Journal of Symbolic Computation 46 (2011) 150–182

Contents lists available at ScienceDirect

Journal of Symbolic Computation

journal homepage: www.elsevier.com/locate/jsc

Static consistency checking of web applications with WebDSL

Zef Hemel, Danny M. Groenewegen, Lennart C.L. Kats [1], Eelco Visser

Software Engineering Research Group, Delft University of Technology, The Netherlands

ARTICLE INFO

Article history:
Received 16 June 2010
Accepted 29 July 2010
Available online 25 September 2010

Keywords:
Domain-specific language
Web application development
Linguistic integration
Consistency checking
Verification
Static analysis

ABSTRACT

Modern web application development frameworks provide web application developers with high-level abstractions to improve their productivity. However, their support for *static verification of applications* is limited. Inconsistencies in an application are often not detected statically, but appear as errors at run-time. The reports about these errors are often obscure and hard to trace back to the source of the inconsistency. A major part of this inadequate consistency checking can be traced back to the lack of linguistic integration of these frameworks. Parts of an application are defined with separate domain-specific languages, which are not checked for consistency with the rest of the application. Examples include regular expressions, query languages and XML-based languages for definition of user interfaces. We give an overview and analysis of typical problems arising in development with frameworks for web application development, with Ruby on Rails, Lift and Seam as representatives.

To remedy these problems, in this paper, we argue that domain-specific languages should be designed from the ground up with static verification and cross-aspect consistency checking in mind, providing linguistic integration of domain-specific sub-languages. We show how this approach is applied in the design of WebDSL, a domain-specific language for web applications, by examining how its compiler detects inconsistencies not caught by web frameworks, providing accurate and clear error messages. Furthermore, we show how this consistency analysis can be expressed with a declarative rule-based approach using the Stratego transformation language.

E-mail addresses: z.hemel@tudelft.nl (Z. Hemel), d.m.groenewegen@tudelft.nl (D.M. Groenewegen), l.c.l.kats@tudelft.nl (L.C.L. Kats), visser@acm.org (E. Visser).

[1] Tel.: +31 015 278 5175; fax: +31 0 15 278 6632.

0747-7171/$ – see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jsc.2010.08.006

1. Introduction

Web applications are complex software systems that combine many technical concerns, such as database querying, input handling, user interface design, and navigation. Web application frameworks are often used to simplify web development and improve web developer productivity. A web framework consists of a set of APIs built on a general-purpose programming language. Popular web frameworks include JBoss Seam, Lift, Ruby on Rails, and Django. These frameworks enable abstraction over many low-level details of normal web application development, avoiding handwritten boilerplate code, thus increasing developer productivity.

While web frameworks improve the clarity of the application and expressivity of developers that use it, applications containing inconsistencies (faults) often fail late, i.e. at run time or deployment time instead of at compile time. Even inconsistencies in applications written using a framework based on a statically typed language such as Java or Scala are often only revealed at deployment time or at run time. The errors produced when the application fails are often difficult to trace back to their origin and error messages are typically not domain-specific, exposing framework implementation details.

1.1. Causes of late failure

Web frameworks use a combination of high-level APIs, meta-programming techniques, and domain-specific languages to achieve higher developer expressivity. Meta-programming techniques used range from reflection in Scala and Java-based frameworks to extension and adaptation of classes and objects at runtime in frameworks based on dynamically typed languages such as Ruby and Python. Domain-specific languages (DSLs) are used for user interface construction (ASP.NET, JSF), access control policies (rule files), pattern matching (regular expression) and database queries (SQL, HQL).

Domain-specific languages, as used by web frameworks, are not *linguistically integrated* with the rest of the framework. Therefore, in practice, very few consistency checks are performed on *connections* between the application aspects defined in different domain-specific languages, resulting in late failure. Web frameworks based on statically typed general purpose languages can report a limited class of application inconsistencies at compile-time. Modern frameworks, such as JBoss Seam and Scala Lift, cannot identify all inconsistencies during compilation, because the static checks they provide are limited to the type checker of their host language (Java and Scala respectively). Other errors, often inconsistencies between application components defined in separate DSLs, are only reported at deployment time or at run time, resulting in the same issues that web frameworks based on dynamically typed languages have.

Frameworks based on dynamically typed languages, such as Ruby on Rails and Python's Django only provide *runtime* consistency checks. Typically, consistency in these frameworks is not explicitly checked, but rather manifests itself when the faulty code is executed. Consequently, errors are not always easily traced back to the source of the problem, and the messages are often unclear and confusing, relating to the framework implementation and not the actual web application. Many errors – not all – include a stack trace directing the developer to the point in the source code (either the framework's code or the developer's) where the failure occurred. Reported error messages often expose underlying implementation details. For instance, when routing to a non-existing controller in Ruby on Rails, an "uninitialized constant" error is reported that refers to a name-mangled version of the application's controller name.

1.2. Design for consistency checking

One solution to late failure and bad error reporting is to build static verifiers for existing web frameworks. However, developing verifiers is very complicated because the framework was never intended to be statically verified.

In this paper we propose a different solution: web languages should be *designed* to enable static verification of its applications for consistency. We show that linguistic integration of the languages is essential for effective checking of consistency properties that span multiple aspects of the application. Linguistic integration entails that different technical concerns, typically expressed using completely

separate languages, be instead expressed using a single language integrating the syntax and semantics of multiple sub-languages as described by Visser (2007).

We illustrate this approach with WebDSL, a web language integrating a number of sub-languages for different concerns related to the construction of web applications with a rich data model, such as a data modeling language, a user interface language, an action language, and an access control language. Based on linguistic integration, consistency properties that span multiple technical domains can still be statically checked in WebDSL. Important domain concepts, such as entities, pages and templates are first-class language elements in WebDSL ensuring that error messages for consistency violations are always expressed in a domain-specific manner, e.g. "entity not found" rather than "undefined constant".

1.3. Contributions

This paper identifies early, accurate consistency checking of web applications as a problem. It is an important problem since it directly affects the productivity of web developers: with better, more accurate static checks, maintenance of source code can be simplified. Existing frameworks based on general-purpose programming languages provide only a limited number of consistency checks. External tools that provide additional checks are hard to construct and maintain, especially when targeting linguistically separate languages. We argue that only an integrated solution allows for an efficient implementation of static consistency checking.

The contributions of this paper are as follows: (1) An analysis of areas where consistency checks are typically lacking within current web frameworks. (2) An analysis of the quality of failure of three state-of-the-practice web frameworks. (3) A declarative, rule-based approach to linguistic integration and consistency checking. (4) A demonstration of this approach with an implementation in the Stratego transformation language of consistency checking for a (subset of) WebDSL.

Previous papers on WebDSL by Visser (2007) and Hemel et al. (2009) gave an overview of the implementation strategy used for the creation of WebDSL. The present paper focuses on consistency checking, relating it to consistency checks in other frameworks, providing a detailed description of the different static checks performed by the language, showing novel, non-trivial ways a web application can be checked, and describing the rule-based architecture in which these checks are implemented.

We begin this paper with a study of different classes of inconsistencies in web applications, showing how these are checked and reported in major web frameworks. In many cases, these consistency checks are lacking in accuracy and in quality of the error reports. In Section 3 we analyze why this is the case, looking at the implementation of the different frameworks. In Section 4 we explain how to address the discovered problems, and describe solutions applied in WebDSL. In Section 5 we demonstrate how a static checker for a subset of WebDSL can be implemented using rewrite rules in Stratego. Section 6 handles discussion points and describes differences with previous work.

2. Failures in web applications

Modern web applications comprise a number of aspects, often expressed using different domain-specific languages, e.g. HTML for user interfaces and data models using annotated Java code. Our experience with mainstream web development frameworks has been that faults, especially across aspect boundaries, manifest themselves late, e.g. only when the application is run and the specific page is loaded, often resulting in developer annoyance and a decrease in productivity. Not only do failures occur late, they are often difficult to trace back to their origin and provided error messages are not domain-specific and expose implementation details of the framework.

To analyze failures in web application frameworks, we have conducted an experiment investigating the problems in fault manifestation and reporting in the current state of practice. We evaluate four aspects of mainstream web frameworks (data model, user interface, application logic and access control). Through *fault seeding* we register when and how applications built using these frameworks fail. Subsequently, the next section will examine the reasons of failure and how they can be mitigated.

2.1. Web application aspects

Typical modern web applications comprise multiple aspects. Application aspects include the data model, user interface and business logic. To simplify development, frameworks offer specialized languages and APIs for these aspects. For instance, user interfaces are defined using an extension of HTML, data models are defined by annotating classes with persistence annotations, and a rule language is used to declaratively specify access control rules. While the use of specialized languages and APIs enables the separation of concerns, the application aspects are not completely independent. Each aspect contains links to other application aspects. These inter-aspect links are an important cause of the late detection of web application failures.

For our study we selected four common application aspects, which are listed below. This list is not meant to be exhaustive, but we believe it is a representative list of aspects that are typically covered by web application frameworks. Other application aspects have similar issues. For each application aspect we list some common internal and inter-aspect faults.

- **Data model**, web frameworks typically have APIs to define the data model of the web application in a declarative manner. The data model represents the data structures that need to be persisted. Common faults:
  - *Properties of non-existing types*, the data model defines properties of types that do not exist.
  - *Invalid inverse properties*, inverse properties refer to non-existing properties.
  - *Invalid data validation*, rules to validate the values of data model properties are invalid, e.g. the regular expression that checks the zip code format contains a syntax error.
- **User interface** is typically defined using a separate DSL, usually an extension of HTML. Common faults:
  - *Invalid page elements*, the use of tags and controls that do not exist or are used incorrectly.
  - *Invalid element nesting*, incorrectly nesting tags and controls in an invalid manner, e.g. nesting list items outside a list.
  - *Invalid references to data model*, the user interface often presents data from the data model, references to the data model, e.g. entity properties, may be incorrect.
  - *Invalid links to pages*, links to pages within the application do not exist or are linked to with wrong parameters.
  - *Invalid links to actions*, actions to be triggered, e.g. when pushing a button, do not exist or are invoked incorrectly.
- **Application logic** defines the business logic of the application. Common faults:
  - *Invalid references to data model*, properties and types that do not exist.
  - *Invalid redirect from actions*, the user is redirected to pages within the application that do not exist.
  - *Invalid data binding*, form data is bound to entities incorrectly.
- **Access control** defines who can access what parts of the application in a declarative manner. Common faults:
  - *Invalid references to data model*, access control rules link to non-existing data model entities and properties.

2.2. Moment of failure

Application faults should manifest themselves as soon as possible; the sooner the developer knows, the sooner he or she can resolve the problem. Thus, the *moment of manifestation* is an important quality of fault detection in frameworks. Once a fault has manifested itself, the developer has to resolve the problem. Therefore, the *retraceability* of the problem to its source is important; the location of the fault should be clearly indicated in the code. Once the source of the problem has been pin-pointed, the reported error message should indicate what the problem is *in terms of the application domain* and

should reveal as little about the underlying implementation as possible. For instance, when a link to a non-existing page within the application is found, the error should use domain terminology such as "page" and "link" rather than "constant" or "method".

Thus, we can determine the quality of fault detection in frameworks and DSLs by considering three aspects:

(1) The *moment of manifestation*, i.e. the moment the developer is presented with an application inconsistency:
   - *compile time*, detected during compilation of the application;
   - *deployment time*, detected when the application is started or deployed to an application server;
   - *runtime*; detected at the server while the application is running, e.g. when loading a page;
   - or in the *browser*, when an error is only detected when a page is loaded by the client (e.g. mistakes in Javascript, HTML etc.).

(2) Is the error *retraceable* to its origin? Is a source code filename and line number clearly indicated?
(3) *Clarity* and specificity of error message. Are domain-specific terms used in error messages, or do they uncover the underlying implementations?

2.3. Frameworks

We evaluate three mainstream, available web application frameworks that represent the state of the practice in web application development. We discuss other web frameworks and languages in Section 6. We base our study on parts of example applications and tutorials from the websites of the different frameworks. We apply the technique of *fault seeding* by introducing small inconsistencies in parts of the application (often in the form of simple typing errors, simulating what happens when an application is changed or a developer makes a mistake) and observe how the errors manifest themselves.

The selected frameworks are:

- *Ruby on Rails*,[2] representing dynamically typed language frameworks. We chose Rails as a representative of frameworks based on dynamic languages. Other frameworks such as Django for Python are similar in terms of implementation techniques and error handling.
- *JBoss Seam*,[3] a framework based on Java, combining a number of existing Java technologies such as the Java Persistence API (JPA) and JavaServer Faces (JSF). We selected JBoss as a representative of Java-based frameworks. A comparable framework is Spring.
- *Lift*,[4] a web framework based on Scala, a highly expressive object-oriented/functional programming language with a sophisticated type system. Scala is a statically typed language with a very flexible syntax, distinguishing Lift from the two other categories.

In the remainder section we highlight two faults related to the data model and the user interface. A full overview of the cases we studied is given in Appendix. We summarize our results in tables that rank the three quality aspects of moment of manifestation, retraceability, and clarity (labeled *M*, *R*, and *C*).

2.4. Case 1: Consistency of references to the data model

	M	**R**	**C**
Rails	Runtime	+	−
Seam	Runtime	−	+
Lift	Runtime	+	+

User interfaces are typically used to present data from a database. Therefore user interface code contains references to the data model, for instance to show the value of a certain property, or binding a control to a certain entity property.

In Ruby on Rails, references from the user interface to data model properties are constructed through embedded Ruby code. The following example displays the

[2] We evaluated version 2.3.4 of Ruby on Rails, http://www.rubyonrails.org/.
[3] We evaluated version 2.2.0.GA of Seam, http://www.jboss.com/products/seam.
[4] We evaluated version 1.0 of Lift, http://www.liftweb.net/.

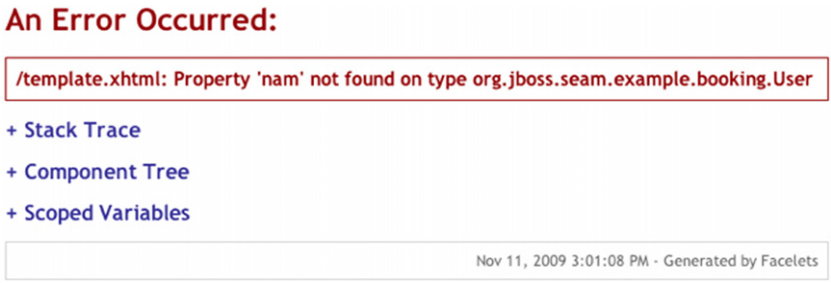

Fig. 1. Seam exception when using an undefined property `nam`.

value of the `name` property of the `post` entity, encoded to be displayed in HTML:

```
<td><%=h post.name %></td>
```

Although references to undefined properties, such as `post.nam` instead of `post.name`, are easily traced back to their source, the reported "undefined method" message is not domain-specific and only reported at runtime.

In Seam, values of entity properties can be injected into a page using the `#{...}` syntax:

```
Welcome #{user.name}
```

When invalid property names are used, a domain-specific runtime exception is reported when the page is loaded ("Property 'nam' not found on type …"), but no indication of the source of the problem is supplied (see Fig. 1).

In Lift, the `name` property of an entity `user` is referenced as follows:

```
<user:name>User name</user:name>
```

When misspelling `name` as `nam`, Lift gives a clear, domain-specific error ("no such property") and reports the line and column number of the error.

All of the tested frameworks report faults in references to the data model only at runtime, when the specific page is loaded.

2.5. Case 2: consistency of links to pages

	M	R	C
Rails	Runtime	+	−
Seam	Browser	−	−
Lift	Browser	−	−

Creating hyperlinks between pages is a fundamental part of the web. While broken links to external websites are hard to avoid, broken links *within* a single web applications should be avoided and, at least in principle, be automatically detected.

Ruby on Rails provides a `link_to` helper for user interfaces:

```
<%= link_to 'Edit', edit_post_path(post) %>
```

The `edit_post_path` method that is called is generated on the fly by convention, the convention taking the form of `<action>_<controller>_path(<args>)`. When the name of this method is constructed incorrectly, a generic "undefined method" error is reported, with accurate code and line and column numbers. This means that the framework is able to detect broken, internal links before they are displayed to the user. However, the error message is not domain-specific.

Seam uses a `s:link` tag to create links to arbitrary URLs. These URLs are not checked by the framework:

```
<s:link id="register" view="/register.xhtml" value="Register New User"/>
```

When the linked page does not exist, the user is presented with a "page not found" error when the link is clicked.

Lift does not have a special construct to define internal links, instead simple `<a href="...">` tags are used. Similar to Seam, links to non-existing pages go undetected until they are clicked.

	Manifestation			Retraceability			Clarity		
Category	**Ra**	**Se**	**Li**	**Ra**	**Se**	**Li**	**Ra**	**Se**	**Li**
Data model									
Properties of non-existing types	R	C	C	–	+	+	–	+	+
Invalid inverse properties	R	D	C	–	+	+	+/–	+	+
Invalid data validation	R	C/D	C/D	–	+/–	+/–	–	+	+
User interface									
Invalid page elements	R	R	R	+	+	–	–	+	–
Invalid element nesting	B	B	B	–	–	–	–	–	–
Invalid references to data model	R	R	R	+	–	+	–	+	+
Invalid links to pages	R	B	B	+	–	–	–	–	–
Invalid links to actions	R	R	R	–	+	–	+	–	–
Application logic									
Invalid references to data model	R	C	C	+	+	+	–	+	+
Invalid redirect from actions	R	R	R	–	–	–	–	–	–
Invalid data binding	R	NA	NA	–	NA	NA	–	NA	NA
Access control									
References to data model	R	R	C	+	–	+	–	–	+
Ra = Ruby on Rails, Se = Seam, Li = Lift B = Browser, C = Compile, D = Deploy NA = Not applicable, R = Runtime									

Fig. 2. A summary of consistency checks in Ruby On Rails, JBoss Seam, and Lift.

2.6. Summary

A summary of our results is shown in Fig. 2. Rather than tally the specific scores of the individual frameworks, we conclude that there are many cases where errors are not reported at the earliest possible opportunity, where errors are not easily traceable to their source, and where error messages are unclear or confusing. In the next section we discuss reasons in the design and implementation of the frameworks that cause these deficiencies.

3. Impact of language and framework design on fault detection

In this section we analyze *why* faults in web applications manifest themselves late in the development process and why failures often have poor retraceability and clarity. The examples of web application inconsistencies in the previous section illustrate that there are many cases where inconsistencies lead to late failure. They may only be reported or otherwise manifest themselves once a definition is *used*, not when it is first compiled or interpreted. In many cases, reported error messages are very generic, revealing details about the implementation of the framework (i.e., revealing leaky abstractions). Error messages also do not always show the origin of the error, as they are reported in various ways and definitions are not directly checked.

The frameworks in our survey have been implemented using different programming techniques and based on different programming languages. In the following subsections we analyze different properties of the frameworks that impact the manifestation of faults.

3.1. Reflection and run-time code manipulation

Reflection and run-time code generation are common techniques for integration and deployment of components in web application frameworks. Based on the dynamic language Ruby, Rails in

particular makes heavy use of these techniques to provide convenient, high-level abstractions. JBoss Seam makes use of reflection techniques to process annotations, particularly to describe the data model.

3.1.1. Ruby on rails

As a typical example of how the dynamic programming approach of Ruby interacts with how failure manifests itself, consider a one-to-many relationship declaration in an entity:

```
has_many :comments
```

This declaration *implies* there is a `Comment` entity defined elsewhere. When the property is used, the Rails framework simply takes the `comments` symbol, strips off the `s` and capitalizes the first character. If no such entity is defined, the developer will receive a "constant not defined" error related to `Comment`, while the application code does not contain any reference to this entity anywhere directly. These indirect error messages can be confusing to the user of the framework. If entity declarations were instead verified directly when the entity was declared, the error could be detected earlier, and would be more easily traced back to the source. The dynamic programming approach taken by Ruby on Rails involves a trade-off between the performance of not checking such properties and ease of use.

Many features of the Rails framework make use of methods which are passed a map with named arguments. This way, arbitrary key/value pairs can be used as arguments for these methods. When a key is mistyped or there is no definition for such a key (as seen with `:confirmation` in Appendix A.2.4), such faults remain undetected unless the contents of the map is explicitly verified by the framework. In the current implementation of Rails, this is often not the case.

3.1.2. JBoss seam

After a JBoss seam application is compiled, framework-specific tools are used to deploy it onto a server environment. Typically, application servers enable web application verification code to be invoked while the application is being deployed. This provides frameworks with the opportunity to perform additional checks that were not already performed by the compiler.

An example of a post-compilation time consistency check is Seam's verification of entity classes and their annotations and embedded regular expressions. Any faults detected in the data model are reported by throwing exceptions. Unfortunately, in practice this seems to cause a domino effect of exceptions being thrown by various components of the application server. This causes enormous stack traces to be recorded in the server logs, in which it is very hard to find the originating error message. Still, by performing these checks while the application is being deployed, Seam avoids runtime failures resulting from certain classes of faults in the data model.

3.2. Linguistic separation

The three frameworks each employ one base language: Java, Ruby, or Scala. They also employ a number of other languages, such as XHTML, regular expressions, or query languages. These languages are linguistically separated in the sense that the compiler for the base language is not aware of the definitions made in the other languages and whether or not they are consistent and correct. Because the compiler cannot pick up these inconsistencies, they can lead to failures as an application is running.

Conceptually, it is appealing to use different languages that each address different technical concerns: each language can be more or less suited for that particular domain. Unfortunately, as these languages have been designed and have evolved separately, there can be redundancy and inconsistency among them. The EL expression language used in JBoss Seam, for example, does not support all features of standard Java expressions, yet it adds some features of its own.

Separate languages also introduce a problem for programming tools, as tools that support one language lack awareness of other languages that are used in a web application. Editors and compilers generally only have a limited "view" of a web application, constrained by the boundaries of a particular language. They do not check inside strings, determine the meaning of annotations, or analyze accompanying XML or XHTML files. Consistency checking for concerns that cross the boundaries of a

language – understanding-in-the-large of a web application – is very hard when different languages are used. Only tools that are specialized to work with a particular set of languages and frameworks (such as IntelliJ IDEA, discussed in Section 6.2) can check for some of these consistency issues. However, as the different languages, frameworks, and tools involved are developed by different groups of people, such a solution is very hard to maintain and even harder to make complete.

Links and redirects in the three frameworks are constructed as simple URL strings. Only in Rails, where links can be constructed using helper methods, are internal links checked for correctness at run-time. The other frameworks do not support any form of consistency checking: bad links only manifest themselves when the user tries to follow them.

3.3. Limited static type checking

Faults manifest themselves at a variety of different stages: at compile time, deployment time, run time, or sometimes only in the browser. Failures early in the development cycle typically require less effort to resolve. Faults that are detected directly at compile time do not require failure-to-fault tracing or running the application to be detected.

Seam and Lift benefit from their statically typed base languages with respect to compile-time detection of faults, while Rails can only provide developers with feedback about faults at runtime. In our study we found that there is a number of negative performance trade-offs when delaying checks until run-time, and that accurately discovering and reporting the origin of errors can be difficult. Still, there were many cases where the Seam and Lift frameworks did not score much better at providing early feedback.

Since Rails is based on Ruby, there is no compilation step, and consistency errors that are reported are always detected at run time. Still, we can distinguish between errors reported when a definition is interpreted and when the definition is used. In many cases, errors are only reported when definitions are used. In our experience, the framework performs very few checks when definitions are made, before they are used elsewhere. When errors are reported, the messages are usually generic Ruby messages (typically, a `NoMethodError`).

Based on compiled, statically typed languages, Lift and Seam can report many errors before an application is deployed. Errors detected by the Java and Scala compiler always clearly indicate their origin. Using an IDE such as Eclipse, compile-time errors can be conveniently marked in the source code using a marker in the editor. Still, the reported error messages are always generic Java or Scala error messages, as the compiler and IDE only follow the static semantics of the host language. Because of this limitation, any language features encoded in strings, such as embedded queries or regular expressions, cannot be checked. Likewise, any references to other elements of an application in the form of strings (such as in the Seam `@OneToMany` annotation) cannot be statically checked. The Java and Scala host languages also do not offer a way to statically constrain the placement of annotations on the right elements of an application, or to avoid conflicting annotations.

A problem with relying on the static type system of the base language is that the errors reported are not specific to the domain of web programming. For instance, instead of reporting an error about an entity property, reported errors may complain about the field of a class. Since Seam and Lift are frameworks and not true languages on their own right, reporting domain-specific error messages is very difficult. Only by the construction of extensions to the already elaborate Java or Scala compilers would it be possible to check such frameworks. Building such extensions is generally a difficult, laborious undertaking, especially for frameworks that rely on reflection techniques and linguistic separation. In Section 6.2 we discuss tools that follow this approach in more detail.

3.4. Run-time consistency checking

Most faults not detected by the compiler or at deployment time are reported at runtime. Some errors are reported directly when a definition is processed by the runtime, others only in particular use cases of the application, manifesting themselves only when a particular action is performed by the user. Such delays in detection are detrimental for developer productivity and,

as regressions may go undetected when not covered by the test suite, the maintainability of an application.

From a framework implementation point of view, runtime consistency checks – at least in principle – make it easy to report accurate, highly specific error messages. However, in practice, traceability of these errors is often lacking, as source location information at run time is scarce, usually limited to the point in the application where the check was performed. There are often many framework calls in between the location of the error and the point where the error is detected, resulting in runtime traces that can be misleading or confusing. Our survey in Section 2 showed that the quality of runtime error messages and their traceability varies widely and is typically worse than for compile-time reported errors.

Seam and Lift perform static checks at compile-time using the standard Java and Scala compilers and perform a limited set of consistency checks at deployment time. This leaves it up to the runtime to perform the remainder of the checks. Thorough, often domain-specific checks that are not performed earlier are performed at run time. These checks guarantee the correctness of any strings in annotations and of string-embedded languages. Both frameworks run on the Java Virtual Machine and use the Java exception tracing mechanism for reporting the origin of such errors. For run-time checks, some of these reported origins relate to the usage sites of inconsistent definitions, but as a last resort they are still helpful in determining the root cause of an error.

Location information provided by exceptions is ineffective for checks that are not performed at the definition site where an error is triggered. This makes it particularly difficult to report accurate location information for errors in annotations, which are heavily used especially in Seam. The Java language provides few means to provide exact location information when the annotations are reflected over at runtime. At most, a class and method name can be provided in any annotation errors that are reported.

3.5. Summary

Providing accurate, static checks at compile-time avoids failures at deployment-time or at run-time. Statically detected faults do not require failure-to-fault tracing and can be reported directly inside an IDE. Still, there are many classes of faults that are not statically detected by the frameworks in our survey. Reasons for this include that they use reflection and run-time code manipulation techniques, linguistically separated languages, and can only use static typing provided in the base language compiler. Instead, many faults are reported at run time, introducing a (small) performance penalty and often resulting in errors that are vague or hard to trace back to their originating fault.

4. Designing for static verifiability

In the previous sections we demonstrated the problems of weak static verification of web applications. We concluded that the cause of this weakness is in the design of the programming languages and frameworks. Static verifiability is an afterthought, delegated to third party tool developers or coped with by test-driven design methodologies. Because static verifiability is not a criterion during design, the resulting language will end up being hard to verify. Our solution is designing a web programming language with static verifiability in mind as exemplified in WebDSL.

WebDSL embraces the notion of having different, specialized languages to address separate concerns. WebDSL provides specialized languages for data modeling, user interface design, and basic data operations. However, through *linguistic integration*, these different languages are combined into one large integrated language. Fig. 3 illustrates the key domain-specific languages that together form WebDSL. The languages are seamlessly integrated, follow the same style of syntax and share common elements, and can be used together in one module, if required.

WebDSL and its sublanguages have been designed as statically checked languages: the *moment of detection* of all consistency checks is at compile time. In fact, using the new WebDSL Eclipse plug-in, errors are detected as the developer writes his code. As the checks are performed directly on the source code, rather than on a deployed application, any reported errors directly relate to the source code,

UI	Language for defining HTML user interfaces
Data models	Language to define persistent data models
Action	Simple language for defining application logic
Access control	Access control rules for specifying the access control policy
Validation	Data validation language
HQL	Database query language
Workflow	Language for defining workflows

Fig. 3. WebDSL sublanguages.

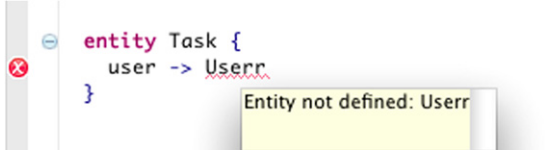

Fig. 4. Property type consistency.

Fig. 5. Inverse annotation.

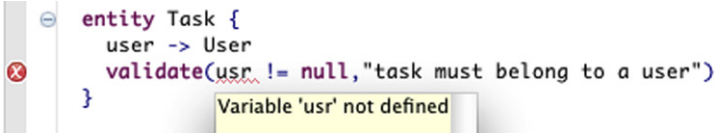

Fig. 6. Data validation.

ensuring proper *retraceability*. Finally, since errors relate to the domain-specific WebDSL language – and not a general-purpose language with a web framework on top – all errors are domain-specific and are explained in terms of the web application domain rather than in terms of the underlying implementation.

For a general description of WebDSL, we refer the reader to our previous work (Visser, 2008; Hemel et al., 2009; Groenewegen and Visser, 2008; Hemel et al., 2008; Groenewegen and Visser, 2009). This section will highlight design decisions where static verifiability was taken into account, in particular the categories from Fig. 2 will be addressed.

4.1. Data model

Data model entities are *first-class language elements* in WebDSL. They are defined as uniquely named top-level elements. The properties of data model entities are statically typed, they can refer to built-in simple types or to defined entities. A shared, static type system across WebDSL sub-languages enables static verification of the use of existing types and properties. Designing the language with entities as first-class language elements enables reporting of domain-specific error messages.

Fig. 4 illustrates the editor's feedback when a non-existing type is referenced in a property in WebDSL. Similarly, Fig. 5 shows that this check also holds for inverse relations. Fig. 6 shows checking of references to entity properties from validation rules.

```
define viewUserTasks(u:User){
  for(t:Task in u.tasks){
    showTsk(t)
    navigate editTsk(t) { "edit" }
  }
}
```

Fig. 7. Template call and navigation.

```
define showTask(t:Task){
  header{
    output(t.title)
  }
  output(t.desciption)
}
```

No property desciption defined for Task

Fig. 8. Template reference to data model.

4.2. User interface

User interfaces in WebDSL are defined using page template definitions. Like data model entities, these are declarative first-class language elements in WebDSL. Templates can call other built-in or user-defined templates. Navigation between pages is expressed using `navigate` elements which create links to other pages within the application. Rather than constructing links through string concatenation, links are defined as typed page calls, for which can be verified that they exist and that the number and type of their arguments are correct. Fig. 7 shows how mistakes in template calls and navigates are reported. Output elements form references to the data model for displaying data (the `output` template name is overloaded for each type). Fig. 8 illustrates that such references are checked as well.

4.3. Application logic

While the model-view-controller pattern is generally considered good style, WebDSL does not impose the use of this pattern in the language. Instead, WebDSL applications typically encapsulate small snippets of application logic directly in user interface code as page actions. Larger pieces of logic can be defined separately in functions. The sublanguage used in these page actions and functions is a Java-like imperative language with a simple API, fully checked by the WebDSL typechecker. Fig. 9 shows a small template that will result in a form with two input fields. Data binding is automatic, any input in the form will update the data model before the action is executed. Redirecting the user to a different page after an action has succeeded is done using the built-in `return` construct. A `return` construct, similar to a `navigate` in the user interface language, takes a page call as its argument.

The incorrect action reference `sav` is reported, as is the page reference `showUserTsks` inside the action.

4.4. Access control

The access control policy of a WebDSL application is defined in access control rules. The access control language reuses the expression language (and its checks) also used in the user interface and application logic. In addition, the page signature syntax is the same as for defining pages, enabling the verification that a rule in fact matches an existing page with correct signature.

Fig. 10 shows how a missing property of the data model is reported.

```
define page editTask(t:Task){
  form{
    input(t.title)
    input(t.description)
    submit sav() { "save" }
  }
  action save(){
    return showUserTsks(t.user);
  }
}
```

Fig. 9. Action logic.

```
rule page editUser(u:User){
  u == principal || u.admin
}
```
No property admin defined for User

Fig. 10. Access control.

4.5. *Verifiability versus flexibility*

Designing for verifiability requires a trade-off with flexibility. Verifiability should be part of the language design considerations, but may impede coverage, i.e. the range of programs that can be expressed. As an example, consider verification of navigation in WebDSL. The interaction between page definitions and navigate statements is verified by controlling the URLs that are generated for pages, and thus required for links to those pages. That is, a URL for a page consists of the name of the page followed by the (identities) of the arguments separated by slashes. For most applications that results in nice readable URLs. If a developer wants to implement a more dynamic scheme this can be realized by creating a single page definition that interprets the URL parameters and dispatches to some appropriate template definitions. However, this results in a loss in the effectiveness of static verification; navigates become calls to the generic dispatch page, rather than to a specific page, which requires the developer to deal with parameter encoding/decoding and verifying consistency. For applications where such flexibility is a requirement, the current WebDSL design is not optimal; it would be better to generalize the current page/navigate paradigm to declaratively specify dispatch schemes that are verifiable.

In practice, WebDSL's verifiability does not impede coverage. The language is used for several web applications that are in production. The largest and most complex WebDSL application to date is researchr.[5] Researchr is a digital library with over a million publication records, including BibTeX import and export, bibliographies, reviewing, tagging, a reputation system, groups, and a messaging system. Researchr's data model consists of over a hundred entities (represented by 140 database tables) and the complete application consists of about 18,000 lines of WebDSL code. The static consistency checking scales to the size of the application, and is even a pre-condition for its maintainability; making changes is not scary since consistency faults introduced are detected at compile-time.

YellowGrass[6] is a free web-based issue tracker. Internally we use it to track WebDSL bugs and other projects within our group use it as well. The official WebDSL website[7] has been built using

5 http://researchr.org.
6 http://yellowgrass.org.
7 http://webdsl.org.

```
signature
  constructors
    Module          : ID * List(Definition) -> Module

    // data model
                    : Entity -> Definition
    Entity          : ID * List(Property) -> Entity
    Property        : ID * Type -> Property
    SimpleType      : ID -> Type

    StringLit       : STRING -> Exp
    Var             : ID -> Exp
    PropertyAccess  : Exp * ID -> Exp

    // user interface templates

                    : Template -> Definition
    TemplateDef     : List(Mod) * ID * List(Param) * List(Element) -> Template
    Page            : Mod
    Param           : ID * Type -> Param

    String          : STRING -> Element
    Navigate        : PageRef * List(Element) -> Element
    Call            : TemplateRef * List(Element) -> Element
    TemplateRef     : ID * List(Exp) -> TemplateRef
    PageRef         : TemplateRef -> PageRef
```

Fig. 11. Signature for NWL, a subset of WebDSL.

WebDSL. It features an editable manual with revision control. TweetView[8] is a twitter archival and search tool that archives tweets about certain topics and attempts to reconstruct conversations around them.

5. Rule-based consistency checking

In the previous sections we have argued that static consistency checks for a linguistically integrated web programming language provide better and earlier feedback to developers. In this section we show that this can be realized using a high-level rule-based specification. We give a formal definition of automatic consistency checking for a subset of WebDSL using rewrite rules in Stratego (Bravenboer et al., 2008) following the style developed for type checking by Hemel et al. (2009). We give a brief introduction to Stratego and the style of consistency checking employed in the paper. The concrete syntax of WebDSL is defined using SDF grammars (Visser, 1997), but in this paper we focus only on the abstract syntax and semantics of the language.

5.1. Language definition

We illustrate static consistency checking in WebDSL using a subset of the full language focusing on the two examples from Section 2: references to the data model in user interface templates, and consistency of references to user interface templates and pages. Fig. 11 defines the abstract syntax of the subset of WebDSL we are considering using an algebraic signature, which consists of typed term constructors corresponding to language constructs. Fig. 12 illustrates the definition with the concrete and abstract syntax of a fragment of a WebDSL program.

The data model of a WebDSL program is defined using *entity declarations* (`Entity`), which consist of a name and a list of properties (`Property`), each having a name and a type. Expressions are constants (`StringLit`), variables (`Var`), or access to the values of properties of objects (`PropertyAccess`).

8 http://tweetview.net.

```
module blogpost
entity Post {
  title   : String
  text    : WikiText
  author  : User
}
define page post(p : Post) {
  header{ output(p.title) }
  output(p.text)
  navigate editpost(p) { "Edit" }
}
```

```
Module("blogpost",[
  Entity("Post",
    [Property("title",  SimpleType("String")),
     Property("text",   SimpleType("WikiText")),
     Property("author", SimpleType("User"))]
  ),
  TemplateDef([Page()], "post", [Param("p", SimpleType("Post"))],
    [Call(TemplateRef("header"),
            [Call(TemplateRef("output", [PropertyAccess(Var("p"), "title")]))]),
     Call(TemplateRef("output", [PropertyAccess(Var("p"), "text")])),
     Navigate(PageRef(TemplateRef("editpost", [Var("p")])),[String(""Edit"")])]
  )
]
```

Fig. 12. Concrete and abstract syntax for fragment of a WebDSL program.

The user interface of a WebDSL program consists of *template definitions* (`TemplateDef`), which have a name, list of parameters, and list of template elements. The elements compose the output of the template from the objects passed as parameters. This is mostly achieved by reference to other templates. Some of these templates are primitives. For example, the `output` template presents the value of an object, and the `input` template is used to create input form elements.

Template *page* definitions have the `Page` modifier and produce a complete web page. Non-page template definitions define partial pages that are used to compose pages. There are two ways in which template definitions refer to other template definitions. A template call (`Call`) *inlines* the body of a referenced template in the calling template. A page reference (`PageRef`) is used to produce a link to *navigate* to the corresponding template (which must be a page definition).

5.2. Static consistency checking

The language is designed to support static consistency checking. References to other elements of a program are *explicitly* encoded in the syntax of the language. For example, instead of encoding an expression retrieving the value of a property of an object as a string literal, the user interface language can use expressions to produce such values. The identifiers used in these expressions are typed and property accesses can be checked against the data model. Similarly, references to user interface templates are explicit calls that can be checked for existence of the called template and the proper typing of the arguments passed; in contrast to the composition of URLs from strings (which is akin to pointer manipulation in C).

The WebDSL compiler translates WebDSL programs to Java programs. Before code generation, the source code is statically checked for consistency violations. WebDSL is also supported by an Eclipse editor plug-in, which displays error messages and warnings in the editor, providing immediate feedback about consistency errors to the developer (see previous section). Code generation and static checking in the compiler and in the Eclipse plug-in are implemented in the Stratego transformation language.

Static checking is divided into three parts. *Name resolution* determines which identifier uses refer to which declarations. *Type analysis* computes types (and other properties) of composite expressions. *Consistency checking* applies constraints to sub-terms, producing error messages when violations are encountered. In the next subsection we give a brief introduction to Stratego. In the following subsections, we discuss the definition of name resolution, type analysis, and consistency checking.

5.3. Stratego

Stratego is a language for program transformation based on the paradigm of term rewriting with programmable rewriting strategies introduced by Visser et al. (1998). Stratego transformations operate on first-order terms of the form

```
t ::= x                 // variables
    | "..."             // string literals
    | i                 // integer constants
    | c(t1, . . . ,tn)     // constructor applications
    | [t1, . . . ,tn]      // lists of terms
    | (t1, . . . ,tn)      // tuples of terms
```

Basic transformations are defined by means of conditional *term rewrite rules* of the form

```
r : t1 -> t2 where s
```

with `r` the name of the rule, `t1` and `t2` first-order terms, and `s` a *strategy expression*. A rule applies to a term when its left-hand side `t1` matches the term, and the condition `s` succeeds, resulting in the instantiation of the right-hand side pattern `t2`. Otherwise the application *fails*.

In addition to checking applicability constraints, the condition of a rule can perform computations the results of which are used in the right-hand side of the rule. For example, in the rule schema

```
r : t1 -> t2 where t3 := <s> t4
```

the term `t4` possibly containing variables from `t1` is transformed by the application of a strategy `s` and the result is matched against the pattern `t3`, possibly binding variables, which may be used in the right-hand side `t2`.

More complex transformations can be created by composing rules using *strategies*. A strategy is essentially a partial function from terms to terms. If a strategy is not defined on a term it is said to *fail*. Failure arises from the failure of rewrite rules to apply to terms. Strategies are composed from basic combinators such as the identity transformation `id`, sequential composition `s1; s2` and deterministic choice `s1 <+ s2`. From these basic combinators new combinators can be defined using (parametric) strategy definitions. For example, the definitions

```
try(s) = s <+ id
repeat(s) = try(s <+ repeat(s))
```

define the combinator `try(s)` that attempts to apply a strategy `s` to a term, and restore the term if `s` fails, and `repeat(s)` that applies a transformation `s` as often as possible to a term. While the strategies above apply a transformation to the root of a term, *term traversal* strategies apply transformations to sub-terms. The basis of term traversal strategies are *one-level* traversal operators such as `all(s)`, which applies a strategy `s` to each direct sub-term of a term. For example, the definitions

```
bottomup(s) = all(bottomup(s)); s
alltd(s) = s <+ all(alltd(s))
```

introduce the `bottomup(s)` strategy that applies `s` to each sub-term in a bottom-up (post-order) fashion, while `alltd(s)` applies `s` to an outermost frontier for which `s` succeeds.

Context-sensitive transformations can be expressed by means of *dynamic* rewrite rules (Bravenboer et al., 2006), which are instantiated at run-time, as illustrated by the following schema:

```
r : t1 -> t2
    where rules( dr : t3 -> t4 )
```

The dynamic rule `dr` is defined when `r` is applied to a term matching `t1`. Any variables that `t3` and `t4` share with `t1` are then inherited by the instantiation of `dr` (concrete examples follow below).

5.4. Name resolution

In textual software languages, program units are identified by name — hence, names are known as identifiers. Declarations introduce names and definitions bind names to meanings — often

```
strategies

  declare-all = alltd(declare-def); rename-all

rules

  declare-def:
    ent@Entity(x, prop*) -> Entity(x, prop*)
    with rules( EntityDeclaration : x -> ent )

  declaration-of :
    SimpleType(x) -> <EntityDeclaration> x

  declare-def :
    def@TemplateDef(mod*, x, param*, elem*) -> TemplateDef(mod*, x, param*, elem*)
    with sig := <signature-of> def;
         rules(
           Template : x -> def
           Template : sig -> def
         )

  signature-of :
    TemplateDef(mod*, x, param*, elem*) -> (x, <param-types>param*)

  param-types :
    TemplateDef(mod*, x, param*, elem*) -> <param-types> param*

  signature-of :
    TemplateRef(x, e*) -> (x, t*)
    where t* := <map(type-of)> e*

  declaration-of :
    ref@TemplateRef(x, e*) -> def
    where def := <signature-of; Template> ref
```

Fig. 13. Name resolution for top-level declarations.

declarations and definitions are combined in one construct. Definitions are applied by invoking their name. In the language of Fig. 11 there are four kinds of identifiers. Entity declarations introduce named entities. Properties identify the attributes of entities. Template definitions identify user interface components. Template parameter names identify their arguments. Corresponding to these declarations, we have the following uses of identifiers. Type expressions are references to entities (and primitive types). Variables are references to entity objects (or primitive values). Property access expressions retrieve the value of a property of an object. Template references invoke a template.

An important source of inconsistencies is the use of names that do not correspond to definitions, or the use of names of existing definitions in the wrong place or in the wrong way. Thus, the first task of a consistency checker is to resolve the use of names, identifying for each application which declaration it invokes. We distinguish two types of identifiers, i.e. identifers with global scope and identifiers with local scope. We can distinguish further layers, associating name spaces with modules, but we will ignore such layers here, but note that they can be expressed with the same approach.

The rules in Fig. 13 define name resolution for the top-level definitions in our language, that is entity declarations and template definitions. The `declare-def` rules introduce the dynamic rules `EntityDeclaration` and `Template`, mapping identifiers to definitions. The `EntityDeclaration` rule maps the name of an entity to the complete abstract syntax representation of the corresponding entity declaration. Note that `x@t` denotes a simultaneous match to a variable (`x`) and a term pattern (`t`). The `declaration-of` rule maps a type expression to the corresponding entity declaration, *provided* the `EntityDeclaration` rule is defined for the type name. If not, the `declaration-of` rule simply fails.

Similarly, the `Template` dynamic rule maps the name of a template definition to its complete AST representation. Since non-page template definitions can be overloaded there is also a mapping from the *signature* of a template to its definition. The signature of a template definition is a pair of its name and the list of its parameter types. The `declaration-of` rule produces the template

```
strategies

  rename-all = alltd(rename)

rules

  rename :
    Param(x, t) -> Param(y, t)
    with y := <rename-var>(x, t)

  rename-var :
     (x, t) -> y
     with y := x{<new>}
     with rules(
            RenameId : x -> y
            TypeOf   : y -> t
          )
  rename :
    Var(x) -> Var(y)
    where y := <RenameId> x

  rename :
    TemplateDef(mod*, x, param1*, elem1*) -> <declare-def>
    TemplateDef(mod*, x, param2*, elem2*)
    with {| RenameId:
            param2* := <rename-all> param1*;
            elem2*  := <rename-all> elem1* |}
```

Fig. 14. Name resolution for local identifiers.

definition corresponding to a template reference by computing its signature. Computing the signature of a template reference requires type analysis (`type-of`) to determine the type of the argument expressions. The `declare-all` strategy applies the `declare-def` rules to all top-level definitions, using the `alltd` strategy, thus creating dynamic rule mappings for each.

For the identifiers with global scope we have assumed that for each identifier (or signature) there is a single declaration that corresponds to it. Identifiers with local scope are different in that an identifier can be used in multiple scopes, corresponding to different declarations. In the language of Fig. 11, the only local identifiers are the names of template parameters. The same parameter name can be used in multiple template definitions. To distinghuish multiple uses of the same identifier, name resolution of locally scoped identifiers is implemented as a *transformation* that *renames* these identifiers to a unique name.

Fig. 14 defines the `rename-all` strategy defining renaming for our web language, applying a top-down traversal looking for terms that it can apply the `rename` transformation to. The `rename` rules transform identifier declarations and uses in order to use unique names. The rule for `Param` renames a template parameter to a unique name using the `rename-var` rule, which given an identifier `x` and a type `t`, creates a unique new name `y`, which is `x` with as annotation a freshly created string. Thus, we create a new unique term, but retain the original name of the identifier for use in error messages. Furthermore, `rename-var` defines dynamic rule `RenameId` to rename the original identifier to its new name, and `TypeOf` that maps the new identifier to its type `t`. The `rename` rule for variables (`Var`) uses the `RenameId` rule to replace a variable `x` with the corresponding unique name `y`.

To actually distinguish identifiers defined in different scopes, the `rename` rule for `TemplateDef` uses a dynamic rule scope (`{|R:s|}`) to limit the bindings of the `RenameId` dynamic rule to the traversal of the template elements in the body of the definition.

```
rules

  type-of :
    StringLit(x) -> SimpleType("String")

  type-of :
    Var(x) -> t
    where t := <TypeOf> x

  type-of :
    PropertyAccess(e, f) -> t2
    where t1 := <type-of> e
    where ent := <declaration-of> t1
    where Property(f, t2) := <lookup-property(|f)> ent

  lookup-property(|f) :
    Entity(x, prop*) -> <fetch-elem(?Property(f,_))> prop*
```

Fig. 15. Type analysis.

5.5. Type analysis

After name resolution we can map identifiers to their declarations (or types). Expressions compose new things (values, templates) from basic things (constants) and the things represented by identifiers using composition operators. Type analysis computes the type of such expressions so that we can determine if these compositions are consistent with the internal or user-provided definition of operators. The language of Fig. 11 has only simple expressions, consisting of string literals, variables, and property access. The other kind of expressions are the template `Elements`. Their composition is checked directly by consistency checking rules below.

Fig. 15 defines the `type-of` rule, which computes the types of expressions. The type of a string literal is `String`; other constants are treated similarly. The type of a variable is the type from its declaration, which we obtain using the `TypeOf` rule. The type of a property access `e.f` is determined by first computing the type `t1` of `e`. The declaration of that type is some entity `ent`, which should have a property `f` with type `t2`, which is the type of `e.f`. Any of the steps in this computation may fail; `e` itself may not have a type, the type `t1` may not be declared, or the corresponding entity may not have a property named `f`. In all these cases the application of `type-of` fails.

5.6. Consistency checking

Name resolution and type analysis set the stage for definition of consistency checking rules. The `check` rules in Fig. 16 define the main constraints for our language, *and* produce an error message explaining the failure to comply to a constraint. For brevity we have omitted rules that check unique definitions, e.g. that a name can be used for at most entity, or that an entity may not have two properties with the same name.

A constraint checking rule is a regular Stratego rule of the following general form:

```
check :
  context -> (target, message)
  where assumption
  where assumption
  where require(constraint)
```

The rule applies to some `context`, i.e. a subterm of the program we are checking. The `where` clauses first test some (zero or more) assumptions about the context. If these assumptions hold, the `constraint` is tested. If the constraint *fails*, the check rule *succeeds*, i.e. an error has been

```
strategies

  analysis = declare-all; collect-all(check)

rules

  check :                                                                    // 1
    t@SimpleType(x) -> (x, $[Type '[x]' is not defined])
    where require(<is-simple-type> t)

  check :                                                                    // 2
    e@Var(x) -> (<id>, $[Variable '[x]' not declared])
    where require(<type-of>e)

  check :                                                                    // 3
    e1@PropertyAccess(e2, f) -> (f, $[[<pp>t] has no property '[f]])
    where t := <type-of> e2
    where require(<type-of>e1)

  check :                                                                    // 4
    TemplateRef(x, e*) -> (x, $[Reference to undefined template '[x]'])
    where not(<is-primitive-template> x)
    where require(<Template> x)

  check :                                                                    // 5
    ref@TemplateRef(x, e*) -> errors
    where not(<declaration-of>ref)
    where def := <Template> x
    where errors := <zip; filter(check-arg); not(?[])> (e*, <param-types> def)

  check-arg :                                                                // 6
    (e, t) -> (e, $[Argument of type '[<pp>t]' expected (not of type '[<pp>t2]')])
    where t2 := <type-of> e
    where require(<eq>(t, t2))

  check :                                                                    // 7
    ref@TemplateRef(x, e*) -> [(x, $['[x]' expects [l] arguments; [k] provided])]
    where not(<declaration-of>ref)
    where def := <Template> x
    with k := <length>e*
    with l := <param-types; length> def
    where require(<eq>(k, l))

  check :                                                                    // 8
    PageRef(ref@TemplateRef(x, e*)) -> [(x, $[Navigation to template (not a page)])]
    where def := <declaration-of> ref
    where require(<is-page-def> def)

  constraint-warning :                                                       // 9
    Call(ref@TemplateRef(x, e*), elem*) -> [(x, $[Page definition is used as template])]
    where def := <declaration-of> ref
    where require(not(<is-page-def> def))

  check :                                                                    // 10
    Call(TemplateRef("input", [e]), []) ->
    (e, $[Argument of input should be variable or property access])
    where require(<is-lvalue> e)

  is-lvalue = ?Var(_) <+ ?PropertyAccess(_,_)
```

Fig. 16. Consistency checking rules.

detected — `require` is an alias for `not`. If an error is found, the rule returns a pair of the `target`, a subterm of `context`, and an appropriate error message. The `analysis` strategy in Fig. 16 defines the static consistency checking for our language. It first applies the `declare-all` name resolution strategy to the program, and then collects all consistency violations by applying the `check` rules using the `collect-all` strategy. Note that check rules can be defined without dependency on a

particular traversal or order of application; all context information needed to check the assumptions and constraint are provided by name resolution and type analysis rules.

Rules 1–4 define the definedness of types, variables, property access, and template references. The remaining rules check further consistency properties of template references. Rules 5–7 check the types and arity of the arguments of template references. Rule 8 checks that links (`PageRef`) are to page definitions and not to internal templates. Rule 9 gives a warning if a template inlines a page definition. Rule 10 checks that the parameter of a call to the primitive `input` template is an l-value, i.e. an assignable expression.

Note that checking of terms is *context-free*, i.e. all occurrences are checked irrespective of their context. For instance, the use of expressions as arguments of template calls is covered by rules for expressions. It is not necessary to define a rule checking that arguments to a template reference are well-typed expressions; only the interaction between the expression and the template reference needs to be checked.

5.7. Summary

We have illustrated how a language design that integrates sub-languages covering different (technical) domains allows checking of their consistent use. A key property of the language design is to choose explicit representations of elements, instead of programmatic encodings; e.g. explicit page references instead of string manipulation to construct URLs make it possible to check that only links to existing page definitions are created.

Given such a language design, the verification of the consistency of a web application can be expressed using declarative consistency checking rules comprising of name resolution, type analysis, and check rules composed by strategies.

6. Discussion and related work

6.1. Consistency checking capabilities integrated into languages and frameworks

Cooper et al. (2006) describe Links, another domain-specific language for the web. Similar to WebDSL, it consists of a number of sublanguages that are linguistically integrated and are compiled to a combination of server and client-side code. Although the language is statically typed, the paper does not describe static verification of Links applications.

Meijer et al. (2006) developed LINQ for the .NET platform. Language INtegrated Query is an extension of C# and VB.NET that provide a generic query syntax that aims to replace string-encoded SQL queries and other types of query languages such as XPath for XML. LINQ queries are statically verified by the compiler. While LINQ is a good first step, other string encoded languages remain on the .NET platform, such as regular expressions. Other general purpose languages with powerful type systems are powerful enough to add database query support as an internal DSL, type-checked by the host language. Spiewak and Zhao (2009) demonstrate how this can be achieved with Scala and Bringert et al. (2004) how it can be done with Haskell. However, error messages of the latter two frameworks are expressed in terms of Scala and Haskell type errors, rather than domain concepts.

Brabrand et al. (2002) introduced Bigwig, a domain-specific language for developing interactive web applications, which they call web services. One of the core ideas of Bigwig is that its services are session based. The services are not viewed as a collection of pages but as sequences of interactions between client and server. Such an abstraction avoids the broken page link issue discussed in Section 2, while limiting URL flexibility. The Bigwig compiler provides a number of static guarantees. Particularly interesting are the guarantees about dynamically created documents. The compiler checks that input fields always match the code that receives the input, i.e. each name property of an `<input>` tag should be handled by server-side code (Sandholm and Schwartzbach, 2000). This particular problem does not apply to WebDSL, because such input names are generated by the compiler. Besides guarantees for form inputs, Bigwig also guarantees that all documents being generated dynamically are valid XHTML 1.0, as described by Brabrand et al. (2001). WebDSL enforces consistency checks for many HTML

elements, but not a strict XHTML compliance, which is future work (see Section 6.5). The successor to Bigwig, JWIG (Møller and Schwarz, 2009), does not add additional types of analysis. The difference is that the analysis is applied in the context of a Java embedding instead of an external DSL.

Thiemann (2002) describes WASH/CGI, a Haskell library to build web applications. The Haskell type system is used to statically verify certain application properties, such as navigation links. This is easy to do, because pages in WASH are just functions, and navigation links are function calls. We downloaded WASH, but were not able to compile and test it. However, we suspect that not all application code is checked statically. For instance, callback attributes contain Haskell expressions embedded in strings. In addition, because the Haskell compiler does not know about domain-specific concepts such as pages, the error messages will not be expressed in domain terminology, but rather in terms of the Haskell type system.

In 1996 already Atkins et al. (1999) discussed the advantage of domain-specific languages in terms of static verification of web applications. The language they proposed, MAWL, enables the definition of form-based applications and performs static checks between the definition of views and the application's logic. However, Mawl is very limited in the aspects it covers, it only covers logic and user interface definitions. It does not cover aspects such as access control, data modeling, data validation and workflows with multiple participants.

The WebDSL language we described is designed to generate full-featured web applications from a single, high-level specification. In contrast, several model-driven methodologies for creating web applications have been proposed in recent years, including OOHDM (Schwabe et al., 1996), SHDM (Lima and Schwabe, 2003), WebML (Ceri et al., 2000), UWE (Koch et al., 2001), OOWS (Pastor et al., 2003), and Hera (Vdovjak et al., 2003). Many of these model-driven methodologies have evolved into tools that provide partial code generation, for example UWE4JSF (Kroiss et al., 2009) for UWE, HyperDe (Nunes and Schwabe, 2006) for SHDM, WebRatio (Brambilla et al., 2007) for WebML, OOWS (Valderas et al., 2007), and Hera-S (van der Sluijs et al., 2006) for Hera. These solutions generate only a skeleton application that targets a conventional web application platform. Developers can edit these, relying on these frameworks (as discussed in Section 2) to perform consistency checking for the application as a whole. The model-driven solutions used to design the skeletal application can only perform partial consistency checking, and are oblivious of any handwritten code added to it.

Comai et al. (2002) describe a tool for statically verifying consistency properties on XML-based WebML models. Although WebML is a visual language, the models are stored in an XML-based textual representation. The tool can be used to report erroneous patterns in those XML-based models. To verify correctness of an application, syntactic and semantic checks are performed. As WebML is a graph-based language, certain consistency properties are a natural part of the syntax: for example, links to other pages in the application can be checked by checking the syntax. The semantic checks discussed in the paper are addressing issues specific to the WebML language. Like many other model-driven approaches, WebML generates only skeletal applications, and cannot perform full consistency checking once custom code is added to an application.

In previous work Bravenboer et al. (2007) described StringBorg, a generic approach to embedding a DSL in a host language, for instance by adding SQL and regular expression support to Java. The host and embedded language become linguistically integrated and therefore static verification can be performed on the newly created combination of the languages. StringBorg is a specialization of the MetaBorg approach of Bravenboer and Visser (2004) for embedding languages using SDF grammars (Visser, 1997), which has also been used for the construction of WebDSL, as described by Visser (2007).

6.2. External consistency checkers

For many frameworks, it is technically feasible to provide better consistency checks and better feedback to developers than provided by the reference implementations, as observed in Section 3. External third-party tools can sometimes improve consistency checking and feedback, often integrating into IDEs and providing cross-language consistency checks not performed by the reference implementation.

JetBrains develops IDEs for a number of different languages. With IntelliJ IDEA, they support the Java language, but also provide specialized support for frameworks such as the JBoss Seam framework,

Struts, and GWT (JetBrains, 2009a). The IDE provides features such as content completion and error checking in JSP definitions and provides consistency checks and feedback not available with the reference implementation. Another JetBrains IDE is the Web IDE (JetBrains, 2009b), which provides support for a variety of languages that are commonly used together, including PHP, HTML, CSS, JavaScript, and SQL. While it provides only limited static checking for these languages, it provides an integrated environment for all these languages together, even though they are independently developed and maintained.

Tatlock et al. (2008) describe Quail, a tool for deep type checking of queries embedded in strings. The tool specializes on the Java language and performs safety checks of queries embedded in string literals, rather than introducing an embedded language. The authors showed that their tool can check most types of queries constructed as strings, but a small category of runtime-constructed (concatenated) strings remains unchecked. The embedded language approach applied for WebDSL and StringBorg does not have this limitation, but cannot be used with the embedded strings in the standard Java language as it was not designed for those checks.

External consistency checker tools can improve consistency checks of frameworks, and have one major advantage over the integrated consistency checkers discussed in Section 6.1 as well as those of WebDSL: they can be used with existing, industry-accepted frameworks. However, as these checkers are developed independently from the framework they analyze, they do have a number of disadvantages in terms of completeness and correctness:

- *Uncoordinated development by independent teams can lead to inconsistencies.* In addition to keeping up with the latest versions, maintaining correctness and providing complete support is increasingly difficult as more components developed by independent teams come into play.
- *Thorough framework-level consistency checking is never complete.* Whereas our approach checks a single *language*, these tools check *frameworks*. Frameworks can interact with other frameworks of external parties (e.g., a unit testing framework), new language features, and data types. The tool vendor cannot anticipate all these interactions. As a result, some of the more sophisticated consistency checks can only be implemented as heuristics.

Furthermore, these independently developed checkers pose a number of challenges to their developers, requiring significantly more effort to develop and maintain than built-in consistency checkers:

- *The language and frameworks are complex.* The complexity of the language and frameworks make their analysis very complex. Domain specific languages are typically much smaller and simpler and consequently easier to analyze.
- *The source source language and framework are not designed to enable checking.* This makes it considerably harder to implement many classes of consistency checks. An example of this is the string-embedded queries of Java, checked by Quail: only by a sophisticated data-flow analysis can these queries be checked, and completeness cannot be guaranteed.
- *Supporting and keeping up with multiple versions of languages and tools requires considerable effort.* These tools must support different, independently developed versions of languages and frameworks, and combinations thereof. This places a large burden with the tooling developers, even if the goal is only to support the most recent versions.
- *Reuse of the reference compilers and interpreters is very hard.* It takes a lot of effort to effectively reuse the reference compiler and interpreter implementations (say, the Java compiler and JSF/XML processors). These tools already implement components required for consistency checks, but they have not been designed for reuse by external consistency checkers.

6.3. Finding faults by unit testing

To manage the lack of static checking in web applications, unit testing is often proposed as a way to check different consistency properties in web applications. However, while unit tests are a highly effective, indispensable way of identifying regressions in an application, they do not provide the same level of accuracy, completeness, and the swiftness of static consistency checks. There are

two approaches to unit testing: either strictly testing a single unit of code, making heavy use of mock objects; or writing tests that cross more than one unit of code (sometimes called cross-tests or integration tests). Strict unit tests implicate one unit of code: if a test fails, the offending code is easily identified. Writing strict, explicit unit tests for basic consistency properties is laborious and impractical. Only cross-tests, testing more than one unit, are effective at checking consistency between different modules. Still, these tests are typically not complete in testing all consistency properties. They also do not clearly implicate a particular piece of source code, like static checks or even strict unit tests can do. By applying strict test-driven development it is possible to implicate the *most recent edit* of an application as the cause of the failure of a test, but not a particular line or statement.

Static consistency checks, more so than unit tests or other runtime checks, excel at rapid and accurate error reporting. Found inconsistencies can be reported before deploying or running an application, and are always associated with a particular location in the source code. When used with an integrated development environment (IDE), any constraint violations can be reported by displaying error markers in the source code editor. This allows developers to quickly adapt their code to fix mistakes, or can guide them through the process of making larger changes, when the application may be in a state where it cannot be deployed or executed.

6.4. Previous work

Key abstractions provided by the WebDSL language are in the areas of data modeling, user interface specification, and data operations. For a detailed overview of higher-level abstractions, built on top of these core concepts, we refer the reader to earlier papers: Visser (2007) and Hemel et al. (2009) gave an overview of the basic design and implementation of WebDSL, Groenewegen and Visser (2008) described the access control language, Groenewegen and Visser (2009) described data validation, and Hemel et al. (2008) described the workflow language. In contrast to these earlier papers, the present paper focuses on consistency checking, showing how it compares to consistency checking in other languages, and describing how consistency checking is implemented.

Static consistency checking and IDE integration are a powerful combination: an IDE that supports a statically checked language can report any errors directly in the editor. In previous work, Kats et al. (2009) and Kats and Visser (2010) reported on the construction of IDE plugins for the Eclipse environment using SDF (Visser, 1997) and Stratego (Bravenboer et al., 2008), particularly focusing on the constructing of an IDE for WebDSL. In the present paper, we focus on the semantic checks of the WebDSL language and the underlying semantic (Stratego) rules.

6.5. Future work

While the WebDSL compiler checks a lot of properties, it is not yet complete. WebDSL applications are not currently guaranteed to produce validating HTML, for instance. This is something we intend to investigate. Also, declarative rules could describe nesting restrictions of user-defined templates.

WebDSL is optimized for the construction of form-based interactive web information systems. It is currently not very well suited for building applications that mainly rely on heavy client-side JavaScript work. Improving support in this area will provide an opportunity for verification of Rich Internet Applications.

We also intend to investigate how we can further simplify the definition of compilers with static verification in Stratego, e.g. by even more declaratively defining scoping rules.

7. Conclusion

In this paper we demonstrated that timely, accurate and adequate error reporting is problematic in current state-of-practice web frameworks, such as Ruby on Rails, Lift and Seam. While certain frameworks report some application inconsistencies at compile-time, many are only discovered later, at deployment or run time. The lack of consistency checking in otherwise statically checked

languages can be contributed to the linguistic separation of these frameworks. Aspects of the applications are defined in separate DSLs whose consistency is not checked with the rest of the application.

In this paper we argued that DSLs should be designed from the ground up to enable static verification by linguistically integrating its sublanguages. Based on static verification and linguistic integration, the WebDSL language provides consistency checks that are reported at compile-time, can directly be traced back to their source, and provide clear, domain-specific error messages. We showed examples of error messages given by the WebDSL compiler. Subsequently we detailed the architecture and implementation of a consistency checker for a simplified version of the WebDSL language.

Acknowledgement

We would like to thank the reviewers of a previous version of this article for their constructive comments that have helped to improve the presentation. This research was supported by NWO/JACQUARD project 638.001.610, *MoDSE: Model-Driven Software Evolution*, and 612.063.512, *TFA: Transformations for Abstractions*.

Appendix. Consistency checking in web application frameworks

A.1. Data model consistency checking

Web applications typically store data in a database. To simplify data persistence, the three frameworks abstract over database architectures, allowing developers to define a data model consisting of *entities* with *properties* and *relationships* between these entities. These can be one-to-one, one-to-many, or many-to-many relationships. In this subsection we study consistency checks of entity types, relations, and data validation constraints that may be specified for the data model.

A.1.1. Consistency of property types

	M	**R**	**C**
Rails	Runtime	−	−
Seam	Compile	+	+
Lift	Compile	+	+

All three frameworks map their data models to relational databases by default. In relational databases, for each column in a table (i.e., each entity property) an existing type (i.e., a primitive type or the type of another entity) has to be specified.

In Ruby on Rails, entities, their properties, and types are defined in *database migration* scripts. Database migrations create the initial database and apply data migrations as the application evolves. In the following example, we define an migration that creates a `posts` table with three properties: `name` of type string, `title` of type string, and `content` of type text:

```
create_table :posts do |t|
  t.string :name
  t.string :title
  t.text :content
end
```

When a migration creates a property with an undefined type, (e.g. `t.strin`), no column is generated in the table for the property, nor is an error reported during the migration. The error is only detected when the property in question is *used* somewhere else in the application. Depending on the use of the property this may result in a range of errors, e.g. a `NoMethodError` is thrown when rendering an input control for the `:name` property. The error in does not lead back to the migration script in which this mistake was originally made, the error is therefore not only unclear, it is also not easily retraceable to the source of the problem.

In Seam and Lift, data models are defined as annotated Java/Scala classes, where entity properties are defined as fields. Consequently, when undefined property types are referenced, a compile-time error is reported by the Java or Scala compiler. The exact location of the error is clearly marked, and the error message – while not using the terms "entity" or "property" – is clear.

A.1.2. Consistency of entity relationships

	M	R	C
Rails	Runtime	−	+/−
Seam	Deploy	+	+
Lift	Compile	+	+

To define relationships between two entities *A* and *B*, the data model must specify a property of type *B* in *A* and an *inverse* property that links entity *B* back to *A*. For instance, in the context of an online discussion board, a topic has many messages. A `Topic` entity would therefore define a `messages` property, and a `Message` entity a `topic` property, modeling the inverse of the relationship. This inverse property must explicitly specify the nature of this relationship (one-to-one, etc.).

In Rails, inverse properties are declared using `belongs_to`, `has_one`, `has_many` and `has_and_belongs_to_many` calls:

```
class Topic < ActiveRecord::Base
  has_many :messages
end
```

This example defines a one-to-many relationship from topics to messages. It implies there must be a `Message` entity, which has a field named `topic_id`, referring back to this topic.

Rails enforces the convention of naming inverse properties by pluralizing the entity they refer to: i.e., `Message` becomes `:messages`. When this convention is not followed, or when an entity is referred to that does not exist, no error is reported when the database is initialized or migrated. However, when the property is used, a `NoMethodError` is reported, tracing back the error to wherever the property was used rather than the entity declaration that was inconsistent.

In Seam, inverse columns are defined using the `@OneToMany` annotation (in case of a one-to-many relationship) specifying the inverse property with the `mappedBy` attribute:

```
@OneToMany(mappedBy="auction")
public Set<Bid> getAllBids() {
  return allBids;
}
```

If the `mappedBy` property does not exist or is misspelled (e.g. as `aauction` instead of `auction`), an exception occurs when the application is deployed. While the actual error message tends to "drown" in an enormous stack trace, the actual message reported is accurate and specific:

```
mappedBy reference an unknown target entity property:
org.jboss.seam.example.seambay.Bid.aauction in
org.jboss.seam.example.seambay.Auction.allBids
```

While no line number or filename is supplied, a class name and property is supplied, which makes it relatively easy to find.

Lift has no support for persistent inverse properties. Instead, it allows inverse properties to be defined using a query:

```
def entries = Expense.findAll(By(Expense.account, this.id))
```

Any errors in the inverse property name (`Expense.account` in this case) are found at compile time are easy to trace back to the origin of the problem. The error message is generic, but clear.

A.1.3. Consistency of data validation

	M	R	C
Rails	Runtime	−	−
Seam	Compile/Deploy	+/−	+
Lift	Compile/Deploy	+/−	+

Most web frameworks allow developers to specify data validation constraints to validate user input. Examples of such constraints are constraints on the length of the input, or requiring a particular property to be set.

In Rails, validation constraints can be defined in entity classes. An example is the `validates_presence_of` construct, which defines that a field must be set:

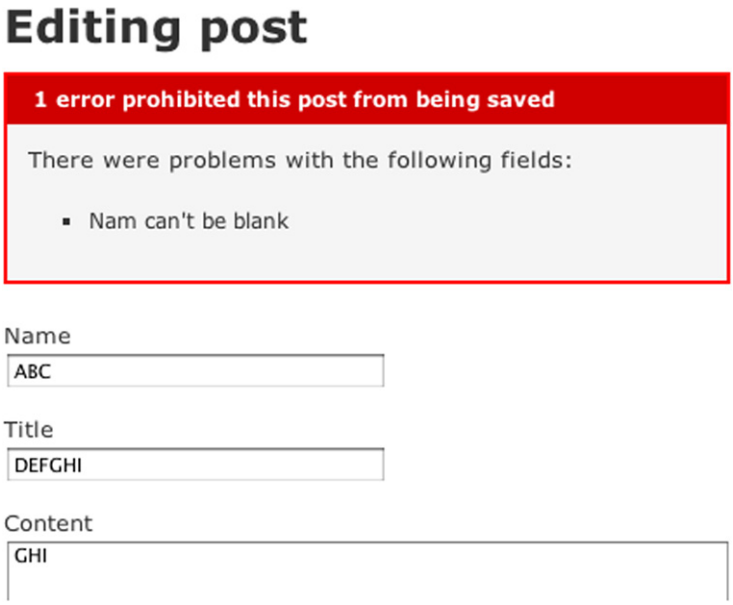

Fig. A.1. Ruby on rails validation error.

```
class Post < ActiveRecord::Base
  validates_presence_of :name
end
```

Constraint rules are not checked for validity but are still used in the user interface of an edit form in the application and when the application attempts to save an input. For example, if a presence constraint is specified for a non-existent property `nam`, Rails simply reports that property has not been set (see Fig. A.1). As Rails fails to report an error directed to the developer about this problem, the message does not provide location information of the source of the problem and does not clearly state the underlying problem.

In Lift and Seam, property validation is defined using validator annotations that are mostly checked at compile time. However, certain types of validators, such as regular-expression validators require a regular expression to be encoded as a string. The following Seam example demonstrates this:

```
@Pattern(regex="^\\w*$", message="not a valid username")
public String getUsername() {
   return username;
}
```

A syntactically incorrect regular expression such as `^[\\w*$` is not detected at compile time. Instead, it is detected when the application is deployed, printing a long stack trace in which a `PatternSyntaxException` is reported. While the regular expression in question is printed, no indication is given about the location of the error.

A.2. User interface consistency checking

The user interface of web applications is generally implemented using a combination of HTML and CSS. All three frameworks leverage HTML directly to create the user interface. They do extend HTML with additional tags or escapes to the framework language. Proper (X)HTML has a strict syntax and clearly defines how page elements (tags) can be nested. However, browsers are very liberal when it comes to the interpretation of HTML. Therefore, faulty HTML code can result in surprising interpretations. By checking the validity of page elements and element nesting before a page is sent to a browser, interpretation problems can be avoided.

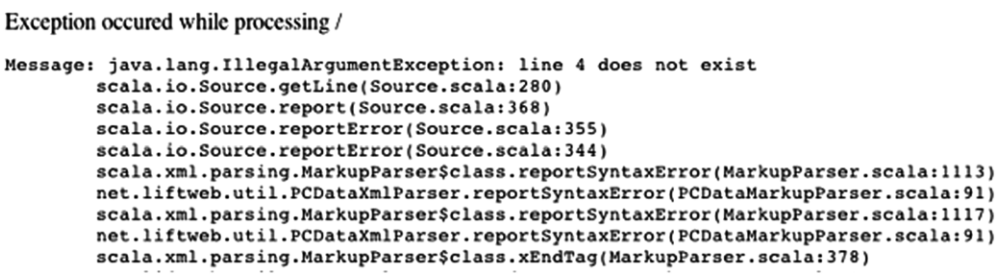

Fig. A.2. Lift exception when opening invalid tag.

A.2.1. Usage of valid page elements

	M	R	C
Rails	Runtime	+	−
Seam	Runtime	+	+
Lift	Runtime	−	−

While none of the frameworks check if the HTML tags used are valid, they typically do perform checks on their own framework-specific extensions to HTML. This subsection focuses on these special page elements.

Rails' default template language ERB does not use standard XML-style tags for defining dynamic page elements, but instead uses escapes to Ruby code. The following code generates a link to another page:

```
<%= link_to "My Blog", posts_path %>
```

Using an undefined `linkto` page construct (instead of `link_to`) results in an undefined *method* error, instead of reporting an invalid *page element*. As Ruby simply checks for general errors instead of a domain-specific ones, a conceptual mismatch arises when reporting such errors. Still, the error does pinpoint exactly the line where the error occurs.

Seam uses XML tags to render controls and realize control flow within the user interface. The following code renders a label.

```
<h:outputLabel id="UsernameLabel" for="username">Login Name</h:outputLabel>
```

When using an undefined page element, say `h:outputLabe` instead of `h:outputLabel`, the following error is reported when the user interface is loaded:

```
/home.xhtml @23,54 <h:outputLabe> Tag Library supports namespace:
    http://java.sun.com/jsf/html, but no tag was defined for name: outputLabe
```

While it is reported at run-time, the error provides a clear domain-specific error message and clear location of source of the error.

Lift, like Seam, uses XML tags to define dynamic page elements and page flow:

```
<lift:surround with="default" at="content">
  <h2>Welcome to your project!</h2>
  <p><lift:helloWorld.howdy /></p>
</lift:surround>
```

Using an undefined element `lift:surrond` instead of `lift:surround` can result in confusing errors as illustrated in Fig. A.2. The error appears when the user interface is loaded, and cannot be traced back to its origin.

A.2.2. User interface element nesting

	M	R	C
Rails	Browser	−	−
Seam	Browser	−	−
Lift	Browser	−	−

While all three frameworks base their user interface specifications on HTML, they do not check HTML validity, i.e. the correctness of tags and their nesting. For instance, when a `<td>` tag is used outside a `<table>` tag, none of the frameworks reports an error. When the page is loaded in the browsers, the invalid tag is simply ignored, a silent error. Unlike Rails, Lift and Seam do check whether the defined user interface is a well-formed XML document.

A.2.3. Consistency of references to the data model and to pages

We discuss the consistency of references to data model entities and to other pages in Section 2.4.

A.2.4. Consistency of action and controller binding

To submit information in a form, a target controller or action has to be specified to handle the action. The three frameworks handle this in different ways.

	M	R	C
Rails	Runtime	−	+
Seam	Runtime	+	−
Lift	Runtime	−	−

Rails provides a convenient way to generate a form at run-time that can be used to create or edit entities, using the `form_for` construct:

```
<% form_for(@post) do |f| %>
...
    <%= f.submit 'Update post' %>
<% end %>
```

This construct does not explicitly specify an action that should be used when the form is submitted. Instead, it follows the convention that entity controllers should have a `create` action for creating an entity, and an `update` action to edit it in case it already existed. An error is reported when submitting the form for an object for which the controller defines no `update` action (perhaps it provides a `modify` action instead). Rails then reports an unknown action error: "No action responded to update. Actions: create, destroy, edit, index, new, show, and modify". Although no file or line number is provided, the error message is domain-specific and helpful.

Rails provides additional options when binding a form to an action. One option is the ability to let the user confirm the invocation of an action, e.g. when clicking a "Destroy" link. To this end, the `:confirm` keyword is used. However, when the `:confirm` keyword is mistyped (e.g., as `:confirmation`), Rails does not detect this in any way. The keyword is simply *ignored*, resulting in immediate deletion of the entry, without any confirmation:

```
<%= link_to 'Destroy', post, :confirmation => 'Are you sure?',
            :method => :delete %>
```

In Seam, the `commandButton` element links a form to controller actions:

```
<h:commandButton id="change" value="Change"
   action="#{changePassword.changePassword}"/>
```

As these elements are part of view templates, they are not checked at compile-time. Possible errors, such as links to undefined actions, are only detected at runtime, once the button is used. Using an undefined controller in the `action` attribute results in a Seam Debug screen; when scrolling down the actual exception can be seen:

```
Exception during request processing:
Caused by javax.servlet.ServletException with message:
    "#{changePassword.changePasswor}:
javax.el.MethodNotFoundException:
    /password.xhtml @37,91 action="#{changePassword.changePasswor}":
Method not found: Proxy to jboss.j2ee:ear=jboss-seam-booking.ear,
    jar=jboss-seam-booking.jar,name=ChangePasswordAction,service=EJB3
    implementing [interface org.jboss.seam.example.booking.ChangePassword]
    .changePasswor()"
```

The supplied `MethodNotFoundException` is hardly descriptive or domain-specific, but the error be traced back to its origin as the filename and line and column numbers are provided.

A.3. Logic, action, and controller consistency checking

The logic of a web application is typically defined in controllers, sometimes subdivided into actions. Like the user interface part of the web applications, controllers contain references to other parts of the

applications, such as the user interface and data model. Consistency checking can ensure that these references are valid and remain valid as an application is changed.

A.3.1. Consistency of data model references

	M	R	C
Rails	Runtime	+	−
Seam	Compile	+	+
Lift	Compile	+	+

Controllers use references to the data model to persist data to the database, to read or write properties, or to perform queries.

In Rails, references to undefined entity types are reported as "uninitialized constant" errors when the code is invoked at runtime. The error exposes implementation details of the framework and can be confusing to developers, especially since the framework internally prefixes the entity name with the controller name. For instance, when an undefined entity `E` is referenced from controller `C`, the following error is reported: "uninitialized constant C::E". Still, the accompanying stack trace refers back to the code in which the error occurred, so the error can be traced back to its source. Nonexistent properties are reported in a similar fashion, but identified as an "undefined method".

In Seam and Lift, controllers are written in Java and Scala, which are statically checked at compile time. References to undefined entity types are reported as "X cannot be resolved to a type". And non-existing properties in Scala are reported as "not a member of type X".

A.3.2. Consistency of redirects to pages

	M	R	C
Rails	Runtime	−	−
Seam	Runtime	−	−
Lift	Runtime	−	−

Similar to links in views, it is also common for controller code to redirect the user to a different page or controller.

In Rails, redirecting the user to different controllers and actions is done using `redirect_to`:

```
redirect_to :action => "index"
```

When an incorrect action name is used, for example by using `"home"` instead of `"index"`, an unexpected error occurs when the controller is invoked: "RecordNotFound" error: "Couldn't find Post with ID=home". Apparently, when an action is not defined, the action name is interpreted as an entity identifier in some cases. The error is reported as part of the `show` action of the controller, but there is no reference to the location of the actual error.

In Seam, redirects to pages are performed by returning the URL as the return value of an action:

```
return "/index.xhtml";
```

Lift has a `redirectTo` method for this purpose:

```
redirectTo("/index.html")
```

Similar to links in views, these redirects are not checked and specifying a redirect to an undefined page simply result in "404 not found" errors for the end-user.

A.3.3. Consistency of data binding

	M	R	C
Rails	Runtime	−	−
Seam	N/A	N/A	N/A
Lift	N/A	N/A	N/A

Forms can be used to create or modify entities in the data base. By specifying a *data binding* between form elements and entity properties, frameworks can directly interpret the results of a submitted form, creating or updating an entity.

In Rails, data can be bound to entities by passing the map containing the HTTP (POST/GET/PUT) request values to the constructor of a new object:

```
@post = Post.new(params[:post])
```

However, if a mistake is made in the expression, e.g. by inappropriately using :get when the form was changed to use a POST request or simply mistyping submit method, no error is reported. The result is that no data is bound to the properties of `@post` at all, often resulting in a validation error and empty input fields as can be seen in Fig. A.3.

Fig. A.3. Rails reports validation errors when making errors in data binding.

In Seam and Lift, controls are attached to an entity property and perform data binding themselves, they retrieve the value from the property and write back the value when a new value is entered. Therefore no data binding faults are possible, other than referring to non-existing properties and entities (discussed in Appendix A.3.1).

A.4. Access control consistency checking

Access control can be used to restrict parts of web application to authenticated users. Access control rules that depend on the database (as they typically do) contain data model references that should be checked for consistency. Some frameworks allow access control rules to be defined separately from the user interface and controllers. Any bindings to pages and actions should also be checked for consistency. (We will not discuss these bindings here since they are treated in very similar to bindings from other parts of the application.)

A.4.1. Consistency of data model references

	M	**R**	**C**
Rails	Runtime	+	−
Seam	Runtime	−	−
Lift	Compile	+	+

Rails uses the `before_filter` construct to invoke a method before actions within a controller are invoked:

```
before_filter :authorize

def authorize
  auth_user = User.find_by_id(session[:user_id])
  unless auth_user && auth_user.age > 10
    redirect_to(:controller => "accessDenied", :action => "accessDenied")
  end
end
```

Errors are found when the authorization method is invoked, and are reported with a clear indication of the source of the error.

In Seam, access control rules can be defined in a separate rule language:

```
rule CreateBlog
  no-loop
  activation-group "permissions"
when
  mbr: Member()
  acct: MemberAccount(member.memberId == mbr.memberId)
  check: PermissionCheck(target.memberId == mbr.memberId,
                         action == "createBlog", granted == false)
then
  check.grant();
end
```

This DSL is not verified at compile-time. When a property is referred to that does not exist, e.g. `target.memberid` instead of `target.memberId`, the error is reported at the level of the page, instead of in the rule file: "RuntimeDroolsException: Exception executing predicate target.memberid == mbr.memberId". A location in the source code is supplied, but this refers to the location where the problem occurred, not the actual origin of the error: the rule file.

In Lift, access control rules are expressed using Scala expressions, in which invalid references to the data model are detected at compile time.

References

Atkins, D.L., Ball, T., Bruns, G., Cox, K.C., 1999. Mawl: a domain-specific language for form-based services. IEEE Trans. Softw. Eng. 25 (3), 334–346.

Brabrand, C., Møller, A., Schwartzbach, M.I., 2001. Static validation of dynamically generated html. In: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering. PASTE 01, Snowbird, Utah, USA, June 18–19, 2001. ACM, pp. 38–45.

Brabrand, C., Møller, A., Schwartzbach, M.I., 2002. The <bigwig> project. ACM Trans. Internet Techn. 2 (2), 79–114.

Brambilla, M., Comai, S., Fraternali, P., Matera, M., 2007. Designing web applications with WebML and WebRatio. Web Eng.: Model. Implementing Web Appl. 221–260.

Bravenboer, M., Dolstra, E., Visser, E., 2007. Preventing injection attacks with syntax embeddings. A host and guest language independent approach. In: Lawall, J. (Ed.), Generative Programming and Component Engineering. GPCE 2007. ACM, New York, NY, USA, pp. 3–12.

Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E., 2008. Stratego/XT 0.17. A language and toolset for program transformation. In: Experimental Software and Toolkits. Sci. Comput. Programming 72 (1–2), 52–70 (special issue).

Bravenboer, M., van Dam, A., Olmos, K., Visser, E., 2006. Program transformation with scoped dynamic rewrite rules. Fund. Inform. 69 (1–2), 123–178.

Bravenboer, M., Visser, E., 2004. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In: Schmidt, D.C. (Ed.), Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2004. ACM Press, Vancouver, Canada, pp. 365–383.

Bringert, B., Höckersten, A., Andersson, C., Andersson, M., Bergman, M., Blomqvist, V., Martin, T., 2004. Student paper: Haskelldb improved. In: Haskell'04: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell. ACM, New York, NY, USA, pp. 108–115.

Ceri, S., Fraternali, P., Bongio, A., 2000. Web modeling language (webml): a modeling language for designing web sites. Comput. Netw. 33 (1–6), 137–157.

Comai, S., Matera, M., Maurino, A., 2002. A model and an xsl framework for analyzing the quality of webml conceptual schemas. In: Spaccapietra, S., March, S.T., Kambayashi, Y. (Eds.), Conceptual Modeling — ER 2002, 21st International Conference on Conceptual Modeling, Tampere, Finland, October 7–11, 2002, Proceedings. In: Lecture Notes in Computer Science, vol. 2503. Springer, pp. 339–350.

Cooper, E., Lindley, S., Wadler, P., Yallop, J., 2006. Links: web programming without tiers. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (Eds.), Formal Methods for Components and Objects, 5th International Symposium. FMCO 2006, Amsterdam, The Netherlands, November 7–10, 2006, Revised Lectures. In: Lecture Notes in Computer Science, vol. 4709. Springer, pp. 266–296.

Groenewegen, D.M., Visser, E., 2008. Declarative access control for WebDSL: combining language integration and separation of concerns. In: Schwabe, D., Curbera, F. (Eds.), Eighth International Conference on Web Engineering. ICWE 2008. IEEE CS Press, pp. 175–188. Best paper award.

Groenewegen, D.M., Visser, E., 2009. Integration of data validation and user interface concerns in a dsl for web applications. In: van den Brand, M., Gray, J. (Eds.), Software Language Engineering, Second International Conference. SLE 2009, Denver, USA, October, 2009, Revised Selected Papers. In: Lecture Notes in Computer Science. Springer.

Hemel, Z., Kats, L.C.L., Groenewegen, D.M., Visser, E., 2009. Code generation by model transformation. A case study in transformation modularity. Softw. Syst. Model. 9, 375–402.

Hemel, Z., Verhaaf, R., Visser, E., 2008. WebWorkFlow: an object-oriented workflow modeling language for web applications. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (Eds.), Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems. MODELS 2008. In: Lecture Notes in Computer Science, vol. 5301. Springer, Heidelberg, pp. 113–127.

JetBrains, 2009a. Intellij idea. http://www.jetbrains.com/idea/.
JetBrains, 2009b. Web ide. http://www.jetbrains.com/webide.
Kats, L.C.L., Kalleberg, K.T., Visser, E., 2009. Domain-specific languages for composable editor plugins. In: Vinju, J., Ekman, T. (Eds.), Proceedings of The Ninth Workshop on Language Descriptions, Tools, and Applications. LDTA 2009. In: ENTCS. Elsevier.
Kats, L.C.L., Visser, E., 2010. The spoofax language workbench. rules for declarative specification of languages and IDEs. In: Rinard, M. (Ed.). Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2010, October 17–21, 2010, Reno, NV, USA.
Koch, N., Kraus, A., Hennicker, R., 2001. The authoring process of the uml-based web engineering approach. In: First International Workshop on Web-Oriented Software Technology.
Kroiss, C., Koch, N., Knapp, A., 2009. Uwe4jsf: a model-driven generation approach for web applications. In: Gaedke, M., Grossniklaus, M., Díaz, O. (Eds.), Web Engineering, 9th International Conference. ICWE 2009, San Sebastián, Spain, June 24–26, 2009, Proceedings. In: Lecture Notes in Computer Science, vol. 5648. Springer, pp. 493–496.
Lima, F., Schwabe, D., 2003. Modeling applications for the semantic web. In: Lovelle, J.M.C., Rodríguez, B.M.G., Aguilar, L.J., Gayo, J.E.L., del Puerto Paule Ruíz, M. (Eds.), Web Engineering, International Conference. ICWE 2003, Oviedo, Spain, July 14–18, 2003, Proceedings. In: Lecture Notes in Computer Science, vol. 2722. Springer, pp. 417–426.
Meijer, E., Beckman, B., Bierman, G.M., 2006. Linq: reconciling object, relations and xml in the .net framework. In: Chaudhuri, S., Hristidis, V., Polyzotis, N. (Eds.), Proceedings of the ACM SIGMOD International Conference on Management of Data. Chicago, Illinois, USA, June 27–29, 2006. ACM, p. 706.
Møller, A., Schwarz, M., 2009. Jwig: yet another framework for maintainable and secure web applications. In: Filipe, J., Cordeiro, J. (Eds.), WEBIST 2009 — Proceedings of the Fifth International Conference on Web Information Systems and Technologies. Lisbon, Portugal, March 23–26, 2009. INSTICC Press, pp. 47–53.
Nunes, D.A., Schwabe, D., 2006. Rapid prototyping of web applications combining domain specific languages and model driven design. In: Carr, L., Roure, D.D., Iyengar, A., Goble, C.A., Dahlin, M. (Eds.), Proceedings of the 15th International Conference on World Wide Web. WWW 2006, Edinburgh, Scotland, UK, May 23–26, 2006. ACM, pp. 889–890.
Pastor, O., Fons, J., Pelechano, V., 2003. OOWS: a method to develop web applications from web-oriented conceptual models. In: Web Oriented Software Technology. IWWOST'03, pp. 65–70.
Sandholm, A., Schwartzbach, M.I., 2000. A type system for dynamic web documents. In: POPL, pp. 290–301.
Schwabe, D., Rossi, G., Barbosa, S.D.J., 1996. Systematic hypermedia application design with oohdm. In: Hypertext 96, The Seventh ACM Conference on Hypertext. Washington DC, March 16–20, 1996. ACM, pp. 116–128.
Spiewak, D., Zhao, T., 2009. Scalaql: language-integrated database queries for scala. In: van den Brand, M., Gasevic, D., Gray, J. (Eds.), Second International Conference. SLE 2009, Denver, CO, USA, October 5–6, 2009, Revised Selected Papers. In: Lecture Notes in Computer Science. Springer.
Tatlock, Z., Tucker, C., Shuffelton, D., Jhala, R., Lerner, S., 2008. Deep typechecking and refactoring. In: Harris, G.E. (Ed.), Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2008, October 19–23, 2008, Nashville, TN, USA. ACM, pp. 37–52.
Thiemann, P., 2002. Wash/cgi: server-side web scripting with sessions and typed, compositional forms. In: Krishnamurthi, S., Ramakrishnan, C.R. (Eds.), Practical Aspects of Declarative Languages, 4th International Symposium. PADL 2002, Portland, OR, USA, January 19–20, 2002, Proceedings. In: Lecture Notes in Computer Science, vol. 2257. Springer, pp. 192–208.
Valderas, P., Pelechano, V., Pastor, O., 2007. A transformational approach to produce web application prototypes from a web requirements model. Int. J. Web Eng. Technol. 3 (1), 4–42.
van der Sluijs, K., Houben, G.-J., Broekstra, J., Casteleyn, S., 2006. Hera-s: web design using sesame. In: Wolber, D., Calder, N., Brooks, C.H., Ginige, A. (Eds.), Proceedings of the 6th International Conference on Web Engineering. ICWE 2006, Palo Alto, California, USA, July 11–14, 2006. ACM, pp. 337–344.
Vdovjak, R., Frasincar, F., Houben, G.-J., Barna, P., 2003. Engineering semantic web information systems in hera. J. Web Eng. 2 (1–2), 3–26.
Visser, E., 1997. Syntax definition for language prototyping. Ph.D. Thesis. University of Amsterdam. http://swerl.tudelft.nl/bin/view/EelcoVisser/SyntaxDefinitionForLanguagePrototyping.
Visser, E., 2007. Domain-specific language engineering. In: Lämmel, R., Saraiva, J., Visser, J. (Eds.). Generative and Transformational Techniques in Software Engineering, GTTSE 2007, Universidade do Minho, Braga, Portugal. International Summer School GTTSE 2007, Pre-Proceedings. pp. 265–318.
Visser, E., 2008. WebDSL: a case study in domain-specific language engineering. In: Lämmel, R., Visser, J., Saraiva, J. (Eds.), International Summer School on Generative and Transformational Techniques in Software Engineering. GTTSE 2007. In: Lecture Notes in Computer Science, vol. 5235. Springer, Heidelberg, pp. 291–373.
Visser, E., Benaissa, Z.-e.-A., Tolmach, A., 1998. Building program optimizers with rewriting strategies. In: Proceedings of the third ACM SIGPLAN International Conference on Functional Programming. ICFP 1998. ACM Press, pp. 13–26.