on the client-side, but before it has been committed to the field value of the element.

However, if this event is triggered too frequently, the performance of the web application may suffer. Consequently it is not guaranteed that the event will ever fire, or that it will fire regularly (for example, upon every character keystroke) – but this is to guarantee the performance and usability of the web application. This event may be buffered during periods of intense input or performance reasons and to prevent excessive network requests.

5.6.3 onAccess

The *onAccess* event is defined for all Scopes and Accessible elements, and is triggered when the Scope is accessed or the element is rendered[34], or. For example, a control contained within a Frame will cause *onAccess* to be fired whenever the control is rendered on the frame.

When a Scope is accessed, the *onAccess* Event is triggered for its parent Scope before the event is triggered for the current Scope, unless the current Scope has is the root Internet Application element. This cannot be circumvented; it is therefore not possible to prevent the *onAccess* Event of the root Internet Application from triggering. This allows parent Scopes to selectively prevent access to its contained elements, as discussed later in Section 5.10, and is fundamental to the implementation of Gates.

5.6.4 onInit

The *onInit* event is defined for all Scopes, and allows the scope to be initialised for the first time. It is triggered when an instance of that Scope is initialised, and is guaranteed to trigger *before* the *onAccess* event of the Scope is triggered.

Since onInit is *only* called when the scope is initialised, it is guaranteed that the root Internet Application will only trigger its *onInit* event once throughout the lifecycle of the application. The triggering of *onInit* events also follows a containment hierarchy identical to the triggering semantics of *onAccess* events.

5.6.5 onClick

The *onClick* event is defined for all visual elements (see Section 5.12) to describe scenarios where the user clicks a rendered element. This event is once again derived from the definition of the DOM Level 3 *click* event [292].

For composite visual elements, the *click* event will *bubble up*[35] the containment hierarchy of Visible Things, as per the notion of DOM event bubbling discussed earlier in Section 4.2.6 [291]. For

[34]The *onAccess* event could therefore have been named *onRender*.

[35]Unlike the DOM definition, there is no way to cancel a bubbled event in IAML. Describing this scenario is future work, but a workaround can be provided by using state variables and conditions.

visual elements such as Buttons, *onClick* also covers the scenario where the button is clicked using the `Enter` key, as this simulates a pointer click.

5.6.6 onSent

The *onSent* event is only defined for Email elements (discussed further in Section 5.11.1), and represents the successful *delivery* of a given e-mail. It is important to note that the *onSent* event does not cover a successful *receipt* of an e-mail, but only a successfully delivery.

For example, if the e-mail is mistakenly deleted by an intermediary mail daemon, or later deleted by a spam filter, this event may fire but the e-mail is still not received. If the e-mail is lost during delivery, the *onFailure* event may be triggered in the future to signal a failed delivery; that is, an *onFailure* event may occur after an *onSent* event.

It is not possible to guarantee that any given e-mail message has been successfully delivered *and* received by a particular recipient. While *Delivery Status Notifications* (delivery receipts) [209] and *Message Disposition Notifications* (read receipts) [128] can be used to verify a received email, not all e-mail clients support this extension, and the user may disable these notifications for privacy or performance reasons. Another attempt is through the inclusion of *web beacons* – invisible images that are uniquely identified to track user activity, as discussed by Lawton [177] – which may also be disabled for for privacy or performance reasons.

5.6.7 onFailure

The *onFailure* event is in some ways the opposite of the *onSent* event, as it represents the unsuccessful *receipt* of a given e-mail instance. It is not guaranteed that this event will fire in the case of a delivery failure, unless this failure information is provided to the IAML implementation. For example, if the e-mail is mistakenly deleted by a mail daemon or deleted by a spam filter without any return notification, this event may not fire.

An implementation of the IAML metamodel may allow the definition of an “email timeout”, describing the length of time before a queued e-mail (which has not yet triggered *onSent*) is tagged as a failed delivery and *onFailure* is instead triggered. This would allow the implementation to guarantee that an e-mail will eventually either pass or fail within a specified period of time.

5.6.8 onIterate

The *onIterate* event is defined for all Domain Iterator model elements, as discussed earlier in Section 5.5.3. It allows the application to update parts of the application when the current result changes; for example, by using “previous” and “next” buttons on a results page. As described later in Section **??**, this event is fired when the instance pointer of the iterator changes, or the contained Domain Instance is reloaded.

5.6.9 Client-side Events

In IAML, there is no distinction between client-side and server-side events, and events are intended to execute identically on both platforms according to the triggering definitions of the events. For example, if the field value of a visual element is modified, then the *onChange* event *must* be triggered for this visual element, even if the element is not currently visible or rendered.

This transparency between client-side and server-side events allows a model developer to focus on the functionality of an element, without needing to keep track of the rendering state of the entire application. This reduces the complexity of the language, and is one of the design goals of IAML. However, this approach places a technical burden on the web application implementation, as now this implementation needs to emulate the rest of the application even if there the client-side interface is not currently being displayed.

5.7 Operations

As discussed earlier in Section **??**, operation modelling refers to the modelling of the lower-level operations that make up higher-level behaviours, and this modelling may occur through a number of mechanisms. IAML supports modelling behaviours through two model elements, each a subtype of the abstract class Operation.

IAML defines a range of built-in Builtin Operations which can be used in both composite operations, and directly in the web application itself. For example, the Input Text Field has the primitive conditions `hide` and `show`, which change the visibility of the given visual element. A full description of each of these operations is outside the scope of this chapter; the interested reader is instead referred to Appendix **??**.

However, it is not possible to define a non-trivial web application without having to express some aspects of complex behaviour and logic; in many cases, this represents the domain logic of the web application. This logic may be expressed independently of the IAML model instance – for example, through the manual extension of the generated web application – but it may be preferable to model this logic directly within the same instance, to obtain model-driven benefits such as code generation and model instance verification.

5.7.1 Modelling Complex Behaviour

As an example, consider a web application that needs to hide an Input Text Field if the value of a separate text field is zero, or show the field otherwise. This behaviour may be implemented using only ECA Rules, Parameters and Complex Terms as illustrated in Figure 5.23[36]. Here, an instance of a Complex Term used as a Condition for one ECA Rule is reused as a Parameter.

However, there are a number of negative aspects that may arise when using this simplistic approach to model complex behaviour, that need to be considered.

1. Conceptually, complex behaviour would no longer be represented by an Action, but would instead be represented by the constraints operating on a particular ECA Rule as it applies to a single instance of an Event. This has important ramifications: most importantly, it means that an ECA Rule instance must support the definition of *many* Event triggers, otherwise complex behaviour must be duplicated for each triggering Event.

   This also means that an Action would no longer represent "a specification of parameterized behaviour" as defined earlier in Section 5.3.3, but would instead represent a single form of predefined behaviour that cannot be specified by the model developer. This also means that behaviours could not be externally referenced – if the *onChange* and *onInput* events should perform the same action, for example, the behaviour would need to be duplicated.

[36]This model example also illustrates the use of the XQuery Function `fn:not` in order to implement negation.

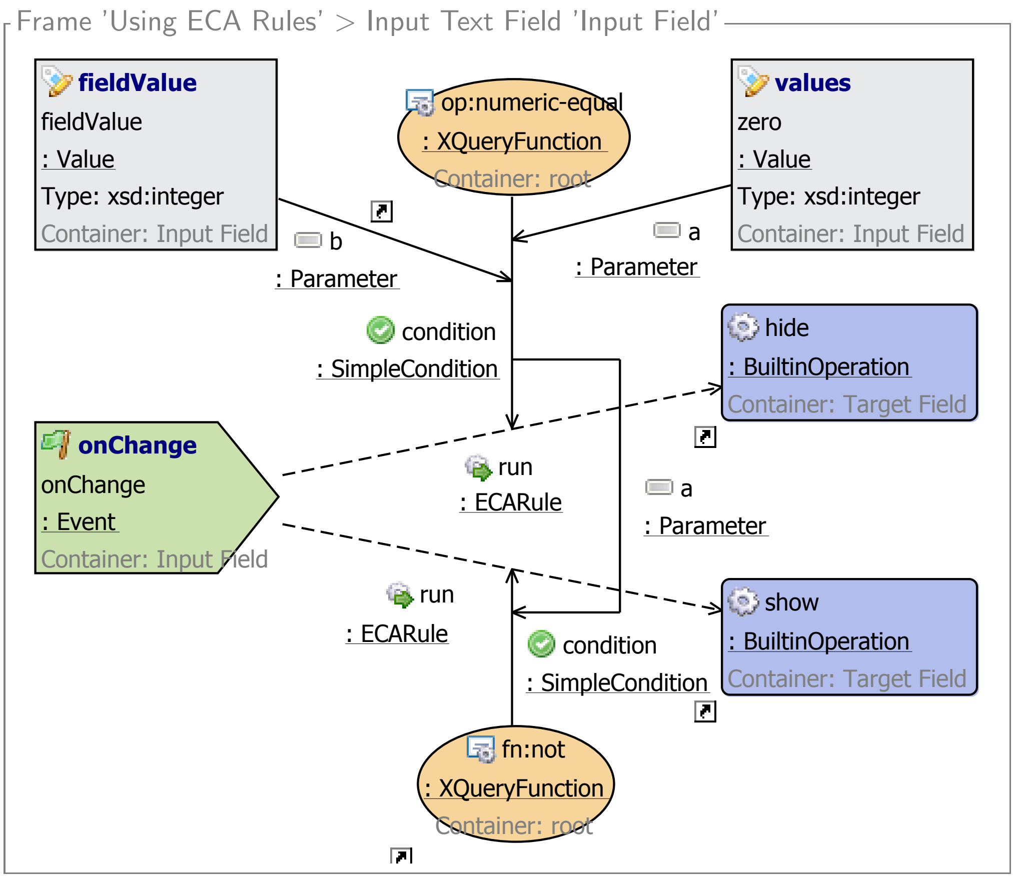


Figure 5.23: Representing complex behaviour within IAML using two ECA Rules

2. As the complexity of a particular behaviour increases linearly, the complexity of each ECA Rule increases exponentially. For example, consider a behaviour which may execute one of eight different operations based on the evaluation of three boolean input variables. In this example, eight ECA Rules would need to be provided, each with a different expression of the same input variables. This represents a significant amount of developer effort, and may result in logic errors.

3. The visual complexity of Figure 5.23 is a consequence of the visual notation used, where every model element has a corresponding node or edge. This issue is discussed in further detail later in Section **??**. A new visual notation would need to be proposed, and this notation may need to incorporate grouping related elements, a hybrid textual/visual syntax, or directly supporting event algebra [325] and behaviour composition operators.

   For example, an ECA Rule could specify a textual condition "*[op:numeric-equal(text.fieldValue, 0)]*" on the edge, which would implement the intended logic. This approach is extremely similar to the use of OCL within UML [224], and would require the definition of a mapping between model instance elements to references within the textual syntax. However, this approach would vastly simplify the definition of complex conditions on a rule.

4. With this approach it is not possible to specify procedural-based behaviour such as loops[37],

[37]As discussed earlier in Section 3.1.2, Fowler [89] argues that domain-specific languages should not exhibit Turing completeness through general-purpose language concepts such as loops. IAML follows this guideline with the exception of

or concurrency-based behaviour such as multithreading. Some procedural behaviour may be emulated using special element wrapper structures – for example, a loop may be implemented by allowing an ECA Rule to re-trigger its triggering Event.

5. Finally, this concept of *event-oriented behaviour* implies that the lowest level of the hierarchical design of the IAML metamodel, as illustrated earlier in Figure 5.2, would be focused on the definition of components. In order to improve the manageability of the complexity of the model instance, it may instead be desirable to support a further hierarchical decomposition into an *operation modelling* layer.

It is therefore more desirable to support modelling complex logic and behaviours within an Action, than on the Conditions of an ECA Rule. Two approaches to modelling complex behaviour will be considered, with respect to the modelling language development principles discussed earlier in Section 2.7: the reuse of an existing textual expression language; or the reuse of a subset of UML.

Reusing an Existing Textual Expression Language

There are many existing languages for defining expressions and behaviour, and these could be used to define Operations within IAML. For example, a new model element OCL Operation could be added to the model, and contain an OCL operation [222]; similarly, a Java Operation could support a Java method.

The main issue with using external languages for defining expressions and operations is of the resulting implementation. For each target platform where the expression must be evaluated, either an instance of that language must be available to evaluate the expression directly, or – if there is no instance of the language in that target platform – the expression must be translated into another form where it *can* be evaluated. For example, an OCL Operation would require an OCL interpreter for both Javascript and PHP, as discussed later in Section 7.6.3.

Reusing UML Activity Diagrams

As discussed earlier in Section 4.3.1, UML activity diagrams may be used to model the lower-level object and control flow of an activity. While the same behaviour may be implemented though UML interaction sequence diagrams, Bennett et al. [23, pg. 106] argue that activity diagrams are more useful to model business behaviours. Some lower-level constructs such as arithmetic are however not supported, and it is not possible to consistently translate these models into code. The metamodel specific to UML activity diagrams is also very large; at the time of writing, this included 52 separate modelling concepts, and 31 common behaviours [224].

As the intent of an IAML Action is based upon the UML *Activity* model element, UML activity diagrams should be well-suited towards the intent of operation modelling within IAML. UML activity diagrams allow for the complex behaviour of the example illustrated earlier in Figure 5.23 to be implemented as an independent *Activity*. In Figure 4.8, the same complex behaviour of Figure 5.23 has been expressed with a UML activity diagram.

It is not desirable to directly use UML activity diagrams within the IAML metamodel due to the size of the UML metamodel. However, it would be desirable to reuse aspects of the UML activity

describing Activity Operations, as general-purpose language concepts vastly simplify the implementation of domain logic.

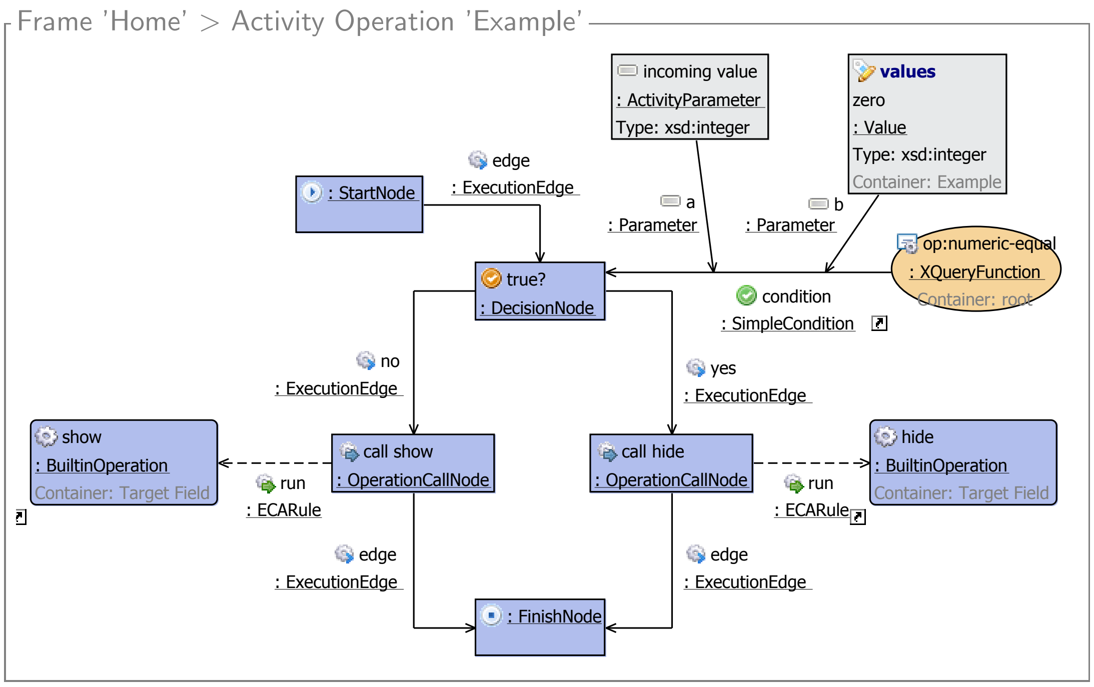


Figure 5.24: The same complex behaviour of Figure 5.23 expressed with an Activity Operation

diagram through metamodel restriction, where a smaller subset of the UML activity diagram is reused along with their concepts, and concrete execution semantics are defined for this smaller metamodel.

TODO Jens question: two logic models?

5.7.2 Activity Operations

Model developers may therefore describe the complex behaviour of web applications using Activity Operations, composed of operational elements derived from the UML activity diagram metamodel. At 23 model elements, the metamodel necessary to define Activity Operations is much smaller than the UML activity diagram metamodel. Due to lack of space, this metamodel is provided instead Appendix **??**, along with the behavioural definitions of each metamodel element.

An Activity Operation allows the complex behaviour defined earlier in Figure 5.23 to be implemented as the Activity Operation represented here in Figure 5.24; it is important to note the structural similarities between this example model and the UML activity diagram representation of the same behaviour in Figure **??**. It is also important to note that this figure does not include the single ECA Rule instance used to execute the Action.

Replacing the Default Behaviour of Model Elements

Throughout the definition of the IAML metamodel, all operations defined for each model element is defined as an instance of the Operation abstract class; at runtime, these references are provided instances of Builtin Operations through model completion rules. This design is intentional, as Activity Operations – which are also subtypes of Operation – may replace these builtin operations to replace default functionality. For example, it would be possible for a developer to replace the logic of the *update* operation of Input Text Field to also trigger additional functionality, or to prevent the text field from ever being updated through that operation.

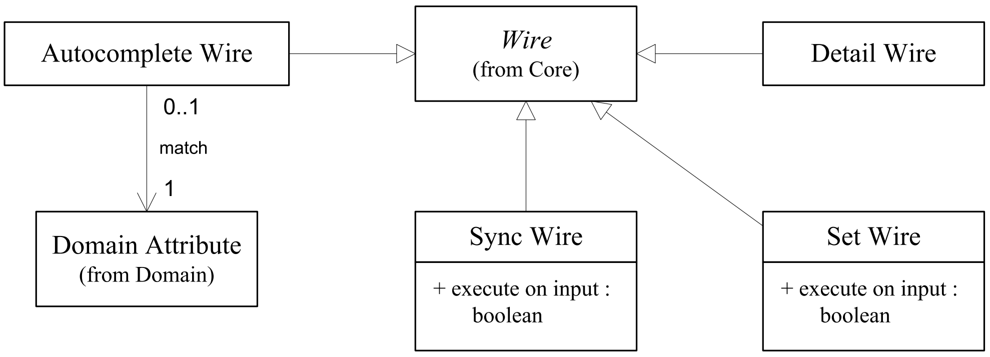


Figure 5.25: UML class diagram for the *Wires* package of IAML

5.8 Wires

As described earlier in Section 5.3.5, common web application design patterns are implemented as instances of Wires, which are then used to complete the model instance according to the design of the Wire through model completion. Each different design pattern is implemented as a subtype of the Wire abstract class, illustrated here in Figure 5.25.

Depending on the design pattern behind them, Wires may either be unidirectional or bidirectional. A unidirectional Wire has both a *source* and a *target*; whereas a bidirectional wire simply has *targets*. In some cases, it may be possible to replace a bidirectional wire with two unidirectional wires[38].

It is important to note this requirement as the distinction between Actions and Wires; an Action has a defined functionality and behaviour defined as part of its instance, whereas a Wire is an abstract description of higher-level behaviour, which is later implemented by ECA Rules through model completion. A Wire instance will *not affect* the generated application without first performing model completion on the model instance.

It is also important to note that Wires are a distinct concept from ECA Rules. ECA Rules specify the actions that a specific event may conditionally execute when the event is triggered; whereas Wires specify higher-level behaviours which are implemented through many individual model elements. For example, the implementation of a single Sync Wire will usually include at least four ECA Rules between the two connecting elements, as discussed later in Section **??**.

In this section, four different Wires will be discussed. The inference rules necessary to complete their functionality is not discussed here due to lack of space, however the inference rules for each Wire is discussed in detail in Appendix **??**; additionally, a summary of the number of inference rules used to implement each Wire is provided separately in Appendix **??**.

5.8.1 Sync Wire

As discussed earlier in Section 4.9, one of the most common design patterns in web applications is the need to keep the values of two elements synchronised. A Sync Wire can be used in situations such as keeping user interface forms synchronised across different interfaces; or for synchronising both a client-side interface and its server-side representation within a database.

An element may be connected to any number of Sync Wires; it is therefore possible that a *Sync Wire loop* may exist within a web application. This is allowable within IAML, and it is a requirement

[38]For example, replacing a bidirectional Sync Wire with two unidirectional Set Wires.

of the model completion and code generation frameworks to implement Sync Wires in such a way that the application will not result in any infinite loops.

An additional feature of a Sync Wire is to automatically keep the *children* of two elements synchronised as well. That is, if an Input Form with three contained text fields is connected to an empty Input Form, the Sync Wire will automatically populate the second Input Form with text fields in order to keep these forms synchronised[39]. These generated text fields will then subsequently be connected to the text fields within the original form using additional Sync Wires.

An example of a Sync Wire is illustrated earlier in Figure 5.22; here, an instance of a *named user* may be modified through an Input Form. The behaviour and functionality of a Sync Wire is implemented through model completion rules, and the interested reader is referred to the detailed decomposition of this process in Appendix **??**.

5.8.2 Set Wire

This unidirectional wire focuses on another common design patterns in IAML models; the need to keep the value of one object updated to another object, but not vice versa. A Set Wire can be used in situations such as keeping a client-side interface updated with changes in its server-side representation within a database, or for keeping a local database synchronised with a remote one. An example of a Set Wire is illustrated later in Figure **??**; here, a new user instance is used to populate the fields of an Email.

Similar to a Sync Wire, a Set Wire may also be used to construct scenarios of looping Set Wires, and these two wire types may be linked together. A Set Wire will also automatically keep the *children* of two elements synchronised in a unidirectional manner; a Set Wire will generate Labels instead of the Input Text Fields generated by a Sync Wire.

5.8.3 Detail Wire

As discussed earlier in Section 5.5.3, a Domain Iterator may be used to iterate over the contents of a data source. This Domain Iterator can be connected to a user interface using a Set Wire; for example, if a Domain Iterator is connected via a Set Wire to an Input Form, upon model completion the form will contain a summary of labels of the current instance within the Domain Iterator.

However, it is often the case that more detail for a particular instance needs to be displayed; or rather, a simple interface needs to be used when navigating over a result set, which may then be expanded into a full interface. The unidirectional Detail Wire represents a way of "zooming in" to a particular instance at runtime.

While the generated *target view* is normally connected to the source Domain Iterator via a Set Wire – that is, the detailed view is only provided in a read-only fashion – it may be desirable to instead replace this wire with a Sync Wire. In this way, the *target view* becomes a way to *update* the details of a selected instance as obtained via an iterator.

Example

The use of a Detail Wire is best illustrated through an example from the Ticketiaml example application. In Figure 5.26, a Detail Wire is connected from a *Ticket* Domain Iterator to a *View Ticket* Frame.

[39]It is not desirable for this to always occur; for example, password fields in a database should *not* be displayed to anonymous users. IAML supports this scenario by designing *overridable elements*, as discussed later in Section 7.5.2.

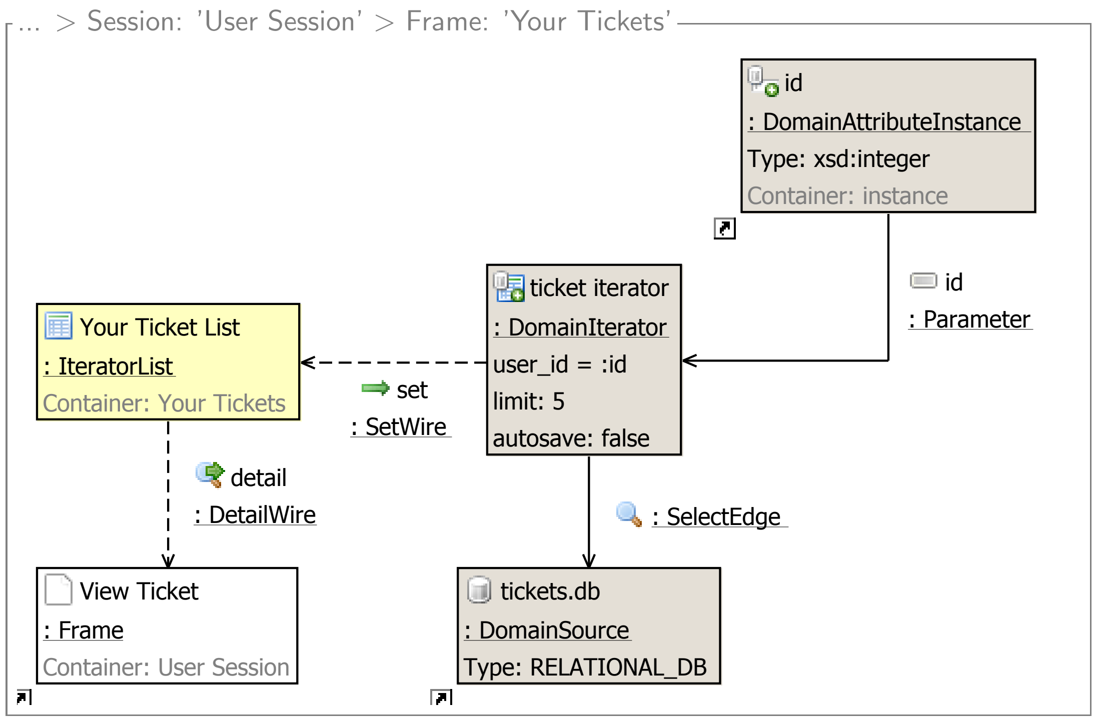


Figure 5.26: Ticketiaml: Connecting a Detail Wire to a Domain Iterator

This will allow the user to view more details about each selected *Ticket* instance illustrated through the Iterator List. In particular, model completion will have the following effects:

- The *Your Ticket List* Iterator List will now include a Button. When a user clicks on this Button, the user will be redirected to the *View Ticket* Frame (the *detail frame*). As expected, this behaviour is subsequently implemented through an ECA Rule.
- The *View Ticket* Frame will include an Input Form to view the details of the selected *Ticket*. This form will be populated with the same instance that the user selected in the first Frame, by defining a new Domain Iterator selected via a query based on an incoming Query Parameter, and connecting this iterator to the form via a Set Wire.

5.8.4 Autocomplete Wire

As discussed later in Section **??**, one of the motivating use cases for IAML was to support *autocomplete*. Autocomplete allows a user to complete an input field by performing a quick inline search based on a search query, which occurs asynchronously without the user having to leave the current page. IAML consequently defines the Autocomplete Wire model element to support this common design pattern.

Autocomplete was one of the first well-known features of AJAX websites, as illustrated by the Gmail web application here in Figure 5.27. A search box allows the user to navigate through a large address book by typing in the first few characters of a name, e-mail address, or both. The search results pop up underneath the input text, and can be navigated by using the keyboard. Once selected by the keyboard or mouse, the resulting address is inserted into the input text field, and another address can then be searched.

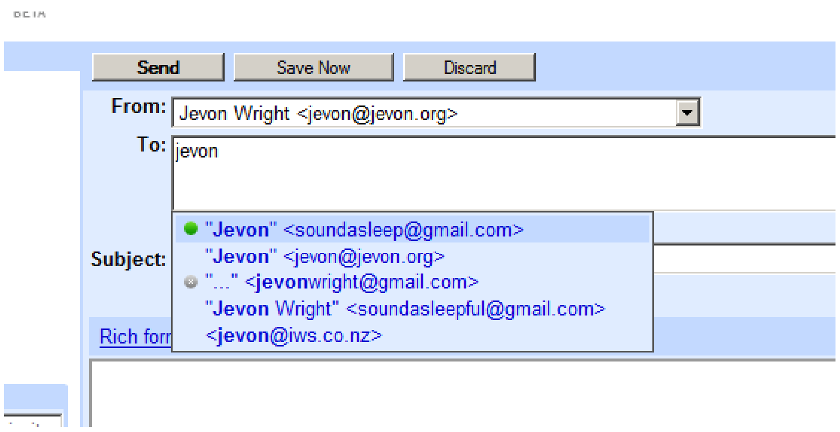

Send
Save Now
Discard
From:
Jevon Wright <jevon@jevon.org>
To:
jevon
"Jevon" <soundasleep@gmail.com>
"Jevon" <jevon@jevon.org>
"..." <jevonwright@gmail.com>
"Jevon Wright" <soundasleepful@gmail.com>
<jevon@iws.co.nz>
Subject:
Rich forr

Figure 5.27: An instance of autocomplete in Gmail

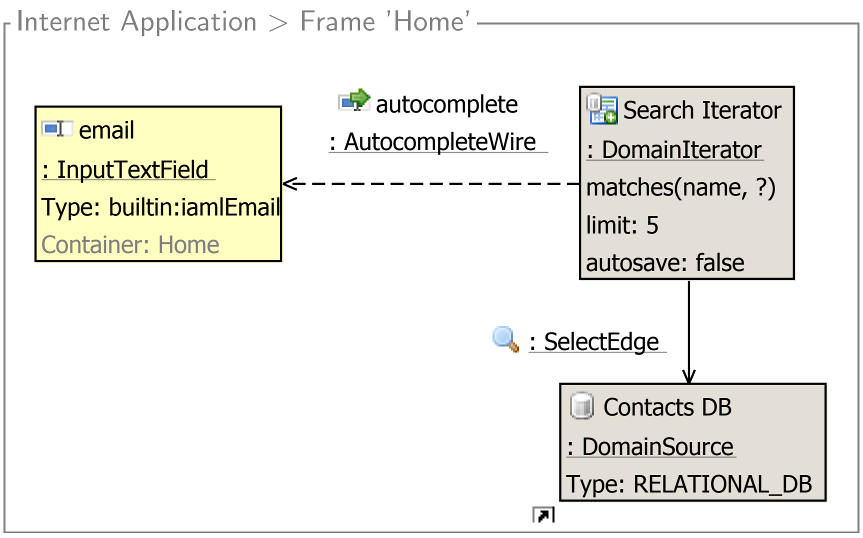
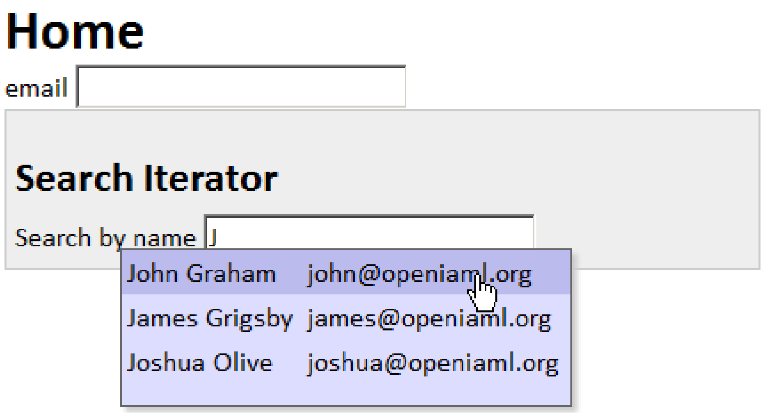


Figure 5.28: Autocomplete implemented in IAML using an Autocomplete Wire

Example

Autocomplete behaviour may be implemented within IAML using an Autocomplete Wire as illustrated in Figure 5.28. Here, a Domain Iterator is specified as the source, and an Input Text Field as the target. This target field will be updated with the value of a selected attribute once it has been searched for and selected by the user. The match reference of the Autocomplete Wire is set to the "name" Domain Attribute of the underlying Domain Type.

The resulting application generated for this example model is dependent on the implementation of the modelling language, code generation framework, and its runtime libraries. With respect to the proof-of-concept implementation of IAML discussed later in Chapter 7, this example model will generate an interface similar to Figure 5.29. Here, the user has entered in the search query "J", and a list of three matching results have been displayed; if the user clicks on one of these, the resulting e-mail address will be inserted into the *email* Input Text Field.


Home

email

Search Iterator

Search by name J

John Graham	john@openiaml.org
James Grigsby	james@openiaml.org
Joshua Olive	joshua@openiaml.org

Figure 5.29: The user interface for an Autocomplete Wire, as generated by the proof-of-concept implementation of IAML

5.9 Users and Security

As discussed earlier in Section 4.8, both users and security are important domain concepts for web applications. A modelling language for RIAs should not only support the design of secure applications, but also simplify common scenarios involving users and their roles and permissions.

5.9.1 Roles and Permissions

Based on the four security models proposed earlier in Section 4.8, IAML uses the Role-Based Access Control (RBAC) security model as described by Sandhu et al. [258]. However, the restriction that users may only possess permissions via a role is relaxed; a user may obtain *any* permission regardless of roles. This approach is taken to increase flexibility; for example, one identified use case for RIAs allows a web application to assign specific permissions to individual users[40], bypassing roles altogether. IAML defines the metamodel elements Role and Permission to represent the RBAC concepts of roles and permissions respectively.

Definitions

Ceri et al. [44] argue that while some user information is generic – for example, e-mail addresses and passwords – other user information is domain-specific. User profile modelling is therefore very similar to the existing domain modelling approach supported by Domain Types, as discussed earlier in Section 5.5.1.

IAML therefore reuses these domain modelling concepts to support user modelling as illustrated in Figure 5.30. In particular, a Role is a subtype of a Domain Type; the Domain Attributes of the Role represent the *profile attributes* of that role; and Roles may be defined in a multiple inheritance hierarchy in order to define a *role hierarchy*.

In order to support the common user modelling functionality as described above, all Roles in a model instance inherit a *default role* containing default user attributes, such as an `iamlEmail`-typed Domain Attribute named “email”, and a `string`-typed Domain Attribute named “password”[41]. By

[40] Use Case 36: *User Content Security*.
[41] This datatype is typed to `string` until the datatype `iamlPassword` is defined, as discussed earlier in Section 5.4.2.

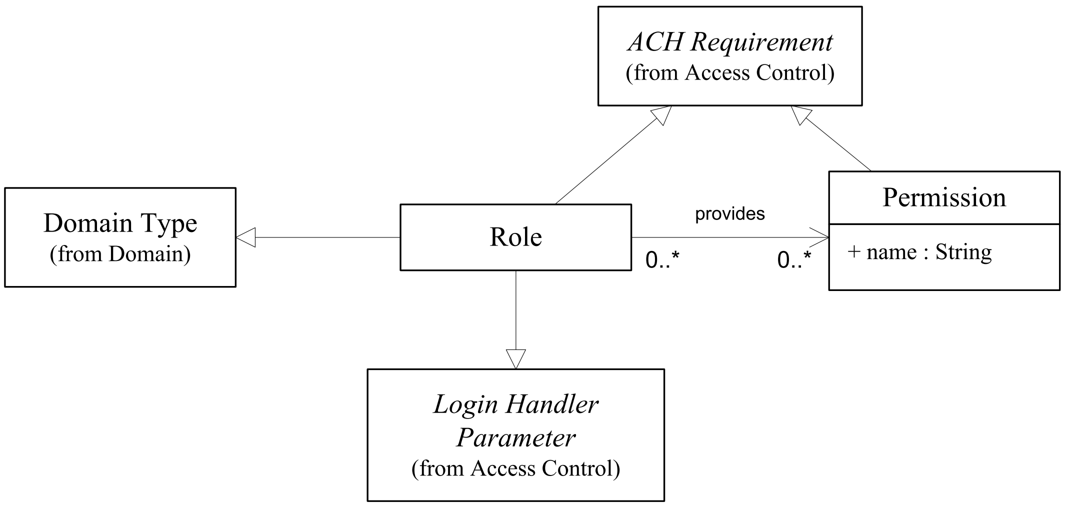


Figure 5.30: UML class diagram for the *Users* package of IAML

defining a default role, model developers do not need to manually define user profile attributes that are common to most web applications.

User Instances

As user modelling is based upon domain modelling, IAML reuses the concepts of data access in Section 5.5.3 for defining the semantics of user instances. That is, user information may be stored in any valid Domain Source (such as a relational database), and a Domain Iterator may be used to select valid *users* in a web application according to a specific Role[42]. A Domain Iterator may also be used to create new user instances.

5.9.2 Login Handlers

While it is possible to use Domain Iterators and Gates directly to prevent access to Scopes, access control is a very important aspect of web applications, and there should be modelling support for these common scenarios. A Login Handler can be used to prevent access to a Scope based on an incoming requirement, and three requirements are defined in IAML.

Similar to a Wire, the functionality of a Login Handler is implemented through *model completion* so a developer may modify the generated behaviour. In particular, a Login Handler will generate login and logout pages, and add Gates or ECA Rules to redirect the user to these Frames if the current user does not satisfy the requirements of the handler. These model completion rules are discussed in further detail in Section **??**.

While a Gate also prevents access to a Scope, the key difference is that a Login Handler provides most of the repetitive scaffolding for common authentication scenarios – such as user logins, or passwords – whereas with a Gate this functionality must be implemented manually by the model developer. A Login Handler controls access through two mechanisms, as illustrated in the package diagram in Figure 5.31: secret keys obtained from Values, and domain objects and users obtained from Domain Iterators.

A *secret key* Login Handler requires that the user provides a single *password*. This is the simplest

[42]In this case, the Schema Edge from the Domain Source will point to a Role rather than a Domain Schema.

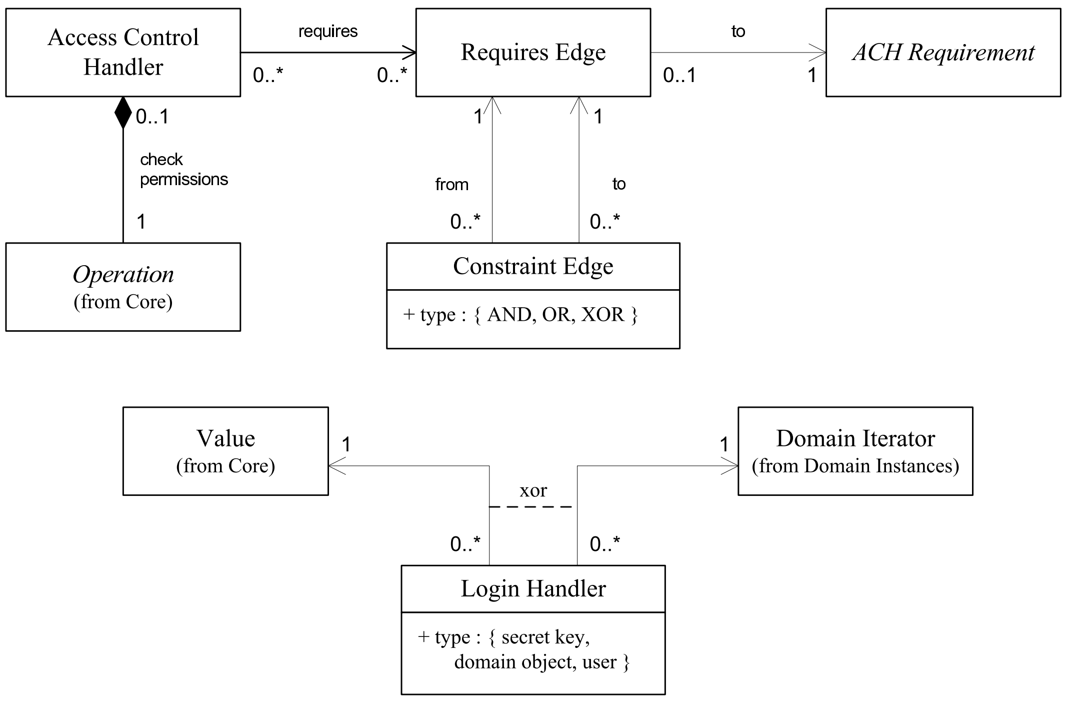


Figure 5.31: UML class diagram for the *Access Control* Package

form of a Login Handler which does not require a valid user instance. Alternatively, a *domain object* Login Handler requires that a valid instance of a Domain Type exists, as specified by the incoming parameters of the Login Handler. This parameter must be an Domain Instance or Domain Iterator.

5.9.3 Access Control Handlers

An Access Control Handler may be used to restrict user access to a Scope to a particular set of Roles and/or Permissions, and is also implemented through the model completion rules discussed later in Section **??**. An Access Control Handler is similar to the Login Handler, except that a Login Handler requires the creation of login and logout pages; a Access Control Handler is only focused on the composition of complex role and permission requirements, and the two model elements may be used simultaneously. The package diagram for the model elements necessary to implement a Access Control Handler is also illustrated in Figure 5.31.

An Access Control Handler is also similar to a Gate (Section 5.10.4), except that Gates are only focused on satisfying incoming Conditions. A Gate may be used in the implementation of a Access Control Handler (i.e., through model completion), except that the model developer would have to implement their own check permissions Operation manually.

Example

The Login Handler and Access Control Handler model elements may be used together to implement complex security requirements; in particular, the Login Handler provides functionality to allow users to login, and the Access Control Handler provides functionality to check the roles and permissions of the current user.

This pattern has been used in the Ticketiaml example to implement user security, as illustrated in Figure 5.32. The Login Handler ensures that a *User* has logged on, and the Access Control Handler

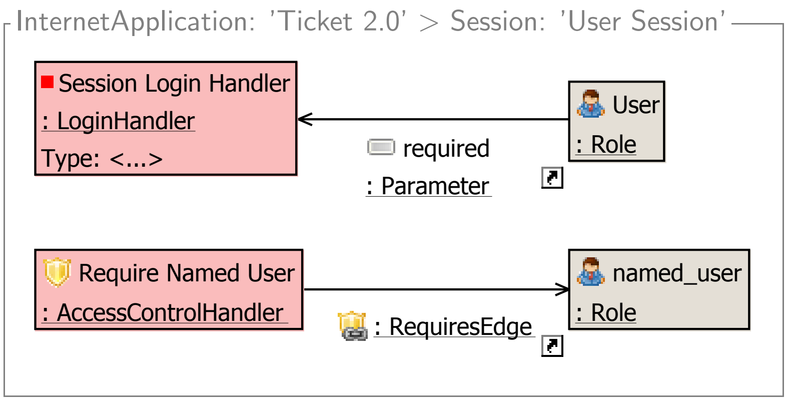


Figure 5.32: Ticketiaml: Protecting access to a Scope via an Login Handler and Access Control Handler

ensures that the *User* also possesses the *named_user* Role.

5.9.4 OpenID

While password authentication is a common web application scenario for user authentication, more recently the concept of *decentralised authentication* has become popular, and in particular the OpenID standard [247]. OpenID allows for users to control their own identity verification, and this improves both reliability, security and privacy. Modelling OpenID verification is an important use case[43] of Rich Internet Applications.

Web applications that use OpenID do not need to store sensitive personal information like e-mails or passwords in order to perform identity verification, but can instead store a single URL which will independently perform this service. By design, OpenID only handles identity verification, in order to reduce the scope of the project and increase security. The OAuth standard [127] may instead be utilised to deal with user profile management.

On a separate level, OpenID authentication can be considered as merely a *specific type* of authentication; other types of authentication include Google Accounts or Microsoft Passport, and it should be simple to include these as separate authentication protocols. Furthermore, it may be possible to unify all these different authentication methods into a single "authentication" datatype, such as the datatype `iamlIdentity` proposed earlier in Section 5.4.2.

It is important to let the developer decide which types of authentication methods they support, rather than simply assuming that the developer would want all authentication types. For example, OpenID does not provide identity management, and is easier to subvert or modify than a Google Account or a Microsoft Passport identity due to its decentralised nature; consequently there may be some scenarios of authentication where the developer would consider OpenID identity verification as too insecure.

Rationale behind the type `iamlOpenIDURL`

Many options were considered in methods of representing OpenID authentication. One option was to define a particular type of Domain Attribute with an isSecure attribute which would represent authen-

[43]Use Case 50: *Single Sign-In Solutions*.

tication attributes. However this would mean OpenID authentication could only apply to instances of Domain Types.

Another option was to define a subtype of Gate – such as an OpenID Gate model element – which would restrict access to a Scope if a valid OpenID was not provided to the gate. However this approach is too platform-dependent – this would mean that each separate authentication method would have to have a separate Gate subtype within the IAML metamodel.

The decision was made to extend the existing datatype framework and provide a type `iamlOpenIDURL` to represent valid OpenID instances. This meant that much of the existing gates and validation frameworks could be reused; for example, a Gate requiring a valid `iamlOpenIDURL`-typed Value instance is one implementation of the proposed OpenID Gate.

This approach may break an assumption that cast instances can always be recast back to an original type. One can consider this as a digital signature on the instance type; if the signature is lost by converting the instance into a different format, then the digital signature (and consequently validity) of the original instance is lost.

5.10 Scopes

As discussed earlier in Section **??**, the ability to model lifecycle and handle lifecycle events may be a useful technique in the development of RIAs. IAML supports lifecycle modelling by defining *scopes* through the abstract model element Scope, and assigning various lifecycle events to these scopes.

The scopes in IAML are based on the possible lifecycle layers of Rich Internet Applications [319] discussed earlier in Section 4.4.1. Each Scope has different initialisation and storage semantics; for example, an Internet Application is accessible to all users, whereas a Session is only accessible by a single browser instance. Scopes in IAML therefore follow a hierarchy, where one Scope can contain another Scope, as illustrated in the package diagram in Figure 5.33.

A Scope may contain any number of value instances, including primitive type instances (Values, Section 5.4.3) and complex type instances (Domain Iterators, Section 5.5.3). The semantics of how these instances are represented, stored and shared across different users form the *storage semantics* of that Scope.

When a Scope is rendered, the lifecycle events for that Scope must be executed before any elements within the current Scope are rendered. However, the lifecycle events for any parent Scopes – that is, up the scope hierarchy – must be executed *before* the lifecycle events for the current Scope. This allows for the designer to protect access to children scopes through parent scopes; for example, in order to access a Frame, the current user must first verify their identity through a parent Session.

5.10.1 Internet Application

An instance of a Internet Application represents an entire web application, and is consequently the root element of the model (and the indirect parent of all model elements). Because *onInit* is only ever called once, this event can be used to perform application initialisation, such as registering web services, initialising databases, and so on. The structure of the Internet Application model element is discussed in further detail later in Section **??**.

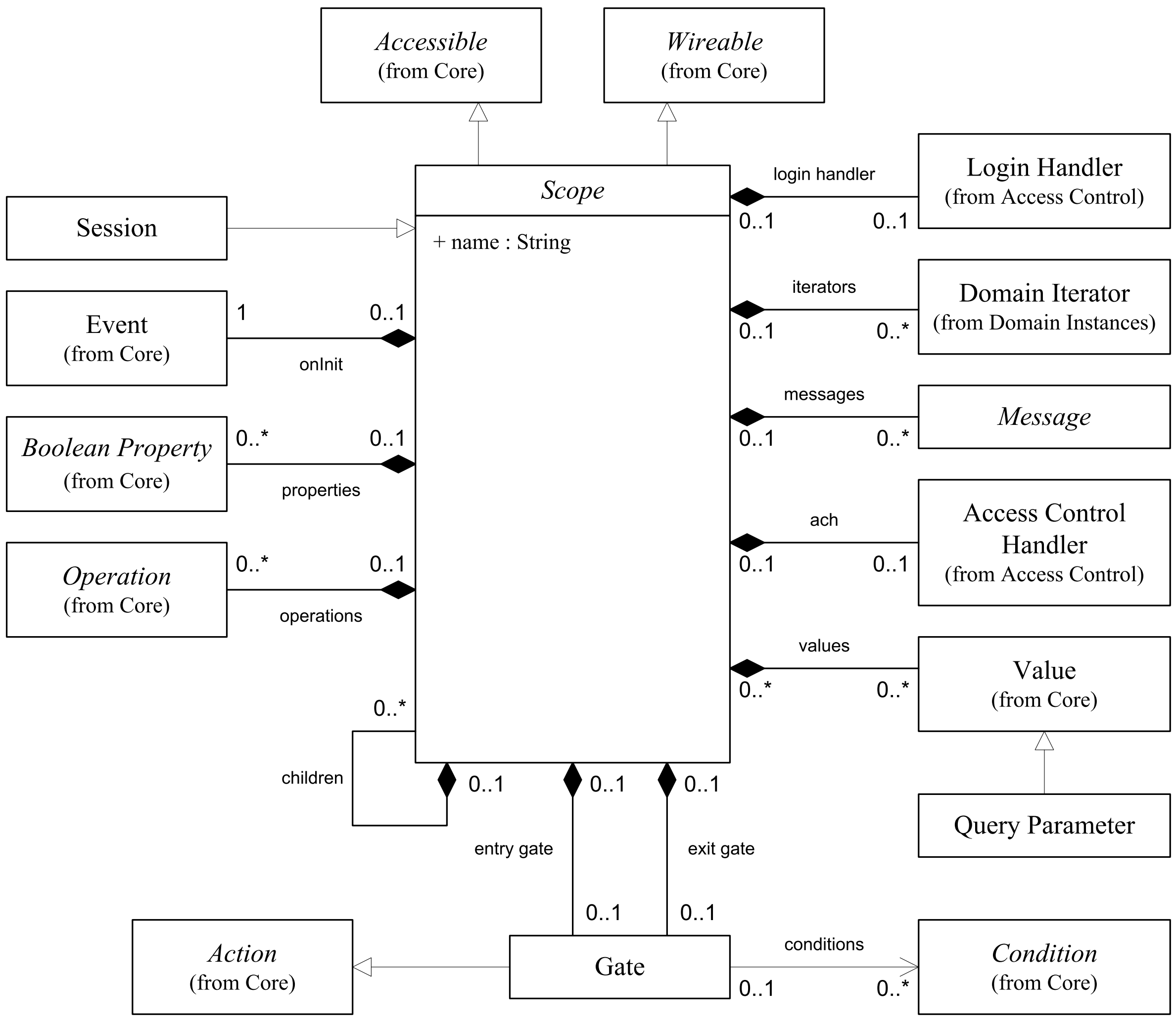


Figure 5.33: UML class diagram for the *Scopes* package of IAML

5.10.2 Session

An instance of a Session represents a user session in the web application; that is, a "sequence of Web transactions issued by the same user during an entire visit to a Web site", as discussed by Cardellini et al. [40]. Because a web application can support multiple Sessions, it is possible for a user to possess many session instances at any point in time; this may prevent data leakage, as data between the Sessions are by default independent.

5.10.3 Failure Handlers

It is necessary to support the graceful failure of errors in a web application. In IAML, the concept of a *failure handler* is used as a form of exception handling, and each Scope in the web application has an associated failure handler. In the case of an exception, this failure handler is executed.

An outgoing ECA Rule from a particular Scope with the name "`fail`" is defined as a *failure handler* for the given scope. If an error occurs during the execution of the given Scope, then the failure handler for that Scope is executed according to the semantics of the ECA Rule (Section 5.3.3). For example, if an Operation executed via a ECA Rule *fails*, all subsequent Action Edges in the *current execution set* are cancelled and the *failure handler* for the Scope containing the source of the

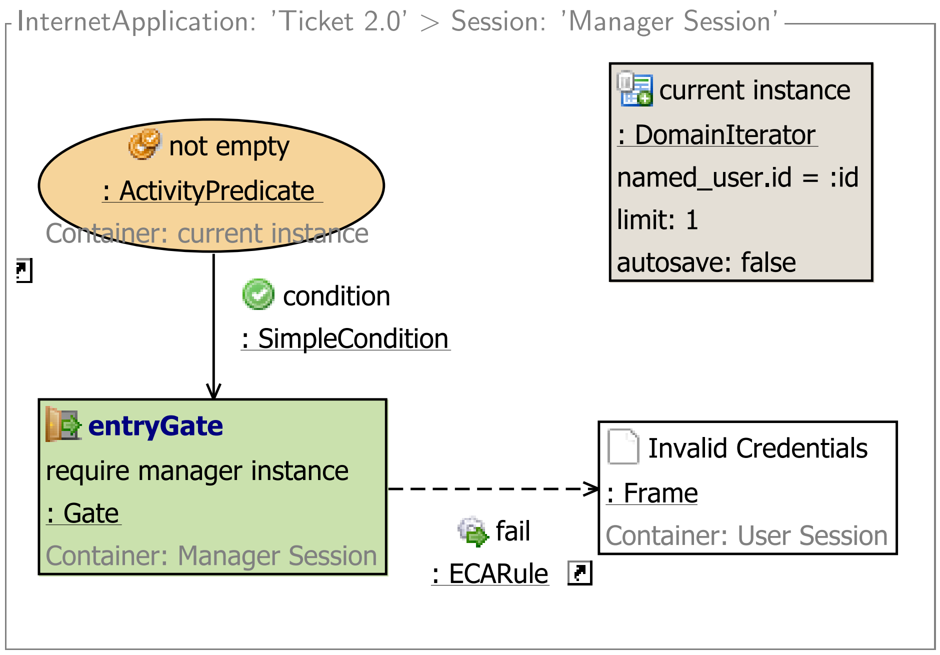


Figure 5.34: Ticketiaml: Protecting access to a Scope via an entry Gate

ECA Rule is executed.

If the current Scope does not define a *failure handler*, then the *failure handler* of the containing Scope is used as the failure handler. If an exception is not caught through the containment hierarchy of Scopes, then the exception is caught by the runtime environment and the web application will display a runtime error.

Since exceptions are essentially a form of triggerable event, it may be desirable to instead implement exception handling through the event modelling approach of IAML. This remains future work; for example, an event trigger such as *onException* could be introduced into the IAML metamodel, using the same failure handler semantics as discussed before.

5.10.4 Gates

One common use case in web application development is to selectively prevent access into portions of the web application, without first obtaining the appropriate permissions. Another is to force a user to view an advertisement before they may proceed with the current page.

In IAML, a Gate may be used to address these scenarios, to restrict access into, or out of, a Scope according to some condition. However, a web application developer should use Gates with caution, as web application users may become frustrated if it is too difficult to satisfy a condition, and simply abandon the web application.

Two types of Gates are defined. An entry Gate restricts access *into* a given Scope, and will prevent entry until a particular Condition is satisfied. Likewise, an exit Gate restricts access *out of* a given Scope, and will prevent exit until a particular Condition is satisfied. The functionality behind Gates are strongly dependent on the method in which the Scope may be accessed.

An exit Gate cannot actually prevent the user from closing down their web browser, or manually navigating to an external URLs. This is a restriction of the current implementation of web browsers. An exit Gate may still be used to prevent any further interaction within the modelled web application.

In the Ticketiaml application, an entry Gate is used to implement a role check for the *manager* Session. As illustrated in the partial visual model in Figure 5.34, this Gate will fail if the incoming Condition – a Predicate defined as part of the Domain Iterator within the same session – is not true. Consequently, if a logged-in user tries to access any Frame within the protected Session, they will be redirected to the *Invalid Credentials* Frame.

5.10.5 Query Parameter

A Query Parameter is an extension of the Value model element discussed earlier in Section 5.4.3, and represents a parameter from the current browsers' URI request. For example, a request URI `target?name=value` would set the "name" Query Parameter to the value "value". Conceptually a Query Parameter is different from a Value; the Value belongs to the containing Scope and persists across calls within that scope, whereas a Query Parameter is global and is entirely dependent on the request URI. Therefore, it is necessary to have a separate element to distinguish the two.

5.10.6 Discussion

During the design of the IAML metamodel, other model elements within the IAML language were also considered as potential Scopes. An Operation could be considered a Scope; here, the *onAccess* event would be fired whenever the operation is executed. A Visible Thing could also be considered a Scope, and would follow the *onAccess* event semantics of Frame.

Similarly, a request itself could be modelled as a Scope [319], but this would require a precise definition of a user request. For example, can only requests triggered explicitly by the user be considered a request, or would requests via AJAX also be considered a request? Such a Scope could define events for when the request begins, finishes, or is redirected to another URL.

The idea of supporting user-definable scopes was also considered; for example, allowing the developer to specify a "within the administration area" scope, or to designate scopes using tagging. This was not one of the identified use cases of RIAs, and the implementation of this concept should not be difficult as future work. However it is not possible to describe arbitrary events without requiring an event modelling language, as discussed earlier in Section 5.3.3.

The IAML metamodel does not define pre- or post-lifecycle events, as these definitions have not proven necessary. For example, an *onFinish* event for Sessions could be triggered when a Session times out [217, pg. 114]; however, one cannot guarantee this event will ever be triggered.

5.11 Messaging

Messaging refers to the ability to encapsulate content into a form which may be sent from a source to a recipient. Web applications are mostly *pull-based*; that is, they provide content in response to a request, rather than actively pushing out content (*push-based*). However, web applications still have a number of ways of pushing out content, even though their underlying implementations may be pull-based. The most common technique is through sending e-mails[44].

[44]Use Case 33: *E-mailing Users*.

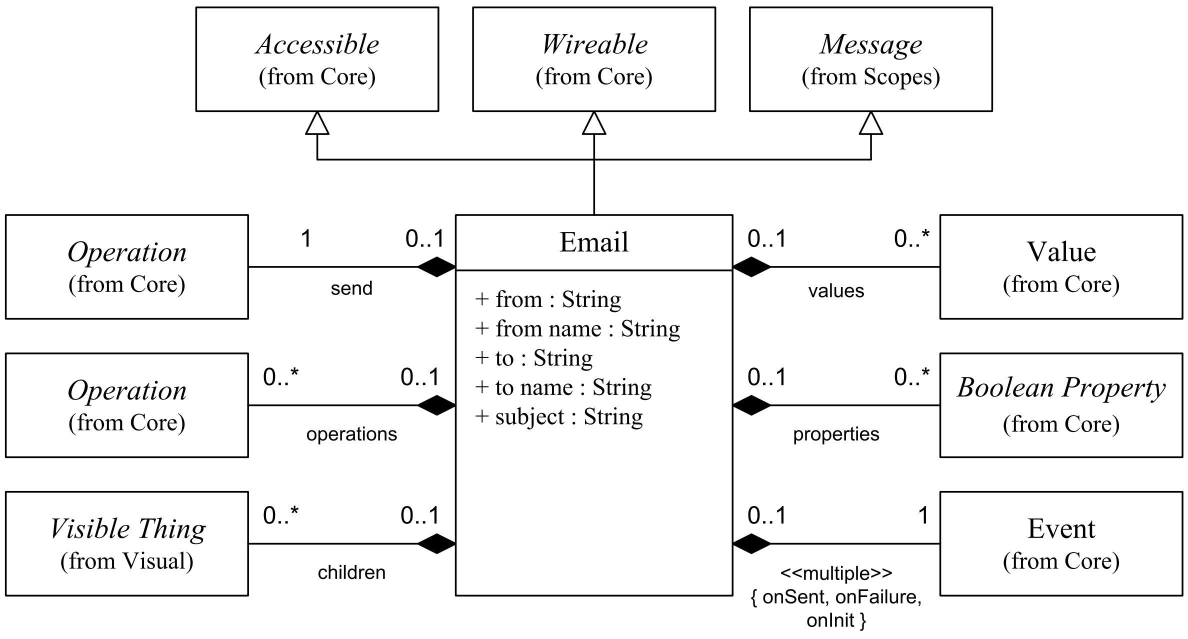


Figure 5.35: UML class diagram for the *Messaging* package of IAML

5.11.1 Email

An e-mail or *mail message*, as defined by RFC 2821 [162], is a message sent from a sender to a receiver through a number of intermediary servers. The message itself has a number of headers and a *body* which may be encoded according to MIME [96]. IAML supports the composition and sending of e-mails through the Email model element, the structure of which is represented in Figure 5.35.

The Email model element is a wrapper for a number of Values, each representing the various required headers for the e-mail message; and also contains events and operations necessary for sending the actual e-mail, such as the send Operation. The Email model element also defines a number of element-specific attributes, which are used as default values if a similarly-named Value is not present or not set. As an Email is Wireable, its content may be synchronised through Set Wires.

When an Email is composed in order for delivery, the *body* of the Email will be composed of all contained Properties and their values at the time the Email was sent. The format of this body may use a *custom template* if this template is specified and supported by the language implementation.

5.11.2 Future Work

There are a number of other ways of delivering messages in web applications, as identified in the list of use cases in Appendix **??**. As the proof-of-concept implementation of IAML has been strictly limited to Basic RIAs as discussed earlier in Section 5.1.1, these messaging types have not been implemented in IAML. However, they will be briefly discussed here as potential future work.

1. A short message may be sent to a user via a text message or SMS for a mobile device[45]. IAML could therefore define a Text Message modelling element to support this scenario, which would likely be similar in composition to the Email element above. That is, there could be *onSent* and *onFailure* events defined, however there would be no need for a subject Value, as SMSes do not support subject fields.

[45]Use Case 32: *Mobile Phone Communication.*

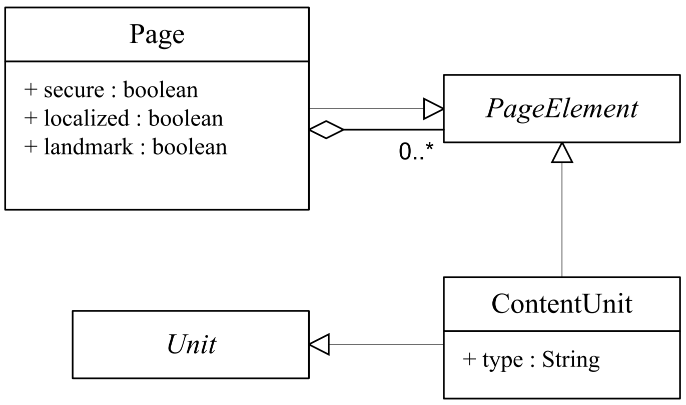


Figure 5.36: Instantiation of user interface elements in WebML: an excerpt from the WebML metamodel [210]

2. Web applications may communicate with web services[46], as discussed earlier in Section **??**. There are different technologies that may be used in communicating with web services; the service may be defined formally according to standards such as SOAP [296] and XML-RPC [314], or in a proprietary format using JSON [56].

5.12 User Interface Modelling

As discussed earlier in Section 2.3.1, the user interface is a very important aspect of RIAs as their interactivity and richness depends directly on its interface. *User interface modelling* is supported in IAML to describe the composition and structure of user interfaces, and is an important part of the modelling language.

5.12.1 Existing Approaches in Modelling User Interface Types

WebML defines different types of *Content Units* such as *Entry Units* (similar to Input Forms) and *Index Units* (similar to Iterator Lists) [43]. However within its UML metamodel, these different types are treated as "black-box plugins to the three existing models, rather than constituents of an independent modeling layer" [210]. To illustrate this design, an except from the WebML metamodel as described by Moreno et al. [210] is illustrated in Figure 5.36.

The WebML approach is similar to defining a Visible Type element within the metamodel, to allow different types of elements to be loaded through a library. The WebML approach achieves this very weakly[47][48]; by defining a *type* of type `String`, there is no guarantee that the given type of ContentUnit will actually exist in the system, and all aspects of the implementation must essentially implement their own type checking system. However, this does solve the issue of defining behavioural semantics, because all of the semantics are moved into third-party plugins.

UWE, on the other hand, defines separate types of UIElements for each type of interface element within the metamodel, such as Button and Text. An excerpt from the UWE metamodel as described

[46]Use Case 27: *Web Service*.

[47]This can be considered a specific instance of the *Primitive Obsession* design anti-pattern discussed by Fowler [87, pg. 81–82], where the differences between different types are expressed via a primitive, rather than through an object type hierarchy causing issues with referential integrity.

[48]**TODO:** Should this anti-pattern have a specific name in this thesis? Such as *Aversion to Object Type Inheritance*.

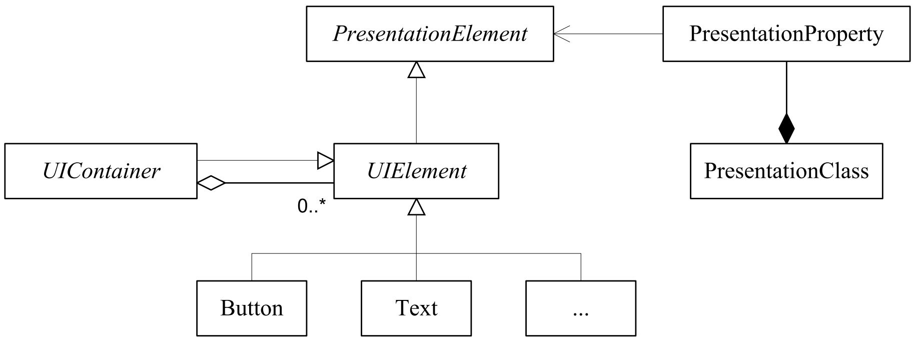


Figure 5.37: Instantiation of user interface elements in UWE: an excerpt from the UWE metamodel [171, pg. 13]

by Kroiß and Koch [171] is illustrated in Figure 5.37. Instances of the user interface elements are notated in UWE model instances using stereotypes [171, pg. 22]. UML itself does not provide any standard way of modelling user interfaces, and extensions such as UML*i* have been proposed [59].

As these two investigations show, both approaches can be used to model the user interface of a web application. More research is necessary to identify when one approach is beneficial over the other. However, the design of the user interface modelling aspect of IAML is most similar to the WebML approach.

5.12.2 Visible Thing

In IAML, the *Composite* design pattern [103] is used to design user interfaces, as illustrated in the UML class diagram in Figure 5.38. Visual elements may be used to compose interfaces, and these elements themselves may be composed of other visual elements. The composite design pattern also aligns nicely with the hierarchical modelling approach in IAML, as top-level visual elements can hide the complexity contained within.

The Visible Thing type is the abstract type for all visual elements in the language. Since all visual elements are designed to be renderable to the user, all Visible Things have additional events related to user interactivity – such as the *onClick* and *onInput* events discussed earlier in Section 5.6 – and are all Changeable (and therefore have a field value). This chapter will briefly introduce each subtype of Visible Thing within IAML; these are discussed in further detail in Appendix **??**.

It is also necessary to define the way in which a user interface is *rendered* to the user. The spatial layout of a user interface is important, so each Visible Thing also specifies its render order. While this spatial information is placed within the model instance itself, it may be desirable to move this information into a separate layout model in the future to encourage the separation of concerns between the interface layout and the underlying model.

Button

A Button represents a clickable button in the user interface, and is derived from the HTML `BUTTON` element specification [290], If triggering the *onClick* event does not change the state of the web application – for example, it does not modify any databases or interact with any external component – then the Button may be rendered as a text hyperlink.

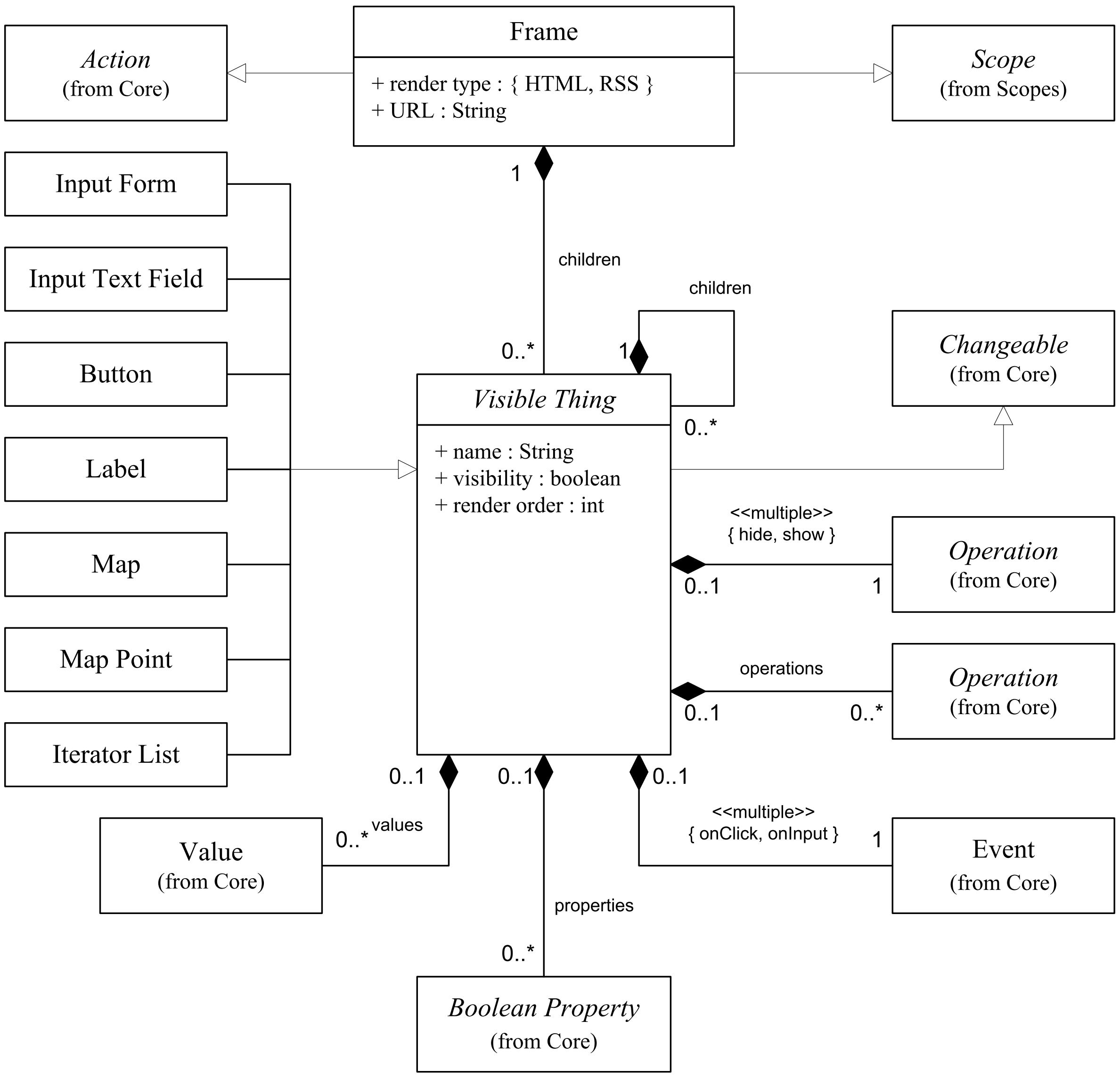


Figure 5.38: UML class diagram for the *Visual* package of IAML

Label

A Label represents a static block of text that is not user-editable, but may still be modified programmatically and trigger *onClick* events. This element is derived from the HTML `LABEL` element specification.

Input Text Field

A Input Text Field represents a text field that accepts text input from the user, and is derived from the HTML `INPUT` element specification [290].

Input Form

An Input Form represents a group of related input elements, and is derived from the HTML `FORM` element specification [290]. A Input Form is particularly useful when using a Sync Wire or Set Wire, as discussed earlier in Section 5.8.1. In particular, if a Domain Iterator is connected via one of these wires to an empty Input Form, the form will be populated with the necessary user interface elements to read or to edit the given iterator instance.

Map

A Map represents a geographical area as an interactive map – that is, the map can be navigated from within the browser. Compared to the other defined Visible Things, including the Map model element is particularly interesting, as it has no analogy in either HTML 4.01 [290] or the upcoming HTML 5 standards [299]; however, if a platform-independent map element is introduced into an upcoming HTML standard, the IAML metamodel will be automatically forward-compatible. A Map may contain many Map Points in order to show multiple locations simultaneously.

Map Point

A Map Point represents a single geographical point, whereas a Map represents a single geographical area. A Map may contain any number of Map Points, all which will be rendered within the current Map as separate points; however, a Map Point does not need to be contained by a Map, in which case the point is rendered individually. In order to model an indeterminate number of Map Points within a Map, an Iterator List may be used as discussed in the next section.

Iterator List

All of the Visible Things discussed so far in this section only support single instances; that is, the number of these elements that may be rendered is defined at design time, and cannot be controlled at runtime. A Iterator List allows for a set of Visible Things to be rendered an arbitrary number of times, controlled at runtime via an instance of the Domain Iterator model element.

For example, if an Iterator List contains a single Label and the connected Domain Iterator has five results, then five Labels will be rendered at runtime. If the Domain Iterator is empty, then the Iterator List will also be empty.

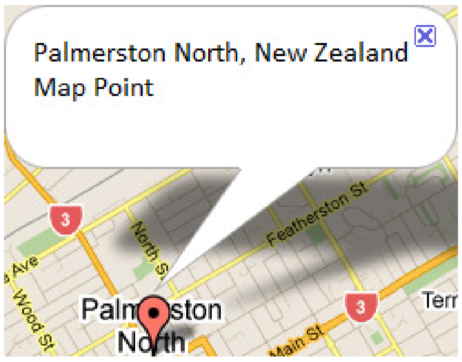
Model Element	Sample Representation
Button	Button
Input Text Field	Input Text Field
Input Form	**Input Form** Input Text Field Button
Label	Label
Map	
Map Point	 Palmerston North, New Zealand Map Point
Iterator List	**value** / **key** Scope / Frame Visible Thing / Input Form Visible Thing / Label

Table 5.4: Sample visual representations of interface modelling elements in IAML

5.12.3 Sample Representations

A sample rendering for each Visible Thing subtype is illustrated in Table 5.4. In this summary, the visual representations of Map and Map Point are those provided by the Google Maps mapping component.

5.12.4 Frame

A Frame is used in IAML to represent a block of content that may be accessed and rendered independently by a web browser, and as an Action may be connected by an ECA Rule to support user navigation. By default, the interface to a Frame will be provided through the combination of web technologies such as HTML and Javascript; however, if the render attribute of the Frame is set to `RSS20`, then the content of the Frame will be available as an RSS 2.0-formated feed[49] [255].

The concept of a Frame includes both the concepts of a top-level page, and a block of content within another Frame. Top-level pages can also specify a custom URL in order to specify the intended URL of the Frame. An important piece of future work is in supporting the definition of sub-frames, and the composition of these with parent Frames in order to support richer web applications. This technique could be used to implement functionality such as wizard-based workflows [285].

By specifying that an Frame is an Action, an ECA Rule may also be used to redirect the browser to a different Frame. Any incoming Parameters to the connecting ECA Rule are therefore passed as Query Parameters to the target Frame.

Example

Within Ticketiaml, the implementation of the *Browse Events* page is implemented through the definition of a Frame consisting of a number of contained Visible Things, as illustrated in the partial model of Figure 5.39. Here, an Domain Iterator is used to populate an Iterator List of matching events via a Set Wire, which then populate a Map via another Set Wire. This iterator is also restricted via an incoming parameter from an Input Text Field.

5.13 Internet Application

In the previous sections of this chapter, each of the modelling concepts used in the IAML metamodel have been defined. The final remaining element that needs to be defined is the Internet Application element, which is the root element of any IAML model instance. An Internet Application defines the Scopes, Domain Types (including Roles), Domain Sources, Permissions and Predicates of the modelled RIA, as discussed later in Section **??**.

5.14 Visual Modelling Metaphors

As discussed earlier in Section 5.1.5, the development of a visual modelling language is an important design goal for IAML. One part of the design of the IAML visual modelling language included the selection or design of *visual metaphors* – the reuse of familiar concepts outside of their conventional

[49] Use Case 26: *Data Feeds*.

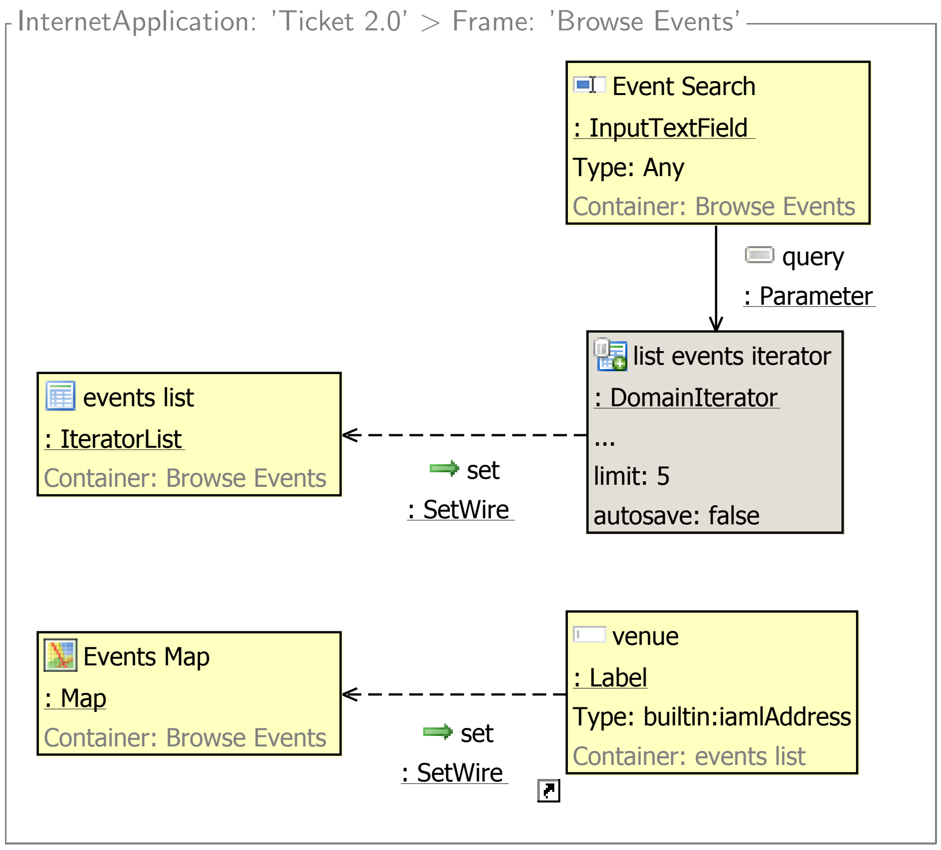


Figure 5.39: Ticketiaml: Designing the user interface for the *Browse Events* page using Visible Things

meaning to express a similar concept – in order to reduce the mental load for developers, and improve system accessibility, as discussed earlier in Section 3.3.1.

Each of the hierarchical layers proposed earlier in Section 5.2.2 can afford their own visual metaphor. For example, the operation modelling layer can reuse the visual metaphors from UML operation modelling [224]. It is important for a visual modelling language to explicitly state these intended visual metaphors; a summary of these metaphors are provided in Table 5.5, similar to the visual metaphor summary of Grundy et al. [121].

Layer	**Contents**	**Proposed Visual Metaphor**
Overview	Domain sources, sessions, frames	UML deployment diagram
Navigation	Frames, sessions, access control, login handlers	Navigation model
Interface	Visual elements, interface events	User interface builder
Domain	Types, type relationships, attributes	UML class diagram
Operations	Activity nodes, conditional nodes, execution flow, data flow	UML activity diagram

Table 5.5: Visual Metaphors for a Rich Internet Application modelling language

5.14.1 Overview Layer

The top layer of the model instance, the *overview layer* represents the overall components within the final RIA. This includes elements such as database and domain sources, sessions, user databases and so on. The UML deployment diagram [224] seems to be an appropriate visual metaphor to adapt, as this layer describes the composition and distribution of application elements, and does not describe any navigation or behaviour.

5.14.2 Navigation Layer

This layer is concerned with the navigation between different pages, or targets, in the system. It is also concerned with sessions and access control. Within the development of web applications, *sitemaps* are often developed, which are hierarchical trees of content connected through navigation.

A *navigation model*, similar to the UWE navigation model [169], is therefore the ideal visual metaphor for this layer. A navigation model illustrates the intended top-level navigation structures between top-level elements in the application, similar to a sitemap.

5.14.3 User Interface Layer

The contents of a page, on the other hand, are inherently visual in Rich Internet Applications. This layer is oriented most towards visual designers. Consequently, a user interface metaphor is chosen; here, visual elements can be dragged and dropped onto the page. The spatial layout of the contents of the diagram could dictate the final layout of the page[50].

As discussed earlier in Section 5.12, UML does not support any form of user interface modelling except through extensions to the language such as UML*i* [59]. Therefore the user interface visual metaphor used is the metaphor used in user interface builders such as VisualAge for Smalltalk [183].

5.14.4 Domain Modelling Layer

As discussed earlier in Section 5.5, domain modelling in IAML is strongly derived from the UML class diagram model [224] as Domain Types have an inheritance hierarchy and each contain attributes and references to other types. The UML class diagram visual metaphor therefore seems the most appropriate metaphor to reuse as the domain modelling layer visual metaphor in IAML.

5.14.5 Operation Modelling Layer

As discussed earlier in Section 5.7, operation modelling in IAML is strongly derived from the UML activity diagram model [224]; in particular, an Activity Operation contains a number of Activity Nodes connected with others through Execution Edges and Data Flow Edges. Reusing the UML activity diagram visual metaphor as a metaphor for operationa modelling therefore seems appropriate.

5.15 Unification of Type Instantiation and Classification

During the development of the IAML metamodel, a discrepancy was identified between domain instance modelling and visual element instance modelling. For example, a Domain Instance must refer-

[50]For example, the top-to-bottom layout of the visual model should be used to derive the top-to-bottom rendering order of the generated user interface.

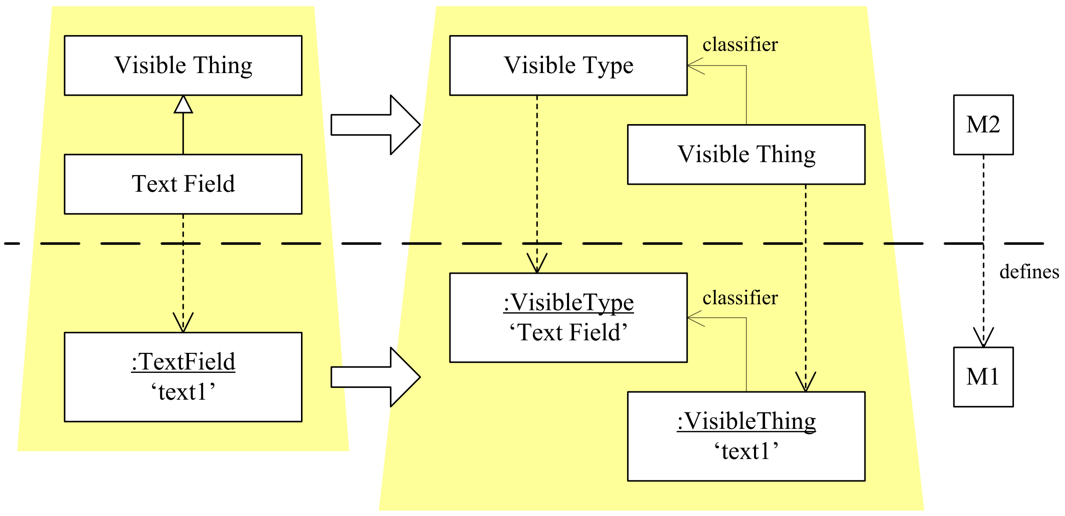


Figure 5.40: The consequences of unifying instance types within the metamodelling architecture of the MDA

ence a Domain Type as a classifier, whereas an Input Text Field has no such classifier reference. The fundamental difference between these two approaches is that a Domain Type *classifies* the Domain Instance, whereas a given text field instance is *defined by* the Input Text Field type.

In terms of the metamodelling architecture of the MDA, this would change the IAML metamodel and the resulting instances of the metamodel as illustrated in Figure 5.40. This refactoring would introduce the following benefits:

1. A Frame no longer has to have separate references to Visible Things and Domain Instances. This means that the instance specification model is more unified. This approach is also more aligned with the underlying infrastructure of UML models [223, pg. 54].

2. Parts of the IAML metamodel can be simplified, as visual element instances such as Input Text Fields and Buttons would be refactored out into the model instance layer. This would mean that a *library* of visual elements could be provided, perhaps automatically.

3. A model instance developer could in the future develop their own Visible Types, allowing IAML to adapt to new standards and innovations in RIAs. For example, the recent acceptance of video as a first-level interface element with HTML 5 [299] could ostensibly be independently defined by a developer as a Visible Type named "Video". As discussed in Section 6.2.1 and Section 6.4.3, EMF metamodels and Xpand templates can already be extended by third parties, so this scenario can already be supported[51].

If these changes were applied to the metamodel mentioned previously in Figure **??**, the resulting metamodel would likely look similar to Figure 5.41. However, this metamodel change would result in the following negative effects:

1. Parts of the IAML metamodel must become more complex. Two additional types, Visible Type and Instance Specification, must be introduced, along with additional references.

[51]The extensibility of the proof-of-concept implementation of IAML is discussed later in Section 9.2.8.

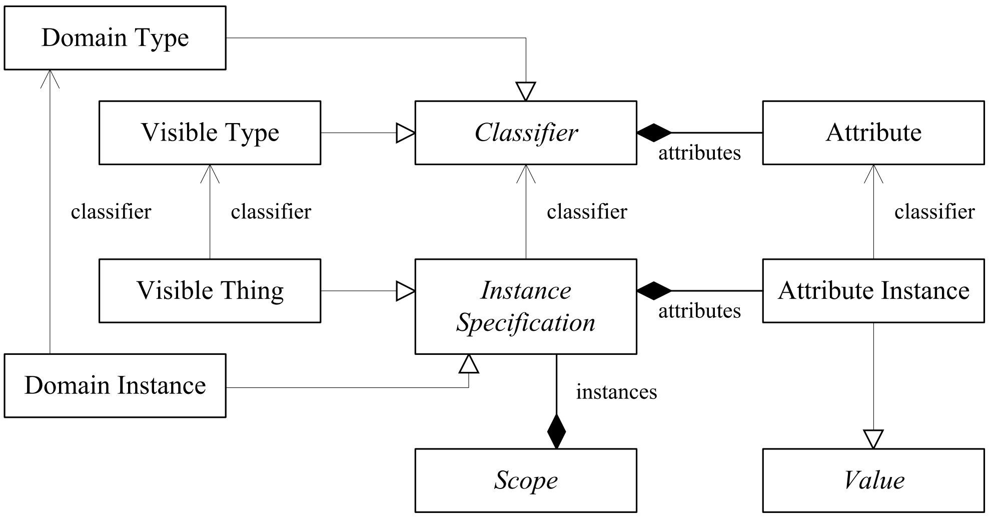


Figure 5.41: A proposed refactoring of the IAML metamodel to unify the instantiation of Domain Instances and Visible Things

2. There is no benefit in moving instances of Visible Types to the model instance layer unless developers can create their *own* instances of Visible Types. In the existing metamodel, different Visible Types have differing event behaviours and user interface appearance. It would therefore be necessary to also support the full definition of the Visible Types' corresponding semantics. This includes:

   (a) Definition of arbitrary events. As discussed in Section 5.3.3, this would require the definition of an event modelling language, which is outside the scope of this thesis.

   (b) Definition of code generation templates. A user-defined user interface element still needs some way to be rendered. This would therefore require the definition of a templating language, which is also outside the scope of this thesis.

   Conversely, no support in defining these semantics for Domain Types was necessary, because all user-defined Domain Types *have the same behaviour and semantics*. That is, the behavioural difference between a “User” and a “Gig” is only the Domain Attributes inside; whereas the behavioural difference between a “Button” and a “Video” is fairly substantial.

   Alternatively, the WebML “black box” approach as discussed earlier in Section 5.12.1 could be used here, where the model developer would define implementations completely independently from the modelling language according to an interface. However, this means that a “Video” element could not define its own events, such as the *onPlay* and *onPause* events of the HTML 5 `video` element [299].

3. It would no longer be possible to distinguish between different interface elements according to their notation. For example, to a model instance developer an instance of a Button would appear identical to a Input Text Field, except for a label describing the classifier. It is not appropriate for a metamodel element to define its own notation – for example, a icon field on Visible Type – as this breaks an underlying MVC model [170].

Finally, it was found that this unification process was not necessary in order to implement the benchmarking application *Ticket 2.0*. This discussion is provided later in this thesis in Chapter 8.

5.15.1 Generalising the Unification of Instance Types

From the previous discussion, it is desirable to extrapolate the underlying reasoning to propose rules on instancing in metamodels. That is, to understand when it is appropriate to have type definitions within the model instance layer, or when to keep it (or move it into) the metamodel layer.

- The definition of a type, such as Domain Type, should reside within the model instance layer when some or all of these criteria are met:
  1. There are none, or few, differences in behaviour between instances of the different types.
  2. It is expected that different model instances will have their own domain-specific instances of the types. It is likely that a developer will have to implement their own instance of the type.
  3. There will be a significant number of unique instances of the type over the lifetime of the language within a suite of model instances of that language.
- Alternatively, the definitions of a range of type instances, such as Visible Things, should reside within the metamodel instance layer when some or all of these criteria are met:
  1. There are substantial differences in behaviour between instances of the different types, that cannot be described within the current language, or the effort necessary to describe and implement these behaviours separately against a modelling language is too great.
  2. It is expected that different model instances will refer to predefined or built-in instances of the type, with third-party instances of the type considered an exception.
  3. There will be very few unique instances of the type over the lifetime of the language within a suite of model instances of that language.

Alone, none of these criteria may be strong enough to warrant the investment necessary to move the instance type definition to another layer. However, they may be useful in the development of new modelling languages that require the definition of types, and the subsequent instantiation of these types. During the development of the IAML metamodel, no such guide was available.

5.15.2 Applying Instance Unification to the IAML Metamodel

Other than Domain Types and Visible Things, there are a number of other elements in the IAML metamodel that may be defined either as user-defined instances of types, or instances of user-defined types. The guidelines introduced in the previous section can be used to make this decision, and in this section this investigation will be performed.

Type Instantiation through Classifiers

With respect to the instance unification guidelines discussed in the previous section, the following IAML concepts should support the definition of both types, and instances of these types, within a single model instance:

1. Domain Types: Different model instances will have their own domain-specific Domain Type instances, and there will be many unique types defined across all model instances. There is no difference in behaviour between instances of different Domain Types.

2. Functions and Predicates: The latest version of XQuery defines 180 different Functions [303], and most web applications will need to also define their own libraries of Functions. It is fairly simple to describe the differences in functionality of different Functions; new instances of Functions can be described using languages such as OCL [222], or within the IAML metamodel itself using Activity Predicates, as discussed in Section 5.7.

Type Instantiation through Instances

Similarly, the following IAML concepts should only support the definition of instances in a single model instance, with the definition of the types of these instances residing in the metamodel definition for that model instance:

1. Visible Things: The behavioural differences between different types of user interface elements are significant, and the effort necessary to support these behaviours is significant. Most web applications will use a common range of user interface elements.

2. Wires: The behaviour of Wires are described with a complex set of model completion rules, as discussed in Section 5.8. As discussed by Wright and Dietrich [320], it is desirable for third parties to define their own model completion rules, however this process is not simple. Allowing Wires to be typed would mean that the IAML metamodel would need to support the elements of model completion directly – in other words, a model completion rule modelling language.

3. Actions: As discussed in Section 5.3.3, there are only two different types of Actions in the IAML metamodel – Operations and Frames for navigation – and it is not clear if there may be any other types of Actions for RIAs.

4. Events: As discussed in Section 5.6, a modelling language that supported the third-party definition of events would need some method for defining the *semantics* of the events themselves. This would therefore require the definition of an event modelling language, which is outside the scope of this thesis.

Type Instantiation through either Classifiers or Instances

There is one IAML concept that may be defined using either of these two instance unification guidelines, as it is not yet clear which approach would be stronger. In the current IAML metamodel, the different types of Scopes are defined as part of the IAML metamodel, as discussed earlier in Section **??**.

1. Scopes: The only behavioural difference between Scopes is the method in which the Properties of that Scope is stored. However, it is not expected that a web application will have a wide range of Scopes, and most developers will simply use the builtin Scopes such as Session and Frame.

Discussion

As discussed earlier in Section 5.1.4, the benchmarking application *Ticket 2.0* may be used to verify the design decisions taken in the development of the IAML metamodel. As discussed later in Chapter 8, this implementation suggests that these design decisions were correct, as no additional types needed to be defined within the language, and all of the types defined were unique to the application itself.

5.16 Conclusion

In this chapter, the complete formal definition of the Internet Application Modelling Language has been provided, including the metamodel structure, constraints operating upon model instances, and the functional semantics of each model element when translated into a generated application. The next chapter will focus on the implementation of this language into a CASE tool using existing model-driven development technologies.

Chapter 6

Implementation Technologies

As discussed earlier in Section 5.1.5, an important design goal of IAML is to provide a proof-of-concept implementation of the modelling language, in order to validate the metamodel. This implementation will be used in the evaluation of the modelling language; serve as a reference implementation for future work; and may increase acceptance of the language by the development community.

A range of technologies – from metamodelling environments to code generators – may be used in such an implementation, and each technology has many existing implementations. In this chapter, each of these technology implementations will be evaluated, and the best technology implementation will be discussed. This combination of selected technologies will form the basis of the proof-of-concept implementation of IAML as discussed in the next chapter.

6.1 Common Comparison Criteria

Each technology in this chapter will initially be evaluated against a suite of common criteria. For example, the execution or runtime environments of a technology can be described and documented; and project quality metrics can be applied if source code is available.

6.1.1 Execution Environment Comparison Properties

In this chapter, the *execution environment* refers to the environment in which the technology is written and executes. This environment can be different from the implementation language; for example, most Eclipse plugins are written in the Java language, but need to be executed within the Eclipse framework [102]. For the proof-of-concept implementation of IAML it would be desirable to select technologies implemented with the same languages and executed within the same environments. The following common properties will be discussed for each technology in this chapter:

- *Language:* What general-purpose programming language is the technology implemented in? To simplify the implementation of a proof-of-concept environment, it would be beneficial to select components that are all implemented within a single language, as no cross-platform integration would be necessary.

- *Runtime Environment:* Applications using the particular technology may need to be integrated within another environment or container at runtime. This additional dependency increases the overhead of using the technology, but may also provide a range of services necessary to simplify an implementation. For programming languages this does not refer to the development environment, but any runtime that compiled instances of the language must rely on.

- *Open Source:* Is the technology available under an open-source license approved by the Open Source Initiative[1]? As discussed earlier in Section 2.7.4, open source software can reduce defects [206], improve functionality and increase developer accessibility when compared to closed source software.
- *Version:* This property simply states the version of the technology evaluated.

Eclipse Framework

Many of the technologies discussed in this chapter are provided as *plugins* to the Eclipse Framework – a software development framework designed to support a wide variety of development languages and approaches through a rich plug-in interface, as discussed by Steinberg et al. [275, pg. 3] and Gamma and Beck [102]. These plugins are implemented as OSGi *bundles* [279] – discussed earlier in Section 4.4.1 – allowing components to be installed and removed at runtime in a flexible manner. All Eclipse plugins therefore have the Eclipse framework as a *runtime environment* dependency.

6.1.2 Open Source Comparison Criteria

Some technologies reviewed in this chapter are provided under open source licenses, and it is therefore appropriate to also evaluate the quality of these technologies according to its open source approach. Two existing frameworks for evaluating open source components are the *Qualification and Selection of Open Source Software* (QSOS) [232] and the *Business Readiness Rating* (OpenBRR) [231] frameworks.

Deprez and Alexandre [63] evaluated both of these frameworks and found that while QSOS provides more extensive rating criteria and supports versioning, OpenBRR provides scoring with less ambiguities and results can be tailored towards a specific context. At the time of writing, however, the progress of OpenBRR appears to have stalled, with no documentation released since 2005[2]. Nevertheless, in this thesis the OpenBRR will be used to form the basis of open source evaluation criteria.

The most recent release of OpenBRR [231] defines 28 metrics categorised into eleven categories, along with a *representative set* of metrics to illustrate the model. The authors argue that these representative metrics should be tailored towards the particular domain. In this thesis, no such tailoring was performed as it was not necessary; the full range of metric values were present through the evaluations, so no reweighting was necessary.

While every category in the OpenBRR ranking system represents an important part of the overall ranking, five of these categories have been selected as particularly important for the proof-of-concept implementation – *quality*, *support*, *documentation* and *adoption*. These categories are those most likely to suggest a simple implementation process, and are described by the OpenBRR standard [231] as follows:

- *Quality:* "Of what quality are the design, the code, and the tests? How complete and error-free are they?"

[1]The Open Source Initiative: `http://www.opensource.org`.

[2]At the time of writing, the OpenBRR website `http://www.openbrr.org` simply states that "we will soon be updating this site with new content."

- *Support:* "How well is the software component supported?"
- *Documentation:* "Of what quality is any documentation for the software?"
- *Adoption:* "How well is the component adopted by community, market, and industry?"
- *Architecture:* "How well is the software architected? How modular, portable, flexible, extensible, open, and easy to integrate is it?" In particular, a system with a high *architecture* score will most likely support third-party extensions.

The categories of *usability*, *performance*, *security* and *stability* are not too relevant for proof-of-concept implementations, as they would not have as much impact as these selected categories. Additionally, the *community* category is very related to the *support* and *adoption* categories, so may be omitted. The individual metric values for each technology is provided in this thesis in Appendix **??**.

In the following sections, the overall *OpenBRR Rating* of each technology implementation will be calculated, which defines a score from a minimum of 28 (unacceptable) to a maximum of 140 (excellent)[3]. An average score of each of the five important categories discussed above will be provided as a numeric value.

Applying OpenBRR to Closed-Source Projects

At first glance, it seems possible to apply the OpenBRR metrics to closed-source projects. For example, metrics such as frequency of releases, types of documentation, and activity on mailing lists can all be derived without needing access to open-source artefacts or methodologies. However, other metrics such as number of bugs, the number of security vulnerabilities and the number of unique contributors simply cannot be derived from a closed-source development methodology. Consequently, closed-source projects are simply given an OpenBRR ranking of *n/a* in the following chapters.

6.2 Metamodelling Environments

A number of metamodelling environments exist in academia and industry to assist the quick development of model-driven platforms and technologies. It is preferable to reuse an existing metamodelling environment over developing an environment from scratch, as a metamodelling environment represents a significant amount of development effort, and it does not seem like a finished environment would be an improvement over existing environments.

There are many existing metamodelling environments as discussed by Fowler [89, pg. 140][4], such as MetaEdit+ [156], the Meta-Programming System (MPS) by JetBrains [148][5], and the Generic Modeling Environment [179]. However, a full evaluation of each of these metamodelling environments is outside the scope of this thesis. In this section, four popular metamodelling environments – initially selected based on their industry support and integration with other technologies – will be investigated and evaluated according to a number of evaluation criteria.

[3]In OpenBRR, each criteria is given an integer ranking between 1 (unacceptable) to 5 (excellent) [231]. A future standard of OpenBRR may benefit in changing the scale to use 0 (unacceptable) to 4 (excellent), as this can make comparisons simpler.

[4]Fowler uses the term "language workbench" to describe a metamodelling environment.

[5]The first public release of JetBrains MPS was released in 2009, and as such was not an option when this research was started.

This investigation is particularly important since the metamodel forms the “core” of the resulting development environment, and the choice of metamodelling environment will affect other aspects of the implementation, such as code generation and verification technologies.

6.2.1 Eclipse Modeling Framework

The *Eclipse Modeling Framework* (EMF) is a Java-based meta-modelling suite for developing metamodels within the Eclipse framework [275]. It is well supported by the Eclipse foundation, and is used as the meta-modelling technology behind a number of other Eclipse-based projects, such as GMF. EMF focuses on the serialisation and functionality of the metamodel, and does not provide a graphical editor for model instances natively; instead, the GEF and GMF components discussed later in Section 6.3 can be used to provide a graphical editor [119].

EMF exists as part of the extensive *Eclipse Modeling Project*, which (at the time of writing) consisted of eleven sub-projects and over 130 components[6]. These sub-projects include graphical modelling, model transformations and model verification. Furthermore, as part of the Eclipse project, integration with other Eclipse technologies – such as version control, development environments and project management – is fairly straightforward.

Meta-models may be loaded from a diverse range of sources [275], such as Ecore model instances created by the EMF model editor; annotated Java source code; XMI models [220] exported from UML CASE tools, such as Rational Rose or ArgoUML [287]; XML Schemas [295]; or metamodels created dynamically at runtime. Ecore model instances can also be created graphically through the *Ecore Tools* graphical editor. Verification is supported by the *EMF Validation* framework, and constraints can be defined using a number of languages, including OCL constraints [222], XML Schema constraints, and manually-implemented Java code generated through scaffolding [275].

A loaded metamodel can then be used to generate the necessary Java scaffolding source code of the corresponding model, using code generation written according to Java Emitter Templates (JET). The generated Java scaffolding may be modified manually, with the generator controlled by a `@generated` annotation[7].

Alternatively, the functionality of the metamodel can be provided through *Dynamic EMF*, which is a simple interpretive implementation of EMF designed for sharing simple objects [275, pg. 36–38]. EMF also supports the extension of existing metamodels by allowing additional metamodels (resources) to be loaded at runtime.

Once created, this scaffolding can be used to create new model instances as simply as creating new Java objects. The EMF framework deals with all the necessary logic behind the implementation, natively supporting serialisation to XMI [220]. EMF metamodels are usually executed within an Eclipse environment, but may be used in a standalone environment.

The underlying meta-metamodel for EMF metamodels is the Ecore metamodel, which is defined in terms of itself. This self-definition allows for Ecore to reside at the M3 layer of the metamodelling architecture of MDA. As discussed earlier in Section 3.1.5, the Eclipse Modeling Framework was built on EMOF [**?**] and model instances can be serialised directly to EMOF [275, pg. 40].

[6]This component count was calculated by getting all non-release components from Bugzilla: `https://bugs.eclipse.org/bugs/query.cgi?classification=Modeling&query_format=advanced`.

[7]When an EMF-generated class is regenerated, only fields and methods with an unmodified `@generated` annotation may be modified or removed by the generator [275, pg. 305–308]; this approach follows the code generation principles proposed by Fowler [89, pg. 126].

```
<eClassifiers xsi:type="ecore:EClass" name="Value">
  <eAnnotations source="http://www.eclipse.org/emf/2002/GenModel">
    <details key="documentation" value="Represents a single value,
        accessible and modifiable at runtime."/>
  </eAnnotations>
  [...]
</eClassifiers>
```

Listing 4: EAnnotation documentation in Ecore

EMF models also support model annotations using the EAnnotation model element. These annotations are particularly helpful when providing metamodel documentation, as illustrated in Listing 4[8]. If necessary, these annotations can be stored and accessed at runtime [275].

6.2.2 ArgoUML

ArgoUML is a free, open-source modelling tool written in Java which was originally developed as a UML diagram editor [251]. While it was originally designed to only support the development of UML diagrams, it has since been extended to support UML Profiles, and thus is a form of a metamodelling environment. It has been used to implement other model-based prototypes, such as the discontinued *ArgoUWE* editor for UWE [165], and *ARGOi* for UML*i* [58, 59].

The ArgoUML meta-model is based on all UML 1.4 diagrams [219]; and as such, software based on ArgoUML also needs to describe their meta-model in terms of UML 1.4 concepts. Internally, model instances are instances of JavaBeans [251], and can be serialised to and loaded from XMI. At the time of writing, ArgoUML had no plans for supporting recent revisions of UML such as UML 2.0 [224].

Custom metamodels are defined in ArgoUML by implementing UML Profiles [165], which may be implemented visually using the graphical editor of ArgoUML itself and loaded through XMI files. However this approach is extremely limited, and profiles are usually supplied as Java class instances instead[9]. These are then provided to ArgoUML at runtime through additional JAR files, known as *modules*. These are loaded at startup, with the architecture of these modules following the *Dynamic Linkage* design pattern [115]. As ArgoUML metamodels are Java class instances, it is not possible to extend these profiles arbitrarily, hindering metamodel extensibility.

One benefit of using ArgoUML is that it provides a free graphical CASE tool to any new models, which will be discussed later in Section **??**; in fact, it is only possible to create model instances graphically in ArgoUML, as illustrated in Figure 6.1. Model instance verification is supported through the concept of *critics* – implemented as Java class instances – which are used to highlight potential problems in a model instance [251]. ArgoUML provides the Dresden OCL toolkit [**?**] to implement OCL constraints, but as profiles are designed to be implemented through Java, OCL constraints for a particular profile must be integrated manually.

ArgoUML is best suited for developing graphical editors for metamodels entirely defined within UML 1.4, as any functionality not directly supported by a UML Profile must be implemented manually. The fact that ArgoUML does not provide easy integration with other model-driven technologies, such as code generation or model completion, consequently suggests that developing a proof-

[8]**TODO:** Ensure that this documentation is correct once Appendix **??** is completed.

[9]UML Profiles in ArgoUML are represented as instances of the `org.argouml.profile.Profile` abstract class.

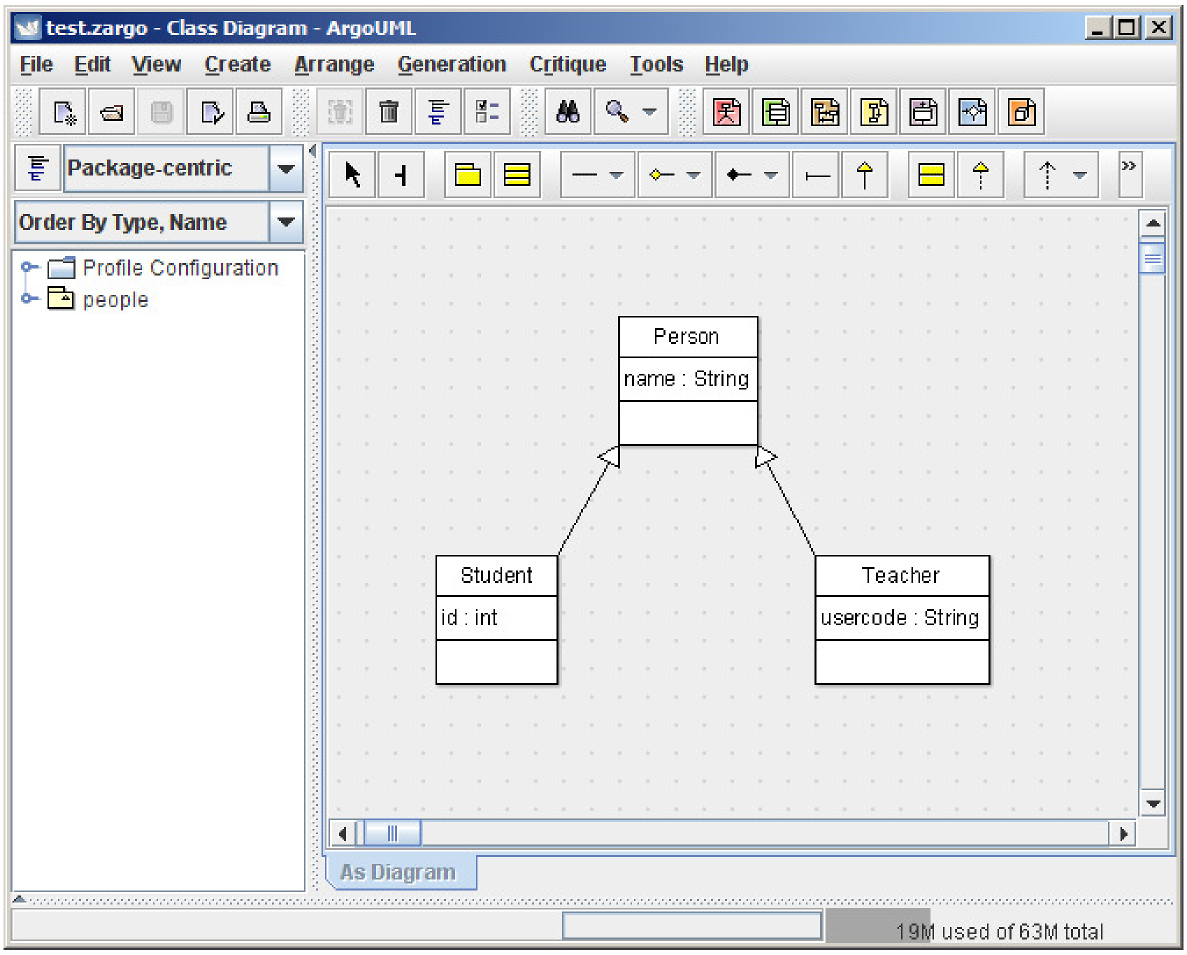


Figure 6.1: Using ArgoUML to define a UML class diagram

of-concept implementation of IAML using ArgoUML would be very difficult and require significant development effort.

6.2.3 Whitehorse

In Visual Studio 2005, Microsoft introduced a model-driven environment for developing applications. This framework was originally codenamed *Whitehorse* [52] and released under the name *DSL Tools*, but is currently released under the name of the *Visualization and Modeling SDK* (VMSDK). The environment is implemented with, and generates to, .Net-based languages such as C#.

Whitehorse provides a graphical interface for developing the structure of domain-specific languages, as well as visualising the created model instances of the developed language. The generated language supports model verification only in terms of a validation framework[10], and constraints must be written in .Net; there is no support for OCL constraints.

In Whitehorse, a DSL can only be defined using the provided DSL editor. The *Managed Extensibility Framework* can be used to extend existing metamodels, however this extensibility is not provided automatically and there are a number of files that must be created manually [204].

The underlying meta-metamodel of Whitehorse-developed metamodels is the *Domain Model Framework* (DMF) [52] – a technology that fulfills a similar purpose as EMF, however is not related to any

[10]Domain-specific language verification uses the *Microsoft.VisualStudio.Modeling.Validation* namespace.

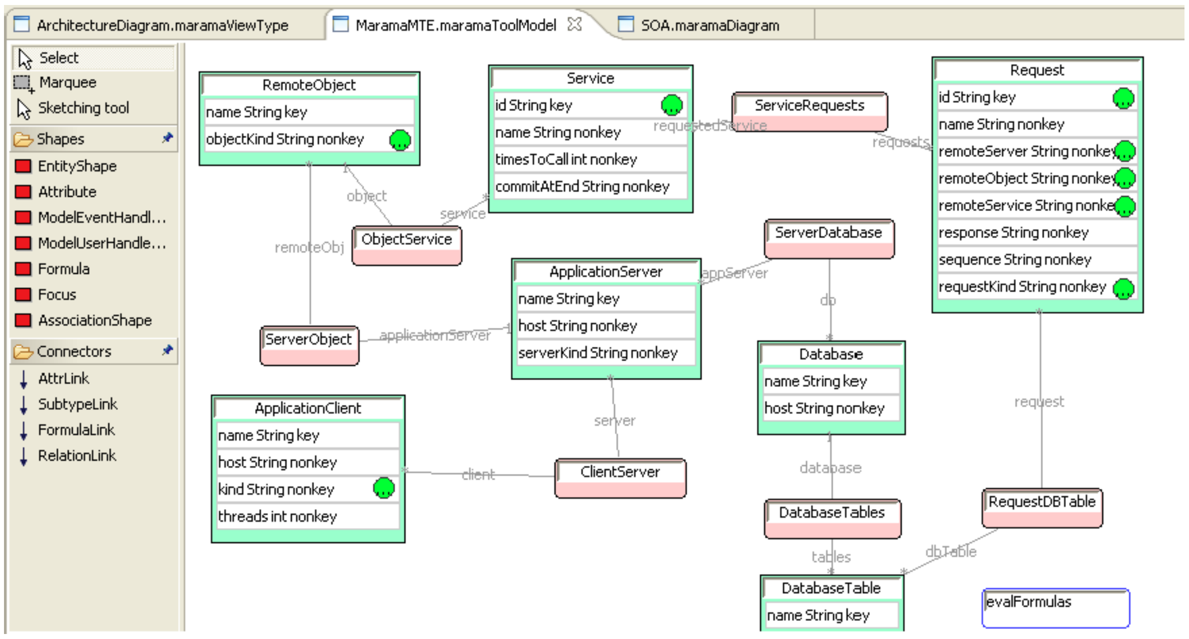


Figure 6.2: Defining a metamodel instance using Marama, adapted from Grundy et al. [122]

version of MOF at all. Similarly, the meta-metamodel behind DMF – which in this thesis is tentatively named the *DslDefinitionModel*[11] – does not appear to be defined in terms of itself. This step is necessary for a meta-metamodel to be a valid M3 layer of the Model-Driven Architecture, as discussed earlier in Section 3.1.5.

Since Whitehorse was announced, there appears to have been very little academic or industry uptake of these technologies. However, the recent release of Visual Studio 2010 [**?**] has reorganised the DSL software development kit into the *Visual Studio Visualization and Modeling SDK*. This new release supports the *ModelBus* model integration framework [30], and supports the serialisation of model instances into the XMI format. Previously, model instances could only be serialised using a proprietary XML-based format.

In this thesis, Whitehorse is the only metamodelling environment evaluated that is closed-source, and the metamodelling environment can only be used if a valid Visual Studio license is available. A modelling environment generated through Whitehorse still requires the Visual Studio IDE to execute, and thus end users also require a licensed copy of Visual Studio to host the environment.

6.2.4 Marama

The *Marama* project started off as an application to improve the quality of meta-modelling tools generated by the *Pounamu* project [324]. Pounamu was originally a project to specify and generate independent visual modelling tools, and Marama was used to generate an Eclipse-based editor instead [120]; Pounamu now appears to be discontinued.

The Marama project is implemented within the Eclipse environment using the Java language, and generated modelling environments are hosted within the Eclipse environment. The project is based heavily on the EMF and GEF projects discussed in this chapter, and each modelling concept has an associated metamodel. Once a metamodel has been defined, model instances can be created visually

[11]The XML namespace used for metamodel instances defined using the *Whitehorse* development tools is `http://schemas.microsoft.com/VisualStudio/2005/DslTools/DslDefinitionModel`.

using the graphical environment discussed later in Section 6.3.5.

Marama defines its own meta-metamodel for the definition of metamodels, which is referred to in this thesis as *MaramaModelProject* (MMProject), and is adopted from an extended Entity-Relationship paradigm [185]. However, this meta-metamodel is not defined in terms of itself; rather, the meta-metamodel is defined in terms of an Ecore model instance[12] [79]. There is no explicit support for the extensibility of existing metamodels defined within Marama.

Metamodel instances can be defined visually as in Figure 6.2. The *MaramaTorua* project [138] supports the definition of Marama metamodels from XML Schemas or the Microsoft *schema inference engine*, but this is not provided as part of the Marama project itself. Constraints (known as *Formulae* in Marama) can be supplied either as textual OCL constraints, or the OCL constraints can be constructed visually using *MaramaTatau* [187]; user-defined extensions for these constraints can also be provided in Java [185].

6.2.5 Summary

A summary of these modelling environments is provided in Table 6.1. Along with the common comparison criteria discussed earlier in Section 6.1.1 and Section 6.1.2, each environment was evaluated on a number of additional criteria:

- *Visual Definition:* Can a metamodel be implemented graphically? For a metamodelling environment that is intended to support the definition of graphical modelling languages, this feature may be essentially provided "for free" by the environment itself.
- *Graphical Editor:* Does the metamodelling environment provide a graphical interface for interacting with a developed model instance, and is this interface the intended means of end-user interaction? This is not strictly beneficial, as the graphical representation of a model instance should be kept separate from the underlying model instance representation.
- *Meta-model Extensibility:* If a developer creates an instance of a metamodel, can this metamodel later be extended by a third party? That is, can a third party add additional elements and constraints to an existing metamodel? Metamodel extensibility allows a third party to extend the modelling environment or adapt it to a particular domain, and is a driving force behind UML Profiles [100].
- *Meta-metamodel:* What is the underlying meta-metamodel of the developed meta-models? That is, what language or technology is used to represent instances of metamodels used in the modelling environment?
- *MDA-compliant M3:* Is the meta-metamodel compliant to the MDA requirement for a meta-metamodel [163, pg. 88]; that is, has the meta-metamodel been defined in terms of itself?
- *Meta-model Sources:* What kinds of sources may be used to develop a metamodel instance? Being able to import a metamodel from an existing source may simplify the development of the modelling environment.
- *Verification Languages:* Which model instance verification languages does the metamodelling environment natively support, if any? In many cases, the metamodelling environment may be extended with additional languages as discussed later in Chapter **??**.

[12]The *MaramaModelProject* meta-metamodel instance is provided in `project.ecore`.

Requirement	EMF	ArgoUML	Whitehorse	Marama
Language	Java	Java	.Net	Java
Runtime Environment	Standalone	ArgoUML	Visual Studio	Eclipse
Open Source	✓(EPL)	✓(New BSD)	✗	✓(MPL)
Version	2.5	0.28.1	2010	20101022
Visual Definition	✓	✓	✓	✓
Graphical Editor	✗ (GMF)	✓	✓	✓
Meta-model Extensibility	✓	✗	✓	✗
Meta-metamodel	Ecore	UML	DMF	MMProject
MDA-compliant M3	✓	✓	✗	✗
Meta-model Sources	Ecore Annotations UML EMOF XML Schema Dynamic	UML Profiles JARs	DSL Editor	Marama XML Schema Microsoft WS
Verification Languages	Java OCL XML Schema	Java OCL	.Net	Java OCL
OpenBRR Rating	120	83	n/a	60
Average Quality	3.5	4.2		2.5
Average Support	4.5	1.0		1.0
Average Documentation	5.0	2.0		2.0
Average Adoption	4.0	2.0		3.0
Average Architecture	4.0	5.0		1.0

Table 6.1: Comparison of Meta-Modelling Environments

From the evaluation of existing meta-modelling environments, the Eclipse Modeling Framework was found to be the most suitable for the implementation of IAML, due to its openness, industry support, documentation, and easy integration with the Eclipse environment. Marama is very promising, but without the same level of industry support as the EMF project and high quality documentation it may be difficult to adapt the toolset to new environments and situations.

6.3 Graphical Modelling

Graphical modelling provides an environment where an underlying model instance, as described by a metamodel, may be represented as a visual model; this is achieved through the association of shapes and relationships to the metamodel element types, through some form of mapping. This section will evaluate a range of existing graphical modelling environments, with the intent of reusing one of these environments rather than having to develop one from scratch.

A common design pattern found with model-driven graphical modelling environments is that this graphical-to-metamodel mapping process is decomposed into smaller model instances which are later combined. For example, the design of the shapes, and the association of these shapes to model elements, can be described as a "gallery" metamodel to support common shapes across multiple editors

[119, pg. 61].

6.3.1 Graphical Editor Framework

The *Graphical Editor Framework* (GEF) [119] is an Eclipse-based framework which aims to assist in the development of rich graphical editors and user interface views. It extends on its underlying technologies, Draw2D and SWT, to provide a higher-level interface using concepts such as shapes and connections, and also provides a "toolbox" interface that can be used to create new shapes. GEF acts as the controller between an underlying model and the interface, following the Model-View-Controller (MVC) approach [170].

GEF is designed to operate against some type of model, which does not need to be any particular metamodelling framework; in most cases, this model is a set of custom Java classes. GEF is therefore easy to integrate with EMF, as EMF can be used to generate custom Java classes [6]. The framework is used as the basis for a number of other graphical environments, such as GMF and Marama which are both discussed later in this section.

Shapes in GEF are instances of Java classes, and GEF does not provide a metamodel for describing these shapes; this is in contrast to the *gmfgraph* metamodel approach provided by GMF as discussed in the next section. Because the underlying model source is simply Java classes, there is also no notation metamodel.

GEF is therefore not used to generate graphical editors, in contrast to how many of the other technologies in this section are designed; rather, it exists as a runtime wrapper around existing Java software and different forms of model instances. This means that GEF's extensibility relies on the developer manually extending the GEF framework directly, as opposed to supplying the framework with different configuration inputs.

6.3.2 Graphical Modeling Framework

The *Graphical Modeling Framework* (GMF) [119] aims to extend the functionality offered by EMF and GEF into a higher-level graphical editor framework. GMF abstracts the functionality of graphical editors into four metamodels – *gmftool* for the palette tooling definition, *gmfgraph* for the graphical shape definitions, *gmfmap* for the mapping of shapes to model elements, and *gmfgen* for the generation of the editor itself – and provides a higher-level view of a graphical editor than the underlying technologies.

The developer of a graphical editor is given the task of implementing model instances according to these metamodels, each of which are implemented and serialised using EMF. GMF can then compile these definitions into a finished graphical editor, with each editor provided as an Eclipse plugin. To assist in the development of these metamodels, GMF provides automated tools and wizards to simplify the development of these model instances. A screenshot of the *GMF dashboard wizard* is provided in Figure 6.3; this wizard highlights the steps necessary to develop a GMF-based graphical editor.

The translation of these graphical model instances into other model instances is achieved with JET, and the generation of the Java source code for the final diagram editors is implemented using Xpand; both of these technologies are discussed later in Section 6.4. These templates are extensible using *dynamic templates*, which allow the editor developer to customise the generated editors without modifying the GMF framework directly; the implementation of dynamic templates in an editor is discussed later in Section 7.4.2.

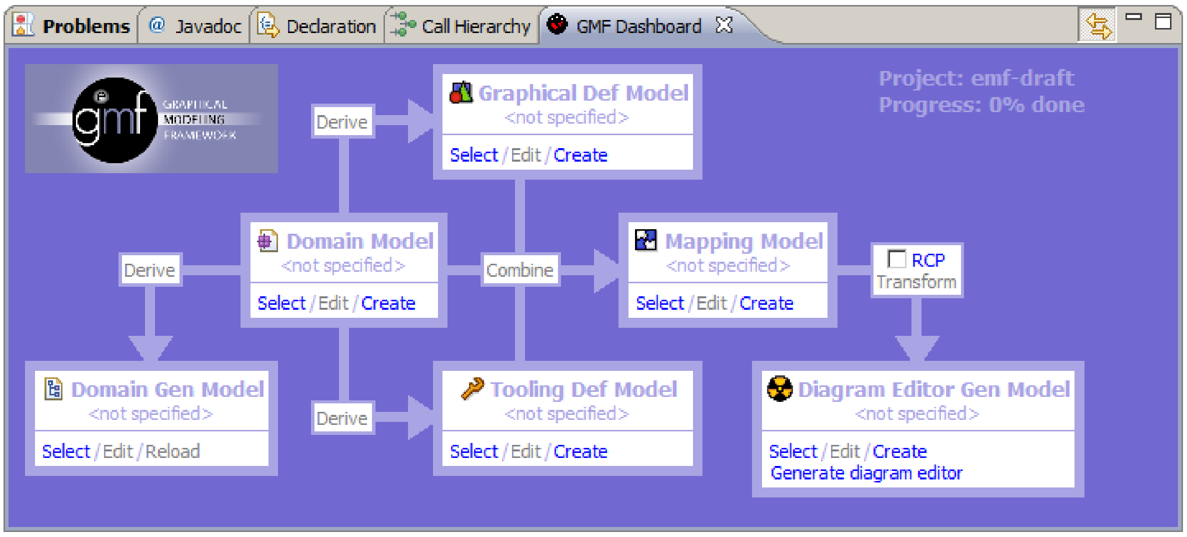


Figure 6.3: The GMF Framework Wizard in Eclipse, illustrating the steps necessary to create a graphical editor

GMF uses the *GMF Notation* metamodel for annotating model instances with the information necessary to visually represent the model instance. By default, the model instance and the associated visual representation model instance are stored in separate resources[13]. Each of these metamodels are implemented as an Ecore metamodel under the EMF framework, as discussed earlier in Section 6.2.1.

6.3.3 ArgoUML

As discussed earlier in Section 6.2.2, ArgoUML was originally developed as a UML diagram editor, which has since been used as a graphical editor for other UML-based models, such as ArgoUWE for UWE. However, since ArgoUML is designed to edit UML diagrams only, it is unlikely that it can be used to edit instances of metamodels that are not related to UML.

The underlying diagram technology for UML is a custom-built *Graph Editing Framework* (GEF*[14]) [251]. GEF* is published as a standalone project under the open source BSD license.

Model element shapes in ArgoUML are not defined according to any metamodel, but are Java instances of GEF classes that are created manually[15]. The underlying model instance and the visual representation are still represented separately; the visual representation is serialised using the *Precision Graphics Markup Language* (PGML) [251], which itself is defined using a DTD [289].

As discussed earlier in Section 6.2.2, extensions of ArgoUML rely on defining *modules* and *plug-ins* [281]. There is no generation stage, and thus there is no graphical editor generator framework that can be extended. However, CASE tools that are based on ArgoUML automatically obtain the notation and layout functionality of the framework, and thus ArgoUML can be considered high-level.

[13]For example, a UML model instance could be stored in `instance.uml`, and the visual representation would be stored in `instance.uml_diagram`

[14]Since the Graphical Editor Framework discussed earlier also uses the GEF initialism, the Graph Editing Framework will instead be referred to in this thesis as GEF*.

[15]For example, the UML model element *FinalState* is implemented by the Java class `org.argouml.uml.diagram.state.ui.FigFinalState`.

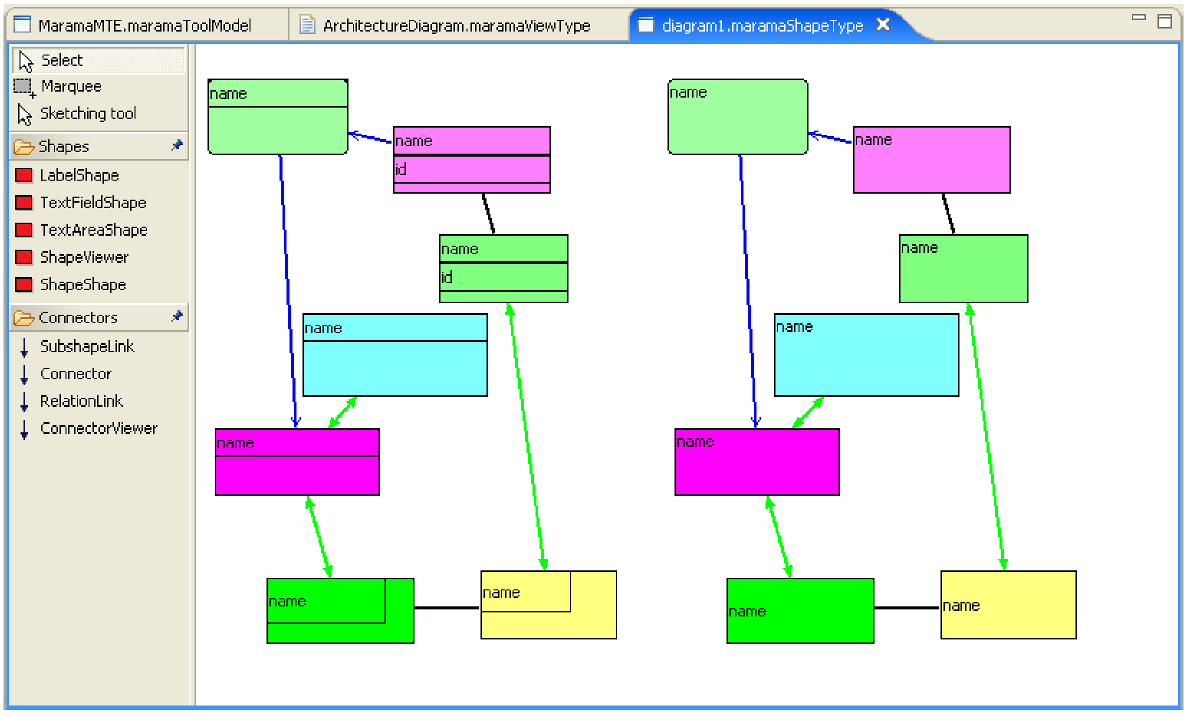


Figure 6.4: The *Shape Designer* of Marama showing the shape design and corresponding concrete view, from Grundy et al. [122]

6.3.4 Whitehorse

As discussed earlier in Section 6.2.3, the *Whitehorse* project was introduced as a way of developing graphical editors for domain-specific languages. The underlying environments of the Whitehorse DSL technologies are the Design Surface Framework (similar to GEF) and the Domain Model Framework (similar to EMF) [52]. Both of these technologies form part of the Visual Studio DSL SDK, and are not independent projects in the vein of GEF and EMF.

The shapes of model elements are represented according to the *DslDesignerDiagram* (DDiagram) metamodel[16]. The diagram model instance is *project-specific* and implies that there is no common shared notation metamodel between different Whitehorse projects; however the underlying model instance and the visual representation are still stored separately. These shapes directly associate with model elements in the underlying metamodel, so it is not possible to define a shared gallery of model element shapes.

When developing a graphical editor using Whitehorse technology, the developer does not need to concern themselves with concepts such as layout or printing, and thus can be considered a high-level framework. The graphical component of Whitehorse does not directly support extensible templates, and changes to the graphical editor must be performed manually in the .Net language.

6.3.5 Marama

Discussed earlier in Section 6.2.4, the *Marama* project is designed to provide all the aspects necessary to develop a graphical model-driven environment. The key difference between Marama and GMF is that the associations between the metamodel and the graphical environment, as well as the constraint

[16]No XML namespace is provided for this metamodel, but the .Net namespace for this metamodel is defined as `microsoft.VisualStudio.Modeling.DslDesigner.DslDesignerDiagram`.

Requirement	**GEF**	**GMF**	**ArgoUML**	**Whitehorse**	**Marama**
Language	Java	Java	Java	.Net	Java
Runtime Environment	Eclipse	Eclipse	ArgoUML	Visual Studio	Eclipse
Open Source	✓(EPL)	✓(EPL)	✓(New BSD)	✗	✓(MPL)
Version	3.5.2	2.2	0.28.1	2010	20101022
Diagram Technology	Draw2D	GEF	GEF*	DSF	GEF
Model Source	Java	EMF	JavaBeans	DMF	EMF
Shape Metamodel	-	gmfgraph	-	DDiagram	Pounamu
Notation Metamodel	n/a	Notation	PGML	Custom	MDiagram
Automatic Layout	✓	✓	✓	✓	✓[?]
Printing Support	✗	✓	✓	✓	✗
Export Diagrams	✗	✓	✓	✓	✓
Extensible Generator	✗	✓	✗	✗	✗
OpenBRR Rating	99	98	83	n/a	60
Average Quality	3.7	2.5	4.2		2.5
Average Support	3.5	4.0	1.0		1.0
Average Documentation	4.5	4.0	2.0		2.0
Average Adoption	4.0	3.5	2.0		3.0
Average Architecture	1.5	4.0	5.0		1.0

Table 6.2: Comparison of Graphical Environments

definitions, are all intended to be implemented visually. For example, the Marama *shape designer* in Figure 6.4 adapted from Grundy et al. [122] allows for the graphical definition of a shape; this shape can then be used by the Marama *view definer* to associate metamodel elements with defined shapes.

The Marama project was started before the GMF framework had been released, and the project had to manually implement much of the functionality provided by GMF, such as validation decorators, complex shapes and automatic layout algorithms [?]. At the time of writing, the primary source of Marama documentation was its user manual [185]. The underlying metamodel for defining shapes, tooling and views in Marama is Pounamu, and this metamodel is represented in terms of a collection of DTD instances. The diagram notation metamodel *MaramaDiagram*[17] (MDiagram) associates shapes to model element instances for a given model instance, and is represented in terms of an Ecore model instance [79].

The implementation of Marama through a combination of DTD and Ecore model instances highlights a strength of model-driven approaches, in that many different model sources can be integrated together. However, it may be preferable in the future to migrate the shape metamodel to an Ecore representation, as DTDs are not M3-compliant in the MDA, as discussed earlier in Section 3.1.5; Ecore metamodels can provide stronger structural constraints than DTDs; and Eclipse-based tooling support for Ecore is much stronger than those for DTDs.

6.3.6 Summary

A summary of these graphical environments is provided in Table 6.2. Along with the common comparison criteria discussed earlier in Section 6.1.1 and Section 6.1.2, each environment was evaluated

[17]This name is derived from the root element in the Marama `diagram.ecore` metamodel.

on a number of additional criteria:

- *Diagram Technology:* What is the underlying technology used for *rendering* the graphical interface – that is, the diagram editor – of the generated tool?

- *Model Source:* What is the underlying technology used for representing the model instances that are being modified by the graphical editor?

- *Shape Metamodel:* Is there a metamodel defined for describing the shapes, or figures, of the graphical editor with an abstracted model? As discussed at the beginning of this section, model-driven graphical modelling environments will often have a separate metamodel for these shapes in order to support a reusable gallery of shapes.

- *Notation Metamodel:* When a model instance is created graphically, there needs to be some means of storing the diagrammatic elements of the visual representation, such as positions, sizes, fonts and colours. Is the graphical notation instance data stored as a model instance with a corresponding separate metamodel? That is, is this graphical environment implemented according to a model-driven development approach? If a separate metamodel is defined, the represented model instance does not have to be polluted with notation information.

- *Automatic Layout:* Does this environment provide built-in support for automatically laying out diagrams? As discussed by [**?**], automatic layout algorithms can vastly improve the usability and efficiency of creating visual model instances, but the implementation of these algorithms can be difficult and would ideally be provided as part of the graphical environment.

- *Printing Support:* Does this environment provide built-in support for printing diagrams via a printer?

- *Export Diagrams:* Does this environment provide built-in support for exporting diagrams to image files? If the environment supports sending images to a printer, then implementation of this feature should be straightforward.

- *Extensible Generator:* Is this graphical environment intended to be extended by developers? Can the editor developer interact with or modify the logic and templates behind the generation of the final editor? This extensibility can allow an editor developer to add functionality without having to modify generated code.

With all of these factors considered, and also considering the earlier decision in Section **??** of using EMF as a meta-modelling environment, the GMF framework within Eclipse was found to be the most suitable environment for the development of a graphical editor for IAML.

While Marama is very similar to GMF, there are two major factors that influenced the decision to use GMF instead: firstly, as an Eclipse Foundation project, GMF has more use in industry and a higher quality of documentation, and this is reflected in the OpenBRR evaluations. Secondly, GMF generated editors are designed to be extensible, which will simplify the implementation of custom extensions (such as model completion, discussed later in Section 6.5).

6.4 Code Generation

Another design goal of the proof-of-concept implementation of IAML is the implementation of a code generator to translate IAML model instances into executable web applications, forming an important part of the evaluation in Chapter 8. Code generation is a model transformation which translates a platform-independent model at a higher level of abstraction, into platform-specific models at lower levels of abstraction, as discussed earlier in Section 3.1.7.

As with the rest of this chapter, there are a number of existing technologies for implementing model-driven code generation, and this section will be used to discuss and evaluate these technologies for suitability. As our previous two technologies are both heavily dependent on the Eclipse environment, it must also be easy to integrate a selected code generation technology into an Eclipse environment. This means that other environments such as Visual Studio and ArgoUML will not be considered.

6.4.1 XSLT

As discussed earlier in Section 3.1.7, *Extensible Stylesheet Language Transformations* (XSLT) [297] can be used to translate XML instances according to some simple translation rules. Since EMF models are often represented as XML files, and can otherwise be exported into XMI, it would be possible to define XSLT files to translate a given IAML model instance into a web application. A model instance can be navigated using the XPath language [301], and queried using the XQuery language [302].

The XSLT approach may be executed according to an underlying metamodel if the source metamodel is first translated to an XML Schema instance – for example, translating an XMI-serialised model valid to an Ecore metamodel, into an XML model valid to an XML Schema. With regards to EMF-based metamodels, XML Schema is less expressive than Ecore; some metamodel information cannot be translated to XML Schema constructs, such as bidirectional references or target reference types [275, pg. 179–235].

XSLT instances are serialised using the general-purpose machine-readable language XML, and are consequently more verbose than domain-specific textual languages. This makes it difficult to develop stylesheet instances without adequate tool support. XML stylesheets do not directly support third-party extensibility, as all component stylesheets must be *explicitly* included and exist at the time of compilation.

Since XSLT is simply a specification, there are many XML *template engine* implementations for a number of languages, and stylesheets can be executed within any runtime environment. Providing an OpenBRR evaluation against a range of these template engines is outside the scope of this thesis, and no such evaluation will be provided. As discussed by Gottlob et al., four popular existing engines include XALAN, Saxon, XT and IE6 [114].

6.4.2 JET

Java Emitter Templates (JET) [119], part of the Eclipse Model to Text (M2T) project, are a template-based code generation technique. The syntax of JET templates are very similar to Java Server Pages (JSP) [139], as illustrated in Listing 5; in fact, the template engine was built using source code from the JSP implementation *Tomcat* [275, pg. 375]. JET may be used within an Eclipse environment, but this environment is not necessary; JET may be used in a standalone environment, as the only runtime requirement is EMF, which may also be provided in a standalone fashion.

```
<%@ jet imports="java.util.* org.openiaml.model.model.*" %>
<%@ include file="copyright.javajetinc" %>
<% InternetApplication root = (InternetApplication) argument; %>
<html><body>
<h1>List of Pages</h1>
<ul>
<% for (Iterator i = root.getScopes().iterator(); i.hasNext(); ) {
  Scope s = (Scope) i.next();
  if (s instanceof Frame) { %>
    <li><a href="<%=((Frame) s).getUrl()%>"><%=s.getName()%></a></li>
<%    }
} %>
</ul>
</body></html>
```

Listing 5: Internet Application code generation using a JET template

JET templates are evaluated against instances of plain old Java objects (POJOs) [149] – allowing them to interact directly with Java code – and can also be used to iterate over XML data. Consequently, Java is both the navigation and query languages used in JET. JET templates also can import existing libraries of templates, known as *taglibs*, to extend the functionality of the code generator.

JET templates can also be extended by third parties through runtime `@include` directives. To generate Java source code from metamodels, the Eclipse Modeling Framework uses JET templates in this fashion, and these generated source codes may be extended by third parties [275, pg. 379][18].

6.4.3 openArchitectureWare

The *openArchitectureWare* project (OAW) began as an independent implementation of various model-driven technologies [75]. This included a code generator (*OAW Xpand*[19]), a textual model defintion language (*Xtext*), a model extensibility framework (*Xtend*) and an OCL-like suite of model checks (*Checks*). Each of these technologies are written in Java, and require the Eclipse environment at runtime. These technologies are internally based on EMF model instances, and allows code generation templates to be developed using an underlying metamodel.

OAW Xpand uses a custom language for defining templates, as in Listing 6; the expression sub-language is "a syntactical mixture of Java and OCL" [75], which can also be used to navigate through the model instance separately from Xpand. OAW Xpand supports the concept of *dynamic templates*, which supports a form of aspect-oriented programming. In particular, an existing template can be overridden or extended by a new third party template.

Since this research was started, all development focus on openArchitectureWare has essentially ceased, and the four subprojects have moved into new Eclipse Foundation projects. Consequently, the OpenBRR evaluation of OAW Xpand is not as positive as the evaluations of the other code generation technologies, as there has been no recent development activity.

[18]Within the proof-of-concept implementation of IAML, the JET templates used by EMF are extended to enable XSD references, as discussed by resolved issue 251: *XSD References cause GMF editors to crash with transaction exception*.

[19]In this thesis, the openArchitectureWare implementation of Xpand is named *OAW Xpand* in order to distinguish this implementation from the Eclipse implementation, which is simply named *Xpand* as in the next section.

```
«IMPORT iaml»
«EXTENSION template::GeneratorExtensions»

«DEFINE root FOR model::InternetApplication»
«EXPAND Copyright::copyrightTemplate»
<html><body>
<h1>List of Pages</h1>
<ul>
«FOREACH scopes.typeSelect(model::visual::Frame) AS frame»
  <li><a href="«frame.url»">«frame.name»</a></li>
«ENDFOREACH»
«ENDDEFINE»
```

Listing 6: Internet Application code generation using an OAW Xpand Template

6.4.4 Xpand

As another part of the Eclipse M2T project, *Xpand* [119] started as an Eclipse-branded branch of the *OAW Xpand* component of openArchitectureWare. It appears that the branch was started to satisfy the model-to-text code generation requirement of GMF, as the GMF code generation templates to generate graphical editors could be extended by third parties to add additional functionality to the generated graphical editors.

Since its integration into Eclipse, Xpand has seen a number of improvements, particularly in terms of standardisation, functionality and performance. With the release of GMF 2.2, Xpand was extended to support QVTo model queries [45]. An improved code editor has also been released, along with significant performance improvements. The underlying expression remains a mix of Java and OCL, as discussed in the previous section. As part of the Eclipse Foundation-supported ecosystem, Xpand is still under active development with a wide range of support options.

6.4.5 Summary

A summary of this investigation is provided in Table 6.3. As discussed earlier in Section 3.1.8, QVT and ATL are omitted as neither technology supports *model-to-text* transformations, which is a requirement for code generation. Along with the common comparison criteria discussed earlier in Section 6.1.1 and Section 6.1.2, each technology was evaluated on a number of additional factors:

1. *Navigation Language:* Within a template, what language is used to navigate through a model instance? That is, what language is used to browse through the model instance graph?

2. *Query Language:* Within a template, what language is used to query a model instance within expressions? That is, what language is used to translate a model instance into an expression? If these two languages are different, each language may be simpler and easier to test, but there may be more of a learning curve for new developers.

3. *Underlying Metamodel:* What is the underlying metamodel for the code generation templates? The increased expressibility and enforced structure of a richer metamodel such as EMF may make it easier to write code generation templates, when compared to a lower-level metamodel such as Java.

Requirement	XSLT	JET	OAW Xpand	Xpand
Language	XML	Java	Java	Java
Runtime Environment	n/a	Standalone	Eclipse	Eclipse
Open Source	n/a	✓(EPL)	✓(EPL)	✓(EPL)
Version	2.0	0.8.2	4.3.1	1.0.0
Navigation Language	XPath	Java	Java/OCL	Java/OCL
Query Language	XQuery	Java	Java/OCL	Java/OCL
Underlying Metamodel	XML	Java	EMF	EMF
Third party extensibility	✗	✓	✓	✓
OpenBRR Rating	n/a	105	94	104
Average Quality		4.5	3.7	3.0
Average Support		4.0	3.0	4.0
Average Documentation		4.0	2.5	5.0
Average Adoption		3.0	4.0	3.0
Average Architecture		4.0	4.0	4.0

Table 6.3: Comparison of Code Generation Environments

4. *Third party extensibility:* Does the technology allow third parties – that is, the model developers themselves – to extend the provided code generation templates? This can improve developer usability significantly, as the developer does not need to access the source code of the modelling environment to make changes, but could instead just add extensions to the given environment.

With respect to these technologies, the decision was made to use the Xpand framework to implement code generation within IAML. The proof-of-concept implementation uses the OAW Xpand project, as the Eclipse-branded Xpand project had not yet been announced when work on the proof-of-concept implementation began.

Finally, it must be noted that none of these evaluated approaches natively support any form of round-trip engineering of model transformations, as discussed earlier in Section 3.1.8. It may be possible to implement round-trip engineering support in a model-to-text transformation implementation[20], and this research remains an important area of future work. This limitation will not impact on the proof-of-concept implementation of IAML however, as round-trip engineering is not a design goal of this thesis.

6.5 Model Completion

Discussed earlier in Section 3.2, *model completion* is the automated process of taking an incomplete base model and adding elements to this model to create an *intended model* according to sensible default rules. Model completion is particularly important for the implementation of Wires as discussed in Section 4.9.2, as this means the logic and semantics behind a Wire can be implemented within an additional model completion layer, and the underlying metamodel and resulting code generation layers can remain unchanged.

[20]For example, change models or trace models have already been used by GMF to support traceability on its model-to-model transformations.

There are a number of different technologies that may be used to implement model completion rules. Almost any language that supports some form of inference and logic can be used; this includes rule engines such as Drools [273], Jena [193] and Jess [98]. The Java specification request JSR-94 [268] covers the definition of a Java rule engine API, and most commercial rule engines are implementations of this standard. In this section, these three rule engines will be evaluated by implementing the *default checkbox rule* introduced earlier in Section 3.2.2.

As model completion is designed as a model transformation operating within the same source and target metamodels, model-to-model transformations may also be used to implement model completion. However this section will not deal with the implementation of model completion in these general-purpose transformation engines, as it is expected that a dedicated rule engine grammar and implementation will provide better performance and expressibility. For example, the three rule engines discussed here utilise the Rete algorithm [273] in order to improve performance, whereas this approach is novel in existing model transformation engines [**?**]. Similarly, these rule engines may be used to implement model-to-model transformations, but this discussion is outside the scope of this thesis.

6.5.1 Jena

Jena is an open source Java framework for building semantic web applications by providing a number of RDF-based APIs. Internally, knowledge is represented as an RDF graph according to the W3C recommendations for the Semantic Web [193]. This graph is then extended with components such as the Semantic Web query language RDQL [193, 266] and a hybrid forward/backwards chaining inference engine.

Since Jena is focused on expressing and evaluating RDF triples, it is necessary to translate a model instance into an RDF format in order to implement model completion rules using Jena. This can be done manually, however the translation of a model instance into an RDF format will lose the semantic information of the underlying metamodel – such as type hierarchies – unless this information is also translated.

The Web Ontology Language (OWL) [304] is a standard for representing the higher-level knowledge of data, and in particular can be used to represent a metamodel of a model instance. The *emftriple* project can be used to translate Ecore models into OWL models [134], which can then be loaded as part of an RDF-based model completion process.[21]

There are a number of constraints for the translation of an Ecore model into the OWL format using *emftriple*. The tool assumes that there are unique names for all classes, properties and enumerations in the metamodel; and the tool struggles with external references to Ecore metamodels located within the project[22]. However, future versions of the tool may resolve these problems to support an automated Ecore translation.

Once the Ecore model has been translated into an OWL format, and a model instance represented in EMF is translated into an RDF format, the Jena inference engine can be used as in Listing 7. The *makeSkolem* builtin predicate allows for a new blank node to be generated based on the uniqueness of a particular set of parameters, and is essential to a Jena implementation of model completion.

[21]**TODO:** The inference of EMF-based models using Jena is pretty novel. Should the source code be provided in an appendix, or an online article, or a publication, or similar? The current implementation is on SVN at `http://code.google.com/p/iaml/source/browse/branches/2009-08-owl/org.openiaml.model.owl/src/org/openiaml`

[22]In particular, all references to the Eclipse XSD metamodel had to be removed before the ontology could be generated successfully.

```
[exampleModelCompletion:
  (?P rdf:type iaml:BooleanProperty)
  (?C iaml:children ?P)
  noValue(?E iaml:for ?P)
  makeSkolem(?A, ?P, ?C)
  ->
  (?A rdf:type iaml:Checkbox)
  (?A iaml:id 'generated')
  (?A iaml:isGenerated 'true')
  (?A iaml:for ?P)
  (?C iaml:children ?A)
]
```

Listing 7: Example rule implementation in Jena

However, there is no easy way to implement the insertion cache concept using Jena rules; that is, the new node created by *makeSkolem* is inserted into the model immediately. Since model completion rules are orderless – that is, the order in which they are executed must not matter – the lack of an insertion cache is not fatal for a Jena-based implementation. However, a Jena-based implementation of model completion may need to define a new *makeCachedSkolem* function in order to correctly satisfy the requirements of model completion.

The Jena framework supports the integration of additional rule engines in order to evaluate model instances, such as Pellet [272], Racer [?] and Fact++ [?]. These engines can provide a limited amount of additional functionality and improved performance; however this thesis will only evaluate the default Jena reasoner.

6.5.2 Jess

Jess is a closed-source rule engine written in Java, which is designed to operate both on facts and POJOs. As discussed at the beginning of this chapter, a closed-source component would make the proof-of-concept implementation difficult to redistribute, and would also hinder the implementation of rule engine extensions, such as the *insertion cache*. Jess was a driving force behind the design of JSR-94, and is the reference implementation for the original specification [268]. An R2ML/Jess translator is also available [216].

The underlying model that Jess operates within is sets of facts, which can be inserted and retracted from within the execution environment. Java objects can also be inserted as facts into the rule engine, and if these objects are also JavaBeans (i.e., POJOs), then the bean properties can also be accessed directly as facts. However, the structure of inserted POJOs must be defined manually as part of the rules, as illustrated in Listing 8. Consequently model completion rules can be difficult to implement in Jess if the underlying objects are based on EMF.

Critically, Jess does not currently support multiple interface inheritance [97], where a class can have more than a single supertype. This means that there are certain classes of rules and type hierarchies that simply cannot be described using Jess; and the sample model completion rule proposed by Wright and Dietrich [320] cannot be implemented in Jess without first flattening the inheritance hierarchy.

```
(deftemplate IamlFactoryImpl (declare (from-class IamlFactoryImpl)))
(deftemplate BooleanPropertyImpl (declare
    (from-class BooleanPropertyImpl)))
(deftemplate CheckboxImpl (declare (from-class CheckboxImpl)))
(deftemplate PageImpl (declare (from-class PageImpl)))

(defrule model-completion-example
  "Example rule of model completion."
  (BooleanPropertyImpl (OBJECT ?property))
  (PageImpl (OBJECT ?container))
  (IamlFactoryImpl (OBJECT ?factory))
  (test ((?container getChildren) contains ?property))
  (test (eq (?property getEditor) nil))
  =>
  (bind ?o (?factory createCheckbox))
  (?o setFor ?property)
  (?o setIsGenerated TRUE)
  ((?container getEditors) add ?o)
  (add ?o)
)
```

Listing 8: Example rule implementation in Jess

6.5.3 Drools

Drools, also known as *JBoss Rules*, is an open source business rule management system and inference rule engine implemented in Java [273]. Inference rules are evaluated using an enhanced implementation of the Rete algorithm [273].

The underlying model that Drools operates within is simple POJOs, making it easy to integrate into an existing Java-based software system. This also means that all of the metamodel properties and operations are accessible to a Drools rule. Reflection is also possible using the EMF API.

Along with Jena, the design and implementation of R2ML was heavily supported by Drools [110], and thus the translation of R2ML rules to Drools rules and vice versa is well supported [216]. The Drools engine also supports an implementation of the JSR-94 specification.

In Drools, a model completion rule is implemented as an inference rule, as in Listing 9, as described in Wright and Dietrich [320]. In particular, the use of an *insertion cache* assists in the implementation of the stratification technique of model completion, where newly created model elements are inserted into a cache before being reinserted into the working memory.

6.5.4 Summary

A summary of this investigation is provided in Table 6.4. Along with the common comparison criteria discussed earlier in Section 6.1.1 and Section 6.1.2, each technology was evaluated on a number of additional factors:

1. *Uses POJO model:* Does the underlying model used by the rule engine support POJOs? That is, can the rule engine be inserted into an existing Java application without any additional translation steps necessary? Since EMF model instances are accessible as POJOs, this means that the rule engine can be integrated with an EMF-based modelling environment easily. Any translation

```
rule "Example rule"
  when
    p : BooleanProperty( )
    not ( Editor ( for == p ))

  then
    Checkbox c = handler.generatedCheckbox(p);
    handler.setFor(c, p);
    cache.add(c, drools);

end
```

Listing 9: Example rule implementation in Drools, adapted from Wright and Dietrich [320]

Requirement	**Jena**	**Jess**	**Drools**
Language	Java	Java	Java
Runtime Environment	Standalone	Standalone	Standalone
Open Source	✓(BSD)	✗	✓(Apache)
Version	2.6.4	7.1p2	4.0.7
Uses POJO model	✗	✓	✓
Multiple interface inheritance	✓	✗	✓
R2ML Translator	✓	✓	✓
Custom Rule DSL	✗	✗	✓
OpenBRR Rating	99	n/a	101
Average Quality	3.8		2.3
Average Support	2.0		4.5
Average Documentation	2.5		5.0
Average Adoption	3.0		4.0
Average Architecture	4.0		5.0

Table 6.4: Comparison of Rule Engine Environments

steps necessary will reduce the performance of model completion, and impact the integration and traceability of the environments.

2. *Multiple interface inheritance:* Can the rules use multiple interface inheritance? The IAML metamodel has heavy usage of both multiple interface inheritance. If a rule engine cannot support multiple interface inheritance, a metamodel can be extended to single interface inheritance, but this indicates additional development effort that may not be necessary.

3. *R2ML Translator*: Does a translator exist to translate R2ML rules [109] into this language, and vice versa? An R2ML translator may permit the automatic translation of rules written in one rule engine into another rule engine, improving interoperability.

4. *Custom Rule DSL*: Does this rule engine directly support the definition of a domain-specific language for model completion rules? By providing a DSL for a given set of rules, rule developers can express rules more easily. This is not strictly necessary, as a domain-specific language can be written outside of the scope of the rule engine.

Each of these engines were found to be expressive enough to describe many different types of rules. The UServ Business Rules Model 2005 [110] provides a "common benchmark" set of rules which may be implemented to compare language expressiveness, and each of these evaluated rule engines satisfies these requirements [110, 216].

Even though there are a large number of rule engines in production, there are very few reliable benchmarks published on these engines, making it difficult to compare engines in terms of performance. This may be due to the fact that rule engine performance depends heavily on the type of rules, the size of the rules, and the size of the input data sets. The UServ benchmark is only concerned with expressiveness, and cannot be used to compare performance characteristics.

In IAML, the decision was made to use the Drools rule engine in order to implement the model completion framework. As the engine was open source, it was fairly straightforward to implement the concepts behind model completion (such as stratification), as discussed by Wright and Dietrich [320]. This work evaluated performance metrics against a test suite of sample input models, and a wide range of model completion rules, and found that Drools was a suitable target platform to implement model completion.

6.6 Model Instance Verification

As discussed earlier in Section 3.4, *model instance verification* is the process of evaluating syntactically correct model instances against constraints to verify they are correct with respect to the intended function of the modelled domain. Each of these verification rules may be implemented in a different technology; for example, the structural definition of the language – which includes a type of model instance verification – is implemented as part of the metamodelling environment, whereas reference cycles can be detected using a different framework.

In this section, a simple example constraint will be used to *illustrate* the design and expressibility of each verification language. The constraint *infinitely redirects* is a constraint that identifies web application pages that, when visited, would continually redirect the user in a non-terminating loop. This constraint may be implemented within both functions- or relations-based verification languages, and may also be evaluated using a model checking approach.

6.6.1 Drools

As discussed in the previous section, Drools was evaluated as a potential implementation of the model completion concept. This same approach can also be used to implement model instance verification, by using inference to infer constraint violations. Likewise with the implementation of Drools used in model completion, the constraints can directly evaluate the POJO objects of the model instance under evaluation. Drools does not natively support any form of transitive closure, so this must be implemented manually with the appropriate guards to prevent an infinite loop.

The implementation of *infinitely redirects* in Drools is provided in Listing 17. The first rule defines the inference of a transient property *NavigatesTo* with respect to two Frames; and the second rule defines that the property is transitive. The final rule defines how constraint verification occurs, and also provides information about the source of the violation. Drools does not support the creation of transient facts, so a separate verification metamodel needs to be created and defined; however, this metamodel can be entirely separate of the metamodel under evaluation.

```
rule "Frame navigating to another Frame on access"
  when
    access : Event ( )
    p : Frame ( onAccess == access )
    p2 : Frame ( eval(p != p2) )
    navigate : ECARule ( from == access, to == p2 )
    not ( NavigatesTo ( from == p, to == p2 ))
  then
    NavigatesTo navigatesTo = factory.createNavigatesTo();
    navigatesTo.setFrom(p);
    navigatesTo.setTo(p2);
    insert(navigatesTo);
end

rule "NavigatesTo is transitive"
  when
    p1 : Frame ( )
    p2 : Frame ( eval(p1 != p2) )
    n : NavigatesTo( from == p1, to == p2 )
    p3 : Frame ( eval(p2 != p3) )
    n2 : NavigatesTo ( from == p2, to == p3 )
    not ( NavigatesTo ( from == p1, to == p3 ))
  then
    NavigatesTo navigatesTo = factory.createNavigatesTo();
    navigatesTo.setFrom(p1);
    navigatesTo.setTo(p3);
    insert(navigatesTo);
end

rule "An infinitely redirecting loop"
  when
    p : Frame ( )
    n : NavigatesTo ( from == p, to == p )
    not (Violation ( source == p ))
  then
    Violation violation = factory.createViolation();
    violation.setSource(p);
    violation.setReason("An infinitely redirecting loop");
    verify.failed(violation);
    insert(violation);
end
```

Listing 10: Implementation of *infinitely redirects* in Drools

```
[redirects1:
  (?P rdf:type iaml.visual:Frame)
  (?E rdf:type iaml:EventTrigger)
  (?P iaml:onAccess ?E)
  (?P2 rdf:type iaml.visual:Frame)
  (?W rdf:type iaml:ECARule)
  (?W iaml:from ?E)
  (?W iaml:to ?P2)
  ->
  (?P eg:redirectsTo ?P2)
]

[redirects2:
  (?P rdf:type iaml.visual:Frame)
  (?P2 rdf:type iaml.visual:Frame)
  (?P3 rdf:type iaml.visual:Frame)
  (?P eg:redirectsTo ?P2)
  (?P2 eg:redirectsTo ?P3)
  ->
  (?P eg:redirectsTo ?P3)
]

[ruleInfiniteRedirect: (?v rb:validation on()) ->
  [ (?X rb:violation error('violation', 'infinite redirect', ?X)) <-
    (?X rdf:type iaml.visual:Frame)
    (?X eg:redirectsTo ?X)
  ]
]
```

Listing 11: Implementation of *infinitely redirects* in Jena

The Drools rule engine does not natively support the derivation trace for a given model element, however the rule engine can be extended to support this scenario. As the implementation of Drools is provided as an Eclipse project, it would be relatively straightforward to implement Drools-based model verification within an Eclipse-based development environment.

6.6.2 Jena

Similarly to Drools, Jena was also evaluated for model completion in Section 6.5.1, and similarly can also be used to implement model instance verification. Jena's inference engine explicitly supports the definition of special model verification rules, which can be selectively enabled or disabled [249]. Likewise with the implementation of model completion, it is necessary to translate the metamodel into an OWL representation [304] using the previously-introduced *emftriple* project [134], and translate the model instance into an RDF implementation.

The implementation of *infinitely redirects* is provided in Listing 11. Similarly to the implementation of the constraint in Drools, the rule (*redirects1*) defines the inference of a transient property *redirectsTo*. The second rule (*redirects2*) defines that the property is transitive, as Jena does not directly support a higher-order transitivity operator on relations. The final rule enables the validation engine, and defines the constraint itself.

Once a constraint violation has been detected by Jena, it is possible to obtain a derivation trace to the source of the violation; however, since the source models had to be translated from their original representations, it would require some development effort to reunite the derivation trace with the source models.

6.6.3 OWL 2 Full

A verification constraint can also be represented using OWL 2 [304], which has native support for concepts such as transitive closure and inference, by using class equivalence to select constraint violations. To be more specific, the metamodel designer would define a new class, defined as *equivalent* to all violating instances of a certain constraint. These constraints could then be combined with the generated metamodel ontology, and the exported RDF graph of the model instance, and evaluated using an ontology reasoner to identify the constraint violations.

One possible implementation of the *infinitely redirects* constraint is provided in Listing 12. This constraint is defined through equivalence of a transitive property *instantRedirectTo*. However, since selecting this property requires a self restriction on a transitive property with a cardinality constraint, this falls outside the bounds of OWL 2 DL, and thus this ontology is an OWL 2 Full ontology.

Unfortunately, at the time of writing there were no implementations of an OWL 2 Full ontology reasoner; furthermore, OWL 2 Full ontologies has been shown to be generally undecidable [212], and it is possible that no implementations will ever exist in the future. Consequently it is not possible to use this technology as a verification engine for the proof-of-concept implementation of IAML.

6.6.4 CrocoPat

CrocoPat is a relational reasoner written in C++ [27], developed and released under the LGPL. It uses Binary Decision Diagrams to efficiently infer information, and natively supports higher-order logic through existential `EX`, universal quantification `FA` and transitive closure `TC` operators; properties can also be defined recursively. It has a well-defined axiomatic semantics, but obtaining the derivation trace is not currently possible. CrocoPat was initially designed to evaluate software models for design patterns, and was found to be much faster than alternative approaches, including Prolog and a commercial relational database [27].

Since CrocoPat is written in C++, the reasoner does not yet directly support the interaction with Java objects or POJOs; as a result, a Java object must first be serialised into a textual format according to the RSF/RML languages. This representation can then be concatenated with a rule file and evaluated by CrocoPat. There is no native support for typing, so a type system has to be manually implemented if it is necessary.

The implementation of *infinitely redirects* in CrocoPat is provided in Listing 13. The first definition defines the inference of a transient property of *RedirectsTo* with respect to two Frames; and the second rule defines that the property is transitive. The third rule defines the constraint violation through inference of this property, and finally the reasoner is instructed to find all instances of that violation.

CrocoPat does not natively support the derivation trace for a given property, but it is possible to add this through an extension of the reasoner. As the implementation of CrocoPat is provided as a C++ project, some development effort is necessary in order to implement CrocoPat-based verification within an Eclipse-based environment.

```
// namespaces omitted
Ontology(<http://openiaml.org/verification/2009/infiniteRedirect.owl>
Import(<http://openiaml.org/model0.5>)

// some classes omitted

EquivalentClasses(InfiniteRedirectFrame ObjectIntersectionOf(
  ObjectExistsSelf(instantRedirectTo) visual:Frame))
SubClassOf(InfiniteRedirectFrame iaml:Scope)
SubClassOf(InfiniteRedirectFrame iaml:VisibleThing)
SubClassOf(InfiniteRedirectFrame visual:Frame)

EquivalentClasses(AccessEventTrigger ObjectIntersectionOf(DataHasValue(
  iaml:name "access") iaml:EventTrigger))

// some properties omitted

TransitiveObjectProperty(instantRedirectTo)
ObjectPropertyDomain(instantRedirectTo visual:Frame)
ObjectPropertyRange(instantRedirectTo visual:Frame)

ObjectPropertyDomain(navigatesToFrame iaml:ActionEdge)
ObjectPropertyRange(navigatesToFrame visual:Frame)

SubObjectPropertyOf(SubObjectPropertyChain(iaml:outEdges isActionEdge
  InverseObjectProperty(isActionEdge)) onAccessNavigate)

SubObjectPropertyOf(SubObjectPropertyChain(iaml:to isFrame
  InverseObjectProperty(isFrame)) navigatesToFrame)

SubObjectPropertyOf(SubObjectPropertyChain(iaml:outActions toFrame)
  eventToFrame)

SubObjectPropertyOf(SubObjectPropertyChain(iaml:to isFrame
  InverseObjectProperty(isFrame)) toFrame)

// defines the verification constraint
SubObjectPropertyOf(SubObjectPropertyChain(iaml:eventTriggers
  isAccessEventTrigger InverseObjectProperty(isAccessEventTrigger)
  onAccessNavigate navigatesToFrame) instantRedirectTo)

)
```

Listing 12: Implementation of *infinitely redirects* in OWL 2 Full

```
RedirectsTo(a, b) :=
  Frame(a) & a != "null" &
  EX(e, EventTrigger(e) &
  onAccess(a, e) &
  Frame(b) & b != "null" &
  EX(w, ECARule(w) & trigger(w, e) & target(w, b)));

RedirectsTo(a, b) :=
  Frame(a) & Frame(b) & TC(RedirectsTo(a, b));

InfiniteRedirect(p) :=
  Frame(p) & p != "null" &
  RedirectsTo(p, p);

PRINT ["Infinite redirection"] InfiniteRedirect(p);
```

Listing 13: Implementation of *infinitely redirects* in CrocoPat

```
abstract sig Frame {
  redirectsTo : set Frame
}

one sig Frame_page1 extends Frame { }{ redirectsTo = Frame_page2 }
one sig Frame_page2 extends Frame { }{ redirectsTo = Frame_page3 }
// ...

assert test {
  no p : Frame | p in p.^redirectsTo
}
check test for 3
```

Listing 14: Implementation of *infinitely redirects* in Alloy

6.6.5 Alloy

Alloy [147] is a specification language for assisting in the design and specification of software and software models, implemented using Java and released under the MIT license. Its intended use is in the development of robust modelling language that conform to a given suite of assertions, by trying to derive model instances that are invalid according to those assertions (known as *counter-examples*). Alloy can also be used to discover constraint violations for a single *given* model instance, although this is not its original intent.

Since Alloy is designed for the specification of languages, it does not directly support other sources of models, such as POJOs; models and metamodels must first be translated into the Alloy specification language. The language does not support multiple inheritance, however the Variant design pattern [38] can be used to emulate this design with single inheritance. It is difficult to define primitive data values such as strings and integers; these have to be translated manually into singleton world facts. Finally, Alloy does not support recursive functions, however in many situations these can be emulated using the transitive closure operator `.*` [62].

There are a number of ways in which the *infinitely redirects* constraint can be implemented in Alloy; a simple and efficient implementation is provided in Listing 14. In this approach, most of

```
// initial def
cached List[model::visual::Frame] redirectsTo(model::visual::Frame this) :
  onAccess.listeners.to.typeSelect(model::visual::Frame);

// we need to keep track of which nodes we have visited
cached Boolean doesRedirectTo(model::visual::Frame a,
    model::visual::Frame b, Set[model::visual::Frame] visited) :
  visited.contains(b) || a.redirectsTo().exists( c | c == b ||
      doesRedirectTo(c, b, (visited.add(a)).toSet()));

// definition
infiniteRedirect(model::visual::Frame this) :
  doesRedirectTo(this, this, {}.toSet());
```

Listing 15: Implementation of *infinitely redirects* in openArchitectureWare: Xtend

```
extension validation::InfiniteRedirect;

context model::visual::Frame ERROR "Infinite redirection":
  !infiniteRedirect();
```

Listing 16: Implementation of *infinitely redirects* in openArchitectureWare: Checks

the components of the underlying IAML metamodel are ignored, and only the essential structure is defined. Some properties, such as *redirectsTo*, are pre-calculated by the translation step; this greatly simplifies the computational requirements of the Alloy verification step.

One powerful feature of Alloy is that it natively supports the interactive visualisation of a constraint violation. The integration of this visualisation into a model verification approach would be ideal to illustrate the reasoning behind a specific counter-example. Because Alloy is designed to search for possible counter-examples for a given range of constraints, Alloy can be considered a model checking language.

6.6.6 openArchitectureWare

openArchitectureWare (OAW) was previous discussed in Section 6.4.3 as an implementation of a code generation engine. The framework also supports the definition of constraints through the combination of the *Xtend* language for transient model extensions, and the *Checks* language for constraint definitions, which use an expression language similar to Java and OCL [75].

Likewise with the implementation of OAW used for code generation, the constraints can directly evaluate the POJO objects of the model instance under evaluation. OAW does not natively support any form of transitive closure, and has difficulty supporting recursive constraints; inferred extensions should be explicitly labelled as `cached`, and some recursive definitions will result in an infinite loop.

Nevertheless, the implementation of *infinitely redirects* is provided in Listings 15 and 16. The first model extension defines a property *redirectsTo* which lists the Frames that a given Frame may directly redirect to; the second model extension then defines a property *doesRedirectTo*, which is true if a given Frame will redirect to another Frame. The third model extension *infiniteRedirect* defines a property for which infinite redirection can be detected. Finally, the constraint definition in Checks is used to define the constraint error message.

The default implementation of OAW does not support the derivation trace of inferred properties. However, if the OAW framework is already integrated into a project in terms of a code generation framework, then it is trivial to add these additional verification constraints to the code generation process. Consequently, OAW may be the ideal platform for implementing property-based constraints, as introduced earlier in Section **??**.

6.6.7 OCL

The Object Constraint Language (OCL) [227] is a language recommended by the OMG to define constraints on model instances, and includes support for the definition of operations, and the derivation of attributes and associations. The OCL language does not support the creation of transient properties – the only type of inference supported is derivation – and therefore a separate verification metamodel must be supplied, or the original metamodel must be extended with additional verification properties. The language itself is functions-based, and supports the higher-order logic of function aggregation through the `iterate` expression [222, pg. 211].

The OCL standard is platform-independent, and there are many different implementations of the language. Only two implementations will be considered in this section: the EMF Validation Framework, and the Dresden OCL toolkit. Since the OCL standard does not support property inference, the *infinitely redirects* constraint needs to be implemented using operations as listed in Listing 18. These constraints are derived from the implementation of the *acyclical class inheritance* constraint of UML class diagrams, as discussed earlier in Section 3.1.9.

EMF Validation Framework with OCL

The EMF Validation Framework [72] provides a number of methods to define constraints for a given EMF model. One approach is to define the constraints as part of the metamodel itself [275, pg. 549–565]; this is ideal for simple constraints such as properties, as the constraints are included in the generated metamodel code itself. However, this approach does not automatically translate the constraints into executable code; for example, an OCL constraint will not automatically call the Eclipse OCL implementation. Consequently, this approach will not be considered in this section.

Alternatively, the constraints can be provided by a constraint *adapter*, allowing the model developer or other tools to selectively execute the constraints. This is ideal for more advanced constraints that may be more computationally intensive. This adapter can explicitly define that a given constraint is written in OCL, and such a constraint will then be executed by the Eclipse OCL implementation.

However, constraints defined in this way are not fully functional OCL specifications; they can only be used to check invariant-based constraints that are not recursive, and cannot be used to specify operation bodies. The recursive OCL constraints introduced earlier in Listing 18 therefore cannot be implemented using this *adapter* approach. Since the EMF Validation framework is a member of the Eclipse ecosystem, it is well-supported with rich documentation.

Dresden OCL

The Dresden OCL2 toolkit [**?**] implements the OCL 2.2 language [227] and may be executed within a standalone environment, or integrated into another modelling environment. OCL specifications are

```
rule "Frame navigating to another Frame on access"
  when
    access : Event ( )
    p : Frame ( onAccess == access )
    p2 : Frame ( eval(p != p2) )
    navigate : ECARule ( from == access, to == p2 )
    not ( NavigatesTo ( from == p, to == p2 ))
  then
    NavigatesTo navigatesTo = factory.createNavigatesTo();
    navigatesTo.setFrom(p);
    navigatesTo.setTo(p2);
    insert(navigatesTo);
end

rule "NavigatesTo is transitive"
  when
    p1 : Frame ( )
    p2 : Frame ( eval(p1 != p2) )
    n : NavigatesTo( from == p1, to == p2 )
    p3 : Frame ( eval(p2 != p3) )
    n2 : NavigatesTo ( from == p2, to == p3 )
    not ( NavigatesTo ( from == p1, to == p3 ))
  then
    NavigatesTo navigatesTo = factory.createNavigatesTo();
    navigatesTo.setFrom(p1);
    navigatesTo.setTo(p3);
    insert(navigatesTo);
end

rule "An infinitely redirecting loop"
  when
    p : Frame ( )
    n : NavigatesTo ( from == p, to == p )
    not (Violation ( source == p ))
  then
    Violation violation = factory.createViolation();
    violation.setSource(p);
    violation.setReason("An infinitely redirecting loop");
    verify.failed(violation);
    insert(violation);
end
```

Listing 17: Implementation of *infinitely redirects* in Drools

```
context Frame::allRedirectsTo() : Set(Frame) body:
  allRedirectsTo = self.redirectsTo()->union(
    self.redirectsTo()->collect(p | p.allRedirectsTo())

context Frame::redirectsTo() : Set(Frame) body:
  redirectsTo = if self.onAccess->oclIsUndefined() then OrderedSet{}
    else self.onAccess.listeners.to->collect(p | p.oclIsKindOf(Frame))
    endif

context Frame inv:
  not self.allRedirectsTo()->includes(self)
```

Listing 18: Implementation of *infinitely redirects* in OCL

```
LTLSPEC
  G ((!(navigation_running = 1 -> !(F navigation_finished = 1)))
    U navigation_running = 0)
```

Listing 19: Implementation of *infinitely redirects* in LTL

defined separately from the definition of the metamodel, allowing for constraint verification to occur independently of the model instance.

The latest release of the toolkit – formerly called *Dresden OCL for Eclipse* – includes integration within the Eclipse environment and allows OCL constraints to execute on a variety of models, including EMF-based model instances and POJOs. The Dresden OCL toolkit also forms the basis of ArgoUML's support for verifying specified OCL constraints [281].

6.6.8 NuSMV

NuSMV is a symbolic model checker written in C [48] which supports the evaluation of specifications written in the CTL and LTL specification languages [140]. As opposed to both CrocoPat and Alloy, the definition of a NuSMV specification is implemented as a series of states; the model checker then navigates through these states to try and find counter-examples. Once a counter-example has been identified, the derivation trace is provided to the user in a textual format of state changes. NuSMV supports a concept similar to threads by defining *modules*, which can be executed in any order according to a fairness algorithm; such a concept may make it easier to test the asynchronous nature of RIAs.

Since a NuSMV specification is based around the definition of states and their transitions, a considerable amount of effort must first be expended to translate a given model instance into a system specification. This often involves the definition of system functionality; for example, a web application may define the set of available web pages, buttons that may be clicked, current browser location, state-based functionality of operations and so on. In the domain of web applications, the functionality of the web browser itself would have to be emulated. As a result, these specifications are unlike any of the other approaches mentioned in this chapter, as NuSMV does not interact with types or model instances, and cannot infer any information.

The implementation of the specification for the *infinitely redirects* constraint in LTL, which NuSMV may then evaluate, is provided in Listing 19. This specification states that in all situations (`G`), we must

Requirement	Jena	OWL	Drools	CrocoPat	OAW
Language	Java	n/a	Java	C++	Java
Runtime environment	Standalone	n/a	Standalone	Standalone	Eclipse
Open source	✓(BSD)	n/a	✓(Apache)	✓(LGPL)	✓(EPL)
Version	2.6.4	2.0	4.0.7	2.1.4	4.3.1
Uses POJO model	✗	✗	✓	✗	✓
Supports subtypes	✓	✓	✓	✗	✓
Derivation cause	✓	✓	Manual	Manual	✗
Implementation	✓	✗	✓	✓	✓
Transient properties	✓	✓	✗	✓	✓
L_{ver} expressiveness	Relations	Relations	Functions	Relations	Functions
Higher-order logic	✗	✓	✗	✓	✗
Supports inference	✓	✓	✓	✓	✓
Recursive rules	✓	✓	✓	✓	Manual
Model checking	✗	✗	✗	✗	✗
OpenBRR rating	99	n/a	101	77	94
Average Quality	3.8		2.3	3.0	3.7
Average Support	2.0		4.5	1.0	3.0
Average Documentation	2.5		5.0	3.0	2.5
Average Adoption	3.0		4.0	2.0	4.0
Average Architecture	4.0		5.0	1.0	4.0

Table 6.5: Comparison of Model Instance Verification Environments (1)

never (`!`) have a navigation that never finishes (`!F`), as long as we can only navigate one page at a time (`U`). This example does not include the considerable amount of definitions necessary to model the system; these definitions are included in Appendix **??**.

This specification focuses on the abstract concept of "page navigation" to identify an infinitely redirecting page. This means that any form of page navigation in an IAML model instance must be implemented according to these concepts, but also provides the model developer with a much stronger system. Other concepts such as the behaviour of predicates and operations can be defined entirely within this state-based system, allowing for the specification of more advanced constraints. Such constraints could include that all operations executed within a system must terminate, or that no variable may overflow its specified range (such as integer overflow).

The NuSMV project is released under the open source LGPL license; however development does not occur in public, so it is not possible to obtain accurate measurements for many of the OpenBRR quality measurements. In particular, there is no public bug tracker, so it is not known if there are any critical or security issues in any given release. The OpenBRR evaluation therefore assumes the worst-case values for these metrics.

6.6.9 Summary

A summary of this investigation is provided in Tables 6.5 and 6.6. Along with the common comparison criteria discussed earlier in Section 6.1.1 and Section 6.1.2, each technology was evaluated on a number of additional factors. This evaluation identified that none of these technologies satisfy all of

Requirement	Alloy	EMFV OCL	Dresden OCL	NuSMV
Language	Java	Java	Java	C
Runtime environment	Standalone	Eclipse	Standalone	Standalone
Open source	✓(MIT)	✓(EPL)	✓(LGPL)	✓(LGPL)
Version	4.1.10	1.3.1	3.1.0	2.4.3
Uses POJO model	✗	✗	✓	✗
Supports subtypes	✗	✓	✓	✗
Derivation cause	✓	✗	✗	✓
Implementation	✓	✓	✓	✓
Transient properties	✗	✗	✗	✗
L_{ver} expressiveness	Relations	Functions	Functions	n/a
Higher-order logic	✓	✓	✓	✗
Supports inference	✓	✗	Partial	✗
Recursive rules	✗	✓	✓	✗
Model checking	✓	✗	✗	✓
OpenBRR rating	91	90	72	63
Average Quality	3.7	3.3	3.0	1.7
Average Support	1.0	3.0	1.0	1.0
Average Documentation	2.5	5.0	2.5	2.5
Average Adoption	3.5	2.0	2.0	3.0
Average Architecture	4.0	3.0	3.0	2.5

Table 6.6: Comparison of Model Instance Verification Environments (2)

the desirable properties of a universal verification engine, and each technology has at least one feature that it does not currently support.

1. *Uses POJO model:* Does the underlying model used by the language implementation support POJOs? That is, can the verification engine be inserted into an existing Java application without any additional translation steps necessary? The benefits of this approach are identical to those as discussed in the previous section.

2. *Supports subtypes:* Does the language support the concept of typed elements natively? If the language does not, then the type system of the metamodel will need to be supported through additional predicates, which may introduce development problems or impact performance. If the language cannot support inference, then the range of possible constraints that may be implemented will be heavily restricted.

3. *Derivation cause:* Can the verification engine provide any type of trace for a failed constraint? Tracability will allow for a more informative error message to be displayed to the model developer, improving usability.

4. *Implementation:* Is an implementation of the language provided? This requirement is critical for the technology to be considered as part of the proof-of-concept implementation in Chapter 7.

5. *Transient properties:* Is it possible to define the properties necessary for verification as transient? That is, can constraints define additional properties and predicates without having to

modify the source metamodel? A language that does not support transient properties will need to use a separate verification metamodel at runtime, increasing development effort.

6. *L_{ver} expressiveness:* With respect to the categories defined earlier in Section 3.4.2, is the expressiveness of this verification language based on functions or relations? This may impact on the performance and decidability of constraints expressed within this language.

7. *Higher-order logic:* Does this language support the definition of additional functions or relations using higher-order logic? This may include support for transitive closure over relations, or aggregation over functions.

8. *Supports inference:* Can the language define additional properties or predicates through inference, or can a predicate only be defined within a single definition? Inference may allow the decomposition of a complex constraint into many smaller definitions, supporting constraint reuse.

9. *Recursive rules:* Can the language define recursive properties or predicates – that is, definitions that can recursively call themselves? A language that cannot support recursion may restrict the range of its expressible constraints. In some scenarios, recursion may be implemented using transitive closure instead [62].

10. *Model checking:* Can the language evaluate a constraint by systematically checking all of the possible states of the system, as described earlier in Section 3.4.2? Model checking allows for a much more detailed analysis of the behaviour of the system, but requires significantly more computing power in order to evaluate a constraint.

As discussed earlier in Section 3.4.3, model instance verification will be implemented within three categories of constraint expressiveness, and each category implemented using a single constraint language discussed in this section. To inform this decision, the desirable properties for each of these categories must first be discussed. A summary of this evaluation against each of these criteria is provided here in Table 6.7.

1. *Function Language:* The most important criteria for a function-based verification language are ease of use and speed, since these will be evaluated frequently. Therefore, this technology needs to provide an implementation, and should also support POJOs; support native subtypes; and support the inference of transient properties. Since OAW Checks satisfies all of these criteria and is executed within the Eclipse runtime environment, this technology is the ideal candidate for the verification of model instance properties.

2. *Relation Language with Higher-order Logic:* Relation-based languages may incur a higher performance penalty, but should support recursive rules and transitive closure. It is also important that a derivation cause can be provided to the user, as a recurse constraint violation may be more difficult for a model developer to understand. According to these evaluations, no technology provides all of these features; the CrocoPat engine is however selected as it is preferable to have recursive rules over model checking support.

3. *Model Checking Language:* Finally, an ideal technology for model checking must not only support model checking, but provide a derivation trace for constraint violations. The key difference

Technology	Function Language	Relation Language + HOL	Model Checking
EMFV OCL	#	✗	✗
Dresden OCL	✓	✗	✗
Drools	✓	#	✗
OAW	✓✓	#	✗
Jena	✓	✓	✗
CrocoPat	✓	✓✓	✗
Alloy	#	✓	✓
NuSMV	#	✗	✓✓

Marker	Description
✓✓	Best approach
✓	Satisfies most requirements
#	Requires workarounds
✗	Not supported

Table 6.7: A summary of the suitability of verification languages for addressing each verification approach category

between Alloy and NuSMV is that Alloy tries to find invalid structures, whereas NuSMV tries to find invalid *behaviour* through states. This difference allows NuSMV to be much more expressive for describing complex constraints and more thorough when finding violations, and is therefore selected for the proof-of-concept implementation.

TODO I also did a performance evaluation, but this might be out of the scope of the thesis/evaluation. Should I include it? It may give more weight to which engine is better.

6.7 Conclusion

In this chapter, a range of different technologies have been discussed, each which may be used in the development of a model-driven environment for IAML model instances. For each type of technology, one has been selected as the basis of the first proof-of-concept implementation of IAML. In the next chapter, the details involved with actually implementing the modelling environment will be discussed.

Chapter 7

Proof-of-Concept Implementation

In the previous chapter, a range of model-driven technologies were reviewed and selected to form the basis of the proof-of-concept implementation of IAML. This chapter will fully discuss the effort and details required in actually translating the design of the modelling language into a functional graphical editor. The development history of this implementation is available online at the IAML project page of `http://code.google.com/p/iaml/`, and the public Subversion [50] source code repository at `http://iaml.googlecode.com/svn/trunk/`.

7.1 Introduction

The implementation of this proof-of-concept graphical editor involves the development and integration of a large number of components, such as model completion; code generation; instance verification; and the underlying model instances and IAML metamodel itself. An overall summary of these components and their integration is provided here as a UML component diagram in Figure 7.1, with each technology viewed as a black-box component connected through external interfaces; the rest of this chapter will decompose each of these components into a white-box view [224, pg. 152].

The overall runtime environment of the proof-of-concept implementation of IAML is within the Eclipse framework [102], which provides a rich plugin environment based on OSGi bundles [**?**]. Each of these implementation components may therefore be provided as an OSGi bundle, where each connecting interface is specified through exporting and importing packages.

> For each component in the proof-of-concept implementation of IAML, the corresponding *OSGi bundle ID* for its implementation will be provided using this visual syntax.

7.1.1 Implementation License

As discussed earlier in Section 5.1.5, one of the design goals of the IAML language was to provide the language and proof-of-concept implementation under an open source license. It is undesirable to define a new open source license, as a wide range of licenses have already been defined, and the proliferation of different licenses is a major problem [95].

As mentioned earlier in Section 2.7.4, a full discussion on the integration of different open source licenses is well outside the scope of this thesis, and the interested reader is referred to Michaelson

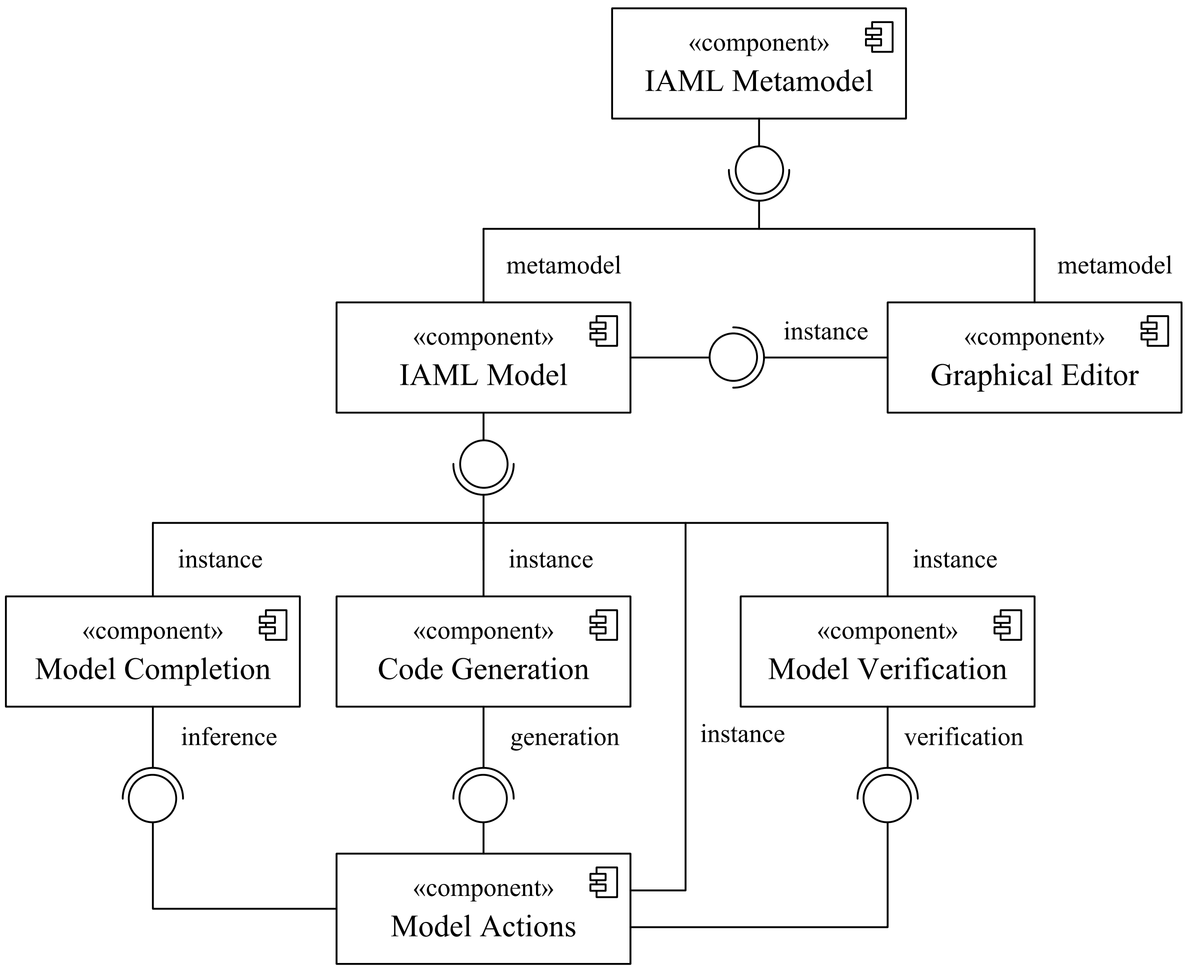


Figure 7.1: Overall UML component diagram of the Proof-of-concept Implementation of IAML

[203]. Other than simply making the source code available, other intentions of licensing this implementation under an open source license were to ensure the project could be integrated into other platforms; that the license is an open source license according to the Open Source Definition [230]; that the license supports the development of *free software* [203]; and that third-party modifications should also be released under the same license to encourage community development.

In this implementation, the decision was made to use the *Eclipse Public License* (EPL) [70], as it satisfies all of these license requirements. It is also trivial for EPL-licensed components to legally work together, as many of the dependencies of the IAML implementation are already licensed under the EPL. Finally, all EPL-licensed code can be reintegrated into the Eclipse project [71], meaning that any extensions or fixes developed as part of this implementation can directly be contributed back into the relevant EPL-licensed component.

7.2 *IAML Metamodel*

The *IAML Metamodel* component is implemented as a single component, and is implemented in EMF 2.5 [275]. This component internally depends on two external components to provide the metamodels for XSD and Ecore. In particular, both the XSD and Ecore metamodels are provided through existing implementations by the Eclipse Foundation [275, 73].

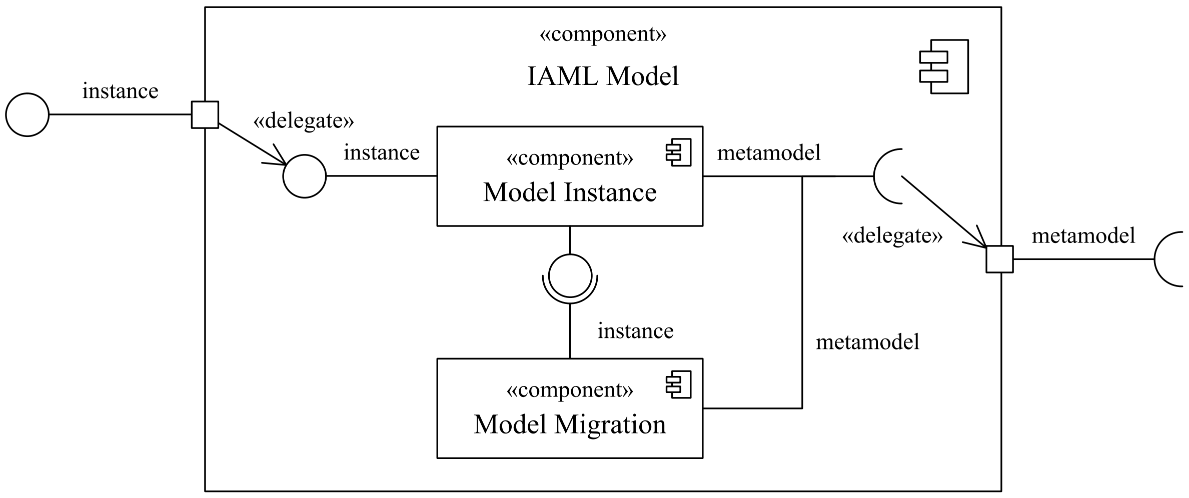


Figure 7.2: UML component diagram of the *IAML Model* Component Decomposition

The corresponding OSGi bundle for the implementation of the *IAML Metamodel (EMF)* component is `org.openiaml.model`. This bundle internally depends on the `org.eclipse.xsd` and `org.eclipse.emf.ecore` bundles, implemented and supplied by the Eclipse Foundation.

7.3 *IAML Model*

The decomposition of the *IAML Model* component is provided in Figure 7.2. This component is made up of two sub-components: a *Model Instance* component; and a *Model Migration* component. The former component provides the functionality to store, interact and serialise a given IAML model instance, with respect to the IAML metamodel; and the latter component provides the functionality to migrate model instances between different metamodel versions.

7.3.1 *Model Instance*

The *Model Instance* component is the most critical component of the entire implementation, as all of the other components in the system interact with IAML model instances. The implementation of this component, however, is simply the generated source code from EMF from the given Ecore-based metamodel. As described by Steinberg et al. [275], this involves automatically creating a `genmodel` model for a given metamodel source – which, in this case, is an Ecore file – and then using this `genmodel` model instance to generate the relevant source codes. IAML model instances are therefore stored as XMI representations within `iaml` files, as illustrated throughout this thesis.

Only one extension needs to be used in the generation of the model instance source code. To simplify the reference of model element instances in IAML model instances, all elements are given an ID; that is, an `EAttribute` with the `ID` property set to `true` [275, pg. 110]. However, these IDs are not generated automatically; the relevant *Factory* for the creation of the element must create its own ID. To implement this, a *dynamic template* is provided to the JET engine[1].

[1]The relevant dynamic template is `org.openiaml.model/templates-emf/model/FactoryClass.javajet`.

> The corresponding OSGi bundle for the implementation of the *Model Instance* component is `org.openiaml.model`. This bundle ID is the same as the *IAML Metamodel (EMF)* component, which is a common scenario of generated EMF models [275, pg. 71].

7.3.2 *Model Migration*

As IAML was developed using an iterative evolutionary process as discussed in Section 5.1.3, existing model instances used for testing the implementation had to be kept valid throughout the development lifecycle with respect to changes to the underlying IAML metamodel. In this thesis, this process is defined as *model migration*, which is "the execution of [an algorithm] on existing domain models to transform them into domain models that are correct in the evolved domain" [**?**, pg. 6]; this is equivalent to a model transformation where the source and target metamodels are different versions of the same metamodelled domain.

The evolution of metamodels, and the subsequent migration of model instances to updated metamodels, is a significant area of academic research[2] [252] and an important aspect of developing a domain-specific language [89, pg. 65]. As IAML was designed and implemented in an incremental fashion, the model migration strategy used in the proof-of-concept implementation is an *incremental migration strategy*, as discussed by Fowler [89, pg. 65]. That is, a suite of migrators are developed over the course of the metamodel's evolution, allowing for changes to build upon existing migrators.

The actual implementation of these migrators followed a naïve approach without references to a model-driven environment. In particular, the migrators are written in Java, and operate on the XMI representations of a model instance within `iaml` files. This approach was used as it was the simplest to implement; there was no need to create a repository of old IAML metamodels, which would have been of significant size[3]; and model migration is not a major focus of this research. Future model migrators should instead follow one of the other migration approaches in order to obtain their benefits.

> The corresponding OSGi bundle for the implementation of the *Model Migration* component is `org.openiaml.model.actions`.

7.4 *Graphical Editor*

The decomposition of the *Graphical Editor* component is provided in Figure 7.3. This component is decomposed into five separate sub-components: *Model Edit*, a component which provides a basic interface to edit model instances; *Diagram Editors*, the set of diagram editors that actually provide the graphical interface; *Diagram Definitions*, the model instances that define the diagram editors, and are used to generate the editors themselves; *Diagram Extensions*, which provides graphical extensions to the diagram editors, without having to modify the editors directly; and *Diagram Actions*, which provide additional actions to the diagram editors themselves.

[2]**TODO:** Grundy said in his recent Massey presentation that model and tool evolution is tricky.

[3]At the time of writing, the IAML metamodel definition file `iaml.ecore` had been modified 239 times within the version control repository.

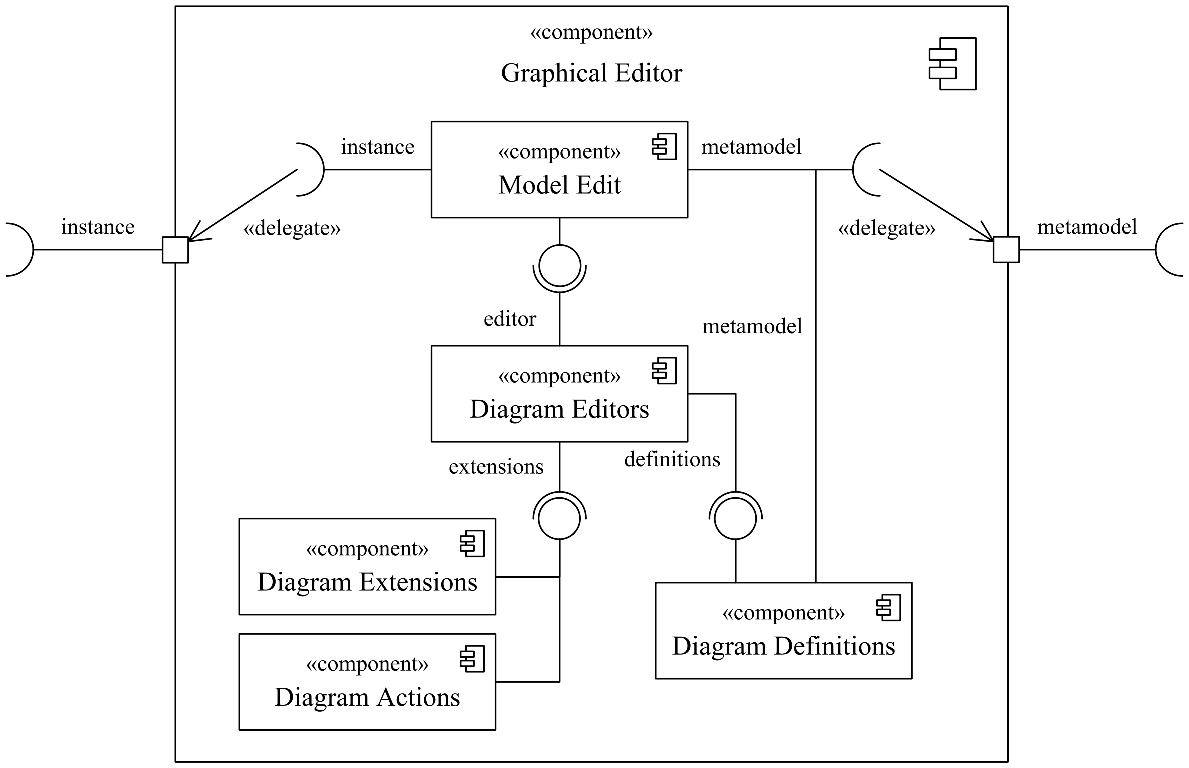


Figure 7.3: UML component diagram of the *Graphical Editor* Component Decomposition

7.4.1 *Model Edit*

Part of the implementation of EMF is *EMF.Edit Support*, which provides a basic user interface to editing model instances, by combining the generated model plugin with the Eclipse UI Framework *JFace* [275, pg. 45–46]. Similarly to the generated model plugin as discussed in Section 7.3.1, an Edit plugin may be generated automatically from the `genmodel` model instance.

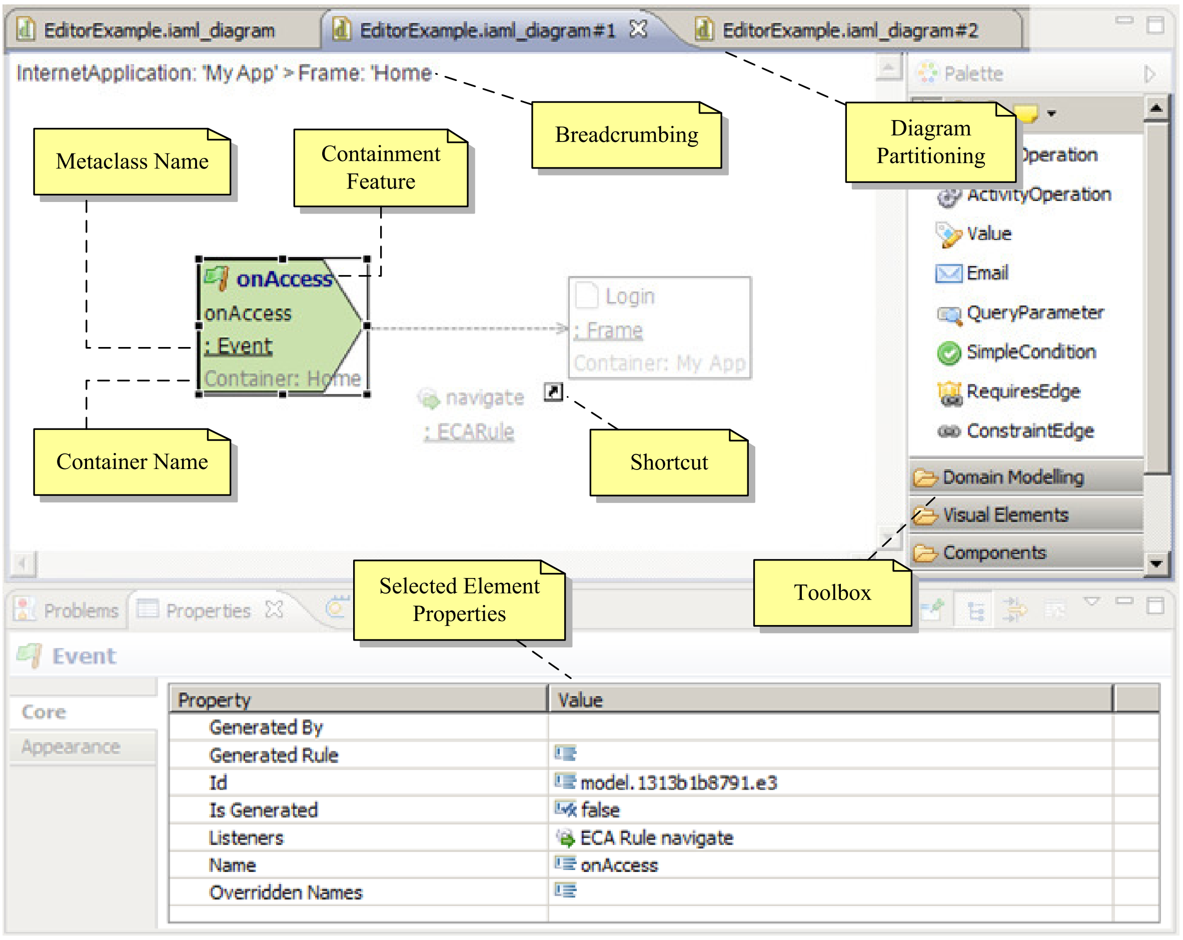
This generated plugin includes a tree-based viewer of a model instance derived automatically from the metamodel structure, and a properties-based element viewer allowing a model instance developer to modify the attributes and references of selected model elements. The generated implementation of this component is fairly crude, but is a necessary requirement for the diagram editors, as discussed in the next section.

The corresponding OSGi bundle for the implementation of the *Model Edit* component is `org.openiaml.model.edit`. This bundle ID follows the standard naming pattern for generated EMF.Edit plugins [275, pg. 649].

7.4.2 *Diagram Editors*

The *Diagram Editors* component represents the actual graphical editors for IAML model instances. As discussed earlier in Section 6.3, these editors are defined according to the Graphical Modeling Framework (GMF). In particular, the graphical editor was implemented using GMF 2.2.

The source code for the diagram editors are generated according to the process described earlier in Section 6.3.2; in particular, the editors are defined according to GMF metamodels, and are provided by the *Diagram Definitions* component. A screenshot of the resulting diagram editor is provided here



Property	Value
Generated By	
Generated Rule	
Id	model.1313b1b8791.e3
Is Generated	false
Listeners	ECA Rule navigate
Name	onAccess
Overridden Names	

Figure 7.4: Implementation of a graphical editor for IAML model instances using the Graphical Modeling Framework

in Figure 7.4.

The visual representation of a model instance is stored independently from the underlying model instance within a `iaml_diagram` file. GMF editors also provide the action "**initialise diagram file**" to initialise a new visual representation of a model instance, which uses the automatic layout logic provided by GMF. Within the IAML implementation of the *Diagram Editors* component, this allows any `iaml` model instance to be transformed into a visual representation.

Hierarchical Modelling

As discussed earlier in Section 5.2.2, model instances are intended to be edited following a hierarchical modelling approach, in order to reduce the complexity of model instance development and to support a number of different visual metaphors. The implementation of hierarchical modelling using GMF was straightforward, but equired some diagram extensions to improve usability.

The basic implementation of hierarchical modelling was provided by GMF's built-in support for *diagram partitioning*; this approach allows for different model elements to appear as the current "root", and model instance developers can navigate through the model instance by opening and closing elements and tabs, as illustrated in Figure 7.4. For example, by double clicking on the representation of a particular Frame instance, a new *Frame* diagram editor window will open, visually representing the *directly contained* contents of that Frame element instance; a number of diagram extensions such as

Top-Level Container	OSGi Bundle ID
Internet Application	`org.openiaml.model.diagram`
Access Control Handler	`org.openiaml.model.diagram.access_handler`
Activity Predicate	`org.openiaml.model.diagram.condition`
Email	`org.openiaml.model.diagram.email`
Frame	`org.openiaml.model.diagram.frame`
Domain Instance	`org.openiaml.model.diagram.instance`
Domain Iterator	`org.openiaml.model.diagram.iterator`
Login Handler	`org.openiaml.model.diagram.login_handler`
Activity Operation	`org.openiaml.model.diagram.operation`
Domain Type	`org.openiaml.model.diagram.schema`
Session	`org.openiaml.model.diagram.session`
Visible Thing	`org.openiaml.model.diagram.visual`

Table 7.1: Generated Diagram Editors in IAML and their associated OSGi Bundle IDs

breadcrumbing and shortcuts extend this functionality, as discussed later in Section 7.4.4.

Only model elements that are intended to contain other elements should have an associated diagram editor; this significantly reduces the number of different diagram editors that are necessary. A list of all of the diagram editor instances, along with their associated OSGi bundle IDs and the type of their root model element – known in GMF as the *top-level container*[4] – are listed in Table 7.1.

7.4.3 *Diagram Definitions*

As discussed earlier, the *Diagram Definitions* component defines the abstract representations of the intended diagram editors according to the GMF metamodels, and these diagram definitions may then be translated into instances of diagram editors as through the process discussed earlier in Section 6.3.2. In particular, four metamodels (*gmftool*, *gmfgraph*, *gmfmap* and *gmfgen*) are defined by GMF, and the GMF framework provides code generation templates implemented in Xpand to translate instances of *gmfgen* models into Java source code.

Only one instance of the *gmfgraph* metamodel is necessary, as there should only be one visual representation of each IAML metamodel element. These graphical definitions are shared across all of the editors as described in Table 7.1. For each of these editors, an associated *gmftool*, *gmfmap* must manually be defined, as each diagram editor has a different set of elements that can be displayed (*gmfmap*), and a different set of elements that can be created (*gmftool*). By combining these three model instances together using the GMF framework, an instance of the *gmfgen* metamodel can be created for each of the diagram editors, and this instance can then be used to generate the source code of the actual diagram editor.

As an example, the Session diagram editor is generated by combining the graphical definitions in the `iaml.gmfgraph` file; the palette tooling definition in the `session.gmftool` file; and the shape

[4]A diagram editor defined GMF with an abstract root element type as a top-level container will generate a warning message of "Top-level diagram container must be concrete" when the editor is generated. This warning is raised as it is not possible for a model developer to create an instance of an abstract element, and occurs when generating the Visible Thing diagram editor, but this warning may be ignored in hierarchical approaches for elements contained directly or indirectly by the concrete root element.

and element mapping definitions in the `session.gmfmap` file, into the editor generation definitions into the `session.gmfgen` file.

> The corresponding OSGi bundle for the implementation of the *Diagram Definitions* component is `org.openiaml.model`, as the diagram definitions are stored in the same location as the *IAML Metamodel* component to simplify development.

SimpleGMF

It was recognised during the development of the *Diagram Definitions* component that many of the GMF model instances shared similar features. For example, the "open diagram behaviours" needed to be consistent across all *gmfgen* instances; mappings in *gmfmap* instances needed to be consistent with the IAML metamodel structure; and tool definitions in the *gmftool* instances needed to be consistent with these *gmfmap* mappings.

In response to these common patterns, a domain-specific modelling language named *SimpleGMF* was developed, which also uses model transformations in openArchitectureWare to simplify the development of these many GMF model instances. This language allows for many common GMF definitions to be defined in a single graphical definition file, and generates all of the model instances of the four metamodels (*gmftool*, *gmfgraph*, *gmfmap* and *gmfgen*) necessary. The full description of SimpleGMF is well outside the scope of this thesis; the interested reader is instead referred to the project homepage at `http://openiaml.org/simplegmf/`.[5]

Visual Notation Cognitive Effectiveness

As discussed earlier in Section 4.10.1, the design of the visual syntax for IAML model instances needs consider the usability of the resulting syntax, and to satisfy the goal of having a one-to-one mapping between elements and notations. With respect to the cognitive effectiveness guidelines proposed by Moody [208], the visual syntax has the following design:

1. **Construct deficit**, where a model concept does not have a corresponding visual notation, is permitted for attributes and references that are rarely used. This is acceptable as all attributes and references are still accessible to the developer through an additional dialog – the *Selected Element Properties* frame, as in Figure 7.4.

2. **Construct redundancy**, where multiple visual notations can represent a single model concept, is not accepted with two exceptions: displaying the metaclass name of elements, and displaying the name of element parents, as discussed in the next section.

3. **Construct overload**, where a visual notation can represent multiple model concepts, is not permitted.

4. **Construct excess**, where a visual notation does not represent any model concepts, is also not permitted.

[5] **TODO:** This could also be published as a paper.

Shape Design

It is necessary to describe the design of the visual notations of IAML model elements, along with the rationale behind each of their designs. This process involves seven parameters: shape type; shape style; icons; colours; line weights; line patterns; and font styles. The design decisions for each of these parameters are partially inspired by the guidelines discussed earlier in Section 4.10 by Moody [208] and Rumbaugh [256].

1. **Shape type** refers to whether a model element is displayed as a node or an edge. All model elements will be represented as nodes, with the references between them as edges; however, certain IAML model elements (such as Wires) represent relationships themselves, and should therefore be displayed as edges to simplify the visual syntax.

   GMF editors are designed to work best when there is a one-to-one mapping between graphical elements and element instances in the underlying metamodel. GMF may represent references between element instances using nodes or edges, however this requires the definition of additional *gmfmap* constraints. Additional types of derived nodes and edges may also be displayed, however requires the definition of both additional *gmfmap* constraints and OCL constraints, which had only recently been supported in GMF 2.3[6]. Due to the resource constraints of this research, it was therefore decided to only support the representation of element instances as nodes or edges, and not the representation of references or attributes.

   All model elements within IAML are displayed as a node, with the exception of elements that represent relationships or flows. That is, edges are used as the syntax for all ECA Rules, Conditions, Parameters, Wires and Constraint Edges. Edges are also used for the operation modelling elements of Data Flow Edges, Execution Edges and External Value Edges; and for the domain modelling elements of Provides Edges, Requires Edges, Schema Edges and Select Edges. References and attributes for a particular model element instance are not represented as nodes or edges, but as labels attached to the shape type of the element itself.

2. **Shape style** only applies to node shape types, and refers to the actual style of the node shape; for example, whether the node is displayed as a rectangle, an ellipse, or another shape. The list of visual shapes currently used for various model elements is illustrated here in Table 7.2. Where appropriate, visual designs are inspired by the similar visual stereotypes defined in the UML 2.0 specifications [224].

3. **Icons** are used as an abstract representation of the node or relationship itself. An icon may be visible in three locations, as illustrated in Figure 7.4: the tree-view model instance editor created by EMF [275]; the element creation palette; and on the element itself in the graphical editor.

   Every model element in IAML is provided with a unique icon, partially satisfying the requirement that each model element must have a unique notation. Moody [208] surprisingly finds that icons are rare in software engineering visual notations, and that most rely on geometrical shapes; however, icons can easily be too subtle, and do not satisfy the guideline proposed by Rumbaugh that instances must be easy to draw by hand [256].

[6]Eclipse bug 256461: *Use ParsingOptions.IMPLICIT_ROOT_CLASS for implicit access to the features of EObject in all OCL queries in GMF*

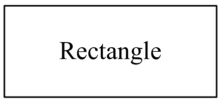
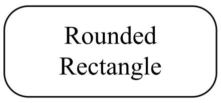
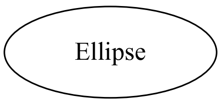
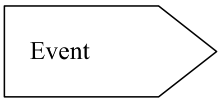
Shape	**Model elements represented**
	The default shape for IAML model elements. This includes visual elements such as Visible Things, domain modelling elements such as Domain Types and Domain Sources, and value instances such as Values.
	This shape is used for Operations.
	This shape is used for Functions and Permissions.
	This shape is used for Events, and is based on the visual notation for UML *SendSignalAction* elements [224, pg. 284].

Table 7.2: Shape styles for the visual representation of IAML model elements

4. **Colours** refer to the colours used in the nodes and edges. For colourblind users or for printing in black and white, the choice of colours can impact on the accessibility of the environment; however, the colour of a visually represented model element does not impact on the behaviour or functionality of that element in any fashion, as elements are distinguished through their metaclass names. There are three categories of colours used:

   (a) **Edges**: One of the guidelines proposed by Rumbaugh [256] is that diagrams "must fax and copy well using monochrome images". Subsequently, all lines and edges used in the IAML visual syntax are coloured black.

   (b) **Nodes**: These colours refer to the overall colours of the node backgrounds. As our implementation is within the Eclipse environment, it is appropriate to reuse the style and design guidelines provided for Eclipse plugins [172]. Seven colours – the maximum number of colours that a user can distinguish [208] – are selected from the Eclipse palette[7], and applied to the background colour of model elements as illustrated here in Table 7.3.

   (c) **Text**: Following the same logic behind the decision of edge colours, all text in the IAML visual syntax is either coloured black, gray, or dark blue. Gray is only used for labels that are read-only, such as includes container names (Section 7.4.4). Dark blue is only used to distinguish the *containment feature* of particular elements, such as Events, as discussed later in the next section.

5. **Line weights** for all nodes and edged within IAML are rendered at the default line weight. As per the guideline proposed by Rumbaugh [256] – "distinctions are not too subtle" – no other line weights are used, as it can be difficult to distinguish between line weights when a diagram is printed or viewed in a different screen resolution.

6. **Line patterns** refer to the pattern used on an edge or border; in the visual syntax of IAML, most

[7]According to the style and design guidelines of Eclipse [172], the predominant colours used in the palette are Blue, Yellow, Green, Red, Brown, Purple, and Beige.

Colour	**Model elements represented**	**Sample**
White	Default shape background color.	⬜
Red	Indicates components, such as Login Handlers and Access Control Handlers.	🟥
Orange	Indicates Functions and Conditions.	🟧
Brown	Used for domain modelling and user modelling, such as Domain Types, Domain Iterators and Permissions.	🟫
Yellow	Indicates scopes, such as Sessions.	🟨
Green	Indicates events, actions, gates, or other sources of actions, such as Events and Gates.	🟩
Blue	Indicates operations and their contents, such as Operations and Activity Nodes.	🟦
Gray	Indicates instances of data, such as Values and Temporary Variables.	◻

Table 7.3: Background colours for the visual representation of IAML model elements

edges and borders are rendered as a solid line. A single form of dashed line is used for elements involving some level of interactivity or behaviour, to ensure distinctions between line patterns are not subtle. In particular, a dashed line is used to represent all ECA Rule, Wire, Data Flow Edge and Constraint Edge instances.

7. **Font styles** refer to the font faces, sizes, weights, and decorations used for text on model elements.

   (a) **Font face**: It is not desirable for an interface to use too many different font faces. Pnly one font face is therefore used across all IAML model elements.

   (b) **Font face**: Similarly, it is not desirable for an interface to use too many different font sizes, and only one font size is used across all IAML model elements.

   (c) **Font weight**: Font weight refers to the measure in which a font is emboldened. In IAML, the two font weights of normal and bold are used, and most IAML model elements are represented using the normal weight. The only two exceptions are the name of the *containment feature* and to highlight overridden elements, as discussed in the next section.

   (d) **Font decorations**: Font decorations refer to whether a font is in italic, or underline, or both. All labels in the IAML visual syntax are represented as undecorated labels, with one exception; the metaclass name of a model element is displayed with an underline as per the UML notation [223, pg. 201], as discussed in the next section.

For each model element in the IAML language, a value for each of these parameters is selected according to the design decisions behind the parameter, and encoded as a diagram definition in a *gmfgraph* model instance. A comprehensive reference to the visual syntax used to represent each modelling element is provided in Appendix **??**.

7.4.4 *Diagram Extensions*

While the GMF framework is powerful, it is still necessary to extend the generated IAML graphical editors to provide additional domain-specific functionality. For example, breadcrumbing and metaclass name extensions would be very difficult or impossible to achieve strictly within the GMF runtime, and these extensions should be provided to the editors in an automated way.

There are three ways in which diagram editors generated through GMF may be extended. One method is by providing extensions through Eclipse extension points, defined by GMF[8]. Another is by modifying the generated source code manually; the GMF framework adheres to the semantics of the `@generated` tag as from EMF, as discussed in Section 6.2.1.

Finally, *dynamic templates* may be provided to the generation framework. These templates use the functionality of dynamic templates provided by Xpand, as discussed in Section 6.4.3. In particular, a template file with the same name and the same location as an existing Xpand template will be used to extend or replace the original template. This is achieved with the `AROUND` operator and the `targetDef.proceed()` command.

Manual modifications to the diagram editors are not used in IAML. This means that no diagram editor code needs to be modified manually, and thus the source code does not need to be committed to the version control repository. Extension points would be an ideal way of modifying diagram editors; extension points allow extensions to be added and removed silently, improving performance and scalability, and reducing errors by defining well-designed interfaces [102]. However, the extension points available at the time could not be used to implement some of the necessary diagram extensions.

The generated diagram editors in IAML were therefore extended through both extension points and dynamic templates. The *Diagram Actions* component represents instances of extensions through extension points, and is discussed later in Section 7.4.5; this section will instead discuss the use of dynamic templates for diagram extensions. Five diagram editor extensions will be discussed in this section: *breadcrumbing*; *shortcuts*; *container names*; *metaclass names*; and *generated element notations*. These dynamic templates are supported by a library of common code to improve performance and reduce dependencies, which is stored in a separate OSGi bundle.

The dynamic templates for the *Diagram Extensions* component are stored within the `org.openiaml.model` OSGi bundle, along with the dynamic templates used for the *Model Instance* component. The library of common code used by the *Diagram Extensions* component is provided by the `org.openiaml.model.diagram.helpers` OSGi bundle.

Breadcrumbing

As discussed earlier in Section 4.1.2, it is important for a hierarchical modelling approach to provide appropriate context. For example, a developer may "zoom into" one of two unrelated Frames may both contain a Input Text Field labelled "name"; context is necessary to inform the developer of which Frame instance they are currently viewing.

Breadcrumbs are a technique used in web applications to provide context information to the user about their current location within a web site, which may improve user satisfaction and the efficiency of site navigation [**?**]. This technique is adapted to the diagram editors of IAML; these breadcrumbs highlight the navigation path from the root model element (the Internet Application) to the currently viewed model element via the implicit *container* references of each model element, as discussed earlier in Section 5.3.2. This breadcrumb is calculated automatically and does not persist as part of the diagram; an example of breadcrumbs is illustrated earlier in Figure 7.4.

[8]One such extension point is the `org.eclipse.gmf.runtime.diagram.ui.layoutProviders` extension point, which is used to define layout providers for diagram editors.

Shortcuts

While breadcrumbing provides context for a particular view of the contents of a model element, it does not provide information about references to model elements outside of the current model element contents. For example, when a Button is intended to navigate to another Frame when the *onClick* event is triggered, this external Frame reference should be visible.

GMF supports this concept through *shortcuts*, where external element references are displayed as normal elements, but with an additional shortcut annotation (↗). These model elements must have a corresponding shape/element mapping in the *gmfmap* definition of the current editor, but does not necessarily have to have a corresponding palette tooling entry in the *gmftool* definition. Additionally, these shortcuts are only displayed for node types, and not for edge types such as Wires.

It was necessary to extend the default GMF implementation of shortcuts. With the default implementation, only elements that can be directly contained in the view element could be added as shortcuts; edges between elements would disappear if they are not directly contained; and edges between shortcut elements did not persist across views. Some of these issues are acknowledged GMF bugs, whereas others are expected functionality.

To resolve these issues, a custom shortcut controller GetShortcuts was implemented, which provides a list of relevant nodes and edges that should be rendered as shortcuts contained within a particular model element instance. This controller is then integrated into the generated GMF editors through the use of dynamic templates.

Container Names

While shortcuts can be used to display a model element outside of the currently viewed container, the shortcut icon is a binary annotation that does not provide context to where the model element is actually contained. To resolve this issue, a reference to the *container* of the particular model element is displayed for every model element. Currently this reference is a label documenting the containers' name, as illustrated in Figure 7.4.

Not all model elements need a reference to their parent displayed; for example, Domain Types and Predicates are global throughout the entire application, and such a parent reference would clutter up the visual representation. Parent labels are therefore only displayed for certain element types[9].

Metaclass Names

As discussed earlier in Section 5.3.4, UML generally uses differences in visual notation to distinguish between the instances of different metaclasses. That is, an Input Form should have a different visual notation to an Input Text Field. However, developing unique visual notations for every model element represents a significant amount of effort. A simple interim solution was to follow the UML syntax for defining instances of classes – as illustrated in the UML infrastructure specification [223, pg. 201] – where a model element may also display the name of the defining metaclass of that element.

An example of how metaclass names are displayed on every model element is illustrated in Figure 7.4. These element instance names are derived automatically from the model element instance class at the time that the model element is displayed, and these labels do not persist in the visual nota-

[9] At the time of writing, these elements were Domain Attribute; Domain Attribute Instance; Event; Operation; Function; Value; Frame; Gate; and Visible Thing.

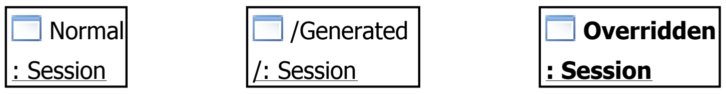


Figure 7.5: Illustrating generated and overridden elements in the IAML editor

tion. The textual format of metaclass names also means that every different model element type may be distinguished, even if the representation is printed without any colours.

Containment Feature

In terms of EMF, a *containment feature* is a reference that may contain other elements. The IAML metamodel was designed to reduce containment feature redundancy, where it should not be possible for a model element to be contained within two or more derived containment features for a particular model element[10].

However, the IAML metamodel does support containment feature redundancy for certain elements; for example, a Changeable element defines the *onAccess* and *onChange* events through containment features earlier in Figure 5.14, and each of these references must hold an Event instance. The functionality of each Event depends on the containment feature that contains it; however within GMF, a shape is defined by the element type itself, and does *not* consider the containing feature[11].

To distinguish between the different containment features which may contain an element, one option is to define a range of element shapes, one for each containment feature; for example, the *onAccess* event could have a slightly modified shape, or the *onChange* event could have a different background colour. However, it is preferable to keep the shape styles consistent, as it is only the containment feature that dictates the differences in functionality.

A *containment feature label* is therefore defined for elements that may be contained through many containment features[12]. These labels are rendered in a dark blue and bold font, and displays the name of the containment feature. An example of this containment feature label is illustrated in Figure 7.4; this label is also provided automatically and does not persist as part of the visual notation.

Generated Element Notations

As discussed earlier Section 7.5, the usability of the implementation of model completion depends on the support of the development environment. For example, a notation is necessary for distinguishing the difference between developer-created elements, and elements created through model completion. Based on the UML concept of *derived* properties with a name prefix of a forward slash “/”, all generated elements are prefixed with this slash, as illustrated in Figure 7.5.

Similarly, a notation is necessary to distinguish elements that have been manually overridden by the developer, as discussed later in Section 7.5.2. One approach to solving this problem is derived from the use of stereotypes or classifiers in UML visual syntax, where textual stereotype labels such as «generated» or «overridden» may be rendered on generated or overridden model element instances.

[10]For example, if a model element could contain any number of Functions, and a subtype of this model element could additionally contain any number of Predicates, then an instance of a Boolean Property could be contained within either of these containment features by an instance of this subtype.

[11]A more detailed discussion on GMF view mapping mismatches between the underlying metamodel and the resulting representation is discussed later in Section 9.2.3.

[12]At the time of writing, these elements were Gate; Event; Value and Builtin Property.

Another approach to solving this problem is illustrated within Mozilla Firefox; overridden configuration values are highlighted in bold, allowing users to quickly identify overridden values. This approach is used in IAML diagram definitions, where all overridden elements are rendered in bold, as illustrated in Figure 7.5. This approach is preferred over using textual stereotypes labels as it is simpler to implement, and reduces the visual complexity of the resulting diagrams.

7.4.5 *Diagram Actions*

Some of the extension points provided by GMF are used to define additional action-based extensions to the diagram editors. Two actions operate on entire collections of visual model representations – i.e. `iaml_diagram` files – as discussed below:

1. **Export to PNG**: GMF provides a builtin command to export a single diagram view to a range of image formats, such as GIF, PNG and SVG. This action extends this functionality to also export *all* of the views of this diagram; that is, every hierarchical level of the diagram, from the root element to every child.
2. **Export to HTML**: This action extends the previous action to also generate a set of HTML files, which provide imagemaps [290] on the exported PNGs to allow a user to navigate through the model instance graphically. This action is used by the *ModelDoc* framework [317] to export navigable example models.

Three actions were also defined to operate on individual model elements within a particular diagram, and can also be applied to collections of individual model elements.

1. **Infer only contained elements**: This action applies model completion to only the selected element, in order to create all of the elements that would be contained by this particular element if model completion were to occur. That is:
   (a) First the entire model is completed, as in the *Infer all elements* action discussed later in Section 7.8.
   (b) The contents of the model are iterated over, and all *generated* elements – that is, Generated Elements where is generated is `true` – that are not contained (directly or indirectly through children) by the target element are removed.
   (c) All *generated* elements that refer to deleted elements are then recursively removed. This process is repeated until the model has stabilised.

   It is important to note that this action is not guaranteed to be safe according to the properties of model completion. In particular, subsequent model completions on this partially inferred model instance may create a different model instance, i.e. $C(X) \neq C(C(X) - Y)$. However, in practice this restriction has not yet proven problematic, and it helps the model developer in overriding the contents of individual model elements.
2. **Remove contained generated elements**: This command similarly operates on a single element, and attempts to remove all *generated* elements that are contained within the selected element. If there are other non-generated elements that refer to the deleted generated element (and its contained children), then the developer is warned that the resulting model may be inconsistent.

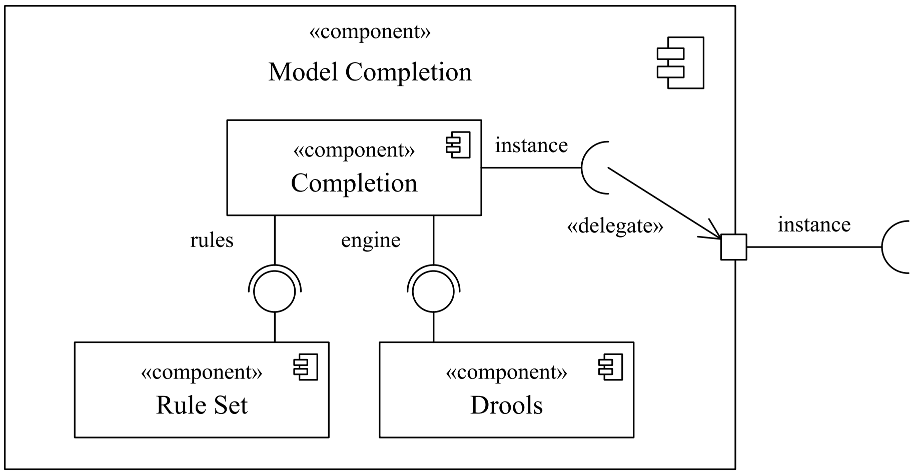


Figure 7.6: UML component diagram of the *Model Completion* Component Decomposition

3. **Move into separate model**: This refactoring action moves the selected diagram elements into a separate IAML model instance, allowing for complex model instances to be split into smaller model instances. This action simply reuses existing EMF functionality [275, pg. 33].

> The corresponding OSGi bundle for the implementation of the *Diagram Actions* component is `org.openiaml.model.diagram.custom`.

7.5 *Model Completion*

As discussed throughout this thesis, model completion may be used to complete additional functionality on a model instance. This includes the implementation of Wires (Section 5.8); completing the Domain Feature Instances within Domain Instances based on their defining Domain Types (Section 5.5.4); or the implementation of client-side input validation based on primitive types (Section 5.4.4).

The *Model Completion* component defines the implementation of the model completion process, as proposed by Wright and Dietrich [320] and discussed earlier in Section 3.2. The underlying rule engine behind the implementation of model completion in IAML is Drools [273], as selected through the evaluation process of the previous chapter in Section 6.5.

The decomposition of this component is provided in Figure 7.6[13]. This component is made up of three sub-components: the *Drools* component, which provides the implementation of the Drools rule engine; the *Rule Set* component, which encodes the logic and intent behind IAML model completion into rules into a format for the rule engine; and the *Completion* component, which connects the rules, engine, and model instance together, and provides the interface necessary to execute model completion on model instances.

[13]This decomposition represents an ideal-world situation, where standards such as RuleML [306] should allow the transparent exchange of different rule engines without loss in functionality; realistically, Drools rules include functionality specific to the engine itself.

The corresponding OSGi bundle for the implementation of the *Model Completion* component is `org.openiaml.model.drools`, and all sub-components provided within this same bundle.

7.5.1 *Completion*

The *Completion* component integrates the set of model completion rules (the *Rule Set* component) into an instance of the rule engine (the *Drools* component), and also extends the rule engine with the requirements necessary to implement model completion according to the requirements proposed by Wright and Dietrich [320]. In particular, the implementation of the *insertion cache* [320] is provided by this component[14].

7.5.2 *Rule Set*

The model completion rules necessary to implement the semantics of the modelling language in Appendix **??** are represented here as the *Rule Set* component, and are implemented within 261 instances of Drools rules. To simplify development, these rules are separated into 21 separate *rule packages*, each within a single file; a summary of these rule packages is provided in Appendix **??**.

As discussed earlier in Section 3.2.1, an important aspect of model completion is ensuring that the rules are well-documented, so that the developer can anticipate the intent of the process. This documentation requirement is addressed by the *ModelDoc* framework [317], where the `@inference` tag can be used to describe the intent of a model completion rule. A sample implementation of one of these rules is provided here in Listing 20; this inline documentation is used later to populate the documentation of the Sync Wire element itself in Section **??**.

One key idea behind model completion is that depending on the intent of the model developer, generated elements may be removed, or model completion may be selectively disabled [320]. The IAML proof-of-concept implementation supports this concept by selectively overriding the generation of elements, using the overridden property of the Generated Element model element.

The behaviour of the *overridden* property is simple. As part of the creation of new modelling elements through model completion, each modelling element is referenced as generated by a particular element[15]. For example, each of the elements generated by a Sync Wire connecting two Input Forms should reference the wire instance as its generator. Conversely, if an element is overridden, then no elements should be generated by that overridden element.

As the overridden property of a Generated Element is stored as a normal attribute within the metamodel, it can be modified as normal by interacting with the properties dialog of a selected element. As described earlier in Section 7.4.4, all overridden elements are represented in the graphical editor in a bold font, to illustrate that the element is overridden.

TODO Discuss *overridden names* property.

7.5.3 *Drools*

The final component necessary to implement model completion is the implementation of the Drools rule engine, as described by Proctor et al. [245]. At the time of writing, version 4.0.7 of the Drools

[14]The implementation of the insertion cache is provided by `DroolsInsertionQueue.java`.

[15]This also means that for an element to be used in the generation of another element, this element must be a subtype of the Generates Elements interface.

```
/**
 * @inference SyncWire When a {@model Changeable} is connected to an
 *    {@model ContainsOperations} by a {@model SyncWire}, the source
 *    element will {@model ECARule call} the 'update' {@model Operation} on
 *    the target when the source {@model Changeable#onChange changes}.
 */
rule "Run instance wire from edit to update (onChange)"
  when
    sw : SyncWire( overriddenNames not contains "run" )
    source : Changeable( )
    target : ContainsOperations( )
    eval( functions.connects(sw, source, target) )

    event : Event( source.onChange == event )
    operation : Operation( eContainer == target, name=="update" )

    not ( ECARule( trigger == event, target == operation, name == "run",
        eContainer == sw ) )

    eval( handler.veto(sw) )

  then
    ECARule rw = handler.generatedECARule(sw, sw, event, operation);
    handler.setName(rw, "run");
    queue.add(rw, drools);

end
```

Listing 20: One Drools rule used in the model completion implementation of the Sync Wire model element

engine was used, which is provided as a collection of JARs.

7.6 *Code Generation*

The *Code Generation* component implements the model-to-text transformation step necessary to translate an IAML model instance into an executable web application. The underlying code generation engine behind this implementation is openArchitectureWare (OAW) [75], as selected through the evaluation process of the previous chapter in Section 6.4. This component relies on a number of sub-components, and the decomposition of the *Code Generation* component is provided in Figure 7.7[16].

This component is made up of six sub-components: the *openArchitectureWare* component, which provides the OAW implementation; the *Templates* component, which provides the code generation templates themselves; the *Runtime Library* component, which provides additional libraries necessary for runtime applications; the *Platform Configuration* component, which provides platform-specific information to the platform-independent IAML model instance; the *Output Formatter* component, which reformats generated source code into a more readable form; and the *Generator* component, which connects all of the components together into a single workflow.

[16]Similarly to the decomposition of the *Model Completion* component, this represents an ideal-world situation where code generation templates may be executed using different code generation engines without any loss in functionality.

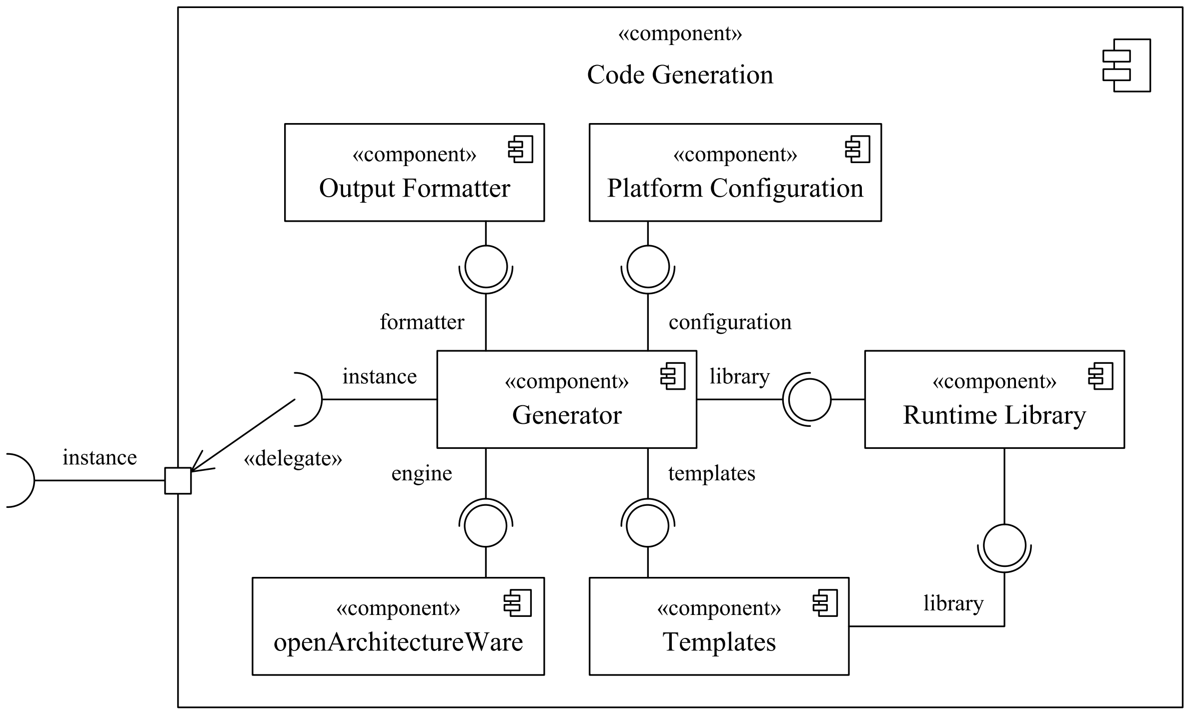


Figure 7.7: UML component diagram of the *Code Generation* Component Decomposition

7.6.1 *openArchitectureWare*

This component simply provides the implementation of the OAW template engine, as described by Efftinge et al. [75]. The code generation templates were implemented with the openArchitectureWare 4.3.1 framework. As discussed earlier in Section 6.4.4, it is desirable to migrate these templates to the recently released Xpand framework, and this remains future work[17]. The OAW library is implemented as a set of Eclipse plugins, and thus is available as a set of OSGi bundles.

The corresponding OSGi bundles for the implementation of the *openArchitectureWare* template engine component are `org.openarchitectureware.dependencies` and `org.openarchitectureware.workflow`, provided by openArchitectureWare.org.

7.6.2 *Templates*

The collection of code generation templates necessary to implement the behaviour of IAML model elements are represented as the *Templates* component, and are implemented using 160 template files. The openArchitectureWare Xpand implementation of one of these templates – `runFrameEvents`, which implements the triggering behaviour of the *onInit* and *onAccess* Events within a Frame – is illustrated here in Listing 21.

The target languages of the proof-of-concept code generation implementation is a combination of PHP 5.3 [180]; Javascript 1.6 [86]; HTML 4.01 [290]; CSS level 2 [300]; and the relational database *sqlite* [233]. At the time that these templates were implemented, the development of HTML 5 had not yet reached the recommendation stage. PHP was selected as the underlying server-side language as it

[17]Issue 71: *Migrate OAW Xpand to Eclipse Xpand*.

```
/**
 * @implementation Action
 *    {@model Action}s are run in order of descending priority; that is,
 *    a higher priority {@model Action} will execute first.
 */
«DEFINE runFrameEvents(String eventName) FOR model::Event-»
  // Frame EventTrigger «this»
  «IF eventName == "init" || eventName == "access"-»
    // Actions sorted by priority
    «EXPAND template::operations::OperationCall::callOperation(true, false)
      FOREACH listeners.sortBy(e|-e.priority)-»
  «ELSE-»
    «throwException("I don't know what to do with frame event " + name)»
  «ENDIF-»
«ENDDEFINE»
```

Listing 21: Implementation of the `runFrameEvents` code generation template in Xpand

```
/**
 * Get the containing {@model Scope} of the given element,
 * or <code>null</code> if none.
 */
model::Scope containingScope(emf::EObject this) :
  null;

model::Scope containingScope(model::NamedElement this) :
  containingScope(eContainer);

model::Scope containingScope(model::Scope this) :
  this;

model::Scope containingScope(model::InternetApplication this) :
  null;
```

Listing 22: Implementation of the `containingScope` model extension in Xtend

was already well-known, and a PHP-based implementation would strongly assist in the proof that the IAML metamodel is platform-independent.

These templates include some model instance extension definitions, using the *Xtend* language [75]. For example, the extension `containingScope` implements the semantics of the *containing scope* definition earlier described in Section 5.10; the source code for this metamodel extension is illustrated here in Listing 22. The collection of templates also require the runtime library, as discussed in the next section.

> The corresponding OSGi bundle for the implementation of the *Templates* component are `org.openiaml.model.codegen.php`.

7.6.3 *Runtime Library*

The *Runtime Library* component provides common libraries necessary for the generated web applications, in order to obtain a number of maintainability-related benefits. In particular, these libraries can be tested independently of the code generation templates; improvements in the libraries can be published independently of the metamodel; code generation templates can be simpler, and generated source code will consequently be simpler; and it is easier to include third-party library components as part of an included library.

Much of the runtime library is the implementation of a single library in two different implementation languages. For example, the typing system of IAML has to be implemented both in the server-side PHP language, and in the client-side Javascript language. For functionality that cannot be executed on the client-side – for example, sending e-mails – callback interfaces must be implemented. Similarly, builtin XPath functions such as `fn:contains` must be implemented in both PHP and Javascript as part of the implementation of XQuery Functions.

The runtime library also includes two third-party components: *PHPMailer* [**?**], which provides a rich interface to send e-mails and is used in the implementation of Email; and the *Prototype Javascript Framework* (prototype.js) [246], which may be used to simplify the development of client-side Javascript within RIAs. The development of this library also included the development of an additional component *html2text*, which provides functionality to transform an arbitrary block of HTML-formatted text into a text-only format, which looks similar to the corresponding HTML representation.

> The corresponding OSGi bundle for the implementation of the *Runtime Library* component are `org.openiaml.model.runtime`, and the third-party PHPMailer library is provided through the OSGi bundle `org.openiaml.model.runtime.phpmailer`.

7.6.4 *Platform Configuration*

There are a number of platform-specific configuration elements that are necessary for the generation of a web application, such as proxy information and API keys. As one of the design goals of the IAML metamodel was to provide a platform-independent metamodel, it is not possible to place this configuration information in these model instances. These platform-specific configuration values are instead defined within a *platform-specific metamodel*, and these model instances are instead provided to the code generation framework.

Currently, the platform-specific model is simply provided as a set of key/value pairs, as discussed later in Appendix **??**. However, this model instance could be represented as an instance of an EMF metamodel, which would afford the resulting platform-specific model instances all of the benefits of a model-driven approach, as described earlier in Section 3.1.3. This approach remains future work[18].

> The implementation of the *Platform Configuration* component is included as part of the *Generator* component within the `org.openiaml.model.codegen.php` OSGi bundle.

[18]Issue 87: *Develop platform-specific metamodel for project properties.*

7.6.5 *Output Formatter*

If a code generation framework is generating source code for a language that is freeform[19][20], the resulting source code is often poorly structured in terms of readability, as the code generation templates do not need to understand the syntax of the target language. While this unreadable source code is still valid program code, it may be desirable to transform this source code into a readable format, to assist in debugging and third-party extensibility.

A common approach to solve this problem is to provide a component to reformat the generated source code; for example, the EMF framework uses Eclipse's Java Development Toolkit (JDT) project to reformat the generated metamodel source code [275, pg. 358]. In openArchitectureWare terms, this formatter is known as a *postprocessor*, which operates on file instances as they are created through the workflow [75].

Because the web applications generated by the IAML code generator are a mix of PHP, Javascript, CSS and HTML, it is difficult to format the source code correctly using a single AST-based parser. Source code of these languages may occur at any time at any arbitrary location; for example, a HTML page may contain a Javascript script, which then contains a PHP instruction within a variable identifier. At the time of writing, no existing parser supported simultaneous formatting for these four languages.

As part of this proof-of-concept implementation of IAML, the *iacleaner* project was developed to apply code formatting to the diverse mix of web application languages used within IAML. This project uses a custom parser which provides the ability to "jump out" between languages, but does not construct an AST. The *iacleaner* project is released as an Eclipse plugin under the Eclipse Public License, and is available online at `http://code.google.com/p/iacleaner/`.

> The corresponding OSGi bundle for the implementation of the *Output Formatter* component is `org.openiaml.iacleaner`.

7.6.6 *Generator*

The final component necessary to implement the code generation component of IAML is the *Generator* component, which orchestrates all of the sub-components together. This component includes an instance of an openArchitectureWare *workflow* [75], which is executed by the template engine to generate the web application source code. This component also includes the user interface actions necessary to execute the code generation on a particular model instance.

> The corresponding OSGi bundle for the implementation of the *Generator* component is `org.openiaml.model.codegen.php`.

7.7 *Model Verification*

The *Model Verification* component defines the implementation of model instance verifiers. As discussed in Section 6.6, there are many different types of verification that may be performed on a model

[19]**TODO:** Reference in a softwarwe engineering book?

[20]For example, languages such as Python and Haskell use whitespace and indentation as part of the language syntax, and are *not* freeform.

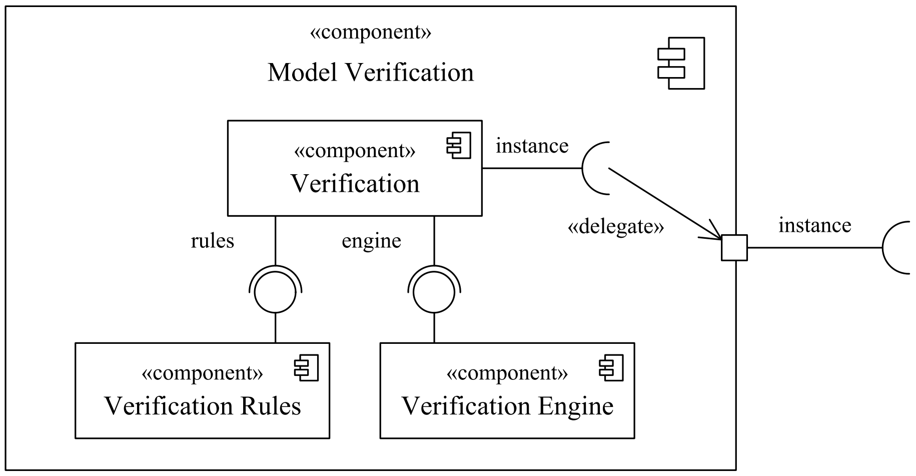


Figure 7.8: UML component diagram of the *Model Verification* Component Decomposition

instance, each with differing requirements and necessary resources; therefore, it is beneficial to have an overall framework for model instance verification.

This component provides this model verification framework, which is implemented as the combination of four sub-components as illustrated in Figure 7.8[21]. Essentially, each type of verification is implemented as a set of verification rules (the *Verification Rules* component), executing on a particular verification engine (the *Verification Engine* component), that operates on a particular model instance (the *Verification* component).

GMF diagram editors integrate constraint violations registered through the EMF Validation framework into the corresponding visual representation of a model instance. When a constraint violation is detected on a model element, a problem is registered in the Eclipse problems view [102], and a graphical annotation is displayed on its visual representation. For example, an invalid Sync Wire which violates the constraint specified in Listing 23 will be displayed to the model developer as illustrated in Figure 7.9.

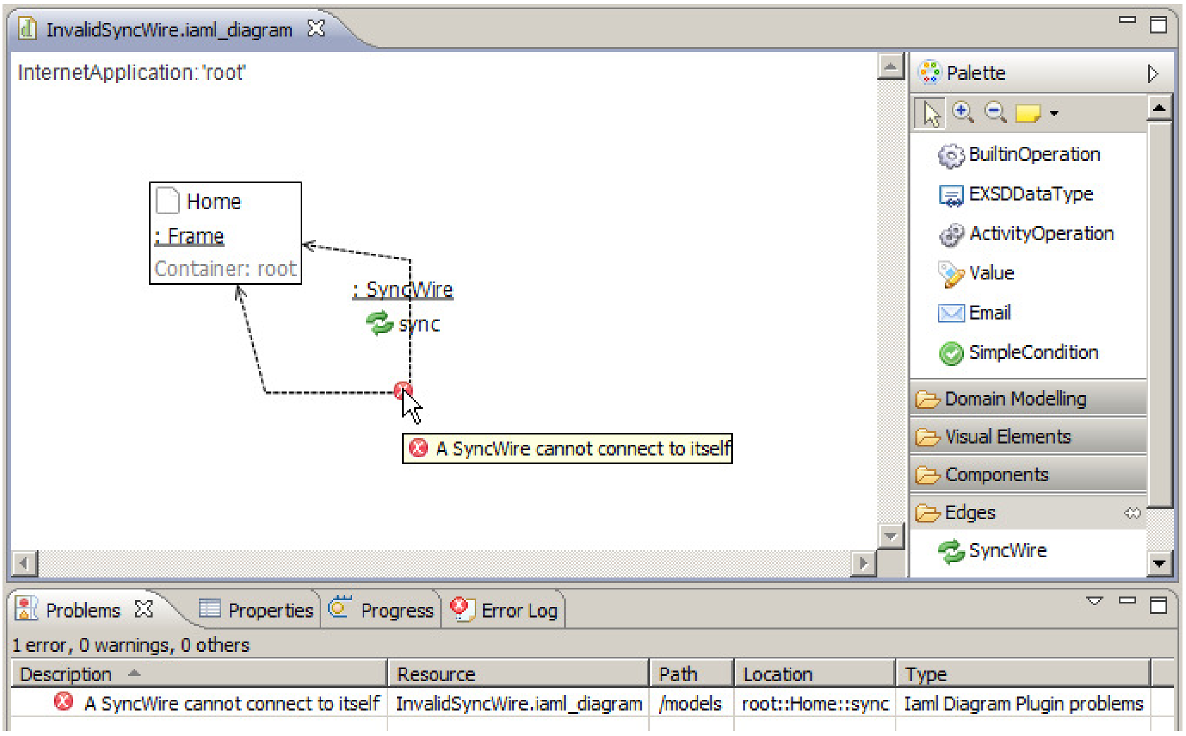
In this section, the implementation of the four selected verification engines as discussed earlier in Section 6.6 – openArchitectureWare Checks, OCL through the EMF Validation Framework, CrocoPat and NuSMV – will be discussed. With the exception of Checks, each of these components are implemented as a separate set of OSGi bundles, allowing for verification to occur independently of the development process as configured by a model developer.

7.7.1 Model Verification with Checks

The Checks language forms part of the openArchitectureWare platform [75], and provides a model instance verification syntax that is a hybrid of Java and OCL. Due to the ease of implementation and framework functionality of openArchitectureWare, this technology was selected in Section **??** as the ideal technology to implement constraints within a function-based verification language.

Consequently, the Checks-based *Verification Engine* component is provided by the openArchitectureWare 4.3.1 framework, and the Checks-based *Verification Rules* component is a collection of 69 constraints. These constraints are provided through the code generation plugin, as model instance

[21] Similarly to the decomposition of the *Model Completion* component, this also represents an ideal-world situation where verification rules may be executed using different verification engines without any loss in functionality, which are one of the aims of the JSR-94 [**?**] and SBVR [**?**] specifications.



A SyncWire cannot connect to itself

1 error, 0 warnings, 0 others

Description	Resource	Path	Location	Type
A SyncWire cannot connect to itself	InvalidSyncWire.iaml_diagram	/models	root::Home::sync	Iaml Diagram Plugin problems

Figure 7.9: Visualisation of model instance constraint violations within a diagram editor

```
context model::wires::SyncWire ERROR "A SyncWire cannot connect to itself":
  from != to;
```

Listing 23: Implementation of a Checks constraint for Sync Wires

verifier is always executed as a prerequisite of code generation. All of the constraints placed on the IAML metamodel throughout Appendix **??** were implemented in terms of Checks constraints. For example, the constraint that a Sync Wire cannot connect the same element together is provided here in Listing 23.

For the Checks implementation of the *Model Verification* component, the *Verification Engine* component is provided by the `org.openarchitectureware.dependencies` and `org.openarchitectureware.workflow` OSGi bundles; and both the *Verification Rules* and *Verification* components are provided by the `org.openiaml.model.codegen.php` OSGi bundle.

7.7.2 Model Verification with OCL in the EMF Validation Framework

While most metamodel constraints are defined and verified using the openArchitectureWare Checks verification component, there are a number of metamodel constraints defined earlier in Chapter 5 which are defined in terms of OCL. It is therefore desirable to reuse these OCL constraints, to improve the robustness of the resulting model instances and prove these constraints are correctly implemented. As discussed earlier in Section 6.6.7, OCL is a functions-based verification language that also supports a form of higher-order logic.

```
<constraint statusCode="1" severity="ERROR" lang="OCL"
    name="Value constraint 1" id="ocl1">
  <message>Values must be type XSDSimpleType or EXSDDataType</message>
  <target class="Value"/>
  <!-- the OCL expression -->
  <![CDATA[
    self.type->isEmpty()
    or self.type.oclIsKindOf(xsd::XSDSimpleTypeDefinition)
    or self.type.oclIsKindOf(model::EXSDDataType)
  ]]>
</constraint>
```

Listing 24: Implementation of a OCL constraint for Values in the EMF Verification Framework

As these constraints are also not defined in a recursive fashion – one of the considerations of the verification technology evaluation discussed earlier in Section 6.6.7 – these constraints may be implemented using the EMF Validation Framework. It is preferable to use this OCL implementation over the Dresden OCL2 toolkit as this framework is a member of the Eclipse ecosystem and well-integrated into EMF-based metamodels. OCL constraints are implemented as part of the `plugin.xml` plugin manifest file of an Eclipse plugin [102][22].

All of the constraints placed on the IAML metamodel throughout Chapter 5 were therefore implemented using the EMF Validation framework. For example, one of the constraints defined in Figure 5.6 – where a Value must define a type of either type XSD Simple Type or EXSD Data Type – is implemented within the EMF Validation framework as an OCL constraint in Listing 24.

For the OCL implementation of the *Model Verification* component, the *Verification Engine* component is provided by the `org.eclipse.emf.validation` and `org.eclipse.emf.validation.ocl` OSGi bundles; and both the *Verification Rules* and *Verification* components are provided by the `org.openiaml.verification.ocl` OSGi bundle.

7.7.3 Model Verification with CrocoPat

The CrocoPat framework [27] discussed earlier in Section 6.6.4 was selected as the ideal technology to implement constraints using a relations-based verification language, as CrocoPat supports the higher-order operator `TC` of transitive closure on relations. As the CrocoPat engine is written in C, the component is not normally provided as an OSGi bundle; but for ease of implementation and component reuse, it has been manually wrapped into a bundle. At the time of writing, only the *infinitely redirects* constraint is implemented; this constraint has been discussed earlier in Section 6.6.4 and illustrated in Listing 13.

For the CrocoPat implementation of the *Model Verification* component, the *Verification Engine* component is provided by the `org.sosy_lab.crocopat.cli` OSGI bundle; and both the *Verification Rules* and *Verification* components are provided by the `org.openiaml.verification.crocopat` OSGi bundle.

[22]**TODO** Need to resolve issue 271: *Cannot use OCL and OAW verification adapters together*.

7.7.4 Model Verification with NuSMV

Finally, the NuSMV framework [48] discussed earlier in Section **??** was selected as the ideal technology to implement model checking. Similarly to the CrocoPat engine, the NuSMV engine is not provided as an OSGi bundle, but is manually wrapped into a bundle.

At the time of writing, only two constraints had been implemented using NuSMV; the *infinitely redirects* constraint as implemented earlier in Listing 19, and a constraint that tries to identify *infinite execution loops* expressed within an Activity Operation. Since the IAML model instance must first be translated into a textual NuSMV input format, a model-to-text transformation must be executed; these transformation templates are also implemented in the openArchitectureWare language.

[**?**] uses a similar approach to implement model checking of design properties of WebML model instances by specifying properties in the LTL language. This approach also uses model transformations to translate a system into a representation suitable for evaluation, and uses a combination of QVT and XSLT to implement these transformations.

For the NuSMV implementation of the *Model Verification* component, the *Verification Engine* component is provided by the `it.itc.irst.nusmv.cli` OSGI bundle; and both the *Verification Rules* and *Verification* components are provided by the `org.openiaml.verification.nusmv` OSGi bundle.

7.8 *Model Actions*

A separate *Model Actions* component integrates the components discussed throughout this chapter, and provides a user interface to access their functionality. In a similar fashion to the user interface provided by the *Diagram Actions* component, this user interface is implemented through context-sensitive menus through the Eclipse framework. Seven actions are defined to operate on model instances, i.e. `iaml` files.

1. **Infer all elements**: This action executes the model completion rules of the *Model Completion* component on the selected model instance, replacing the original model instance.

2. **Generate code**: This action first evaluates model completion against the given model instance, and then executes the code generation templates in the *Code Generation* component to generate an executable web application. By default, this generated code is placed into a directory named `output` within the current Eclipse project.

3. **Generate code and load in browser**: This action extends the *generate code* action to also load the generated application using the system web browser, if one is available. The URL of the generated application is taken from the runtimeUrl property of the Internet Application.

4. **Migrate model to latest version**: This action evaluates the model migrators of the *Model Migration* component on the given model instance. If the current model instance can be suc-

cessfully migrated to the latest version of the metamodel[23], then the original model instance is replaced with the migrated instance.

5. **Remove phantom edges**: During the development process of a model instance, instances of Wires or ECA Rules may lose their from and to references, causing constraint violations since they cannot be visualised. This action searches a model instance for instances of these *phantom edges* and removes them, since they will have no behavioural effect on the generated application.

6. **Rewrite generated element IDs**: As described in the specification for Generated Element, all elements have an associated ID attribute, which is initialised at creation time into a system-unique value[24]; however, these initial values are fairly verbose, such as `model.126aaefae53.29`. This action relabels element IDs to a more human-readable format; for example, an Input Text Field instance may be renamed to `text1`. Future work includes reimplementing the unique ID algorithm to render this operation unnecessary[25].

7. **Export to DOT**: If the elements in a model instance are considered as nodes, and the relationships between the elements as edges, then the model instance can be considered a directed graph. This action exports the selected model instance into a format that may be passed to DOT, as illustrated by Wright and Dietrich [320].

A single action is defined to operate on folders of model instances – or within Eclipse terminology, on instances of IContainer workspace elements.

1. **Migrate all IAML models**: This action recursively searches through the selected container to identify IAML model instances, and attempts to migrate each model instance according to the *Model Migration* component. Any problems identified are aggregated into a single issue, and presented to the developer at the end of the migration process. This action is particularly useful for migrating suites of example models, and was very useful in the development of the test suite of model instances as discussed in Section 7.9.

> The corresponding OSGi bundle for the implementation of the *Model Actions* component is `org.openiaml.model.actions`.

7.9 Tests

A design goal discussed earlier in this thesis in Section 5.1.3 was to develop the proof-of-concept implementation of IAML simultaneously with the production of a suite of automated tests. As described in Section 2.7.3, a well-designed suite of test cases can be used to improve the quality and reliability of the developed software. Consequently, each of the components discussed in this chapter have been implemented according to a range of automated JUnit test cases, within individual OSGi bundles.

In particular, the suite includes tests for the model completion framework and rules; the code generation framework and templates; metamodel and diagram model consistency; release quality tests;

[23] A "successfully migrated model instance" is one which can be loaded through EMF without any warnings or errors.

[24] This unique value is a combination of the package name, the time of package initialisation, and a package-specific counter; this generally guarantees uniqueness with $O(1)$ computational complexity, since the model instance does not need to iterate through the model.

[25] Issue 25: *More descriptive generated IDs*.

and general integration tests. A separate diagram test suite is also defined for testing aspects of the diagram editors themselves, such as the breadcrumbing, container names, and diagram partitioning aspects of the diagram editors.

In order to test the model-driven aspect of the implementation, it was necessary to develop test models that could be evaluated. Consequently, this test suite included the creation of a large suite of test model instances, which at the time of writing included at least 244 model instances. This suite of test models has also been used to evaluate the performance of model completion using the Drools engine, and also to demonstrate the non-trivial nature of model completion [320].

> The corresponding OSGi bundles for the implementation of the *Tests* component are `org.openiaml.model.tests` and `org.openiaml.model.tests.diagram`.

7.9.1 Model-driven Code Coverage

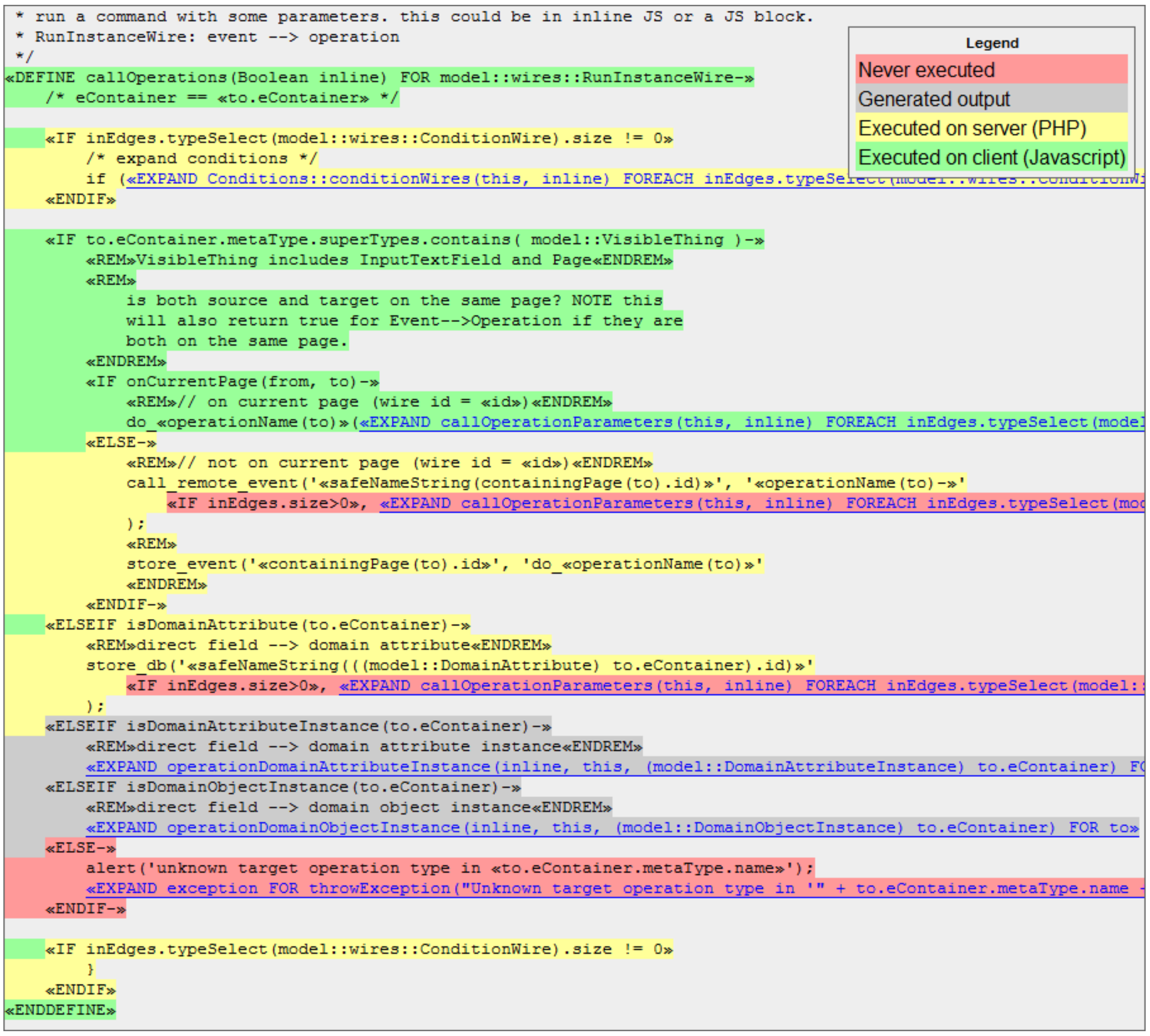
During the development of the proof-of-concept implementation of IAML, a brief experiment was performed where code coverage techniques were applied throughout the code generation process. Because a source IAML model instance is transformed many times – through model completion, code generation, runtime libraries, and executed on any number of target platforms – the concept of *code coverage* can be extrapolated into the concept of *model-driven code coverage*, which applies common code coverage metrics to each individual step of the model transformation process.

As discussed by Nagappan and Ball [215, pg. 417], *code coverage* is "an important metric by which the extent of testing is often quantified," based on the premise that errors cannot be detected within software unless the error is tested. Conversely, code coverage may also be used to identify portions of the source code that is never executed, and to an extent this unused source code may be removed safely in order to improve the simplicity and quality of the software itself.

Code coverage metrics need to be aggregated across a number of different runs of the instrumented software, with each run provided a different set of valid input values. The suite of test cases and test models introduced in this section are designed to exhaustively test the complete implementation of IAML, and therefore form an ideal suite of test data for this process. These code coverage metrics may therefore be automatically captured while these tests are evaluated.

For example, the code generation templates in Section 7.6 may be annotated with four different types of coverage through this aggregation, as illustrated in Figure 7.10: templates that are never used; templates that are generated, but never executed; templates that are executed by the web server; and templates that are executed by the client (web browser). This figure shows that the source code generated by this particular template is executed on both the client and the server; parts of the generated code are never executed; and parts of the template are never used to generate any source code.

The code coverage approach has also been used to evaluate the suite of model completion rules used as part of the *Rule Set* component in Section 7.5.2. This is achieved by extending the behaviour of the *insertion queue* component of model completion to also keep track of the source rule that inserted each element. Model completion rules that are never used across any test case may therefore be candidates for removal from this suite; conversely, this evaluation has shown that all of the current model completion rules summarised in Appendix **??** are necessary to implement the intended behaviour of the IAML metamodel.



```
 * run a command with some parameters. this could be in inline JS or a JS block.
 * RunInstanceWire: event --> operation
 */
«DEFINE callOperations(Boolean inline) FOR model::wires::RunInstanceWire-»
    /* eContainer == «to.eContainer» */

    «IF inEdges.typeSelect(model::wires::ConditionWire).size != 0»
        /* expand conditions */
        if («EXPAND Conditions::conditionWires(this, inline) FOREACH inEdges.typeSelect(model::wires::ConditionWi
    «ENDIF»

    «IF to.eContainer.metaType.superTypes.contains( model::VisibleThing )-»
        «REM»VisibleThing includes InputTextField and Page«ENDREM»
        «REM»
            is both source and target on the same page? NOTE this
            will also return true for Event-->Operation if they are
            both on the same page.
        «ENDREM»
        «IF onCurrentPage(from, to)-»
            «REM»// on current page (wire id = «id»)«ENDREM»
            do_«operationName(to)»(«EXPAND callOperationParameters(this, inline) FOREACH inEdges.typeSelect(model
        «ELSE-»
            «REM»// not on current page (wire id = «id»)«ENDREM»
            call_remote_event('«safeNameString(containingPage(to).id)»', '«operationName(to)-»'
                «IF inEdges.size>0», «EXPAND callOperationParameters(this, inline) FOREACH inEdges.typeSelect(moc
            );
            «REM»
            store_event('«containingPage(to).id»', 'do_«operationName(to)»'
            «ENDREM»
        «ENDIF-»
    «ELSEIF isDomainAttribute(to.eContainer)-»
        «REM»direct field --> domain attribute«ENDREM»
        store_db('«safeNameString(((model::DomainAttribute) to.eContainer).id)»'
            «IF inEdges.size>0», «EXPAND callOperationParameters(this, inline) FOREACH inEdges.typeSelect(model::
        );
    «ELSEIF isDomainAttributeInstance(to.eContainer)-»
        «REM»direct field --> domain attribute instance«ENDREM»
        «EXPAND operationDomainAttributeInstance(inline, this, (model::DomainAttributeInstance) to.eContainer) FO
    «ELSEIF isDomainObjectInstance(to.eContainer)-»
        «REM»direct field --> domain object instance«ENDREM»
        «EXPAND operationDomainObjectInstance(inline, this, (model::DomainObjectInstance) to.eContainer) FOR to»
    «ELSE-»
        alert('unknown target operation type in «to.eContainer.metaType.name»');
        «EXPAND exception FOR throwException("Unknown target operation type in '" + to.eContainer.metaType.name +
    «ENDIF-»

    «IF inEdges.typeSelect(model::wires::ConditionWire).size != 0»
        }
    «ENDIF»
«ENDDEFINE»
```

Legend

🟥	Never executed
⬜	Generated output
🟨	Executed on server (PHP)
🟩	Executed on client (Javascript)

Figure 7.10: Illustrating code coverage of code generation templates by annotating client-side and server-side functionality

7.10 Conclusion

In this chapter, the proof-of-concept implementation of a modelling environment for the development, code generation and verification of IAML model instances has been discussed. In particular, the implementation has been designed as the combination of a number of independent components, which are supplied as OSGi bundles. This implementation will be used to evaluate both the modelling environment and the IAML modelling language itself, as discussed in the next chapter.

Chapter 8

Evaluation

Throughout this thesis the design, description and proof-of-concept implementation of the Rich Internet Application modelling language IAML has been discussed. This chapter will focus on the evaluation of this research and these contributions in terms of the five evaluation criteria discussed earlier in Section 5.1.4.

8.1 Feature Comparison

As discussed in Section 2.3, one of the first steps of this research involved the identification of features that a RIA modelling language should possess. These features were then used to evaluate the range of existing modelling languages for web applications, as published in Wright and Dietrich [319]. This feature comparison will now be re-evaluated on the current IAML implementation, in order to compare this language with existing web application modelling languages, and each evaluation will be reflected in the feature comparison table illustrated later in Table 8.2.

- **Events:** The concept of an event-condition-action rule is a fundamental concept of the IAML metamodel, represented as the associations between Events, Conditions and Actions and expressed as an ECA Rule. Arbitrary events cannot be defined by the model developer, as this would require the inclusion of an event modelling language as discussed earlier in Section 5.6. IAML does not currently support any time-based events.

- **Browser Interaction:** IAML does not implement any form of browser interaction, such as navigation, cookies, identifying user agents or opening new windows. However, many of these requirements have been conceptually designed to fit into IAML as discussed in the next section, and this effort remains future work.

- **Lifecycle Management:** Lifecycle support in IAML is supported by the definition of Events in Scopes – such as the *onInit* and *onAccess* events of Sessions. There is little support for events at the end of a lifecycle – for example, an Email provides an *onFailure* event – and future work may include defining events such as *onDelete* or *onTimeout*.

- **Users:** As discussed in Section 5.9, IAML natively supports the definition of role-based access control mechanisms through Roles providing Permissions. However, there is no modelling support for the interactive collaboration between users.

Modelling Concept	Existing Metamodel	Level of Reuse
Primitive types	XML Schema [295]	Complete
Domain types	EMF Ecore [275]	Partial
Operations	UML activity diagram [224]	Adapted
Users and security	RBAC [258]	Adapted

Table 8.1: Reuse of existing metamodels in the IAML metamodel

- **Security:** As discussed in Section 5.9, access to Scopes can be restricted using combinations of Gates, Access Control Handlers and Login Handlers. Future work includes the development of model verification techniques to verify the security of these approaches.
- **Databases:** IAML supports the definition of primitive types using the XML Schema datatype metamodel, and complex domain types using the EMF Ecore metamodel. Both of these metamodels are platform-independent, allowing for domain-specific data to be represented across any type of database. However, there is not yet any support for common database concepts such as views, joins or sub-selects. There is also no support for modelling offline data, or for uploading files.
- **Messaging:** Messaging is supported through RSS feeds and Emails. As discussed earlier in Section 8.2 support for other types of web services has been designed, but these modelling concept have not yet been implemented.
- **UI Modelling:** A limited range of user interface components, such as Input Text Fields and Maps, are natively supported in the metamodel, as illustrated earlier in Section 5.12. However, there is no way to define new types of user interface components, except through the EMF-based extension of the IAML metamodel. Complex user interface functionality can be simplified into design patterns represented as Wires. Low-level visual properties such as fonts and colours are not part of the IAML metamodel, and generated applications are instead supplied with additional templates, scripts and stylesheets.
- **Platform Independence:** As an important design goal of IAML, the language is designed to be platform-independent with distinct components. Both the metamodel and the model instances are serialised in the XMI format, permitting integration with other model-driven approaches. The proof-of-concept implementation of the code generation component only supports a single set of target platforms, but future work includes focus on additional platforms[1].
- **Standards:** Standardised technologies and formats have been used wherever possible in the development of the IAML metamodel and implementation, and a summary of these reused metamodels is provided in Table 8.1. The implementation of each component is provided as OSGi bundles, and reuse existing implementations wherever possible.
- **Use of metamodels:** The IAML metamodel satisfies both the metamodelling architecture and viewpoint architecture of the MDA, as discussed in Section 3.1.5.
- **Verification:** The proof-of-concept implementation of IAML implements model instance verification using four different technologies, as discussed earlier in Section 7.7. Each technology

[1]This future work is discussed in further detail later in Section 9.2.6.

Feature Category	**WebML**	**UWE**	**W2000**	**IAML**
Events	Ok	-	Poor	Ok
Browser Interaction	Poor	-	-	-
Lifecycle Management	Poor	Good	Poor	Ok
Users	Good	Poor	Poor	Ok
Security	Ok	Ok	Poor	Good
Database Support	Good	Ok	Poor	Poor
Messaging	Good	Poor	Ok	Ok
UI Modelling	Poor	Ok	Ok	Good
Platform Independence	Excellent	Excellent	Good	Excellent
Standards Support	Poor	Excellent	Excellent	Ok
Use of Metamodels	Poor	Excellent	Excellent	Excellent
Verification	Ok	Ok	-	Excellent
Software Support	Good	Ok	Poor	Good

Table 8.2: A comparison of IAML against existing modelling language support for the general feature categories of modelling Rich Internet Applications, adapted from Wright and Dietrich [319]

uses a different verification language with different performance characteristics and resource requirements, allowing a model instance developer to selectively evaluate certain classes of constraints as necessary throughout the development lifecycle.

- **Software Support:** The IAML metamodel has been implemented in Chapter 7 using a wide range of technologies into a proof-of-concept implementation, and this environment can be used to develop web applications. This implementation is provided as free software under an open-source license.

These evaluation results are reflected in Table 8.2, adapted from Table 2.1 earlier in Section 2.4.7. It is important to note that the other languages have not been re-evaluated since it was first published; for example, WebML has recently proposed additional support for more client-side interactivity and user interface components [**?**, 312].

This comparison table suggests that the IAML metamodel and its consequent proof-of-concept implementation satisfy more of the features necessary to model Rich Internet Applications than any other existing language. However, this comparison table also illustrates that there are a number of features, such as database support and standards support, which require attention in the future.

8.2 Modelling Requirements

To improve on the accuracy of the previous evaluation, each of the detailed requirements of RIAs used in the design of the metamodel [318] can be individually evaluated against the current implementation of IAML. These requirements represent the 46 requirements of *Basic RIAs*, and their evaluations are provided here in Tables 8.3 and 8.4.

The implementation of a particular requirement is decomposed into four activities, as illustrated in the legend of Table 8.5: the design of the requirement (**D**); the implementation of the requirement in the IAML metamodel (**M**); the implementation of the requirement with code generation templates, visual syntax and test cases (**I**); and the validation of the requirement using the *Ticket 2.0* benchmarking application (**V**). These four activities represent the steps necessary in the hybrid modelling language software process model, discussed earlier in Section 5.1.3; that is, a requirement must be designed before it can be implemented, and implemented before it can be evaluated.

This detailed evaluation illustrates that all of the detailed modelling requirements of *Basic RIAs* have been considered in the design of the IAML metamodel, but due to time constraints they have not all been fully implemented and validated. Their implementation remains future work, and it is expected that the implementation of the remainder of these modelling requirements would improve each element in the feature comparison evaluation of IAML in Table 8.2 to "excellent".

8.3 Benchmarking Application Implementation

As discussed in Section 2.5, the benchmarking application *Ticket 2.0* was developed in order to evaluate the functionality of a RIA modelling language within a single application. This application combines all of the detailed feature requirements of a *Full RIA* into a single application [318], and can be used to benchmark many different languages by comparing different implementations of the same application.

Such a comparison forms one of the evaluation criteria of the IAML proof-of-concept implementation against a web application implemented manually – that is, one developed with conventional web programming languages such as PHP and Javascript, without using any model-driven technologies. The system metrics discussed earlier in Section 2.6.3 will be evaluated against each application to compare them; this is necessary as Symfony applications are implemented in a variety of different scripting and configuration languages, whereas IAML applications are implemented as XMI model instances with additional templates.

The development metrics will also be split based on generated code, library code, and manually-written code in order to highlight the manual effort necessary for each application, and illustrate how much functionality is generated or provided by the framework. In Appendix **??**, these metrics are re-evaluated against each of the different languages used in each implementation.

8.3.1 Implementation in Symfony

To develop the manual implementation of Ticket 2.0, the Symfony framework [240] was selected as a basis of the approach, as this framework provides a clean approach to web application design through the use of modules, plugins, actions and views. Using a framework to implement the application is appropriate, as complex web applications are rarely implemented from scratch.

The Symfony framework is implemented in the PHP general-purpose language, and provides web applications using HTML, Javascript and CSS. Symfony uses the YAML language to simplify the configuration of framework functionality, as YAML provides a simple human-readable text-based representation of tree-based data [**?**]. Symfony also utilises a significant number of other open-source frameworks, such as Propel or Doctrine for database access [240]; PHPUnit for unit testing [321]; and Prototype or jQuery for a client-side Javascript framework [246, 181].

The implementation of Ticket 2.0 in Symfony as *Ticketsf* was fairly straightforward, and released under an open-source license online[2]. All of the requirements of Ticket 2.0 [318] were successfully

[2]The source code of *Ticketsf* is available online through Sourceforge, and is available under the Eclipse Public License [70] at `http://sourceforge.net/projects/iaml`.

#	**Requirement**	**D**	**M**	**I**	**V**	**IAML Support**
Data						
D1	Static Pages	✓	✓	✓	✓	Frame
D2	View Data	✓	✓	✓	✓	Domain Iterator
D3	Update Data	✓	✓	✓	✓	Domain Iterator
D4	Pagination	✓	✓	✓		Domain Iterator
D5	Provide Data Feed	✓	✓	✓	✓	Frame (RSS only)
D6	Use Web Services	✓	✓	✓		Frame (RSS only)
D9	Web Service Provider	✓				Frame
D10	Uploading Files	✓				Datatypes
D11	Access Server Data	✓	✓	✓	✓	Value, Domain Attribute Instance
D12	Local Variables/Data	✓	✓	✓	✓	Value, Temporary Variable
D13	Cookies	✓				A Value within a new Scope
Events						
E1	Scheduled Events	✓				A timed Event
E2	Client Timer Support	✓				A timed Event
E3	Server Timer Support	✓				A timed Event
E4	Async Form Validation	✓	✓	✓	✓	Datatype validation
E5	Client Form Validation	✓	✓	✓	✓	Datatype validation
E6	Server Form Validation	✓	✓	✓	✓	Datatype validation
E8	Browser-Based Chat	✓				Implemented manually
Users and Security						
S1	User Authorisation	✓	✓	✓	✓	Access Control Handler
S2	Session Support	✓	✓	✓	✓	Session
S3	User Logout	✓	✓	✓	✓	Login Handler
S4	Automatic User Auth	✓				Login Handler extension (see D13)
S5	User Security	✓	✓	✓		Permission
S6	Group Security	✓	✓	✓	✓	Role
S7	Security Levels	✓	✓	✓	✓	Role-based Access Control
S8	Single Sign-In Solutions	✓	✓	✓		`iamlOpenIDURL` type through Gate
S9	Personalisation	✓	✓	✓	✓	Domain Attribute
User Agents						
A1	Browser Identification	✓				Values within a new "browser" Scope
A2	User Redirection	✓	✓	✓	✓	ECA Rule to a Frame
A3	Multiple Browser Support	✓	✓			Additional code generation templates
A4	Multiple Outputs	✓	✓	✓	✓	Frame: HTML, RSS
A5	Client-Side Application	✓	✓	✓		Visible Thing
A7	Back Button Control	✓				Event
A10	Navigation Control	✓				Hash Fragment (like Query Parameter)

Table 8.3: Evaluation of the modelling requirements of *Basic RIAs* against IAML (1), adapted from Wright and Dietrich [318]

#	Requirement	D	M	I	V	IAML Support
Interaction						
T1	E-Mailing Users	✓	✓	✓	✓	Email
T2	E-Mail Unsubscription	✓				Email with library support
T3	Mobile Phone Comm.	✓				Similar to Email
User Interface						
U2	Client-Side Scripting	✓				New client-side Events
U3	Drag And Drop	✓				Event or Wire
U4	Loading Time Support	✓				Event
U5	Keyboard Shortcuts	✓				Event
U6	Opening New Windows	✓				ECA Rule attribute
U7	Pop-Up Dialog Boxes	✓	✓	✓		*alert* Operation
U8	Runtime Interface Updates	✓	✓	✓	✓	Only after callbacks
U10	Modal Dialogs	✓				Visible Thing attribute
U12	Provide External Libraries	✓	✓	✓	✓	IAML runtime library

Table 8.4: Evaluation of the modelling requirements of *Basic RIAs* against IAML (2), adapted from Wright and Dietrich [318]

Key	Requirement	Count
D	Designed: Introduced in Chapter 4	46 (100%)
M	Modelled: Implemented in the EMF metamodel	27 (59%)
I	Implemented: Tested with test cases and examples	26 (57%)
V	Validated: Successfully implemented in the Ticket 2.0 application	20 (43%)

Table 8.5: Legend to the modelling requirements evaluation of IAML

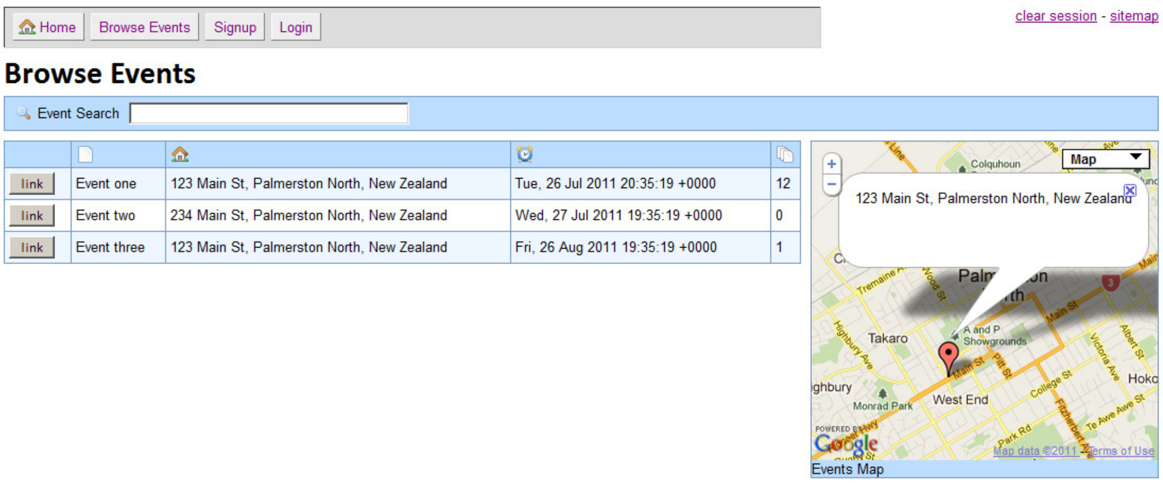
implemented in this application. A number of issues were discovered during the implementation of the language, and these issues were used as inspiration for portions of the IAML design. These issues included:

1. Particular components, such as login over SSL, PDF creation and internationalisation (“i18n”), were supported natively or as *Symfony plugins*. This meant that much of the development effort focused on integrating plugins, rather than implementing components from scratch.

2. Some application requirements such as the Flash-based MP3 player were not available as Symfony plugins, but the development of new plugins to satisfy these goals was fairly straightforward.

3. When designing the ticketing client-side application, an identical server-side application had to be implemented simultaneously. Consequently, some portions of the business logic had to be reimplemented in both PHP and Javascript, because there was no easy way to translate between the two. Web application testing frameworks such as PHPUnit [321] and JWebUnit [132] may be used to verify the consistency of these logics.

4. Each of the different secured parts of the application – for example, the public site, the user site, and the administrator site – are each implemented as a separate “application” according to Symfony. However, each application is intended to work independently of the other, making it difficult to integrate these sites together.

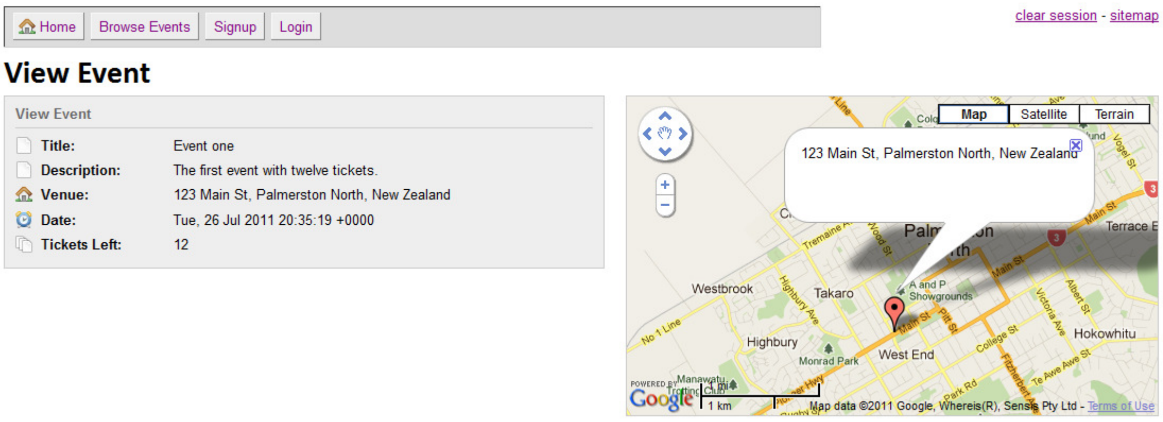


Browse Events

link	Event one	123 Main St, Palmerston North, New Zealand	Tue, 26 Jul 2011 20:35:19 +0000	12
link	Event two	234 Main St, Palmerston North, New Zealand	Wed, 27 Jul 2011 19:35:19 +0000	0
link	Event three	123 Main St, Palmerston North, New Zealand	Fri, 26 Aug 2011 19:35:19 +0000	1

123 Main St, Palmerston North, New Zealand

Figure 8.1: A screenshot of the *Browse Events* page implemented in Ticketiaml



View Event

View Event

Title:	Event one
Description:	The first event with twelve tickets.
Venue:	123 Main St, Palmerston North, New Zealand
Date:	Tue, 26 Jul 2011 20:35:19 +0000
Tickets Left:	12

123 Main St, Palmerston North, New Zealand

Figure 8.2: A screenshot of the *View Event* page implemented in Ticketiaml

8.3.2 Implementation in IAML

The implementation of Ticket 2.0 in IAML as *Ticketiaml* was performed once the IAML metamodel had been completed as discussed in this thesis. The implementation of *Ticketiaml* is similarly released under an open-source license, with the source code available online[3]. The IAML model instance for Ticket 2.0 is also provided in visual and textual form within this thesis in Appendix **??**.

A screenshot of the Ticketiaml implementation of the page *Browse Events* is illustrated here in Figure 8.1. On this page, a list of all of the events in the system are displayed, along with an interactive map showing their addresses. If one of these listed events is selected, the *View Event* page in Figure 8.2 is displayed, which displays more information about the particular event, along with a more detailed map of its address.

As the design of IAML includes support for common reusable patterns of web applications – such as access control, data access, mashups and visual element design – the implementation of these features in Ticketiaml were straightforward. However, other concepts that are not yet implemented as discussed in Section 8.2 – such as populating foreign keys, or overriding or extending the model

[3]The source code of *Ticketiaml* is hosted within the IAML project, and is available under the Eclipse Public License [70] at `http://openiaml.org/ticket20/`.

completion process – were difficult, and their implementation required more effort.

Model completion was used heavily in the implementation of Ticketiaml, such as automatically completing the contents of Input Forms through Sync Wires, and utilising the generated functionality of Login Handlers to protect Scopes. However, in two situations the intent of model completion rules was not clear, and the developer had to read the source code of the model completion rules themselves[4].

Unfortunately, the full requirements of the Ticket 2.0 application could not be completed using the current IAML implementation, and only 22 of the original 123 requirements of the application could be implemented. A number of system resource limits were hit during the implementation of the model instance[5], and no resources were available to optimise the implementation of the language further.

The underlying technologies for model completion and code generation routinely deal with model instances larger than those generated through IAML [**?**], and there are no known theoretical or design limits that would prevent the full implementation of Ticket 2.0 using IAML. Additional development on the IAML implementation is therefore necessary, and performance has already been a development focus[6]. It is important to note that performance is not a focus of this research, however there are a number of approaches that may be used to resolve these problems, as discussed later in Section 9.2.7.

8.3.3 Re-implementation in Symfony

As it would not be possible to evaluate the Symfony-based implementation of Ticket 2.0 with the incomplete IAML-based implementation in any meaningful way, the decision was made to reimplement Ticket 2.0 in Symfony to match the implemented features of *Ticketiaml*. This smaller implementation of Ticket 2.0 was implemented in Symfony 1.4.8 as *Ticketsf-mini*, and exactly matches the features of *Ticketiaml*. The implementation of *Ticketsf-mini* is similarly released under an open-source license, with the source code available online[7].

This reimplementation was fairly straightforward and once again, highlighted the same issues discovered during the implementation of *Ticketsf*; namely, client-side and server-side logic must be implemented in two different languages (PHP and Javascript) and kept consistent throughout the development process. Screenshots of the Ticketsf-mini implementation of the pages *Browse Events* and *View Event* are illustrated here in Figures 8.3 and 8.4.

8.3.4 System Metrics Evaluation

These two different implementations of the same benchmarking application can be compared through the system size metrics described earlier in Section 2.6.3, and these metrics are provided here in Table 8.6. The file-specific metrics can be broken down based on the proportion of manual development effort[8]; the results of these metrics against the manual implementation effort is provided in Table 8.7.

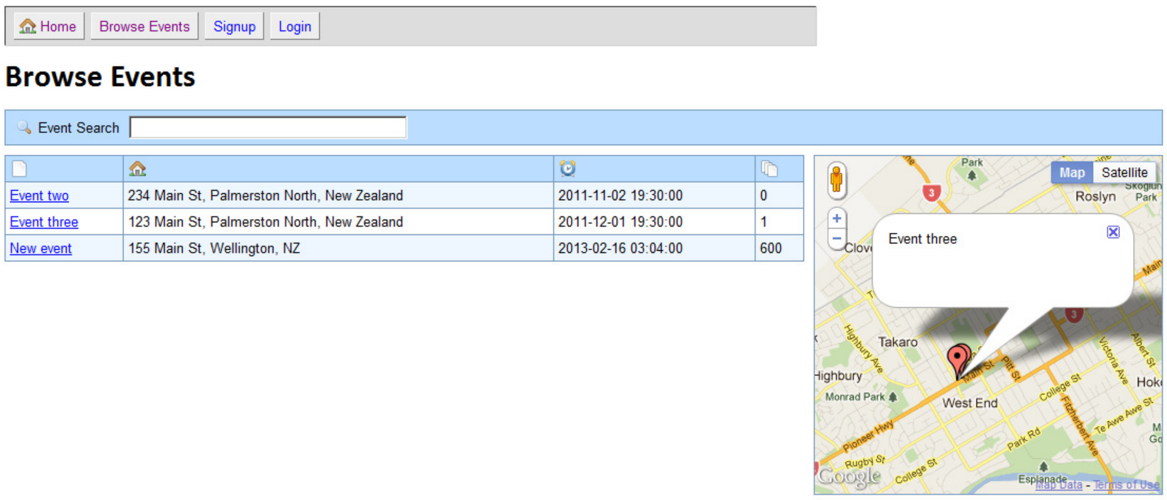
[4]**TODO:** In order to improve the usability of model completion, the model completion rule intentions should be available within the model instance development environment itself.

[5]For example, the architecture-specific maximum heap spaces for the Sun Java virtual machine were repeatedly reached when performing model completion and code generation [159].

[6]For example, issues 184, 210 and 261 have focused on the performance of model completion, and issues 117 169 and 181 have focused on the performance of code generation.

[7]The source code of *Ticketsf-mini* is hosted within the IAML project, and is available under the Eclipse Public License [70] at `http://openiaml.org/ticket20/`.

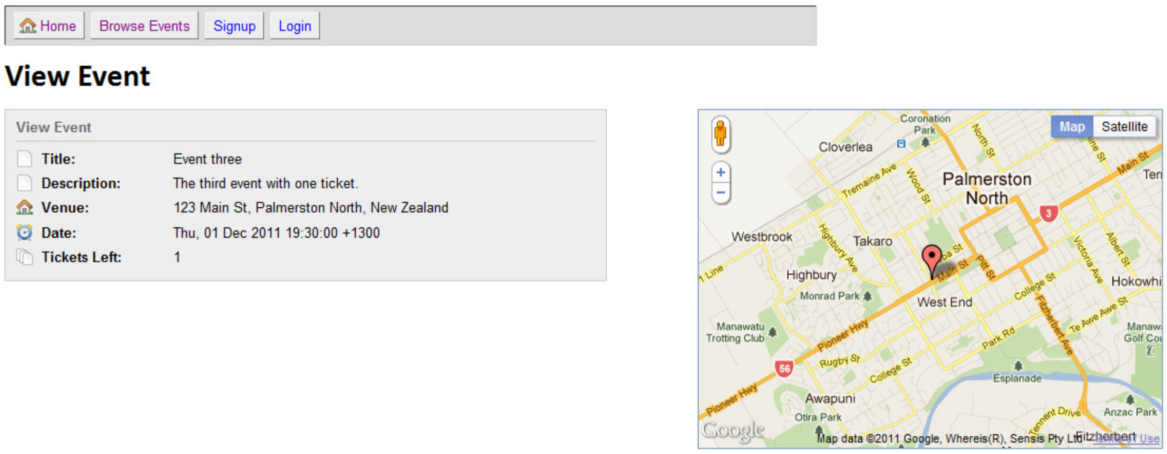
[8]In this evaluation, *manual effort* is defined as code which requires manual development effort in order to implement a requirement. This includes new source code files and manual changes to generated code, but does not include generated source code that does not require any modification, nor runtime libraries provided by the application framework.



Browse Events

Event two	234 Main St, Palmerston North, New Zealand	2011-11-02 19:30:00	0
Event three	123 Main St, Palmerston North, New Zealand	2011-12-01 19:30:00	1
New event	155 Main St, Wellington, NZ	2013-02-16 03:04:00	600

Figure 8.3: A screenshot of the *Browse Events* page implemented in Ticketsf-mini



View Event

View Event

Title:	Event three
Description:	The third event with one ticket.
Venue:	123 Main St, Palmerston North, New Zealand
Date:	Thu, 01 Dec 2011 19:30:00 +1300
Tickets Left:	1

Figure 8.4: A screenshot of the *View Event* page implemented in Ticketsf-mini

Metric	**Ticketsf-mini**	**Ticketiaml**	**Difference**
Tasks	22 (18%)	22 (18%)	-
Time (weeks)	1	1	-
NDev	1	1	-
FC (changes)	229	43	-81%
Rev	21	19	-10%
DTech	9	7	-22%
DMedia	1	1	-

Table 8.6: Comparing two Ticket 2.0 implementations using system metrics: overall development effort

Metric	**Ticketsf-mini**	**Ticketiaml**	**Difference**
Files (files)	45	7	-84%
Size (bytes)	49,390	46,998	-5%
NCLOC (lines)	1,010	606	-40%
ALOC	22.44	86.57	286%

Table 8.7: Comparing two Ticket 2.0 implementations using system metrics: manual effort

These values can be used to make a number of observations on each systems' implementation. These metrics suggest that an IAML-based web application requires less manual development effort with fewer implementation technologies than a Symfony-based web application, and the reduction in implementation technologies are illustrated in Appendix **??**. However, as only one benchmarking application has been evaluated, additional evaluations are necessary in order to strengthen this observation.

Compared to the Symfony implementation, the IAML implementation required significantly fewer file changes (*FC*). This is due to the different architectures of each implementation; IAML source code is designed to be provided as a single model instance with additional platform-specific templates and includes, whereas Symfony source code is designed to be provided across a range of subtypes on generated framework code and classes. This is also reflected in the difference in the number of files (*Files*) between the two implementations, highlighted in the file-based metrics of Table 8.7.

The IAML implementation also required fewer lines of code (*NCLOC*) than the Symfony implementation; this may be due to the differences between PHP scripts and XMI instances; the former is designed to be human-editable, whereas the latter is designed to be machine-readable. This difference is also reflected in the average lines of code per file (*ALOC*). These results are also a consequence of the architecture of the IAML implementation; the flexibility of EMF allows for a model instance to span many XMI files if necessary [275, pg. 404–415].

8.4 Metamodelling Metrics

To evaluate the quality of the underlying metamodel, the suite of metamodel metrics defined earlier in Section 2.6.2 will be evaluated against both the IAML metamodel and a suite of other metamodels

Metamodel	*TNP*	*NoC*	*NoAC*	*TNoR*	*TNoA*	*NoD*	*NoE*	*Nav*	*Cont*	*Dat*
UML	1	247	48	480	106	17	13	0.33	0.38	0.18
GMF Notation	1	69	6	31	68	17	13	0.13	0.52	0.69
XSD	1	57	22	125	98	25	20	0	0.32	0.44
OCL	1	52	9	4	1	0	0	0	0.25	0.20
ModelDoc	1	33	4	62	55	1	1	0.55	0.44	0.47
Ecore	1	20	5	48	33	32	0	0.33	0.38	0.41
IAML	9	103	43	126	39	8	7	0.41	0.52	0.24
UWE [**?**]	10	319	59	533	155	19	15	0.36	0.34	0.23
WebML [**?**]	17	122	21	200	205	12	12	0	0.66	0.51

Table 8.8: Using metamodel metrics to evaluate the IAML metamodel against other similar metamodels, adapted from Monperrus et al. [207] and Vépa et al. [288]

related to model-driven development. The results of this evaluation are provided here in Table 8.8[9]; in this table, each metric has been applied against the most recent Ecore representation of each metamodel.

These metamodel metrics are also applied to Ecore-based implementations of the UWE and WebML metamodels. The UWE metamodel used in this evaluation is the most recent Ecore version of the UWE metamodel as provided through the *UWE4JSF* project proposed by [**?**]. Since the UWE metamodel is defined as an extension to the UML metamodel, the UWE metamodel includes all of the packages, classes and references of the UML metamodel; for example, the UWE metamodel defines the class *PresentationElement* as a subtype of the UML class *Class*.

At the time of writing, the implementation of the WebML metamodel in the latest version of WebRatio (6.1.0) was provided using a proprietary MDA-incompatible architecture (DTDs and XMLs of units) which is not unified into an overall metamodel. However, the AspectWebML project proposed by [**?**] extracted an Ecore-based metamodel from these DTDs, based on work by [**?**]. The WebML metamodel uses a lot of attributes, since they do not distinguish between a PIM and PSM; for example, concepts such as server port numbers are defined as attributes on model elements, whereas they should be defined as platform-specific configuration attributes provided to the code generation templates, or as arguments to the runtime environment.

The metamodelling metrics for the number of classes (*NoC*) and abstract classes (*NoAC*) are also illustrated here in Figure 8.5. These metrics show that the IAML metamodel relies heavily on abstract classes, with a higher proportion of abstract classes than other metamodels. In particular, the IAML metamodel heavily uses model concepts to represent the source and targets of relationships; for example, the interfaces of Requires Edges Source and Requires Edge Destination are used as the source and target of a Requires Edge, as Ecore metamodels do not support the concept of type unions.

The different design architectures of each modelling language are also illustrated here in Figure 8.6, showing the results of the total number of references[10] (*TNoR*) and attributes (*TNoA*) for each modelling language. This illustrates that the WebML metamodel is heavily oriented around attributes, whereas the majority of other languages are oriented around references, and the IAML metamodel fol-

[9]In this table, the results produced by Monperrus et al. [207] could not be reproduced, as they had modified the original Ecore metamodel. Consequently, the metrics provided here have been regenerated from the most recent version of the Ecore metamodel.

[10]**TODO:** The *TNoR* metric is linearly correlated with the *NoC* metric, with an $R^2 = 0.94$. Is this significant enough to mention here?

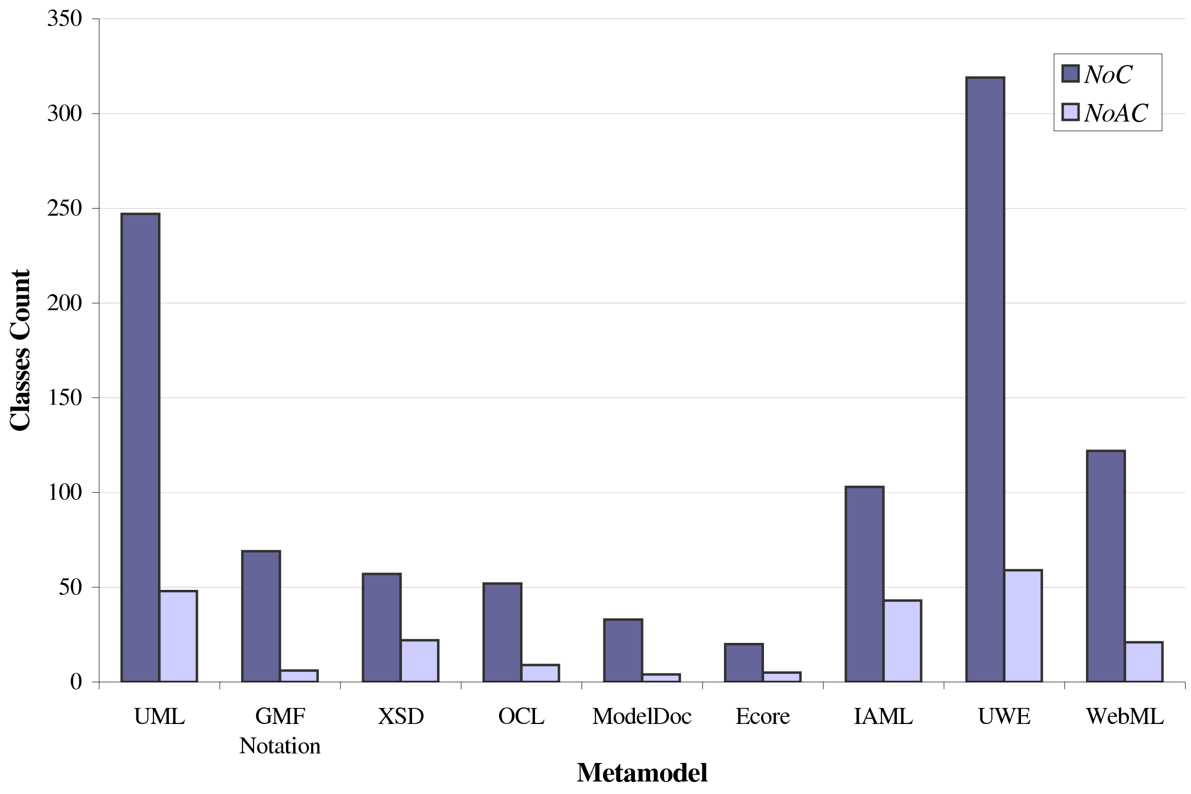


Figure 8.5: Comparison of the number of classes (*NoC*) and abstract classes (*NoAC*) of the IAML metamodel against other similar metamodels

lows a similar architecture. This is also illustrated by the *Dat* metric, which represents the proportion of attributes to structural features in the metamodel.

As discussed by Monperrus et al. [207], the metrics for *Nav*, *Cont* and *Dat* are believed to give information on the modelled domain and the modelling styles and practices, but a balance between these values has not been determined. The *Nav* metric represents the opposite navigability of references through eOpposite, simplifying development of model-driven tools such as code generators. The *Cont* metric represents the proportion of references that are containment references, in that one element can "contain" another element, and may illustrate the modelling architecture used. However, it is not desirable for this metric to be at either of the extremes, suggesting that a modelling language is too unstructured, or conversely too structured.

Importantly, the metamodelling metrics evaluated in this section highlight that the IAML metamodel uses less classes, abstract classes, references and attributes than the existing web application modelling languages of WebML and UWE. This may suggest that the IAML metamodel is simpler than these existing metamodels, but a functionality-based evaluation between these languages is necessary to conclude that this simplicity does not negatively impact the functionality of the language.

8.5 Visual Model Evaluation

As discussed earlier in Section 5.1.4, the visual notation of the IAML metamodel will be evaluated according to two evaluation criteria discussed earlier in Section 4.10, in order to judge the effectiveness of this proposed visual syntax.

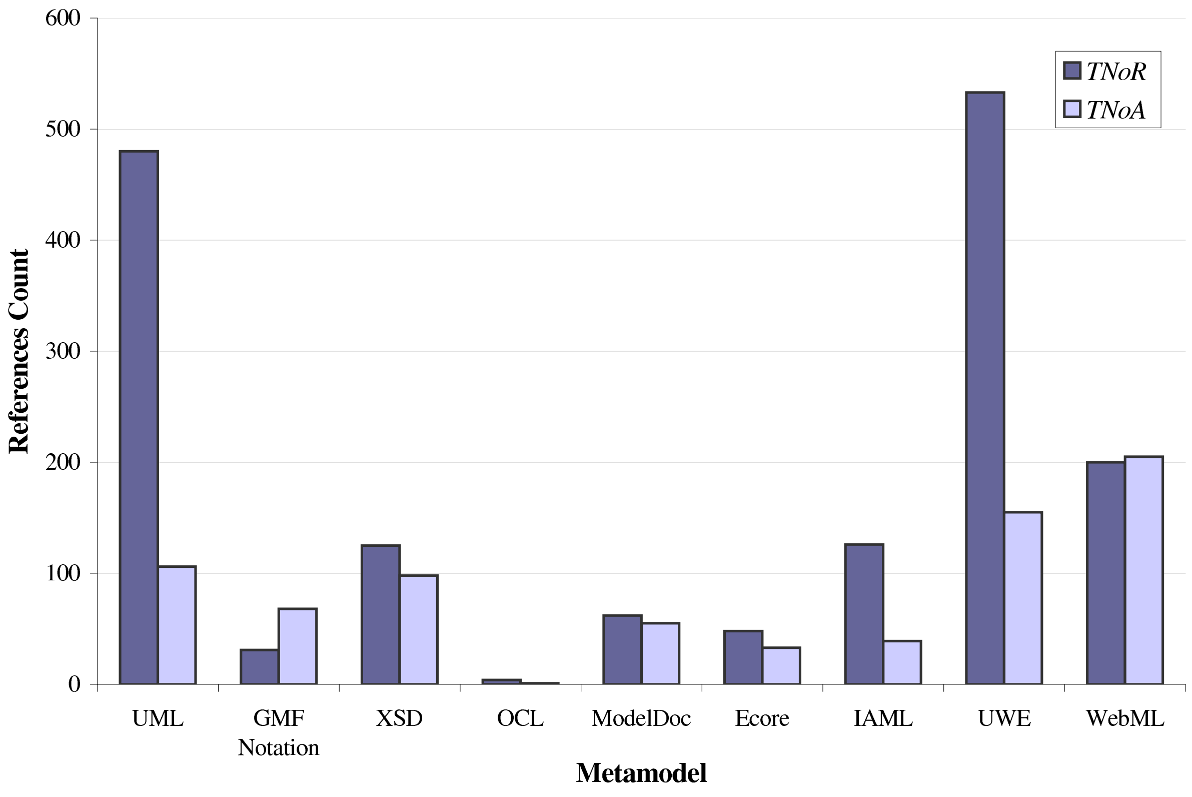


Figure 8.6: Comparison of the total number of references (*TNoR*) and attributes (*TNoA*) of the IAML metamodel against other similar metamodels

8.5.1 Notation Information Capacity

In Table 8.9, the visual syntax of IAML is evaluated for cognitive effectiveness against the information capacity of each visual syntax variable, discussed as earlier in Section 4.10.1. This evaluation shows that the visual syntax of IAML does not exceed any of the maximum information capacities for each of these syntax variables.

For example, there are currently only five shapes used to distinguish between model shapes; only two levels of textual brightness – black, and gray – are used; no textures are used as part of the notation; and seven colours from the Eclipse palette are used for the basis of default colours. However, each of these variables may be individually overridden at request of the user. A number of these variables are not restricted by the syntax, and are defined by the user instead – for example, there is no limit to the number of elements placed horizontally or vertically in a given model instance view.

This evaluation also highlights that the IAML visual syntax only uses five different shapes, yet Moody [208] argues that an unlimited number of shapes can be used in the visual syntax for the language in order to improve usability. In the future, this range of shapes should be extended to new shapes for other modelling elements[11]; for example, Decision Nodes could be represented using a diamond shape, or Domain Sources represented using a database shape.

8.5.2 Cognitive Dimensions Evaluation

The *cognitive dimensions* framework [116] may be used to evaluate a visual notation in a subjective manner, as discussed earlier in Section 4.10.2 and illustrated by Grundy et al. [121]. This evaluation

[11]Issue 231: *Add more shape styles for the visual syntax of model elements.*

Variable	Power	Capacity	IAML Maximum
Horizontal position (x)	Interval	10–15	n/a
Vertical position (y)	Interval	10–15	n/a
Size	Interval	20	n/a
Brightness	Ordinal	6–7	2
Colour	Nominal	7–10	7
Texture	Nominal	2–5	0
Shape	Nominal	Unlimited	5
Orientation	Nominal	4	1

Table 8.9: Evaluation of the information encoding capacity of IAML visual notation, adapted from Moody [208]

has shown that the visual representation of IAML broadly satisfies each of the thirteen dimensions of this framework. Many of these dimensions – such as secondary notation and visibility – are automatically addressed to an extent through the GMF framework. This evaluation is discussed in further detail in Appendix **??**.

8.6 Conclusion

In this chapter, the design and implementation of IAML has been evaluated within five evaluation criteria. The results of these evaluations can be summarised as follows:

1. The IAML metamodel and its proof-of-concept implementation implement more of the features necessary to model Rich Internet Applications than any existing modelling language for web applications. In particular, IAML has strong support for event modelling; platform-independence; reusing metamodels; and the verification of model instances.

2. All of the *Basic RIA* modelling requirements have been used in the design of the IAML metamodel. However due to resource and time constraints only 59% of these original requirements have been implemented in the proof-of-concept modelling environment, and only 43% have been validated as part of the Ticket 2.0 benchmarking application.

3. Since the IAML metamodel does not fully implement the requirements of a *Full RIA*, Ticket 2.0 cannot be fully implemented; resource constraints also prevented the implementation of this application. However, preliminary results from this evaluation suggest that an IAML-based implementation is comparable to a Symfony-based implementation, and reduces the number of development technologies necessary for the implementation of a Rich Internet Application.

4. In terms of metamodelling metrics, the IAML metamodel can be considered simpler than the existing metamodels of UML, UWE and WebML, but more complex than simpler metamodels such as Ecore and OCL. These metrics also suggest that the overall architecture of the IAML metamodel is similar to the architecture of other metamodels.

5. The proposed visual syntax of the IAML metamodel can be considered effective, through evaluating the maximum information capacity of the variables of its visual syntax, and evaluating the overall implementation subjectively against the Cognitive Dimensions framework.

TODO Is there anything else that I should conclude the evaluation chapter with?

Chapter 9

Conclusions and Future Research

In the thesis, the requirement capture, design, development and proof-of-concept implementation of IAML – a new modelling language for the development of Rich Internet Applications – has been discussed in detail. In this chapter, the main contributions of this research to the fields of software engineering and model-driven development will be summarised, along with a brief discussion on some outstanding research questions and future work for this research.

9.1 Research Contributions and Conclusions

9.1.1 Requirements for Modelling RIAs

As discussed earlier in Section 2.3 (pg. 13), the first step for this research was to identify the requirements that an RIA modelling language should satisfy, resulting in thirteen feature categories. This step was necessary as existing web modelling language evaluations were not concerned with the interactive requirements of RIAs.

These categories were then used to evaluate existing modelling languages in terms of their suitability for modelling RIAs, and published by Wright and Dietrich [319]. This evaluation found that no existing web application modelling language was expressive enough to support the requirements of modelling RIAs, and the level of support for modelling events and user interfaces was particularly poor.

Seven representative existing RIAs were also analysed to identify the specific requirements of a RIA modelling language, by identifying the individual use cases that may be involved in the development of an RIA. The functionality of these representative applications were therefore decomposed into a set of 69 use cases, published in this thesis as Appendix **??**.

These use cases were then analysed and translated into a suite of 59 fundamental requirements of RIAs, as published by Wright and Dietrich [318]. This list of requirements forms an important contribution by providing formal definitions of RIA-specific functionality. These requirements were used in the design of the IAML metamodel, and in the future could be used to evaluate existing modelling languages with a greater level of detail than the previously discussed feature comparison.

9.1.2 A Benchmarking Application for Rich Internet Applications

A single RIA benchmarking application, called *Ticket 2.0* [318] was then developed by combining each of these 59 requirements of RIAs into a single application. This forms an important evaluation criteria into the development of RIA modelling languages by proposing a standardised benchmarking application that may be used to evaluate the different implementations of RIA features within different modelling languages.

To illustrate that this hypothetical web application represented a realistic RIA for benchmarking purposes, this application was implemented using the web application framework Symfony [240] in Section 8.3.1 (pg. 218). This application successfully implemented all of the requirements of *Ticket 2.0*, and the subsequent evaluation found that one of the key problems in the conventional implementation of RIAs is in keeping client-side and server-side logic synchronised across two different languages (PHP and Javascript).

Ticket 2.0 was also implemented using the proof-of-concept implementation of IAML in Section 8.3.2 (pg. 221) in order to identify the strengths and weaknesses of the current implementation of IAML. By using design concepts of IAML such as wires and users, it was found that most of the necessary scaffolding for the application could be generated automatically through model completion, also suggesting that the use of model completion is beneficial within the development of RIAs.

Through the proposal of system metrics earlier in Section 2.6.3 (pg. 25), the development expressiveness and productivity of these two approaches could then be evaluated. This subsequent evaluation illustrated that fewer development technologies are necessary for the implementation of an RIA, and that less manual development effort is necessary to implement a subset of a requirements of *Ticket 2.0* when compared to a similar implementation using Symfony. This evaluation also indicated that additional optimisation effort on the proof-of-concept implementation of the language is necessary as future work.

9.1.3 Modelling Language for Rich Internet Applications

The *Internet Application Modelling Language* (IAML) represents a significant contribution as a modelling language for describing and developing RIAs. As evaluated in Section 8.1 (pg. 215), IAML satisfies the design goal of platform-independence; reuses a variety of existing standards where appropriate; and satisfies both the metamodelling architecture and viewpoint architecture of the MDA [214, 163]. This language has been designed to support all of the concepts of modelling Basic RIAs, and the full description of this language forms Chapter 5 and Appendix **??** in this thesis.

The design of IAML includes a number of novel modelling concepts, including some adapted from existing modelling approaches, in order to reduce the effort necessary for web application developers. Each of the concepts described in Chapter 5 represent an important contribution, but a number of these are particularly noteworthy:

1. **Logic-based Core:** The core of the language is derived from first-order logic concepts, and supports Functions, Conditions, Predicates and Complex Terms (pg. 90). By defining these concepts in terms of an existing rigorous mathematical definition, this metamodel core provides a strong foundation for the rest of the language.

2. **Type System:** IAML defines a rich underlying type system for model instances, composed of primitive type modelling adapted from XML Schema datatypes [295] (pg. 99), and complex domain modelling adapted from EMF Ecore [275] (pg. 105). New primitive types may be defined through the derivation of existing types, and the domain modelling approach of IAML supports multiple inheritance.

The type system of IAML is statically checked; objects may only have a single type, allowing for the type validity of a model instance to be checked during development. Types within IAML are also weakly checked; typed objects may be implicitly cast to another type at runtime, and objects may be typed to a *default type* (pg. 102).

3. **Event-Condition-Action (ECA) Rules:** The evaluation of existing modelling languages by Wright and Dietrich [319] found that existing web modelling languages rarely include events as a first-level citizen, despite that RIAs are heavily oriented around events. ECA rules are adapted into IAML to capture this common design pattern, represented as an ECA Rule in Section 5.6 (pg. 115). However, IAML does not support developers defining their own arbitrary events, as this would require the inclusion of an event modelling language.

4. **Wires:** Inspired by the *connection* concept in VisualAge for Smalltalk [183], the concept of a Wire was introduced in Section 5.8 (pg. 122) to support reusing common development patterns for RIAs, such as keeping the values of two model elements synchronised. To support model developers overriding the default behaviour of Wires, this functionality is implemented by evaluating model completion rules against a model instance.

5. **Scopes:** As discussed in Section 5.10 (pg. 130), Scopes permit the model developer to utilise a variety of lifecycle events, such as when a Session is initialised or accessed. Scopes also provides a natural way to define how different elements of data can be differently scoped, along with the associated access and storage semantics of the immediate and indirect contents of each Scope.

6. **Users and Security:** IAML allows model developers to define the potential users of RIAs through the Role-Based Access Control security mechanism [258], represented as Roles and Permissions in Section 5.9 (pg. 126). Through model completion, common functionality such as login and authentication is provided through Login Handlers to reduce the manual development effort necessary for these common RIA use cases.

Through the proposal of metamodelling metrics earlier in Section 2.6.2 (pg. 24), the expressiveness and complexity of the IAML metamodel could be evaluated against a suite of existing modelling languages. This subsequent evaluation in Section 8.4 illustrated that the IAML metamodel is simpler than the existing metamodels of WebML and UWE, and the design of the language follows a similar architecture to many existing languages.

9.1.4 Model Completion

As discussed by Wright and Dietrich [320], a key challenge in the design of a modelling language is in balancing the level of detail expressible in its design; that is, its level of abstractness to its level of flexibility. Software frameworks approach this problem by supporting programming by convention, where documented conventions allow much of the scaffolding to be provided automatically by the framework.

The concept of *model completion* adapts this concept to the model-driven development domain and forms a significant contribution to the field of model-driven development, by representing documented conventions as *non-monotonic* inference rules. These model completion rules may be implemented using the Drools rule engine (pg. 200), and the consistency of this approach is discussed by Wright and Dietrich [320].

Importantly, model completion does not permit any original information in the original model to be removed; this ensures that developer effort can never be overridden. This process may be manually overridden by the developer, and is implemented within IAML by using the Generated Element abstract type as a supertype of all IAML model elements. Model completion is used to complete all of the necessary scaffolding for reusable patterns such as Wires, Login Handlers, and to support the inheritance of Domain Types.

9.1.5 Model Instance Verification

As discussed in Section 3.4 (pg. 52), model instance verification is the process of identifying invalid models from syntactically correct models based on desired correctness constraints. In many cases it is preferable to identify errors within a system model than in its implementation, as the relative cost of fixing and error increases over time [244, pg. 197].

In this thesis, model instance verification was decomposed into three different categories, based on the expressiveness of different verification languages: function-based languages; relations-based languages; and through model checking, with each category possessing different performance characteristics and resource requirements. By implementing model instance verification constraints within each of these different categories, a model developer may selectively evaluate correctness according to their available resources.

Within the proof-of-concept implementation of IAML, model instance verification has been implemented through four different technologies as discussed in Section 7.7 (pg. 206). Simple constraints are evaluated using the EMF Validation framework and Checks; more complex constraints are evaluated using CrocoPat; and the intended behaviour of the modelled applications are evaluated by NuSMV using model checking. As discussed later in Section 9.2.9, the usability and understandability of these implementations has not been evaluated, but remains future work.

9.1.6 Evaluation of Model-Driven Technologies

An important design goal of IAML was to provide a proof-of-concept implementation of the modelling language, and such an implementation involves the integration of a number of different technologies. Each of these technologies – such as model completion, model instance verification, and the metamodelling and graphical environments themselves – may be provided by a number of different technology implementations.

The evaluation of many different implementation technologies for each of these model-driven technologies forms an important contribution of this thesis, and is discussed in detail within Chapter 6. Each implementation was evaluated according to the functionality and expressiveness necessary for each technology; the ease at which each technology may be integrated together; and a qualitative measurement of the implementation quality through the OpenBRR open source evaluation framework [231].

The results of this evaluation were used to select a suite of implementation technologies used for the proof-of-concept implementation of IAML. These technologies included the Eclipse Modeling Framework (EMF) for a metamodelling environment; the Graphical Modeling Framework (GMF) for a graphical modelling environment; the openArchitectureWare Xpand language for a code generation framework; the Drools rule engine for the implementation of model completion; and a selection of four different languages to support different forms of model instance verification, in terms of the necessary expressiveness of different constraints.

As one of the design goals of IAML, a visual syntax was also designed in Section 7.4.3 (pg. 193) in order to support the graphical definition of IAML model instances. The effectiveness and simplicity of this visual syntax was evaluated in Section 8.5 (pg. 226) against both its notation information capacity, and against the Cognitive Dimensions framework.

9.1.7 Other Contributions

Through the process of designing, implementing and subsequent verification of a modelling language for RIAs, a number of smaller contributions throughout this research have also been recognised. These will only be briefly mentioned within this section.

An important consideration for the development of model completion rules is in providing adequate and reliable documentation for these conventions [320]. The *ModelDoc* framework described by Wright [317] was developed to allow this documentation to be automatically loaded from the model completion rules themselves; this documentation can then be combined with other sources of documentation into an authorative documentation source. In particular, this documentation source forms the basis of the IAML metamodel reference in Appendix **??**.

During the development of the IAML visual modelling environment using GMF, it was recognised that many of the GMF model instances used to generate these editors shared common features, yet had to be implemented manually. To reduce the development effort necessary to maintain these graphical editors, the *SimpleGMF* modelling language was developed in Section 7.4.3 (pg. 192) to generate GMF model instances automatically.

In order to reduce the complexity of the implementation of each model-driven technology within the proof-of-concept implementation, a model-driven code coverage technique was developed in Section 7.9.1 (pg. 212). By obtaining these metrics through the evaluation of the exhaustive test suite for IAML, any unused source code – including code generation templates and model completion rules – could be removed to reduce the complexity of the implementation. These code coverage metrics could also be used to improve the test suite itself, or to identify elements of the code generation templates that are never executed within a particular platform.

9.2 Future Research

The research involved in this thesis has included contributions from a wide variety of research areas, and has uncovered a number of unanswered research questions which remain future work; in this section, some of these areas of future work will be briefly discussed. The proof-of-concept implementation itself includes a number of outstanding issues[1], some of which will also be summarised in this section.

9.2.1 Further Evaluation of Existing Web Modelling Languages

As discussed by Wright and Dietrich [318], existing web modelling languages have not been re-evaluated against the list of 59 detailed RIA modelling requirements in Section 2.3.2 (pg. 16), with

[1]The list of outstanding issues is available online at `http://code.google.com/p/iaml/issues/list`.

the exception of the evaluation against IAML in Section 8.2 (pg. 217). Many of these languages have continued to evolve and, in some cases, incorporate limited functionality for RIA concepts. For example, recent work for the actively developed WebML environment includes improved support for modelling client-side data and a new event model, as discussed by Fraternali et al. [91]. A re-evaluation of these web application modelling languages would highlight the strengths and weaknesses of the IAML metamodel with respect to recent developments.

While the development and publication of the *Ticket 2.0* benchmarking application was an important original contribution of this thesis, this application specification only describes the design requirements of *Ticket 2.0*, and does not specify the actual functionality of the benchmark. A web application testing framework such as JWebUnit [132] could be used to develop integration test cases that would independently evaluate the functionality of a certain implementation of *Ticket 2.0*, and guarantee the completeness of each implementation; it would also significantly impact on developer flexibility, as such a test suite would enforce many more constraints on an implementation. The test suite could also be used to accurately profile the implemented applications for performance, and different implementations compared to identify potential areas for future optimisation work.

9.2.2 Modelling Full RIAs

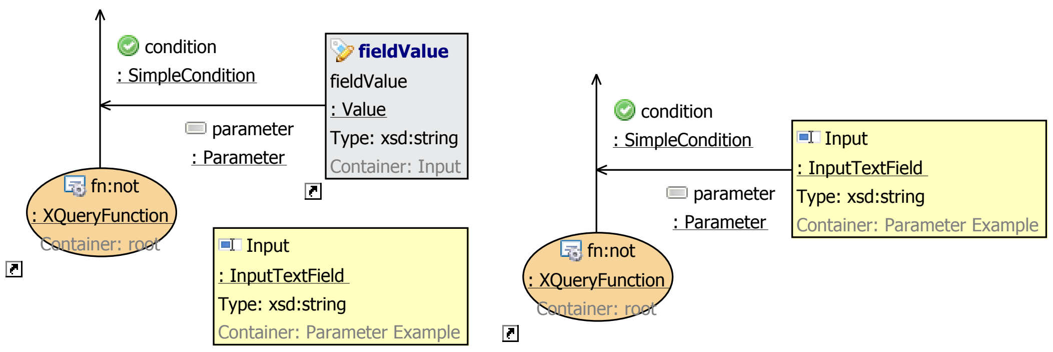
The current design and implementation of IAML only supports the modelling requirements of *Basic RIAs*. As discussed in Section 5.1.1 (pg. 81), this feature set restriction was necessary due to the time and resource constraints of this research project. There do not seem to be any design issues that would prevent the extension of IAML to support modelling *Full RIAs*, possibly through the definition of an extension metamodel. For example, some of the enterprise requirements, such as offline storage and internationalisation, could be defined through annotations on the existing model elements Scope and Visible Thing, rather than the redefinition of their underlying concepts.

9.2.3 Improved Graphical Editor View Mappings

During the development of the proof-of-concept graphical editor of IAML using GMF, an unexpected issue was discovered resulting from the level of abstraction of the interaction between the generated editor and the underlying metamodel. As discussed on Section 1 (pg. 193), GMF-based editor definitions work best when there is a one-to-one mapping between graphical elements and underlying metamodel elements. This means that without extending the generated graphical editor, one cannot easily represent the aggregation or derivation of elements as a single graphical element.

Consider Figure 9.1a, where a Complex Term (in this case, a Simple Condition) is provided the field value of a Changeable element as a Parameter. This field value is defined as the default Value for that element, and it would be preferable to represent this as Figure 9.1b, where the Parameter instead connects the Changeable element directly. For each of these visual representations, the underlying model instance can be exactly the same – the only difference is in the simplification of the syntax.

By default, the generated graphical editor cannot support reducing Figure 9.1a into Figure 9.1b, because the underlying one-to-one mapping model specifies that a Parameter *must* connect to a Parameter Value (and consequently a Value). Consequently, without the manual reconfiguration of the generated graphical editor and the templates that generate it, it is easier to specify that Changeable is a subtype of Value as syntactic sugar as discussed in Section 5.3.6, enforcing additional complexity on the underlying metamodel.



(a) Referencing Values directly (b) Referencing Values indirectly

Figure 9.1: Two approaches in defining a Complex Term that references a Value contained within a Changeable as a Parameter

As the artefacts necessary to generate GMF graphical editors are also model instances of EMF metamodels [119], it may be possible to define a new intermediary graphical mapping metamodel – a *view mapping model* – between these artefacts to formally define this syntactic sugar. Model transformations may then be used to transform the underlying metamodel and view mapping model into the necessary GMF model instances. As SimpleGMF already follows a similar architecture, it may be possible to extend this language into this desired view mapping metamodel.

9.2.4 Integrating Textual Expression Languages in the Visual Editor

The proof-of-concept implementation of the IAML metamodel has been designed to support a visual representation for all of the underlying model instances. However, as shown earlier in Figures 5.23 and 5.24 (pg. 119), this approach can transform simple textual expressions into complex visual representations of the same expression. In languages such as UML, this problem is resolved by allowing constraints to be defined textually; in some cases expressions can be defined in terms of OCL [224], where the main issue is in defining the mapping between textual syntax elements and the underlying model instance.

Since this research was started, the EMFText project was proposed [?] which provides a textual interface to model instances represented within EMF; the resulting syntax is similar to YAML representations of configuration data [?]. The openArchitectureWare framework includes a similar textual interface to EMF model instances called *XText*, however the authors of EMFText argue that *XText* cannot be used to interface with existing metamodels [?]. An investigation into integrating such a textual interface into the visual editor therefore remains future work; such a textual interface should not involve any modifications to the underlying metamodel, as it would be a different representation of the same underlying model.

9.2.5 Extraction of Reusable Components

As illustrated in Chapter 7, the development of the proof-of-concept implementation of IAML required the development of a large number of components. Some of these components, such as the *IAML Metamodel* and *Runtime Library* components, are domain-specific to the RIA domain; others such as the *Model Completion* and *Model Verification* components may be reusable in other modelling domains.

This model-driven development environment generalisation has already been successfully used in the development of the openArchitectureWare framework [75]. As discussed in Section 6.4.3 (pg. 164), this framework integrated a number of existing technologies into a single architecture connected by a workflow. This integration vastly simplified the accessibility of model-driven technologies to other developers, and this architecture is now being integrated into the Eclipse ecosystem. Integrating model completion and model verification into this model-driven architecture could therefore be a very beneficial area of future research and development.

9.2.6 Code Generation Templates for Additional Platforms

While one of the design goals of IAML was to develop a platform-independent modelling language, the code generation templates of the *Templates* component in Section 7.6.2 (pg. 203) only support a single combination of target platforms; in particular, the combination of PHP, Javascript, HTML, CSS and sqlite. To prove that the IAML metamodel is truly platform-independent, it is desirable to reimplement these templates to support additional browser platforms without having to modify any other aspect of the proof-of-concept implementation. For example, as the J2ME standard is designed to be device-independent [282], an implementation of the code generation templates of IAML into a J2ME format would verify that the design of the IAML metamodel is also device-independent.

9.2.7 Incremental Transformations

One of the issues identified during the implementation of the *Ticket 2.0* benchmarking application in Section 8.3.2 is that the entire process is relatively slow and consumes a large number of system resources. This is because both model completion and code generation currently operate as batch processes, meaning that even small changes still require the complete re-evaluation of each model transformation. This slow feedback cycle discourages incremental design processes or those that follow test-driven development [19].

Incremental generation is a code compilation technique that allows source code to be incrementally transformed into executable code as necessary [**?**, pg. 1144–1149], and is an important feature of the Eclipse development environment to reduce development time [102, pg. 146]. Giese and Wagner [107] investigated the development of applying incremental generation to model transformations, proposing bidirectional transformations to support the synchronisation of two models, and found that the corresponding speedup may allow the development of larger model instances.

At the very least, incremental model transformations could be applied to both the model completion and code generation aspects of the IAML proof-of-concept implementation, and would ideally be implemented through the Eclipse plugin framework to support an automated build process within Eclipse itself. For example, Drools supports the assertion and retraction of facts on a running model instance [245], techniques which may support incremental model completion. Similarly, the Eclipse release of Xpand 1.0 has recently announced support for incremental code generation which benefits the transformation of large model instances [142].

9.2.8 Extensibility of the Proof-of-Concept Implementation of IAML

Many of the implementation technologies evaluated in Chapter 6 included an evaluation of the extensibility of each technology, in order to support third-party extensibility of model-driven environments.

For example, both EMF, GMF, Drools and openArchitectureWare all support varying levels of extensibility; and the proof-of-concept implementation of IAML has been implemented through OSGi bundles that use existing Eclipse extension points.

As discussed earlier in Section 3.1.3, integration with other models and environments is an important benefit of model-driven environments. Consequently, an important future research question is on identifying the aspects of which these technologies may be extended, and how the extensibility of these technologies may be impacted by their integration into a single editor. It would also be interesting to investigate how the extensibility of these selected technologies compare to another extensible metamodelling environment such as Marama [120].

9.2.9 User Evaluations on the Proof-of-Concept Implementation of IAML

In this thesis, the interactions between the modelling environment and the model developers themselves have not been evaluated as this investigation fell outside the scope of this research. Many of the novel techniques integrated into the proof-of-concept implementation of IAML, such as the model completion and model instance verification components, would benefit from some form of end-user evaluation.

For example, the benefits and difficulties of the model completion concept need to be evaluated in terms of their usability and understandability, along with identifying the correct level of end-user documentation necessary. Similarly, the benefits and difficulties of using the many different types of model verification approaches needs to be investigated – is it acceptable to simultaneously provide three different approaches? What is the best way to highlight constraint violations identified using model checking? These technology-specific user evaluations could then be composed into an overall understanding of the usability of a modelling environment for developing Rich Internet Applications.

9.2.10 Identifying Metamodel Refactoring Patterns

A number of common activities and patterns emerged as part of the incremental design and development of IAML. For example, something as simple as renaming a class name involved the translation of a variety of source code, model completion rules, code generation rules, test cases, test models, documentation and design documents. Over time a number of patterns emerged, where different types of metamodel changes would involve different amounts of subsequent refactoring effort.

Similarly to the concept of "refactoring" within the software domain [87], similar types of changes applied to a metamodel may be termed *metamodel refactoring*, and the resulting changes necessary for the implementation of that given metamodel termed *coincidental refactoring*. Some of these metamodel refactorings identified through this research are discussed below, grouped into four categories of increasing subjective difficulty.

- **Easy:** *Adding a new attribute, reference, interface or class; creating a new package.*

  Generally, adding something to the metamodel does not require a lot of coincidental refactoring. This is due to the nature of object-oriented software; for an extension, only the extension needs to be considered, and all the underlying code should remain the same.

  Within IAML, none of these refactorings required any direct changes to other components. In particular, the SimpleGMF framework provides all new classes with a default representation within each graphical editor.

- **Moderate:** *Renaming an attribute, reference, interface or class; changing a metamodel namespace; moving a class to another package; removing an empty package.*

  When renaming something in the metamodel, the coincidental changes are fairly straightforward. Generally, existing references simply need to be refactored; there is no change in the internal logic. Automated refactoring tools can often assist in the renaming process – although EMF does not yet directly integrate with the Eclipse refactoring tools.

  Within IAML, these refactorings would often require the graphical editors references to be renamed, and regenerated; code generation templates and model completion rules to be updated; the model migrator updated; and existing model instances migrated. Renaming interfaces was generally the easiest, as since you cannot have an instance of an interface, existing model instances did not need to be migrated.

- **Difficult:** *Merging two attributes or references; removing an interface or class; splitting an interface into two interfaces.*

  These metamodel refactorings require a much more in-depth refactoring, as the developer will have to manually modify the dependent components.

  Within IAML, these changes required extensive development effort on the graphical editors, as graphical elements needed to be removed or merged. Model completion rules would have to be merged, and code generation templates would either be merged or require additional logic. The model migrator would have to be updated and all existing model instances migrated.

- **Very difficult:** *Splitting a reference into two references; removing an attribute or reference; splitting a class into two classes.*

  Within IAML, these changes would require changes on all of the implementation components. Because these metamodel changes are changing the *semantics* of a metamodel element, these refactorings may not trigger compile-time errors. The test suite discussed in Section 7.9 (pg. 211) was extremely beneficial in identifying necessary functional changes that would otherwise have gone unnoticed, and also to identify elements of the ModelDoc documentation that required clarification.

Similarly to software development, modelling languages developed incrementally benefit from the existence of automated refactoring tools or scripts to assist in common changes. For example, something as simple as renaming a model element can result in thousands of necessary changes, including Java source code, model completion rules, documentation and design documents, and test models; with the right development environment architecture, many of these changes could be performed automatically or with little developer input. Further research into this area could include the development of new metamodel refactoring tools, or in new documentation or development methods to increase the robustness of model-driven environments.

9.3 Summary

Model-driven development can be utilised to simplify the development of complex software applications across many platforms, but this research found that no existing web application modelling language could be used to satisfactorily model or develop Rich Internet Applications. The Internet Application Modelling Language (IAML) has been designed and proposed in this thesis as a platform-independent RIA modelling language. This language reuses existing modelling languages where appropriate, and also defines a visual syntax to support the graphical definition of IAML model instances.

The design of this language included the development of a proof-of-concept implementation in the Eclipse framework, to verify that the design concepts of the language benefit the implementation of real-world RIAs. This implementation included the integration of a number of different model-driven technologies – such as a visual modelling interface, code generation templates, model completion rules, and different types of constraints for model verification – in order to improve the usability of developing robust IAML model instances.

The IAML metamodel supports many features not found in other web application modelling languages, such as ECA rules, the expression of reusable patterns through Wires, and a metamodel core based on first-order logic. Through the implementation of the RIA benchmarking application *Ticket 2.0*, these concepts have been shown to simplify the development of real-world RIAs when compared to a conventional implementation through the Symfony framework. This research has also raised a number of interesting research questions that, when answered, may simplify the development of model-driven environments in the future.

Bibliography

[1] Acerbis, R., Bongio, A., Brambilla, M., Tisi, M., Ceri, S. and Tosetti, E. Developing eBusiness Solutions with a Model Driven Approach: The Case of Acer EMEA. In: *Proceedings of the 7th International Conference on Web Engineering (ICWE '07)*, pp. 539–544. 2007

[2] Adobe Systems Incorporated. Adobe AIR, 2011

[3] Adrion, W. R., Branstad, M. A. and Cherniavsky, J. C. Validation, Verification, and Testing of Computer Software. *ACM Comput. Surv.*, vol. 14, pp. 159–192, 1982

[4] Alexander, R. The real costs of aspect-oriented programming? *IEEE Software*, vol. 20(6), pp. 91–93, 2003

[5] Anastasopoulos, M. and Muthig, D. An Evaluation of Aspect-Oriented Programming as a Product Line Implementation Technology. In: *ICSR*, *Lecture Notes in Computer Science*, vol. 3107, pp. 141–156. Springer, 2004

[6] Aniszczyk, C. Using GEF with EMF. Tech. rep., IBM, 2005

[7] Apache Software Foundation. Struts Framework, 2008

[8] Atterer, R. Where Web Engineering Tool Support Ends: Building Usable Websites. In: *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC '05)*, pp. 1684–1688. ACM Press, New York, NY, USA, 2005

[9] Badros, G. J. JavaML: A Markup Language for Java Source Code. *Comput. Netw.*, vol. 33, pp. 159–177, 2000

[10] Baier, C. and Katoen, J.-P. *Principles of Model Checking*. The MIT Press, 2008

[11] Baker, P., Dai, Z. R., Grabowski, J., Øystein Haugen, Schieferdecker, I. and Williams, C. Test Execution with JUnit. In: *Model-Driven Testing*, pp. 149–156. Springer Berlin Heidelberg, 2008

[12] Baldwin, C. Y. and Clark, K. B. *Design Rules, Volume 1: The Power of Modularity*, vol. 1. The MIT Press, 1 ed., 2000

[13] Baresi, L., Colazzo, S., Mainetti, L. and Morasca, S. W2000: A Modelling Notation for Complex Web Applications. In: Mendes, E. and Mosley, N., eds., *Web Engineering*, pp. 335–364. Springer, 2006

[14] Baroni, A. L. *Formal Definition of Object-Oriented Design Metrics*. Master's thesis, Vrije Universiteit Brussel, Brussels, Belgium, 2002

[15] Baroni, A. L. and e Abreu, F. B. A Formal Library for Aiding Metrics Extraction. In: *Proceedings of the International Workshop on Object-Oriented Re-Engineering at ECOOP 2003*. Darmstadt, Germany, 2003

[16] Barr, P., Biddle, R. and Noble, J. A Taxonomy of User Interface Metaphors. In: *Proceedings of the SIGCHI New Zealand Symposium on Computer-Human Interaction*. IEEE Press, 2002

[17] Barry, C. and Lang, M. A Survey of Multimedia and Web Development Techniques and Methodology Usage. *IEEE MultiMedia*, vol. 8(2), pp. 52–60, 2001

[18] Baumeister, H., Knapp, A., Koch, N. and Zhang, G. Modelling Adaptivity with Aspects. In: *Proceedings of the 5th International Conference on Web Engineering (ICWE '05)*, pp. 406–416. 2005

[19] Beck, K. *Test-Driven Development by Example*. Addison-Wesley, 2003

[20] Beck, K., Gamma, E. and Saff, D. *JUnit API*, 2010

[21] Bellettini, C., Marchetto, A. and Trentini, A. WebUml: reverse engineering of web applications. In: *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC '04)*, pp. 1662–1669. ACM, New York, NY, USA, 2004

[22] Bellettini, C., Marchetto, A. and Trentini, A. TestUml: User-Metrics Driven Web Applications Testing. In: *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC '05)*, pp. 1694–1698. ACM Press, New York, NY, USA, 2005

[23] Bennett, S., McRobb, S. and Farmer, R. *Object-Oriented Systems Analysis and Design Using UML*. McGraw-Hill, London, 2nd ed., 2002

[24] Berners-Lee, T. and Fischetti, M. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor*. Harper San Francisco, 1999

[25] Berners-Lee, T., Masinter, L. and McCahill, M. Uniform Resource Locators (URL). RFC 1738 (Proposed Standard), 1994

[26] Beyer, D., Noack, A. and Lewerentz, C. Efficient Relational Calculation for Software Analysis. *IEEE Transactions on Software Engineering*, vol. 31(2), pp. 137–149, 2005

[27] Bézivin, J. In Search of a Basic Principle for Model Driven Engineering. *CEPIS, UPGRADE, The European Journal for the Informatics Professional*, vol. 5(2), pp. 21–24, 2004

[28] Bézivin, J., Dupé, G., Jouault, F., Pitette, G. and Rougui, J. E. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In: *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*. 2003

[29] Blanc, X., Gervais, M.-P. and Sriplakich, P. Model Bus: Towards the Interoperability of Modelling Tools. pp. 17–32. 2005

[30] Boehm, B. W. *Software Engineering Economics*. Prentice-Hall, 1981

[31] Boswell, D., King, B., Oeschger, I., Collins, P. and Murphy, E. *Creating Applications with Mozilla*. O'Reilly Series. O'Reilly, 2002

[32] Bozzon, A., Comai, S., Fraternali, P. and Carughi, G. T. Conceptual Modeling and Code Generation for Rich Internet Applications. In: *Proceedings of the 6th International Conference on Web Engineering (ICWE '06)*, pp. 353–360. ACM Press, New York, NY, USA, 2006

[33] Brambilla, M., Preciado, J. C., Linaje, M. and Sanchez-Figueroa, F. Business Process-Based Conceptual Design of Rich Internet Applications. In: *Proceedings of the 8th International Conference on Web Engineering (WISE '08)*, pp. 155–161. IEEE Computer Society, Washington, DC, USA, 2008

[34] Breslin, J., Passant, A. and Decker, S. *The Social Semantic Web*. Springer-Verlag, Heidelberg, 2009

[35] Bruck, J. and Hussey, K. Customizing UML: Which Technique is Right for You? Tech. rep., 2007

[36] Buckley, A. JSR 14: Add Generic Types to the Java Programming Language. Tech. rep., Java Community Process, 2004

[37] Cardelli, L. A Semantics of Multiple Inheritance. *Information and Computation*, vol. 76(2/3), pp. 138–164, 1988

[38] Cardelli, L. Type Systems. In: Tucker, A. B., ed., *The Computer Science and Engineering Handbook*, pp. 2208–2236. CRC Press, 1997

[39] Cardellini, V., Casalicchio, E., Colajanni, M. and Yu, P. S. The state of the art in locally distributed web-server systems. *ACM Computing Surveys*, vol. 34, pp. 263–311, 2002

[40] Carstensen, P. H. and Vogelsang, L. Design of Web-Based Information Systems - New Challenges for Systems Development? In: *Proceedings of the 9th European Conference on Information Systems (ECIS)*. 2001

[41] Ceri, S., Fraternali, P. and Bongio, A. Web Modeling Language (WebML): A Modeling Language for Designing Web Sites. In: *Proceedings of the 9th International World Wide Web Conference on Computer Networks*, pp. 137–157. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 2000

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

[43] Ceri, S., Fraternali, P., Maurino, A. and Paraboschi, S. One-to-One Personalization of Data-Intensive Web Sites. In: *WebDB (Informal Proceedings)*, pp. 1–6. 1999

[44] Chauvin, M., Tikhomirov, A. and Shatalin, A. GMF 2.2 New and Noteworthy: 2.2 M4, 2010

[45] Chen, P. P.-S. The Entity-Relationship Model – Toward a Unified View of Data. *ACM Trans. Database Syst.*, vol. 1, pp. 9–36, 1976

[46] Chikofsky, E. J. and Cross II, J. H. Reverse Engineering and Design Recovery: A Taxonomy. *Software, IEEE*, vol. 7(1), pp. 13 –17, 1990

[47] Cimatti, A., Giunchiglia, E., Pistore, M., Roveri, M., Sebastiani, R. and Tacchella, A. Integrating BDD-based and SAT-based Symbolic Model Checking. In: *Proceedings of Frontiers of Combining Systems (FROCOS'02)*, *LNAI*, vol. 2309. Springer, Santa Margherita, Italy, 2002

[48] Clements, P. C. A Survey of Architecture Description Languages. In: *Proceedings of the 8th International Workshop on Software Specification and Design*, IWSSD '96, pp. 16–25. IEEE Computer Society, Washington, DC, USA, 1996

[49] Collins-Sussman, B., Fitzpatrick, B. W. and Pilato, C. M. *Version Control with Subversion*. O'Reilly Media, Inc., 2004

[50] Conallen, J. *Building Web applications with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000

[51] Cook, S., Jones, G., Kent, S. and Wills, A. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley Professional, 2007

[52] Cowderoy, A. Measures of Size and Complexity for Web-site Content. In: *Proceedings of the Combined 11th ESCOM Conference and the 3rd SCOPE conference on Software Product Quality*, pp. 423–431. Munich, Germany, 2000

[53] Crespo, Y., Marqués, J. M. and Rodríguez, J. J. On the Translation of Multiple Inheritance Hierarchies into Single Inheritance Hierarchies. In: Black, A., Ernst, E., Grogono, P. and Sakkinen, M., eds., *Proceedings of the Inheritance Workshop at ECOOP 2002*, pp. 30–37. Jyväskylä University Press, 2002

[54] Creswell, J. W. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. Sage Publications, 3rd ed., 2009

[55] Crockford, D. RFC 4627: The application/json Media Type for JavaScript Object Notation (JSON). Tech. rep., The Internet Society, 2006

[56] Czarnecki, K. and Helsen, S. Feature-based survey of model transformation approaches. *IBM Systems Journal*, vol. 45(3), pp. 621–645, 2006

[57] da Silva, P. P. and Paton, N. W. A UML-Based Design Environment for Interactive Applications. In: *Proceedings of the 2nd International Workshop on User Interfaces to Data Intensive Systems (UIDIS '01)*, p. 60. IEEE Computer Society, Washington, DC, USA, 2001

[58] da Silva, P. P. and Paton, N. W. User Interface Modeling in UMLi. *IEEE Software*, vol. 20, pp. 62–69, 2003

[59] David, M. *Flash Mobile: Developing Android and iOS Applications*. Focal Press, 2011

[60] Deitel, H. M. and Deitel, P. J. *Java^{TM}: How to Program*. Prentice Hall Press, Upper Saddle River, NJ, USA, 7th ed., 2006

[61] Dennis, G. Re: Recursion in Alloy 3. Mailing List, 2004

[62] Deprez, J.-C. and Alexandre, S. Comparing Assessment Methodologies for Free/Open Source Software: OpenBRR and QSOS. *Product-Focused Software Process Improvement*, pp. 189–203, 2008

[63] Deutsch, A., Sui, L. and Vianu, V. Specification and Verification of Data-driven Web Services. In: *Proceedings of the 23rd Symposium on Principles of Database Systems (PODS '04)*, pp. 71–82. ACM Press, New York, NY, USA, 2004

[64] Di Martino, S., Ferrucci, F., Paolino, L., Sebillo, M., Tortora, G., Vitiello, G. and Avagliano, G. Towards the Automatic Generation of Web GIS. In: *Proceedings of the 15th annual ACM international symposium on Advances in Geographic Information Systems*, GIS '07, pp. 57:1–57:4. ACM, New York, NY, USA, 2007

[65] Dietrich, J., Hiller, J. and Schenke, B. Take - A Rule Compiler for Derivation Rules. In: Paschke, A. and Biletskiy, Y., eds., *RuleML*, *Lecture Notes in Computer Science*, vol. 4824, pp. 134–148. Springer, Orlando, Florida, 2007

[66] Ditchendorf, T. Fluid, 2010

[67] Dittrich, K. R. and Gatziu, S. Time Issues in Active Database Systems. In: *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*. Arlington, Texas, 1993

[68] Djurić, D., Gašević, D. and Devedžić, V. The Tao of Modeling Spaces. *Object Technology*, vol. 5(8), 2006

[69] Eclipse Foundation. Eclipse Public License, Version 1.0, 2004

[70] Eclipse Foundation. Eclipse Foundation, Inc. Intellectual Property Policy. Tech. rep., Eclipse Foundation, Inc., 2008

[71] Eclipse Foundation. Eclipse Modeling Framework Project (EMF): Validation Framework, 2008

[72] Eclipse Foundation. Eclipse Projects, 2008

[73] ECMA International. ECMA-262: ECMAScript Language Definition, Edition 3. Tech. rep., ECMA International, 1999

[74] Efftinge, S., Friese, P., Haase, A., Hübner, D., Kadura, C., Kolb, B., Köhnlein, J., Moroff, D., Thoms, K., Völter, M., Schönbach, P., Eysholdt, M. and Reinisch, S. *openArchitectureWare Documentation*, 2008

[75] Eichinger, C. Web Application Architectures. In: Kappel, G., Pröll, B., Reich, S. and Retschitzegger, W., eds., *Web Engineering – The Discipline of Systematic Development of Web Applications*, chap. 4. John Wiley & Sons Ltd., England, 2006

[76] Eshuis, R. and Wieringa, R. A Real-Time Execution Semantics for UML Activity Diagrams. In: *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering (FASE '01)*, pp. 76–90. Springer-Verlag, London, UK, 2001

[77] European Smalltalk User Group. About Seaside, 2008

[78] Fahl, W. *Marama Concepts*, 2007

[79] Fayad, M. and Schmidt, D. C. Object-oriented application frameworks. *Commun. ACM*, vol. 40(10), pp. 32–38, 1997

[80] Fenton, N. E. and Pfleeger, S. L. *Software Metrics: A Rigorous and Practical Approach*. International Thompson computer, 2 ed., 1997

[81] Ferraiolo, D. F., Sandhu, R., Gavrila, S., Kuhn, D. R. and Chandramouli, R. Proposed NIST Standard for Role-Based Access Control. *ACM Trans. Inf. Syst. Secur.*, vol. 4, pp. 224–274, 2001

[82] Fielding, R. T. and Taylor, R. N. Principled Design of the Modern Web Architecture. *ACM Trans. Internet Technol.*, vol. 2, pp. 115–150, 2002

[83] Filman, R. E., Elrad, T., Clarke, S. and Akşit, M., eds. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005

[84] Fitting, M. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag New York, Inc., New York, NY, USA, 1990

[85] Flanagan, D. *JavaScript: The Definitive Guide*. Definitive Guide Series. O'Reilly, 2006

[86] Fowler, M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999

[87] Fowler, M. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002

[88] Fowler, M. *Domain-Specific Languages*. Addison-Wesley Professional, 2010

[89] France, R. B., Ray, I., Georg, G. and Ghosh, S. Aspect-oriented approach to early design modelling. *IEEE Software*, vol. 151(4), pp. 173–186, 2004

[90] Fraternali, P., Comai, S., Bozzon, A. and Carughi, G. T. Engineering Rich Internet Applications with a Model-Driven Approach. *ACM Trans. Web*, vol. 4, pp. 7:1–7:47, 2010

[91] Free Software Foundation. The GNU General Public License v3.0, 2007

[92] Free Software Foundation. GCC Runtime Library Exception, 2009

[93] Free Software Foundation. Frequently Asked Questions about the GNU Licenses, 2011

[94] Free Software Foundation. Various Licenses and Comments about Them, 2011

[95] Freed, N. and Borenstein, N. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. RFC 2045 (Draft Standard), 1996

[96] Friedman-Hill, E. Re: JESS: Multi-inheritance. Mailing List, 2001

[97] Friedman-Hill, E. J. *Jess, The Rule Engine for the Java Platform*, 2005

[98] Fu, Y., Dong, Z. and He, X. Formalizing and Validating UML Architecture Description of Web Systems. In: *Workshop Proceedings of the 6th International Conference on Web Engineering (ICWE '06)*. ACM, New York, NY, USA, 2006

[99] Fuentes-Fernández, L. and Vallecillo-Moreno, A. An Introduction to UML Profiles. *The European Journal for the Informatics Professional*, vol. 5(2), 2004

[100] Gabbay, D. M., Hogger, C. J. and Robinson, J. A., eds. *Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 3*. Oxford University Press, Inc., New York, NY, USA, 1994

[101] Gamma, E. and Beck, K. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison-Wesley, 2003

[102] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995

[103] Gane, N. and Beer, D. *New Media: The Key Concepts*. Berg Publishers, 2008

[104] Garfinkel, S. and Spafford, G. *Web Security, Privacy and Commerce*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2nd ed., 2001

[105] Garrett, J. J. Ajax: A New Approach to Web Applications. Tech. rep., 2005

[106] Giese, H. and Wagner, R. Incremental Model Synchronization with Triple Graph Grammars. In: Nierstrasz, O., Whittle, J., Harel, D. and Reggio, G., eds., *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, *LNCS*, vol. 4199, pp. 543–557. Springer Verlag, 2006

[107] Gitzel, R., Korthaus, A. and Schader, M. Using Established Web Engineering Knowledge in Model-Driven Approaches. *Sci. Comput. Program.*, vol. 66(2), pp. 105–124, 2007

[108] Giurca, A. R2ML – The REWERSE I1 Rule Markup Language. Tech. rep., REWERSE Working Group I1, 2006

[109] Giurca, A. and Wagner, G. Rule Modeling and Interchange. In: *Proceedings of the 9th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2007)*, pp. 485–491. 2007

[110] González, A. S., Ruiz, D. S. and Perez, G. M. EMF4CPP: a C++ Ecore Implementation. *DSDM 2010*, 2010

[111] Google Inc. Google Gears, 2007

[112] Gosling, J., Joy, B., Steele, G. and Bracha, G. *The Java Language Specification*. 3rd ed., 2005

[113] Gottlob, G., Koch, C. and Pichler, R. Efficient Algorithms for Processing XPath Queries. *ACM Trans. Database Syst.*, vol. 30, pp. 444–491, 2005

[114] Grand, M. *Patterns in Java: a catalog of reusable design patterns illustrated with UML, volume 1*. John Wiley & Sons, Inc., New York, NY, USA, 2002

[115] Green, T. R. G. and Petre, M. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing*, vol. 7(2), pp. 131–174, 1996

[116] Groenewegen, D. M., Hemel, Z., Kats, L. C. and Visser, E. Webdsl: a domain-specific language for dynamic web applications. In: *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pp. 779–780. ACM, New York, NY, USA, 2008

[117] Groenewegen, D. M. and Visser, E. Declarative Access Control for WebDSL: Combining Language Integration and Separation of Concerns. In: Schwabe, D. and Curbera, F., eds., *Proceedings of the 8th International Conference on Web Engineering (ICWE '08)*, pp. 175–188. IEEE CS Press, 2008

[118] Gronback, R. C. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 2009

[119] Grundy, J., Hosking, J., Zhu, N. and Liu, N. Generating Domain-Specific Visual Language Editors from High-level Tool Specifications. In: *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE '06)*, pp. 25–36. IEEE Computer Society, Washington, DC, USA, 2006

[120] Grundy, J. C., Hosking, J. G., Amor, R. W., Mugridge, W. B. and Li, Y. Domain-Specific Visual Languages for Specifying and Generating Data Mapping Systems. *Journal of Visual Languages & Computing*, vol. 15(3-4), pp. 243 – 263, 2004

[121] Grundy, J. C., Hosking, J. G., Huh, J. and Li, K. N.-L. Marama: an Eclipse Meta-toolset for Generating Multi-view Environments. In: *ICSE*, pp. 819–822. 2008

[122] Gu, A., Henderson-Sellers, B. and Lowe, D. Web Modelling Languages: The Gap Between Requirements and Current Exemplars. In: *Proceedings of the 8th Australian World Wide Web Conference (AusWeb02)*. 2002

[123] Gurevich, Y. Evolving Algebras 1993: Lipari Guide. In: Börger, E., ed., *Specification and Validation Methods*, pp. 9–36. Oxford University Press, Inc., New York, NY, USA, 1995

[124] Haggett, P. and Chorley, R. J. Models, Paradigms and the New Geography. *Socioeconomic Models in Geography*, 1967

[125] Hailpern, B. and Tarr, P. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, vol. 45, pp. 451–461, 2006

[126] Hammer-Lahav, E. The OAuth 1.0 Protocol. RFC 5849 (Draft Standard), 2010

[127] Hansen, T. and Vaudreuil, G. Message Disposition Notification. RFC 3798 (Draft Standard), 2004

[128] Hanson, R. and Tacy, A. *GWT in Action: Easy Ajax with the Google Web Toolkit*. Wiley, Greenwich, CT, USA, 2007

[129] Harel, D. and Rumpe, B. Modeling Languages: Syntax, Semantics and All That Stuff, Part I: The Basic Stuff. Tech. rep., 2000

[130] Havelund, K. and Pressburger, T. Model Checking Java Programs using Java PathFinder. *Software Tools for Technology Transfer*, vol. 2(4), pp. 366–381, 2000

[131] Henry, J., Lane, J. and Wright, J. JWebUnit, 2010

[132] Hettel, T., Lawley, M. and Raymond, K. Model Synchronisation: Definitions for Round-Trip Engineering. In: *Proceedings of the First International Conference on Model Transformation (ICMT '08)*. Springer, 2008

[133] Hillairet, G. emftriple: (Meta)Models on the Web of Data, 2011

[134] Hoare, C. A. R. An axiomatic basis for computer programming. *Commun. ACM*, vol. 12, pp. 576–580, 1969

[135] Hoare, C. A. R. Hints on Programming Language Design. Tech. rep., Stanford, CA, USA, 1973

[136] Huck, J., Morris, D., Ross, J., Knies, A., Mulder, H. and Zahir, R. Introducing the IA-64 Architecture. *Micro, IEEE*, vol. 20(5), pp. 12–23, 2000

[137] Huh, J., Grundy, J., Hosking, J., Liu, K. and Amor, R. Integrated Data Mapping for a Software Meta-tool. In: *Proceedings of the Doctoral Symposium at the 20th Australian Software Engineering Conference (ASWEC 2009)*, pp. 111–120. 2009

[138] Hunter, J. and Crawford, W. *Java Servlet Programming*. Java series. O'Reilly, 2nd ed., 2001

[139] Huth, M. and Ryan, M. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2nd ed., 2004

[140] IEEE. *IEEE Software Engineering Standards: Standard 610.12-1990*, 1990

[141] Irawan, H., Stoll, D., Efftinge, S., Jockel, D. and Zarnekow, S. Xpand: New and Noteworthy, 2010

[142] ISO. *ISO 8879: Information processing — Text and office systems — Standard Generalized Markup Language (SGML)*. International Organization for Standardization, Geneva, Switzerland, 1986

[143] ISO. *ISO/IEC 14977:1996: Information technology — Syntactic metalanguage — Extended BNF*. International Organization for Standardization, Geneva, Switzerland, 1996

[144] ISO/IEC. *ISO/IEC 9126: Software engineering – Product quality*. International Organization for Standardization, Geneva, Switzerland, 2001

[145] ISO/IEC. *ISO/IEC 19503: Information technology – XML Metadata Interchange (XMI)*. International Organization for Standardization, Geneva, Switzerland, 2005

[146] Jackson, D. *Software Abstractions: Logic, Language, and Analysis*. Cambridge, Mass., 2006

[147] JetBrains. JetBrains Meta Programming System

[148] Johnson, R. J2ee development frameworks. *Computer*, vol. 38(1), pp. 107–110, 2005

[149] Johnson, R., Hoeller, J., Arendsen, A., Risberg, T. and Sampaleanu, C. *Professional Java Development with the Spring Framework*. Wrox, Birmingham, UK, UK, 2005

[150] Jouault, F. and Kurtev, I. Transforming Models with ATL. In: *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*. Jamaica, 2005

[151] Jouault, F. and Kurtev, I. On the architectural alignment of ATL and QVT. In: *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC '06)*, pp. 1188–1195. ACM Press, New York, NY, USA, 2006

[152] Kappel, G., Pröll, B., Reich, S. and Retschitzegger, W. An Introduction to Web Engineering. In: Kappel, G., Pröll, B., Reich, S. and Retschitzegger, W., eds., *Web Engineering – The Discipline of Systematic Development of Web Applications*, chap. 1, pp. 1–22. John Wiley & Sons Ltd., England, 2006

[153] Kappel, G., Retschitzegger, W. and Schwinger, W. Modeling Ubiquitous Web Applications: The WUML approach. In: *International Workshop on Data Semantics in Web Information Systems (DASWIS-2001)*. Yokohama, Japan, 2001

[154] Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schneider, M. and Völkel, S. Design guidelines for domain specific languages. In: Rossi, M., Sprinkle, J., Gray, J. and Tolvanen, J.-P., eds., *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09)*, pp. 7–13

[155] Kelly, S., Lyytinen, K. and Rossi, M. MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In: Constantopoulos, P., Mylopoulos, J. and Vassiliou, Y., eds., *CAiSE*, *Lecture Notes in Computer Science*, vol. 1080, pp. 1–21. Springer, 1996

[156] Kelly, S. and Pohjonen, R. Worst Practices for Domain-Specific Modeling. *IEEE Software*, vol. 26, pp. 22–29, 2009

[157] Kent, S. Model Driven Engineering. In: *Proceedings of the Third International Conference on Integrated Formal Methods (IFM '02)*, pp. 286–298. Springer-Verlag, London, UK, 2002

[158] Kessler, P. Why can't I allocate 2GB of heap to the JVM on Windows, Part 2. Blog, 2004

[159] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M. and Irwin, J. Aspect-Oriented Programming. In: *ECOOP*, pp. 220–242. 1997

[160] Kingsley-Hughes, A., Kingsley-Hughes, K. and Read, D. *VBScript: Programmer's Reference*. Wrox. Wrox/Wiley Pub., 2007

[161] Klensin, J. Simple Mail Transfer Protocol. RFC 2821 (Proposed Standard), 2001

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

[163] Kline, K. *SQL in a Nutshell*. O'Reilly Media, Inc., 2001

[164] Knapp, A., Koch, N., Moser, F. and Zhang, G. ArgoUWE: A CASE Tool for Web Applications. In: *First Int. Workshop on Engineering Methods to Support Information Systems Evolution (EMSISE 2003)*. 2003

[165] Kobryn, C. UML 2001: A Standardization Odyssey. *Commun. ACM*, vol. 42, pp. 29–37, 1999

[166] Koch, N. Transformation Techniques in the Model-Driven Development Process of UWE. In: *Proceedings of the 6th International Conference on Web Engineering (ICWE '06)*. ACM, New York, NY, USA, 2006

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

[168] Koch, N. and Kraus, A. The Expressive Power of UML-based Web Engineering. In: *IW-WOST'02*, pp. 105–119. 2002

[169] Krasner, G. E. and Pope, S. T. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Object Oriented Programming*, vol. 1(3), pp. 26–49, 1988

[170] Kroiß, C. and Koch, N. *The UWE Metamodel and Profile – User Guide and Reference*, 2008

[171] Kuleshov, E. and Peter, K. *Eclipse UI Graphics: Style & Design*. The Eclipse Foundation, 2008

[172] Kuuskeri, J. and Mikkonen, T. REST Inspired Code Partitioning with a JavaScript Middleware. In: Daniel, F. and Facca, F., eds., *Current Trends in Web Engineering*, *Lecture Notes in Computer Science*, vol. 6385, pp. 244–255. Springer Berlin / Heidelberg, 2010

[173] Lakoff, G. and Johnson, M. *Metaphors we live by*, vol. 111. Chicago London, 1980

[174] Lang, M. and Fitzgerald, B. Hypermedia Systems Development Practices: A Survey. *IEEE Software*, vol. 22(2), pp. 68–75, 2005

[175] Lano, K. and Bicarregui, J. Semantics and Transformations for UML Models. In: Bézivin, J. and Muller, P.-A., eds., *UML'98 - Beyond the Notation*, *LNCS*, vol. 1618, pp. 107–119. Springer, 1999

[176] Lawton, G. Invasive Software: Who's Inside Your Computer? *Computer*, vol. 35, pp. 15–18, 2002

[177] Leavitt, N. Will NoSQL Databases Live Up to Their Promise? *Computer*, vol. 43, pp. 12–14, 2010

[178] Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J. and Volgyesi, P. The Generic Modeling Environment. In: *Workshop on Intelligent Signal Processing*. IEEE, 2001

[179] Lerdorf, R., Tatroe, K. and MacIntyre, P. B. *Programming PHP*. O'Reilly, 2nd ed., 2006

[180] Lerner, R. M. At the Forge: jQuery Plugins. *Linux J.*, vol. 2009, 2009

[181] Li, K., Hosking, J. and Grundy, J. A Generalised Event Handling Framework. In: *Proceedings of the Knowledge Industry Survival Strategy Workshop at ASE 2009*. 2009

[182] Li, L. *The VisualAge for Smalltalk Primer*. Advances in object technology. Cambridge University Press, 1998

[183] Lieberherr, K. J. and Holland, I. Assuring Good Style for Object-Oriented Programs. *Software, IEEE*, vol. 6(5), pp. 38–48, 2005

[184] Liu, K. *Marama Meta-Tools User Manual*, 2008

[185] Liu, N., Hosking, J. and Grundy, J. A Visual Language and Environment for Specifying User Interface Event Handling in Design Tools. In: *AUIC '07: Proceedings of the Eighth Australasian Conference on User Interface*, pp. 87–94. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2007

[186] Liu, N., Hosking, J. and Grundy, J. MaramaTatau: Extending a Domain Specific Visual Language Meta Tool with a Declarative Constraint Mechanism. *VL/HCC'07*, pp. 95–103, 2007

[187] Manes, A. T. *Web Services: A Manager's Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003

[188] Manolescu, I., Brambilla, M., Ceri, S., Comai, S. and Fraternali, P. Model-driven design and deployment of service-enabled Web applications. *ACM Trans. Inter. Tech.*, vol. 5(3), pp. 439–479, 2005

[189] Manovich, L. *The Language of New Media*. The MIT Press, 2001

[190] Marinilli, M. *Java deployment with JNLP and WebStart*. Kaleidoscope Series. Sams, 2002

[191] Massoni, T., Gheyi, R. and Borba, P. A Model-Driven Approach to Formal Refactoring. In: *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '05)*, pp. 124–125. ACM, New York, NY, USA, 2005

[192] McBride, B. Jena: Implementing the RDF Model and Syntax Specification. In: *SemWeb*. 2001

[193] McConnell, S. *Rapid Development: Taming Wild Software Schedules*. Microsoft Press, Redmond, WA, USA, 1st ed., 1996

[194] Mendelson, E. *Introduction to Mathematical Logic*. Chapman & Hall/CRC, 1997

[195] Mendes, E., Counsell, S. and Mosley, N. Towards a Taxonomy of Hypermedia and Web Application Size Metrics. In: *Proceedings of the 5th International Conference on Web Engineering (ICWE '05)*, pp. 110–123. 2005

[196] Merialdo, P., Atzeni, P. and Mecca, G. Design and Development of Data-Intensive Web Sites: The Araneus Approach. *ACM Trans. Inter. Tech.*, vol. 3(1), pp. 49–92, 2003

[197] Mernik, M., Heering, J. and Sloane, A. M. When and how to develop domain-specific languages. *ACM Computing Surveys*, vol. 37(4), pp. 316–344, 2005

[198] Meservy, T. O. and Fenstermacher, K. D. Transforming Software Development: An MDA Road Map. *IEEE Computer*, vol. 38(9), pp. 52–58, 2005

[199] Metz, C. AAA Protocols: Authentication, Authorization, and Accounting for the Internet. *Internet Computing, IEEE*, vol. 3(6), pp. 75 –79, 1999

[200] Miao, H. and Zeng, H. Model Checking-based Verification of Web Application. In: *Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pp. 47–55. IEEE Computer Society, Washington, DC, USA, 2007

[201] Michaelson, J. There's No Such Thing as a Free (Software) Lunch. *Queue*, vol. 2, pp. 40–47, 2004

[202] Microsoft Developer Network. Extend your DSL by using MEF. Tech. rep., Microsoft, 2010

[203] Mikkonen, T. and Taivalsaari, A. Web applications – spaghetti code for the 21st century. *Proceedings of the 6th International Conference on Software Engineering Research, Management and Applications*, pp. 319–328, 2008

[204] Mockus, A., Fielding, R. T. and Herbsleb, J. D. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Trans. Softw. Eng. Methodol.*, vol. 11, pp. 309–346, 2002

[205] Monperrus, M., Jézéquel, J.-M., Champeau, J. and Hoeltzener, B. Measuring models. In: Rech, J. and Bunse, C., eds., *Model-Driven Software Development: Integrating Quality Assurance*. IDEA Group, 2008

[206] Moody, D. L. The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, vol. 35, pp. 756–779, 2009

[207] Moore, K. Simple Mail Transfer Protocol (SMTP) Service Extension for Delivery Status Notifications (DSNs). RFC 3461 (Draft Standard), 2003

[208] Moreno, N., Fraternali, P. and Vallecillo, A. Webml modelling in uml. *IET Software*, vol. 1(3), pp. 67–80, 2007

[209] Morse, J. M. Approaches to qualitative-quantitative methodological triangulation. *Nursing Research*, vol. 40(2), pp. 120–123, 1991

[210] Motik, B. On the Properties of Metamodeling in OWL. *J. Log. Comput.*, vol. 17(4), pp. 617–637, 2007

[211] Mozilla Foundation. Mozilla Prism, 2009

[212] Mukerji, J. and Miller, J. Model-Driven Architecture Guide, v1.0.1. Tech. rep., Object Management Group, 2003

[213] Nagappan, N. and Ball, T. Evidence-Based Failure Prediction. In: Oram, A. and Wilson, G., eds., *Making Software*. 2011

[214] Nicolae, O., Giurca, A. and Wagner, G. On Interchange between Drools and Jess. *Informatica (Slovenia)*, vol. 32(4), pp. 383–396, 2008

[215] Nussbaumer, M. and Gaedke, M. Technologies for Web Applications. In: Kappel, G., Pröll, B., Reich, S. and Retschitzegger, W., eds., *Web Engineering – The Discipline of Systematic Development of Web Applications*, chap. 6. John Wiley & Sons Ltd., England, 2006

[216] Object Management Group. The Common Object Request Broker: Architecture and Specification, v2.3.1. Tech. rep., 1999

[217] Object Management Group. OMG Unified Modeling Language Specification, v1.4. Tech. rep., 2001

[218] Object Management Group. XML Metadata Interchange (XMI), v2.1. Tech. rep., 2005

[219] Object Management Group. Meta Object Facility (MOF) Core Specification, v2.0. Tech. rep., 2006

[220] Object Management Group. Object Constraint Language Specification, v2.0. Tech. rep., 2006

[221] Object Management Group. OMG Unified Modeling Language (OMG UML), Infrastructure, v2.1.2. Tech. rep., 2007

[222] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure, v2.1.2. Tech. rep., 2007

[223] Object Management Group. MOF Model to Text Transformation Language, v1.0. Tech. rep., 2008

[224] Object Management Group. Meta Object Facility (MOF) Core Specification, v2.4. Tech. rep., 2010

[225] Object Management Group. Object Constraint Language, v2.2. Tech. rep., 2010

[226] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, v1.1. Tech. rep., 2011

[227] Olsina, L., Lafuente, G. and Pastor, O. Towards a Reusable Repository for Web Metrics. *J. Web Eng.*, vol. 1(1), pp. 61–73, 2002

[228] Open Source Initiative. The Open Source Definition

[229] OpenBRR.org. Business Readiness Rating for Open Source. Tech. rep., OpenBRR.org, 2005

[230] Origin, A. Method for Qualification and Selection of Open Source Software (QSOS), v1.6. Tech. rep., 2006

[231] Owens, M. *The Definitive Guide to SQLite*. Apress, Berkely, CA, USA, 2006

[232] Paige, R. F., Ostroff, J. S. and Brooke, P. J. Principles for Modeling Language Design. *Information and Software Technology*, vol. 42(10), pp. 665–675, 2000

[233] Papamarkos, G., Poulovassilis, A. and Wood, P. T. Event-Condition-Action Rule Languages for the Semantic Web. In: *Proceedings of the 2003 International Workshop on Semantic Web and Databases (SWDB '03)*

[234] Park, R. E. Software Size Measurement: A Framework for Counting Source Statements. Tech. Rep. CMU/SEI-92-TR- 20, ESC-TR-92-20, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, 1992

[235] Park, R. E., Goethert, W. B. and Florac, W. A. Goal Driven Software Measurement - A Guidebook. Tech. Rep. CMU/SEI-96-BH-002, Software Engineering Institute, Carnegie Mellon University, 1996

[236] Pastor, O., Fons, J., Pelechano, V. and Abrahão, S. Conceptual Modelling of Web Applications: The OOWS Approach. In: Mendes, E. and Mosley, N., eds., *Web Engineering*, pp. 277–302. Springer, 2006

[237] Plotkin, G. D. The origins of structural operational semantics. *J. Log. Algebr. Program.*, vol. 60-61, pp. 3–15, 2004

[238] Potencier, F. and Zaninotto, F. *The Definitive Guide to Symfony*. Apress, 2007

[239] Preciado, J. C., Linaje, M., Sanchez, F. and Comai, S. Hypermedia Systems Development: Do We Really Need New Methods? In: *Proceedings of the Informing Science + IT Education Conference (IS2002)*. Cork, Ireland, 2002

[240] Preciado, J. C., Linaje, M., Sanchez, F. and Comai, S. Necessity of Methodologies to Model Rich Internet Applications. In: *Proceedings of the 7th IEEE International Symposium on Web Site Evolution (WSE '05)*, pp. 7–13. IEEE Computer Society, Washington, DC, USA, 2005

[241] Preece, J., Rogers, Y. and Sharp, H. *Interaction Design: Beyond Human-Computer Interaction*. Wiley, Indianapolis, IN, 2002

[242] Pressman, R. S. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 5th ed., 2001

[243] Proctor, M., Neale, M., Frandsen, M., Jr., S. G., Tirelli, E., Meyer, F. and Verlaenen, K. Drools Documentation. Tech. rep., JBoss.org, 2008

[244] Prototype Core Team. Prototype JavaScript Framework, 2007

[245] Recordon, D. and Reed, D. OpenID 2.0: a platform for user-centric identity management. In: *Proceedings of the second ACM workshop on Digital identity management (DIM '06)*, pp. 11–16. ACM, New York, NY, USA, 2006

[246] Resnick, P. Internet Message Format. RFC 5322 (Draft Standard), 2008

[247] Reynolds, D. *Jena 2 Inference support*, 2010

[248] Ricca, F. and Tonella, P. Analysis and Testing of Web Applications. *International Conference on Software Engineering*, vol. 0, p. 0025, 2001

[249] Robbins, J. E. and Redmiles, D. F. Cognitive support, UML adherence, and XMI interchange in Argo/UML. *Information and Software Technology*, vol. 42(2), pp. 79–89, 2000

[250] Rose, L. M., Paige, R. F., Kolovos, D. S. and Polack, F. A. An Analysis of Approaches to Model Migration. *In Proceedings of the Joint MoDSE-MCCM Workshop at MODELS '09*, pp. 6–15, 2009

[251] Rossi, G. and Schwabe, D. Model-Based Web Application Development. In: Mendes, E. and Mosley, N., eds., *Web Engineering*, pp. 303–333. Springer, 2006

[252] Rossi, M. and Brinkkemper, S. Complexity Metrics for Systems Development Methods and Techniques. *Inf. Syst.*, vol. 21(2), pp. 209–227, 1996

[253] RSS Advisory Board. RSS 2.0 Specification. Tech. rep., RSS Advisory Board, 2007

[254] Rumbaugh, J. E. Notation Notes: Principles for Choosing Notation. *JOOP*, vol. 9(2), pp. 11–14, 1996

[255] Samarati, P. and di Vimercati, S. D. C. Access Control: Policies, Models, and Mechanisms. *Foundations of Security Analysis and Design*, pp. 137–196, 2001

[256] Sandhu, R. S., Coyne, E. J., Feinstein, H. L. and Youman, C. E. Role-Based Access Control Models. *Computer*, vol. 29(2), pp. 38–47, 1996

[257] Sargent, R. G. Verification and validation of simulation models. In: *Proceedings of the 37th conference on Winter simulation*, WSC '05, pp. 130–143. Winter Simulation Conference, 2005

[258] Schewe, K.-D. The Challenges in Web Information Systems Development in 15 Pictures (Invited Talk). In: *ISTA*, pp. 204–215. 2005

[259] Schleicher, A. and Westfechtel, B. Beyond stereotyping: Metamodeling approaches for the UML. In: Sprague, Jr., R. H., ed., *Proc. 34th Annual Hawaii International Conference on System Sciences (HICSS-34)*. IEEE Computer Society, 2001

[260] Schmidt, D. A. *Denotational Semantics: A Methodology for Language Development*. William C. Brown Publishers, Dubuque, IA, USA, 1986

[261] Schmidt, D. C. Guest editor's introduction: Model-driven engineering. *Computer*, vol. 39(2), pp. 25–31, 2006

[262] Schwinger, W. and Koch, N. Modeling Web Applications. In: Kappel, G., Pröll, B., Reich, S. and Retschitzegger, W., eds., *Web Engineering – The Discipline of Systematic Development of Web Applications*, chap. 3. John Wiley & Sons Ltd., England, 2006

[263] Sciascio, E. D., Donini, F. M., Mongiello, M. and Piscitelli, G. AnWeb: a system for automatic support to web application verification. In: *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE '02)*, pp. 609–616. ACM, New York, NY, USA, 2002

[264] Seaborne, A. RDQL - A Query Language for RDF. Tech. rep., W3C Member Submission 9 January 2004, 2004

[265] Selic, B. The Pragmatics of Model-Driven Development. *IEEE Software*, vol. 20, pp. 19–25, 2003

[266] Selman, D. JSR 94: Java Rule Engine API. Tech. rep., Java Community Process, 2004

[267] Selmi, S. S., Kraïem, N. and Ghézala, H. H. B. Toward a Comprehension View of Web Engineering. In: *Proceedings of the 5th International Conference on Web Engineering (ICWE '05)*, pp. 19–29. 2005

[268] Sendall, S. and Kozaczynski, W. Model transformation: the heart and soul of model-driven software development. *Software, IEEE*, vol. 20(5), pp. 42–45, Sept.-Oct. 2003

[269] Siau, K. and Cao, Q. Unified Modeling Language (UML) - a Complexity Analysis. vol. 12, pp. 26–34, 2001

[270] Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A. and Katz, Y. Pellet: A practical OWL-DL reasoner. *J. Web Sem.*, vol. 5(2), pp. 51–53, 2007

[271] Sottara, D., Mello, P. and Proctor, M. Adding Uncertainty to a Rete-OO Inference Engine. In: *Proceedings of the RuleML International Symposium (RuleML 2008)*, pp. 104–118. 2008

[272] SpringSource. Spring.NET Application Framework, 2007

[273] Steinberg, D., Budinsky, F., Paternostro, M. and Merks, E. *EMF: Eclipse Modeling Framework*. Addison-Wesley Longman, Amsterdam, 2nd ed., 2009

[274] Steindl, C., Ramler, R. and Altmann, J. Testing Web Applications. In: Kappel, G., Pröll, B., Reich, S. and Retschitzegger, W., eds., *Web Engineering – The Discipline of Systematic Development of Web Applications*, chap. 7. John Wiley & Sons Ltd., England, 2006

[275] Strachey, C. Fundamental Concepts in Programming Languages. *Higher-Order and Symbolic Computation*, vol. 13(1/2), pp. 11–49, 2000

[276] Tarski, A. *Introduction to Logic and to the Methodology of Deductive Sciences*. Oxford University Press, New York, third ed., 1965

[277] The OSGi Alliance. *OSGi Service Platform Core Specification, Release 4.2*, 2009

[278] Thomas, D., Hansson, D., Breedt, L., Clark, M., Davidson, J. D., Gehtland, J. and Schwarz, A. *Agile Web Development with Rails*. Pragmatic Bookshelf, 2 ed., 2006

[279] Tolke, L. and Klink, M. *Cookbook for Developers of ArgoUML: An Introduction to Developing ArgoUML*, 2004

[280] Topley, K. *J2ME in a Nutshell* . O'Reilly Media, 2002

[281] Torres, V., Giner, P. and Pelechano, V. Developing bp-driven web applications through the use of mde techniques. *Software and Systems Modeling*, pp. 1–23, 2010

[282] van Gelder, A., Ross, K. A. and Schlipf, J. S. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, vol. 38, pp. 620–650, 1991

[283] van Welie, M., van der Veer, G. C. and Eliëns, A. Patterns as Tools for User Interface Design. In: *Tools for Working with Guidelines*, pp. 313–324. Springer-Verlag, London, Great Britain, 2001

[284] Vlissides, J. M. and Linton, M. A. Unidraw: a framework for building domain-specific graphical editors. *ACM Trans. Inf. Syst.*, vol. 8(3), pp. 237–268, 1990

[285] Voss, A. argo2ecore, 2009

[286] Vépa, E., Bézivin, J., Brunelière, H. and Jouault, F. Measuring model repositories. In: *Proceedings of the First Workshop on Model Size Metrics (MSM '06)*. 2006

[287] W3C Group. Precision Graphics Markup Language (PGML). Tech. rep., W3C Note 10 April 1998, 1998

[288] W3C Group. HTML 4.01 Specification. Tech. rep., W3C Recommendation 24 December 1999, 1999

[289] W3C Group. Document Object Model (DOM) Level 2 Events Specification. Tech. rep., W3C Recommendation 13 November 2000, 2000

[290] W3C Group. Document Object Model (DOM) Level 3 Core Specification. Tech. rep., W3C Recommendation 07 April 2004, 2004

[291] W3C Group. OWL Web Ontology Language Semantics and Abstract Syntax . Tech. rep., W3C Recommendation 10 February 2004, 2004

[292] W3C Group. XML Schema Part 1: Structures Second Edition. Tech. rep., W3C Recommendation 28 October 2004, 2004

[293] W3C Group. XML Schema Part 2: Datatypes Second Edition. Tech. rep., W3C Recommendation 28 October 2004, 2004

[294] W3C Group. Simple Object Access Protocol (SOAP) Version 1.2. Tech. rep., W3C Recommendation 27 April 2007, 2007

[295] W3C Group. XSL Transformations (XSLT) Version 2.0. Tech. rep., W3C Recommendation 23 January 2007, 2007

[296] W3C Group. Extensible Markup Language (XML) 1.0 (Fifth Edition). Tech. rep., W3C Recommendation 26 November 2008, 2008

[297] W3C Group. HTML 5: A vocabulary and associated APIs for HTML and XHTML. Tech. rep., W3C Working Draft 26 February 2008, 2008

[298] W3C Group. Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification. Tech. rep., W3C Working Draft 07 December 2010, 2010

[299] W3C Group. XML Path Language (XPath) 2.0 (Second Edition). Tech. rep., W3C Recommendation 14 December 2010, 2010

[300] W3C Group. XQuery 1.0: An XML Query Language (Second Edition). Tech. rep., W3C Recommendation 14 December 2010, 2010

[301] W3C Group. XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition). Tech. rep., W3C Recommendation 14 December 2010, 2010

[302] W3C OWL Working Group. OWL 2 Web Ontology Language Document Overview. Tech. rep., W3C Recommendation 27 October 2009, 2009

[303] Wagner, G. The Agent-Object-Relationship Metamodel: Towards a Unified View of State and Behavior. *Information Systems*, vol. 28(5), pp. 475–504, 2003

[304] Wagner, G., Antoniou, G., Tabet, S. and Boley, H. The Abstract Syntax of RuleML - Towards a General Web Rule Language Framework. In: *Proceedings of the 2004 International Conference on Web Intelligence (WI '04)*, pp. 628–631. IEEE Computer Society, Washington, DC, USA, 2004

[305] Wagner, G., Antoniou, G., Taveter, K., Berndtsson, M. and Spreeuwenberg, S. A First-Version Visual Rule Language. deliverable I1-D1, Faculty of Technology Management – Eindhoven University of Technology, 2004

[306] Wagner, G., Giurca, A. and Lukichev, S. Modeling Web Services with URML. In: *Proceedings of Workshop Semantics for Business Process Management 2006, Budva, Montenegro (11th June 2006)*. 2006

[307] Wagner, R. *Professional Flash Mobile Development: Creating Android and iPhone Applications*. Wrox, 2011

[308] Wampler, D. Cat Fight in a Pet Store: J2EE vs. .NET. Tech. rep., ONJava.com, 2001

[309] Weaver, J. and Weaver, J. L. *JavaFX Script: Dynamic Java Scripting for Rich Internet/Client-side Applications*. Safari Books Online. Apress, 2007

[310] WebRatio Group. WebRatio AJAX Extension, 2007

[311] Wenz, C. *Essential Silverlight*. O'Reilly, 1st ed., 2008

[312] Winer, D. XML-RPC Specification. Tech. rep., UserLand Software, 1999

[313] Wirth, N. On the design of programming languages. In: *IFIP Congress*, pp. 386–393. 1974

[314] Wong, P. Y. H. and Gibbons, J. A Process Semantics for BPMN. In: *Proceedings of the 10th International Conferenece on Formal Engineering Methods (ICFEM '08)*, pp. 355–374. Springer, Kitakyushu-City, Japan, 2008

[315] Wright, J. ModelDoc: A Model-Driven Framework for the Automated Generation of Modelling Language Documentation. In: *TODO To Appear in the Proceedings of the 8th Asia-Pacific Conference on Conceptual Modelling (APCCM 2012)*. Melbourne, Australia, 2012

[316] Wright, J. and Dietrich, J. Requirements for Rich Internet Application Design Methodologies. In: *Proceedings of the 9th International Conference on Web Information Systems Engineering (WISE 2008)*. Auckland, New Zealand, 2008

[317] Wright, J. and Dietrich, J. Survey of Existing Languages to Model Interactive Web Applications. In: *Proceedings of the 5th Asia-Pacific Conference on Conceptual Modelling (APCCM 2008)*. Wollongong, NSW, Australia, 2008

[318] Wright, J. and Dietrich, J. 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, 2010

[319] Zandstra, M. Testing with PHPUnit. In: *PHP Objects, Patterns, and Practice*, pp. 379–405. Apress, 2010

[320] Zhang, G., Baumeister, H., Koch, N. and Knapp, A. Aspect-Oriented Modeling of Access Control in Web Applications. In: *6th Int. Workshop Aspect Oriented Modeling (AOM'05)*. Chicago, USA, 2005

[321] Zhang, Y. and Xu, B. A survey of semantic description frameworks for programming languages. *SIGPLAN Notices*, vol. 39(3), pp. 14–30, 2004

[322] Zhu, N., Grundy, J. and Hosking, J. Pounamu: A Meta-tool for Multi-View Visual Language Environment Construction. In: *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, VLHCC '04, pp. 254–256. IEEE Computer Society, Washington, DC, USA, 2004

[323] Zimmer, D. and Unland, R. On the Semantics of Complex Events in Active Database Management Systems. In: *Proceedings of the 15th International Conference on Data Engineering*, pp. 392–399. 1999

Index