Non-Monotonic Model Completion in Web Application Engineering

Jevon M. Wright
School of Engineering and Advanced Technology
Massey University
Palmerston North, New Zealand
j.m.wright@massey.ac.nz

Jens B. Dietrich
School of Engineering and Advanced Technology
Massey University
Palmerston North, New Zealand
j.b.dietrich@massey.ac.nz

***Abstract*—Formal models are often used to verify systems and prove their correctness, and ensure that transformed models remain consistent to the original system. However, formal techniques can also be used to add reasoning in the engineering of models, predicting the developers intentions. On a programming level, a similar approach has been used very successfully in several web application frameworks. Promoting formal techniques to the web application domain is useful as web application developers increasingly need to abstract from a growing set of target platforms and technologies.**

We address this need and propose model completion, a formal framework to infer modelling elements. Model completion is a non-monotonic process and formalises the notion of the intended model, permitting the web application developer to focus on application design rather than scaffolding. Benchmarking an implementation of this process for a platform-independent web application modelling language illustrates its significant potential to simplify model-driven development.

***Index Terms*—model-driven development, non-monotonicity, rich internet applications**

I. Introduction

Web application frameworks such as Ruby on Rails [1] and Symfony [2] aim to simplify development of web applications by providing documented conventions, which allow the developer to automatically generate much of the required scaffolding of web applications, yet still offer the developer rich functionality and flexibility if necessary. Frameworks in other domains also successfully use this approach [3]. In the web application field, this scaffolding includes repetitive steps such as form creation, database initialisation, and cross-platform scripting.

We propose applying the same approach used by frameworks to assist a *model developer* to implement model instances; in particular, we permit the developer to specify a smaller *base model* following conventions declared by the *meta-model designer*, which can then be expanded in-place into an *intended model*. This process adds a flexible level of abstraction to a modelling language, similar to the abstractness offered by a framework.

Importantly, we argue that model completion also needs to be non-monotonic. This means that the introduction of new modelling artefacts into a system may require the retraction of previously generated artefacts to reflect this new knowledge. This retains the flexibility of the original modelling language. Non-monotonicity also ensures that the base model is never modified, allowing it to be easily integrated into existing approaches.

In this paper, we discuss the formal semantics of this process to show that non-monotonicity is consistent. We implement the declared conventions into a rule language, and show that a rule engine can adhere to these defined semantics. This *model completion* may then be executed by a rule engine without a significant performance penalty, despite previous work arguing this would be impossible.

In Section II we introduce models and model-driven development, and illustrate how these concepts can assist in the development of complex web applications. We discuss the concepts behind model completion in Section III, and introduce the formal semantics of this process in Section IV, along with two examples illustrating the implementation of non-monotonicity. In Section V, we discuss the implementation of model completion as part of a CASE tool, which we then use in Section VI to evaluate model completion against a rich suite of test models and model completion rules. We discuss some existing work related to model completion and its implementation in Section VII. Finally, we conclude with a discussion of our results and future work in Section VIII.

II. Background

A model can encompass a wide range of concepts; 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 [4]. Models do not need to be separate; they can coexist in different modelling spaces and themselves be abstractions, or instances, of other models [5]. In particular, a model representing another model is its *meta-model*, and the process involved in translating or integrating between different models is a *model transformation*.

In our work we consider a model to be an abstract representation of a system under development, which can assist in implementing the real-world system. For example, UML class diagrams allow us to design an object-oriented software system in terms of classes, inheritance, relationships, attributes and other artefacts [6].

We can also use models to represent different views of a system; for example, a UML activity diagram can represent the flow of an operation, and a UML sequence diagram can represent the flows between different operations. Model-Driven Engineering (MDE) provides a framework to integrate different types of models together; this is the approach taken by UML-based Web Engineering [7].

When designing a modelling language, a key challenge is balancing the level of detail in its design. Too much abstractness will result in a rigid approach that cannot adapt to many situations; too much flexibility will result in models which are large and unmaintainable. In order to add abstractness without sacrificing flexibility, frameworks add abstraction to an existing language, yet retain access to its flexibility when necessary [3].

We are particularly interested in the model-driven development of web applications. The latest generation of interactive web applications named Rich Internet Applications (RIAs) require complex applications that are scalable, flexible and platform-independent [8]. Despite the standardisation of implementation technologies such as CSS and SVG, different platforms often provide incomplete implementations; a significant portion of time spent in web development is focused on resolving these differences.

The development of a web application modelling language to abstract away the technical details of implementation is a promising idea, and is a strong focus of existing research. However, the challenge in balancing abstractness and flexibility of a modelling language for RIAs is particularly evident, as no existing modelling language supports all fundamental RIA concepts [9].

A steady stream of new approaches to client-side applications, such as HTML 5, Google Gears, Mozilla's Prism, Microsoft's Silverlight and Adobe's AIR, continue to be published. By developing a platform-independent model, we can remove the unnecessary technical implementation details and differences between platforms, and provide an abstract model focused on the design of web applications. This model can then be used to generate and deploy the application; this approach is advocated by the OMG's Model-Driven Architecture proposal [10]. Alternatively, virtual machines can be designed to execute a model on a range of platforms, but it is difficult to add additional layers to the already limited capabilities of mobile devices.

Web development for RIAs is particularly difficult due to the wide range of business ideas, technologies, platforms and devices that are rapidly emerging. Consequently, web development requires an agile, extensible and flexible modelling approach. Through non-monotonic model completion, we hope to accommodate abstractness, flexibility and extensibility in web application design.

By predicting the developers intentions, the meta-model designer can automatically complete much of the scaffolding necessary in web application design. The non-monotonicity of model completion permits the developer to override this generated scaffolding with their actual intent if necessary, adding a great deal of flexibility to the design process. We hope that combining this technique with a rich platform-independent model will improve reliability, security and efficiency in designing web applications.

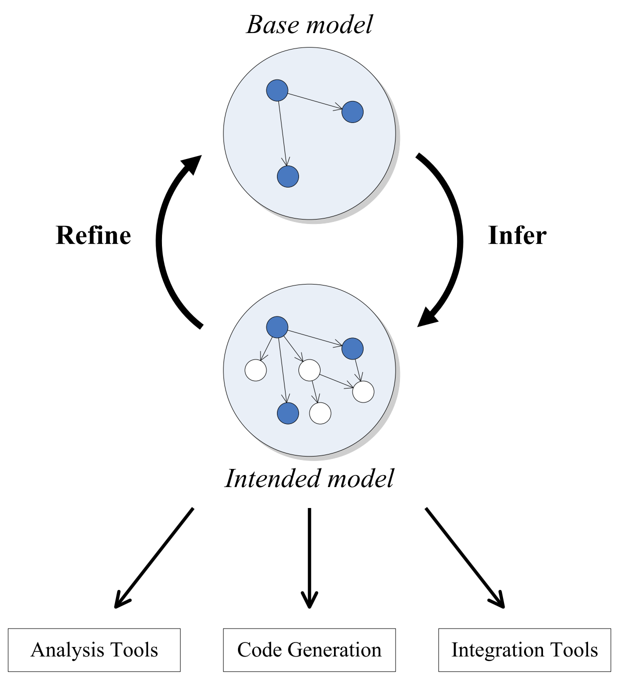

Fig. 1. Model completion within model development

III. Model Completion

Our approach takes many of the concepts from framework development and promotes them into a model-driven environment. In our case, the conventions within the framework are moved into a separate model transformation called *model completion*, illustrated in Fig. 1, which operates entirely under a single meta-model. Essentially, the developer is provided the task of implementing their system requirements without having to completely define all details.

A CASE tool takes this initial model and, following these documented conventions, transforms this model into an *intended model* complete with all necessary detail. Implementing this process within a CASE tool is essential in order to maximise the benefits of using MDE [11]. This model can then be used as input to other tools within MDE, such as code generators or analysis tools [12].

This approach also maps well onto iterative or incremental development processes; once completed, this intended model may be evaluated in order to obtain feedback. This feedback can then be used to refine the base model, starting another development iteration.

In this paper, we only discuss implementations of model completion provided by the *meta-model designers*, rather than the model developers or other third parties. Under most situations, the developer will be provided predefined model completion conventions and will not need to deal with the complexity of defining these conventions. As such, discussing the requirements for developers to contribute and integrate their own conventions is well outside the scope of this paper.

Importantly, the process involved in model completion is usually based on incomplete knowledge of the system under design – known as *non-monotonic reasoning* [13]. 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 technical details of its implementation. It is reasonable to assume that by default, this property should be rendered by a checkbox. However, it may be more appropriate in some cases to represent this property with a drop-down list with the values *yes* and *no*. If the model developer were to add this new knowledge, the default checkbox should be removed and no longer be generated nor replace the new design.

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

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

We restrict the range of retraction within non-monotonic reasoning to only those 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 the developers' effort is never inadvertently discarded.

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 is a sound transformation step, remaining consistent with the original model [14]. As shown in Section V, these semantics can map to a simple implementation within a commercial rule engine.

IV. Semantics

If we consider a model to consist of a set of model artefacts M, we can propose that all possible models are within the universe of possible model artefacts; i.e. $model \in 2^M$. To restrict models to only valid model instances in our domain, we define a meta-model $\mathcal{S}$ to be the valid range of all possible models, $\mathcal{S} \subseteq 2^M$.

For example, consider a model universe 2^M consisting of all possible combinations of properties, property editors and their relationships. Since it does not make sense to have a property editor without a property, we restrict the meta-model $\mathcal{S}$ to only those models where each editor is expressing a property. Any set of model artefacts within this meta-model definition is therefore a valid model instance.

A *model transformation* is the process of converting source models into target models; in this paper, we only consider transformations from one source model to one target model. If we are concerned with two meta-models $\mathcal{S}_1, \mathcal{S}_2$, a model transformation can be portrayed as a function $T : \mathcal{S}_1 \rightarrow \mathcal{S}_2$. If the target meta-model is also the source meta-model, then the transformation can be simplified as a single-model transformation $T_1 : \mathcal{S} \rightarrow \mathcal{S}$.

A. Model Completion

Given a meta-model $\mathcal{S}$, we can formally define a model completion as a function $C : \mathcal{S} \rightarrow \mathcal{S}$ operating within the same meta-model. That is, all completed models will also be valid models in our domain. If we consider model completion as an operation $C(X)$ operating on a model $X \subseteq \mathcal{S}$, there are a number of conditions we need to impose:

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 result in a different model, i.e. $C(X) = C(C(X))$.

These two conditions are part of Tarski's classical axioms for inference operators [15]. However, we do not require monotony; in the face of new information within a base model, previously inferred knowledge may need to be retracted, i.e. $X \subseteq Y \not\Rightarrow C(X) \subseteq C(Y)$.

The main idea behind model completion is to infer new elements and relationships based on the base model, so we define the concept of a *factory function* as a function which introduces a new element or relationship to the model. To support this, we introduce a ranking function that layers elements of the domain (universe) to prevent inference problems. Consequently, we need to introduce additional constraints into the semantics in order to support this technique.

Models can be portrayed as sets of logical formula if we consider artefacts and primitive values as *constants*, and properties and relationships as *relations*. This view is very similar to OWL [16]; here, resources (constants) are typed using classes defined in OWL (the meta-model), and properties and relationships are represented as data and object properties, respectively.

In the following sections, we provide the semantics for the model completion process. Our definition of the semantics of model completion is a variant of the standard semantics of first-order logic. The notation used is based on previous work by Fitting [17].

B. L-Model

Definition 1 (L-Model): Given a language $L(R, F, C, V, =)$ that consists of finite, pairwise disjoint sets of relations R, functions F, constants C and variables V, and a distinguished binary relation symbol = for equality, we define a *L-model* to be a structure $M = \langle D, rank, I, eq \rangle$ where D is an non-empty set, called the domain; $eq \subseteq D \times D$ is a binary relation that is reflexive, symmetric and transitive; $rank : D \rightarrow \mathbb{N}$

is a function that associates domain elements with non-negative integer numbers; and I is an interpretation function that associates:

1) every constant symbol $c \in C$ to a element $c^I \in D$ such that $rank(c^I) = 0$.
2) every n-place function symbol $f \in F$ with a n-ary injective function $f^I : D^n \rightarrow D$ such that $rank(f^I(t_1, ..., t_n) = 1 + max_{i=1..n}(rank(t_i)))$.
3) every n-place relation symbol $P \in R$ with a n-ary relation $P^I \subseteq D^n$.
4) $=$ with eq.

The ranking function allows the effective stratification of terms. Constant terms have a *rank* 0, and the application of functions increments the *rank* of the resulting complex term by 1. Function symbols represent only injective functions, allowing the creation of new model terms; i.e., if $f^I : D^n \rightarrow D$ is a n-ary function, and $t_i \neq t'_i$ holds for some i between 1 and k, then $f^I(t_1, .., t_n) \neq f^I(t'_1, .., t'_n)$ is true.

The ranking function is also necessary in order to prevent inference problems. Consider a rule which creates a new element only if the element did not yet exist; but by creating this element, this rule could never have been fired. Ranking allows us to create new elements within a different space to the current evaluation, resolving this problem.

Intuitively, these functions represent caching; i.e., whenever a function is executed the results are unique for a given set of parameters. The definition of $rank$ for complex terms also enforces that complex terms cannot be interpreted by the same domain elements used to interpret their parts, and implies that functions can only introduce terms with a higher rank. This technique is similar to the use of stratification in logic programming with negation [18].

C. Assignment

Definition 2 (Assignment): Given an L-model $M = \langle D, rank, I, eq \rangle$ and a language $L(R, F, C, V, =)$, an assignment A is defined as a mapping that associates variable symbols $v \in V$ with elements $v^A \in D$. Given an interpretation I and an assignment A, we associate terms with domain elements as follows:

1) $c^{I,A} = c^I$ for constants.
2) $v^{I,A} = v^A$ for variables.
3) $[f(t_1, .., t_n)]^{I,A} = f^I(t_1^{I,A}, .., t_n^{I,A})$.

D. Validity

Definition 3 (Validity): A formula Φ is valid for an assignment A in an L-model M, written $M \models_A \Phi$, iff the following conditions are satisfied:

1) $M \models_A P(t_1, .., t_n)$ iff $\langle t_1^{I,A}, .., t_n^{I,A} \rangle \in P^I$ for atomic formulas.
2) $M \models_A \Phi_1 \wedge \Phi_2$ iff $M \models_A \Phi_1$ and $M \models_A \Phi_2$.
3) $M \models_A \Phi_1 \vee \Phi_2$ iff $M \models_A \Phi_1$ or $M \models_A \Phi_2$.
4) $M \models_A \neg\Phi$ iff not $M \models_A \Phi$.
5) $M \models_A \exists x : \Phi$ iff there exists an assignment A' that assigns the same values to all variables as A except possibly x (a so-called x-variant of A) such that $M \models_{A'} \Phi$ and $rank(x^{A'}) \leq max(rank(t_i^{I,A'}))$, where $\{t_i\}$ is the set of terms occurring in Φ.

A formula Φ is said to be valid in an L-model M, written $M \models \Phi$, iff $M \models_A \Phi$ for all possible variable assignments A.

The last condition constrains the existential quantifiers, otherwise their satisfaction would be prevented by the injective factory functions. Their semantics is only defined with respect to a certain layer within the domain; this layering is achieved through the $rank$ function. These formulas allow us to construct a *rule program* which we can use to implement model completion for a particular modelling domain.

E. Inference

Definition 4 (Inference): Given an L-model $M = \langle D, rank, I, eq \rangle$ and a language $L(R, F, C, V, =)$, the set of models of a set of formula is a function $Mod : 2^L \rightarrow 2^M$ defined as follows: $Mod(X) = \{M | \forall\Phi \in X : M \models \Phi\}$. The inference operator $C : 2^L \rightarrow 2^L$ could then defined as follows: $C(X) = \{\Phi | \forall M \in Mod(X) : M \models \Phi\}$.

This is the classical definition of inference, going back to Tarski [15]. In particular, this definition of C satisfies the three closure conditions of extensiveness, idempotence and monotonicity. However, monotony is not a desirable feature in our domain, as discussed earlier. Non-monotony can be achieved by restricting reasoning to selected models, by defining the *Herbrand model* [19].

F. Herbrand Model

Definition 5 (Herbrand Model): An L-model $M = \langle D, rank, I, eq \rangle$ is a Herbrand model for the language $L(R, F, C, V, =)$ if:

1) D is the set of variable-free terms of L.
2) $t^I = t$ for each variable free term t.
3) $\langle t_1, t_2 \rangle \in eq$ iff t_1 and t_2 are lexically identical.

A better inference operator can then be defined based on selected model(s) such as the Herbrand model. That is, given a selection function $\Sigma : 2^M \rightarrow 2^M$, $\Sigma(m) \subseteq m$, inference can be defined as $C(X) = \{\Phi | \forall M \in \Sigma(Mod(X)) : M \models \Phi\}$.

Multiple mechanisms of reasoning based on selected models have been investigated, including cumulative reasoning based on models selected by preference functions [13], [20], and reasoning in logic programming using stable [21] and well-founded models [18], [22]. Reasoning with these selected models allows the formal definition of intended models.

These semantics do not explicitly mention primitive types and their built-in predicates and functions. They can be easily integrated by using value spaces and lexical-to-value-space mappings. This technique is used by OWL to implement primitive types with formal semantics, which uses the facilities of RDF and XML Schema [23]. Model element types are implemented as unary relations, and type hierarchies can be implemented by adding formulas to infer supertype knowledge based on subtypes.

G. Model Completion Example

To illustrate the model completion process, we will provide two examples of model completion, based on the running *default checkbox rule* example in Section III. We will first illustrate completion, followed by non-monotonicity.

For this first example, we first define a language $L(R, F, C, V, =)$ as in Fig. 2. This language is a small subset of our platform-independent modelling language for web applications, in which we abstract common domain-specific elements away from their implementation.

$$\begin{aligned} R &= \{\mathit{property}, \mathit{editor}, \mathit{editorFor}, \mathit{checkbox}, \mathit{dropdown}\} \\ F &= \{\mathit{newCheckbox}\} \\ C &= \{\mathtt{a}\} \\ V &= \{x\} \end{aligned}$$

Fig. 2. Definition of a language L for a model A

We then define our rule program for the completion operator $C(A)$ as a set of formulas Φ, as in Fig. 3. This rule program is composed of both the initial model $A = \{\Phi_1\}$ and the model completion rules $C = \{\Phi_2, \Phi_3, \Phi_4\}$. In this initial model we only have one element – a *property* – which does not yet have an editor defined.

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

Fig. 3. Definition of the rule program for $C(A)$

From this we can obtain a Herbrand model M which satisfies the completion of this model, as in Fig. 4.

$$\begin{aligned} D &= \{\mathtt{a}, \mathit{newCheckbox}(\mathtt{a}), \\ &\quad \mathit{newCheckbox}(\mathit{newCheckbox}(\mathtt{a})), ...\} \\ \mathit{eq} &= \{(\mathtt{a}, \mathtt{a}), (\mathit{newCheckbox}(\mathtt{a}), \mathit{newCheckbox}(\mathtt{a})), ...\} \\ \mathit{rank} &= \mathit{rank}(\mathtt{a}) = 0, \\ &\quad \mathit{rank}(\mathit{newCheckbox}(\mathtt{a})) = 1, \\ &\quad ... \end{aligned}$$

Fig. 4. A Herbrand model M for $C(A)$

We can summarise the model completion process in Fig. 5. As expected, the completion has inferred a new checkbox editor for the initial boolean property, and the process is extensive: $A \subseteq C(A)$.

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

Fig. 5. Model completion $C(A)$

H. Non-monotonicity Example

In order to illustrate non-monotonicity, we will take the same example as before and follow the same model completion process, but add a drop-down list element to the initial model. This new knowledge will prevent the checkbox element from being created. We first define the language $L(R, F, C, V, =)$, as in Fig. 6.

$$\begin{aligned} R &= \{\mathit{property}, \mathit{editor}, \mathit{editorFor}, \mathit{checkbox}, \mathit{dropdown}\} \\ F &= \{\mathit{newCheckbox}\} \\ C &= \{\mathtt{a}, \mathtt{b}\} \\ V &= \{x\} \end{aligned}$$

Fig. 6. Definition of a language L for model B

We then define our rule program for the completion operator $C(B)$, as in Fig. 7. This rule program is composed of both the initial model $B = \{\Phi_1, \Phi_2, \Phi_3\}$, and the model completion rules $C = \{\Phi_4, \Phi_5, \Phi_6\}$. Importantly, the initial model $B \supseteq A$, but the completion rules remain unchanged.

$$\begin{aligned} \Phi_1 &= \mathit{property}(\mathtt{a}) \\ \Phi_2 &= \mathit{dropdown}(\mathtt{b}) \\ \Phi_3 &= \mathit{editorFor}(\mathtt{a}, \mathtt{b}) \\ \Phi_4 &= \mathit{property}(x) \land \neg\exists y : \mathit{editor}(y) \land \mathit{editorFor}(x, y) \\ &\quad \rightarrow \mathit{checkbox}(\mathit{newCheckbox}(x)) \\ &\quad\quad \land \mathit{editorFor}(x, \mathit{newCheckbox}(x)) \\ \Phi_5 &= \mathit{checkbox}(x) \rightarrow \mathit{editor}(x) \\ \Phi_6 &= \mathit{dropdown}(x) \rightarrow \mathit{editor}(x) \end{aligned}$$

Fig. 7. Definition of the rule program for $C(B)$

From this we can obtain a Herbrand model M which satisfies the completion of this model, as in Fig. 8.

$$\begin{aligned} D &= \{\mathtt{a}, \mathtt{b}, \mathit{newCheckbox}(\mathtt{a}), \mathit{newCheckbox}(\mathtt{b}), ...\} \\ \mathit{eq} &= \{(\mathtt{a}, \mathtt{a}), (\mathtt{b}, \mathtt{b}), \\ &\quad (\mathit{newCheckbox}(\mathtt{a}), \mathit{newCheckbox}(\mathtt{a})), ...\} \\ \mathit{rank} &= \mathit{rank}(\mathtt{a}) = 0, \\ &\quad \mathit{rank}(\mathtt{b}) = 0, \\ &\quad \mathit{rank}(\mathit{newCheckbox}(\mathtt{a})) = 1, \\ &\quad ... \end{aligned}$$

Fig. 8. A Herbrand model M for $C(B)$

The introduction of the new knowledge resulted in a different completed model $C(B) \not\supseteq C(A)$, illustrated in Fig. 9, due to the injective nature of functions. This clearly demonstrates our desired non-monotonicity condition of model completion.

$$\begin{aligned} B &= \{\mathit{property}(\mathtt{a}), \mathit{dropdown}(\mathtt{b}), \mathit{editorFor}(\mathtt{a}, \mathtt{b})\} \\ C(B) &= \{\mathit{property}(\mathtt{a}), \mathit{dropdown}(\mathtt{b}), \mathit{editorFor}(\mathtt{a}, \mathtt{b})\} \end{aligned}$$

Fig. 9. Model completion $C(B)$

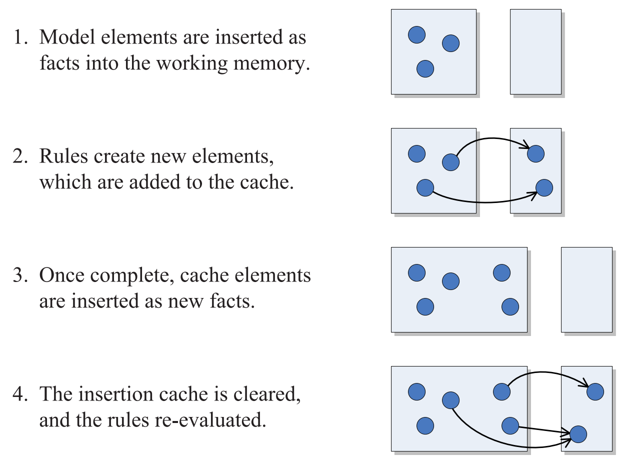
1. Model elements are inserted as facts into the working memory.
2. Rules create new elements, which are added to the cache.
3. Once complete, cache elements are inserted as new facts.
4. The insertion cache is cleared, and the rules re-evaluated.

Fig. 10. Recursively creating levels of elements

V. Implementation

As part of a larger research work, we are designing and implementing a platform-independent modelling language for RIAs. This domain-specific language is called the *Internet Application Modelling Language*, which aims to include fundamental web concepts as first-class modelling concepts. This language is implemented using the Eclipse Modeling Framework (EMF) environment within Eclipse [24], forming a part of a CASE tool for the design, development, generation and deployment of RIAs. A full description of this language is far outside the scope of this paper.

Model completion has been used extensively within our work to simplify the development of RIAs. In our current implementation, model completion is executed by the high-performance JBoss Drools rule engine [25] using individual rules implemented in the DRL dialect; however, Drools supports a wide range of rule sources, including from databases, web services and custom grammars.

Translating our semantics of model completion into this implementation is straightforward. The only major challenge is in implementing the $rank$ function. By default, a rule creating a new model element would be inserted immediately into the working memory; without this insertion, the new model element cannot trigger additional rules recursively, as the inference process is usually only evaluated once.

In our work, we have modified this approach to place newly inserted elements into an *insertion cache*, and evaluate our set of rules repeatedly. In each iteration, the cache is cleared, and newly created elements are instead stored in this cache. These elements in this cache are those of the next rank level. Once the rule evaluation is complete, these new elements are inserted into the working memory, and the process begins again. This process is illustrated in Fig. 10.

Implementing this insertion cache into standard Drools rules is straightforward; a sample implementation of the *default checkbox rule* example is illustrated in Fig. 11. A *handler* object maps injective functions to the EMF framework, which instantiates new objects.

```
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
```

Fig. 11. Example rule implementation in Drools

We can safely declare that the process has completed once the insertion cache is empty following an evaluation iteration. Since our rule program can only insert new elements, and the model completion process is extensive, the final model cannot change without the insertion of additional facts into the working memory. We are also satisfied that our implementation satisfies the semantics detailed in Section IV.

However, to prove that a given set of rules satisfy termination is much more difficult. In the worst case, solving this problem would require a solution to the halting problem, as rules can recursively generate new elements which trigger rules later. It is up to the meta-model designer to ensure these rules eventually terminate; in our work, we limit the number of iterations to a fixed number, which we investigate in the next section.

VI. Evaluation

As part of the process in designing our modelling language, we have designed a comprehensive suite of test models representing individual model concepts. Each test model represents a use case of our modelling language, derived from our earlier work on investigating the requirements of RIAs [26]. These test models are used to ensure the modelling language is expressive; checking for potential conflicts between inference rules; to assist in meta-model refactoring; for developing a code generation framework; as input for integration tests; and as a rich suite of example models for designers and developers. Each model is serialised by EMF as an XMI file, simplifying transportation, versioning, and integration with other tools.

At the time of writing, our test model suite consisted of 110 test models. Ideally we would use a suite of model instances sourced from industry, but such a corpus is not yet available. However, as part of the evaluation process, this suite of models attempts to cover every abstraction used in the modelling process, so this should be a reasonable data set on which to make observations against. The suite of models used may be browsed online at `http://openiaml.org`.

EMF model instances may be interpreted as directed graphs, allowing us to compare different model instances abstractly. Object instances are translated into vertices, and references (including children references) between object instances are translated into edges between these vertices. This is similar to translating EMF models into an RDF graph [27].

	Minimum		Median		Mean		Maximum		Std Dev	
Metric	Initial	Final	Initial	Final	Initial	Final	Initial	Final	Initial	Final
Elements	2	3	11.5	107	15.1	207.5	76	2,094	13.2	338.9
Attributes	9	9	48	201	58.8	391.7	310	3,934	51.0	635.8
Non-default Attributes	4	6	23	166	28.0	321.8	148	3,219	24.7	520.2
Distinct Attribute Values	6	8	24	135	29.3	240.8	138	2,176	22.7	353.6
References	0	0	12	164	16.5	351.4	112	3,632	21.5	584.7
Children	1	2	10.5	106	14.1	206.5	75	2,093	13.2	338.9
Distinct Types	2	3	9	19	8.9	21.8	19	41	3.7	8.9
Min Degree (References and Children)	0	0	0	0	0.3	0.0	1	1	0.5	0.1
Max Degree (References and Children)	1	1	4	10	5.0	14.0	25	57	3.5	8.6
Children Depth	1	2	3	5	3.2	4.6	4	6	0.7	0.6
Cycles (References and Children)	0	0	2	2	3.0	4.8	20	44	3.6	8.0
Diameter (References and Children)	1	2	5	12	5.6	14.2	16	27	2.9	6.5
Average Completion Time (ms)		0.0		8.6		117.6		5,213.6		696.8

TABLE I
SELECTED MODEL METRICS OF INITIAL AND COMPLETED TEST MODELS ($n = 110$)

In order to assert that these test models are in fact useful instances on which to make observations against, we have investigated each of these test models and their interpreted graphs against certain complexity metrics. A selection of these metrics can be found in Table I. In these metrics, we can identify the degree, depth and diameter of our test models, illustrating the range of complexity and that these models are not trivial. We also expect the possibility of reference cycles in input models.

Since a model instance can be translated into a graph, this graph can be visualised. In Fig. 12 we take one of the smaller test models and illustrate the effect of model completion; each node or edge represents a domain-specific RIA concept. The elements within the initial model are highlighted, and the root element of the model – the element which directly or indirectly contains all other elements in the model – is indicated with a double line. References are represented with dashed lines, and children represented with solid lines; attributes are ignored, as they do not add any useful information to the visualisation. In this visualisation, we can see that model completion automatically completes much of the repetitive (and structurally similar) elements of a model.

A. Model Completion

At the time of writing, our model completion implementation is represented by a suite of 131 model completion rules in the DRL dialect of Drools. These rules are used to complete models in our modelling language to the level of detail necessary for code generation. The suite of model completion rules used may also be browsed online at `http://openiaml.org`, illustrating the effort required and techniques used.

Importantly, our suite of test models and completion rules can be utilised to suggest an upper bound for the fixed limit of inference iterations to evaluate. By executing model completion whilst simultaneously keeping track of where new elements are created, we can record the distribution of the $rank$ function. The aggregated results of model completion against our test models is provided in Fig. 13. In particular, the maximum $rank$ value found from all test models is 9; we therefore conservatively suggest an iteration limit of 20.

This graph also illustrates an interesting property of our test models; most elements are usually generated in the second iteration of model completion. This is a property of the metamodel and the rules used. For example, in our work, the first iteration is usually used to declare the signatures of permissible operations of model elements; in the second iteration, the default contents of the operation (temporary variables, execution paths, data flows, conditions, etc.) are defined, and involves many more elements. It would be interesting to investigate which types of elements are generated at each iteration.

In Table I, we re-evaluate the same model metrics against the completed models. As expected, we can see that model completion consistently increases the complexity of models in terms of every metric. In particular, the number of elements, attributes, references and children all increase significantly. It is interesting to note that none of our completed models have more than six layers of children (*children depth*).

We can also use these results to understand the potential benefits of using model completion. Fig. 14 illustrates that on average, the intended model contains 1082% more elements than the base model. This represents a significant saving of developer effort.

In our implementation, model completion is not a significant performance hit, with the average time taken on a reasonable machine well under one second. A summary of the generation times observed is provided in Table I; each test model was completed ten times, and the average time selected. Upon further investigation, we found that the performance of model completion is mostly dependent on the rules used to complete the model, rather than the number of elements in the initial

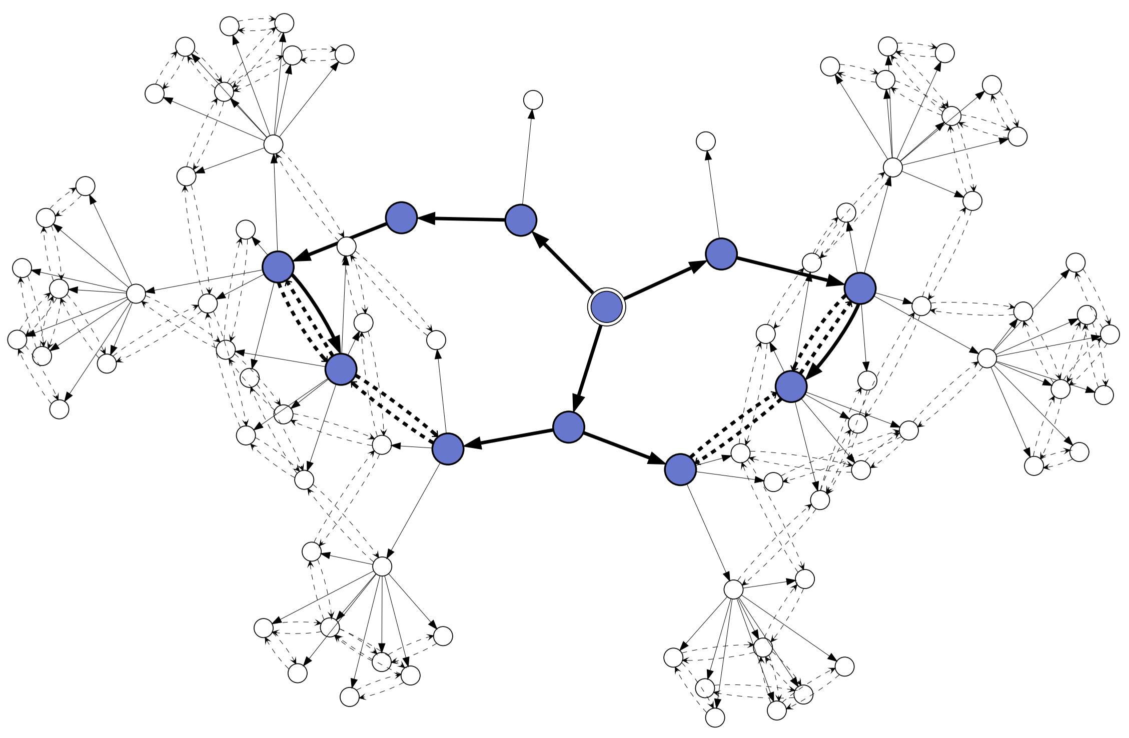

Fig. 12. The test model *PropertiesFileWithInputForm*; the initial model is highlighted

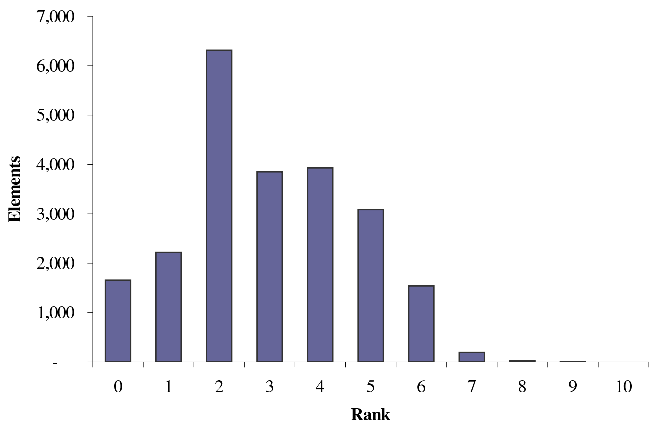

Fig. 13. Aggregation of $rank$ values from all completed test models

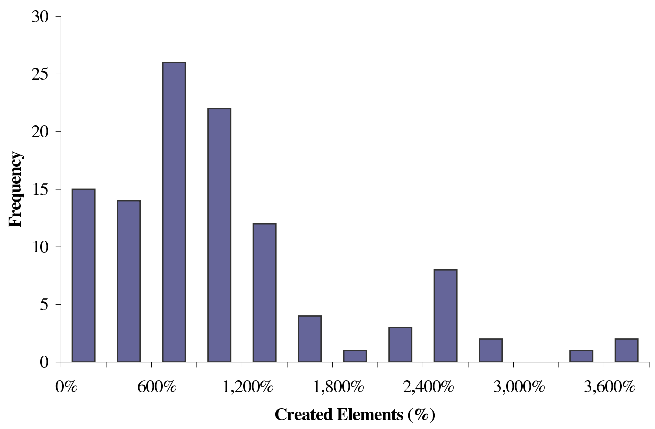

Fig. 14. Histogram of element generation frequency

model; in particular, the number of existential clauses used in each rule. This result is consistent with related work, which we discuss in the next section.

VII. Related Work

The Tefkat language [28] follows a similar approach to our work, by defining rules which create model elements. Tefkat does not support rule patterns based on negative self existentials however, as this was found to introduce significant performance cost. In our work this operator did not introduce a significant performance cost; all of our model completion rules contain at least one negative existential clause. This is probably due to the Rete algorithm used in Drools [25].

Alternatively, one can consider model completion as a series of incremental model transformations operating recursively. Triple graph grammars (TGG) permit the decomposition of a model transformation into a series of incremental model transformations; these can be used in order to realise a performance benefit within model transformation [29]. Grunske et al uses the Tefkat language to implement TGG [30], and found that there was a significant performance cost to match the left-hand side of the incremental rules. While we also discovered this relationship in our work, we did not incur any significant performance costs.

There are many existing implementations of non-monotonic reasoners within industry. Prolog naturally supports non-monotonic reasoning by means of negation [31]; the Jena semantic framework supports non-monotonic inference [32]; and the Pellet reasoner supports the non-monotonic epistemic operator **K** [33]. Non-monotonic inference is not limited to any particular domain; it has been applied to business rules, contracts, legal reasoning and e-commerce negotiations [34].

The WebML approach offers model-driven web application development through a single model which must be implemented in full detail [35]. Conversely, UML-based Web Engineering addresses model-driven web application development by defining at least six computation-independent, platform-independent and platform-specific meta-models. Instances of these models are then combined through a model transformation chain to generate the final web application [7].

Without the support of knowledge inference, different types of software tools are necessary to support model development. In the web engineering community, the case tool for WebML offers reusable modules as a solution to this problem [36], but this introduces problems with round-trip engineering of the system. It also presents difficulties with adapting to new technologies and approaches.

To the extent of our knowledge, no existing work in the field of model-driven web application development has investigated the use of non-monotonic reasoning. Implementing the concepts covered in this paper could provide a simple yet flexible approach to existing approaches of model-driven web application development.

VIII. Discussion

In this paper, we have shown that implementing model completion with industry-standard software has resulted in a high-performance environment, yet one which adheres to formal semantics. Existing tools may benefit from integration with commercial rule engines; for example, the Tefkat language could be implemented directly in Drools, and we expect this might increase its performance.

By using model completion, we can automatically adapt to new technologies and implementations. Since the completed model does not need to modify the base model, improvements in model completion rules can be adapted directly without having to reverse-engineer generated models. As our platform-independent language encompasses more RIA concepts over time, we will investigate the stability of model completion rules over its evolution.

Our implementation provides rich opportunities for model and platform extensibility. For example, a model developer or third party could contribute their own inference rules, or disable existing rules. As mentioned earlier, this discussion is far outside the scope of this paper, but essentially we expect that extensions could consist of any combination of meta-model, inference rules or CASE tool extensions, integrated by the Eclipse framework.

Within the concept of model spaces [5], the work covered in this paper may co-exist within any space. For example, model completion could be used on meta-models instead, assisting the development of meta-models or ontologies. Our approach is not limited to any particular environment either, and only depends on tool support; model completion could be implemented as part of a UML profile, or as a pre-compilation step within a model-driven development environment.

One important area that has not been discussed is on how to document the model completion conventions themselves. In order for model completion to be effective, model developers need to have a detailed and authoritative documentation source, yet one which is still accessible and helpful; merely providing the source code to the rules used would not suffice. This is an interesting area of future research.

Model completion does not need to be limited to an operation performed only at the end of the model development lifecycle; it may be desirable to selectively merge parts of the completed model into the base model, as part of a rapid application development environment. This also remains an area of future research, and a preliminary approach has been implemented in our CASE tool.

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

References

[1] D. Thomas, D. Hansson, L. Breedt, M. Clark, J. D. Davidson, J. Gehtland, and A. Schwarz, *Agile Web Development with Rails*, 2nd ed. Pragmatic Bookshelf, 2006.

[2] F. Potencier and F. Zaninotto, *The Definitive Guide to Symfony*. Apress, 2007.

[3] J. M. Vlissides and M. A. Linton, "Unidraw: a framework for building domain-specific graphical editors," *ACM Trans. Inf. Syst.*, vol. 8, no. 3, pp. 237–268, 1990.

[4] P. Haggett and R. J. Chorley, "Models, Paradigms and the New Geography," *Socioeconomic Models in Geography*, 1967.

[5] D. Djurić, D. Gašević, and V. Devedžić, "The Tao of Modeling Spaces," *Object Technology*, vol. 5, no. 8, 2006.

[6] L. Heaton, "Unified Modeling Language (UML): Superstructure Specification, v2.0," Object Management Group, Tech. Rep., 2005. [Online]. Available: http://www.omg.org/cgi-bin/doc?formal/05-07-04

[7] N. Koch, "Classification of Model Transformation Techniques Used in UML-based Web Engineering," *Software, IET*, vol. 1, no. 3, pp. 98–111, 2007.

[8] J. C. Preciado, M. Linaje, F. Sanchez, and S. Comai, "Necessity of Methodologies to Model Rich Internet Applications," in *Proceedings of the 7th IEEE International Symposium on Web Site Evolution (WSE '05)*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 7–13.

[9] J. Wright and J. Dietrich, "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.

[10] J. Mukerji and J. Miller, "Model-Driven Architecture Guide, v1.0.1," Object Management Group, Tech. Rep., 2003. [Online]. Available: http://www.omg.org/cgi-bin/doc?omg/03-06-01

[11] S. Kent, "Model Driven Engineering," in *Proceedings of the Third International Conference on Integrated Formal Methods (IFM '02)*. London, UK: Springer-Verlag, 2002, pp. 286–298.

[12] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[13] D. M. Gabbay, C. J. Hogger, and J. A. Robinson, Eds., *Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 3*. New York, NY, USA: Oxford University Press, Inc., 1994.

[14] K. Lano and J. Bicarregui, "Semantics and Transformations for UML Models," in *UML'98 - Beyond the Notation*, ser. LNCS, J. Bézivin and P.-A. Muller, Eds., vol. 1618. Springer, 1999, pp. 107–119.

[15] A. Tarski, *Introduction to Logic and to the Methodology of Deductive Sciences*, 3rd ed. New York: Oxford University Press, 1965.

[16] W3C Group, "OWL Web Ontology Language Reference," W3C Recommendation 10 February 2004, Tech. Rep., 2004. [Online]. Available: http://www.w3.org/TR/owl-ref/

[17] M. Fitting, *First-Order Logic and Automated Theorem Proving*, D. Gries, Ed. New York, NY, USA: Springer-Verlag New York, Inc., 1990.

[18] A. V. Gelder, K. A. Ross, and J. S. Schlipf, "The Well-Founded Semantics for General Logic Programs," *Journal of the ACM*, vol. 38, pp. 620–650, 1991.

[19] J. Mei, S. Liu, A. Yue, and Z. Lin, "An Extension to OWL with General Rules," in *Proceedings of the RuleML International Symposium (RuleML 2004)*, 2004, pp. 155–169.

[20] S. Kraus, D. Lehmann, and M. Magidor, "Nonmonotonic Reasoning, Preferential Models and Cumulative Logics," *Artificial Intelligence*, vol. 44, no. 1-2, pp. 167–207, 1990.

[21] M. Gelfond and V. Lifschitz, "The Stable Model Semantics For Logic Programming," in *Proceedings of the 5th International Conference on Logic Programming*. MIT Press, 1988, pp. 1070–1080.

[22] T. C. Przymusinski, "Every Logic Program has a Natural Stratification and an Iterated Least Fixed Point Model," in *Proceedings of the 8th Symposium on Principles of Database Systems*. Philadelphia, PA: ACM, 1989, pp. 11–21.

[23] I. Horrocks, P. F. Patel-Schneider, and F. V. Harmelen, "From SHIQ and RDF to OWL: The Making of a Web Ontology Language," *Journal of Web Semantics*, vol. 1, p. 2003, 2003.

[24] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*, 2nd ed. Addison-Wesley Longman, Amsterdam, January 2009.

[25] D. Sottara, P. Mello, and M. Proctor, "Adding Uncertainty to a Rete-OO Inference Engine," in *Proceedings of the RuleML International Symposium (RuleML 2008)*, 2008, pp. 104–118.

[26] J. Wright and J. Dietrich, "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.

[27] G. Hillairet, F. Bertrand, and J.-Y. Lafaye, "MDE for Publishing Data on the Semantic Web," in *Proceedings of the First International Workshop on Transforming and Weaving Ontologies in Model Driven Engineering (TWOMD 2008)*, 2008, pp. 32–46.

[28] M. Lawley and J. Steel, "Practical Declarative Model Transformation with Tefkat," in *Proceedings of Model Transformations in Practice Workshop, MoDELS Conference*, 2005, pp. 139–150.

[29] H. Giese and R. Wagner, "Incremental Model Synchronization with Triple Graph Grammars," in *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, ser. LNCS, O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, Eds., vol. 4199. Springer Verlag, 10 2006, pp. 543–557.

[30] L. Grunske, L. Geiger, and M. Lawley, "A Graphical Specification of Model Transformations with Triple Graph Grammars," in *First European Conference on Model Driven Architecture - Foundations and Applications*, ser. LNCS, vol. 3748. Springer, 2005, pp. 284–298.

[31] K. R. Apt, *From Logic Programming to Prolog*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

[32] B. McBride, "Jena: Implementing the RDF Model and Syntax Specification," in *SemWeb*, 2001.

[33] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A practical OWL-DL reasoner," *J. Web Sem.*, vol. 5, no. 2, pp. 51–53, 2007.

[34] G. Antoniou, "A Nonmonotonic Rule System Using Ontologies," in *Proceedings of the International Workshop on Rule Markup Languages for Business Rules on the Semantic Web*, 2002.

[35] S. Ceri, P. Fraternali, A. Bongio, M. Brambilla, S. Comai, and M. Matera, *Designing Data-Intensive Web Applications*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[36] R. Acerbis, A. Bongio, M. Brambilla, and S. Butti, "WebRatio 5: An Eclipse-Based CASE Tool for Engineering Web Applications," in *Proceedings of the 7th International Conference on Web Engineering (ICWE '07)*, 2007, pp. 501–505.