Flow Analysis of Code Customizations

Anders Hessellund and Peter Sestoft

IT University of Copenhagen, Denmark
{hessellund,sestoft}@itu.dk

Abstract. Inconsistency between metadata and code customizations is a major concern in modern, configurable enterprise systems. The increasing reliance on metadata, in the form of XML files, and code customizations, in the form of Java files, has led to a hybrid development platform. The expected consistency requirements between metadata and code should be checked but often are not, so current tools offer surprisingly poor development support. In this paper, we adapt classical data flow analyses to detect inconsistencies and provide better static guarantees. We provide a formalization of the consistency requirements and a set of adapted analyses for a concrete case study. Our work is implemented in a fast and efficient prototype in the form of an Eclipse plugin. We validate our work by testing this prototype on actual production code; preliminary results show that this approach is worthwhile. We found a significant number of previously undetected consistency errors and have received very positive feedback from the developer community in the case study.

1 Introduction

Complex enterprise systems increasingly use metadata in the form of XML files for configuration. This facilitates maintenance and allows developers to gain a better overview by focusing on the *what* of the system rather than on the *how*. However, metadata cannot tell the whole story and especially for business logic requirements, it is often necessary to add custom code (e.g. in Java) to implement specialized functionality. This is frequently done through *code customizations*. A code customization is a small code snippet with a predefined interface that can be plugged into the base system. The relation between metadata and code customization is that metadata declares the existence of specialized business logic and the code customization provides an implementation. A code customization fulfills concrete requirements but at the same time introduces new consistency constraints on the system: Metadata and code must agree on proper use of common names and types. Current tools are surprisingly poor at managing these consistency constraints and the errors that arise from violating them.

In this paper, we claim that some of these problems can be eliminated by adapting classic data flow analyses to framework-specific code customizations. We propose a set of data flow analyses for a concrete case study: The Apache Open For Business (OFBiz) [1] enterprise resource planning (ERP) system.

J. Vitek (Ed.): ECOOP 2008, LNCS 5142, pp. 285–308, 2008.
© Springer-Verlag Berlin Heidelberg 2008

These analyses are implemented in an Eclipse plugin and have been applied to a production quality installation of OFBiz. Our prototype has found a large number of consistency errors in this code base. In this paper, we show how our prototype can locate the source of each error and help developers increase the quality of their code customizations. The prototype has been released to the OFBiz developer community and received very positive feedback [2, 3].

The contributions of this work are:

- A formalization of the consistency constraints between metadata and code customizations in OFBiz.
- A set of framework-specific adaptions of dataflow analyses based on this formalization.
- A working implementation of these analyses in the form of an Eclipse plugin.
- An empirical validation of the tool by analyzing production code and eliciting feedback from OFBiz developers.
- A discussion of the limitations of the analyses, and the trade-off between soundness (no false negatives) and precision (only few false positives).

Section 2 below shows a motivating example from the out-of-the-box version of OFBiz, and section 3 provides more background on this case study with an emphasis on its size and complexity. Section 4 formalizes the implicit consistency requirements in OFBiz, and sections 5 and 6 describe the flow analyses used to realize these consistency requirements. Section 7 describes our prototype Eclipse plugin that implements these analyses, and section 8 presents and discusses empirical results from applying this tool to the case study. Section 9 discusses wider perspectives and implications of this work, and section 10 considers related work.

2 Motivating Example: Code Customizations in OFBiz

The OFBiz framework exposes a range of services. A service can be described in a *service definition* such as that shown in listing 1.1. A service definition contains metadata about a service, such as its name and where it is implemented, and describes the service's input and output in the form of attributes. For each attribute, it states its name and type as well as whether this attribute is mandatory or optional. Services are often implemented in Java code snippets, called *code customizations*. A code customization must conform to the service interface given by the service definition. *Conformance* is in this case defined as accepting the same input and returning the same output as specified in the service definition.

Listing 1.1 provides an actual example of the `buildPdfFromSurveyResponse` service from the `Content` module in OFBiz v.3. This service creates PDF-files based on online surveys. The actual implementation of a service is typically written in a more expressive language such as Java. As stated in line 3 of listing 1.1, the `buildPdfFromSurveyResponse` service is implemented in Java by the method `buildPdfFromSurveyResponse`, shown in listing 1.2. The service

```
1 <service name="buildPdfFromSurveyResponse" engine="java"
2         location="org.ofbiz.content.survey.PdfSurveyServices"
3         invoke="buildPdfFromSurveyResponse">
4         <description>Build Pdf From Survey
             Response</description>
5         <attribute name="surveyResponseId" type="String"
             mode="IN" optional="false" />
6         <attribute name="outByteWrapper"
             type="org.ofbiz.entity.util.ByteWrapper"
             mode="OUT" optional="false" />
7 </service>
```

Listing 1.1. The `buildPdfFromSurveyResponse` service definition states that the service is implemented in Java by the `buildPdfFromSurveyResponse` method in the `PdfSurveyServices` class. The service has a mandatory input attribute `surveyResponseId` and a mandatory output attribute `outByteWrapper`.

```
 1 public static Map buildPdfFromSurveyResponse
 2 (DispatchContext dctx, Map context) {
 3   GenericDelegator delegator = dctx.getDelegator();
 4   Map results = ServiceUtil.returnSuccess();
 5   String surveyResponseId =
         (String)context.get("surveyResponseId");
 6   String contentId = (String)context.get("contentId");
 7   try {
 8     if (UtilValidate.isNotEmpty(surveyResponseId)) {
 9       GenericValue surveyResponse =
10         delegator.findByPrimaryKey("SurveyResponse",
11           UtilMisc.toMap("surveyResponseId",
                 surveyResponseId));
12     }
13      ...some 45 lines of code left out ...
14     ByteWrapper outByteWrapper =
15         new ByteWrapper(baos.toByteArray());
16     results.put("outByteWrapper", outByteWrapper);
17   } catch (GenericEntityException e) {
18     ServiceUtil.returnError(e.getMessage());
19   }
20   return results;
21 }
```

Listing 1.2. The `buildPdfFromSurveyResponse` method implements the service declared in listing 1.1. The `context` map declared in line 2 contains input attributes and the `results` map in line 4 contains the output attributes. An input attribute is read in line 5 and an output attribute is set in line 16.

declares two mandatory attributes: an input attribute `surveyResponseId` of type `String` and an output attribute `outByteWrapper` of type `ByteWrapper`.

There are three expected kinds of consistency constraints between listing 1.1 and 1.2, detailed in sections 2.1 through 2.3 below. These constraints are not stated explicitly in the OFBiz documentation and are not checked until runtime where violations can lead to unpredictable behaviour.

2.1 A Java Implementation Must Exist

The service definition says that there must exist a `PdfSurveyServices` class with a `buildPdfFromSurveyResponse` method that implements the service. Checking this constraint is a prerequisite for checking the two other kinds of constraints; since this is fairly easy to do it is not discussed further.

2.2 Only Declared Input Attributes May Be Accessed

The service definition contains a mandatory input attribute `surveyResponseId` of type `String`. Input attributes are supplied to the method via the `context` map in line 2 in listing 1.2. The implementation code should only access keys in this map that correspond to declared input attributes, such as `surveyResponseId`. Lines 5 and 6 in listing 1.2 show a correct access to a declared input attribute `surveyResponseId` and an incorrect access to an undeclared input attribute `contentId`.

2.3 All Declared Output Attributes Must Be Assigned

The example service definition contains a single mandatory output attribute, `outByteWrapper`. Clients of this service can assume that on successful execution, this key is present in the `results` map. Hence the service implementation should make sure that either the key is present or an error message is returned. In listing 1.2, the `outByteWrapper` attribute is set in line 16. However, the `findByPrimaryKey` method in line 10 may throw a checked exception which would prevent the attribute from being set. The `catch` block in line 18 creates a map containing an error message but does not return this map and hence has no effect. This type of subtle error is hard to spot and potentially leads to erroneous output of the service.

Note that the two errors described above are genuine consistency errors found in the release version of OFBiz (November 2007).

2.4 Current Development Tools

A key concern in OFBiz development is to ensure that the definition and implementation of a service are consistent. Development is traditionally done using normal Java- and XML-editors, so there is no tool-support for checking the three above-mentioned kinds of consistency constraints. This is because traditional tools such as XML schema conformance checking and Java type checking do not reveal constraint violations that involve *both* XML and Java artifacts. This lack of tool support causes slow development, costly maintenance,

and errors in deployed OFBiz products, as shown in a previous survey [4] of the OFBiz user forums, issue tracking system, and so on. That consistency is a major concern is further evidenced from the positive feedback that we have received from the OFBiz community on the release of our initial prototype [2, 3]. Before we describe analyses and tools developed, we briefly introduce the overall case study, OFBiz.

3 Case Study: Apache Open for Business (OFBiz)

The OFBiz [1] project is an open source enterprise resource planning (ERP) system. The cornerstone of the project is a J2EE-based framework. The base framework is implemented in Java and can be configured using XML files conforming to 17 different schemas. These schemas can be considered as separate domain-specific languages tailored for individual concerns in OFBiz development, such as user interface, data model, services, workflow etc. The use of these schemas is described in greater detail in [4]. Apart from XML configuration files, code customizations written as small Java code snippets can be added to realize specialized functionality that is beyond the scope of ordinary configuration. Finally, the framework uses HTML and a template engine to render user interfaces, and also uses scripting languages, such as BeanShell script [5], for minor tasks.

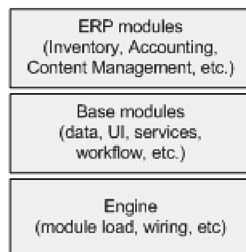

Fig. 1. The 3-layer architecture of OFBiz

The framework has a three layer architecture as shown in figure 1. The bottom layer is a base engine that handles loading and wiring of modules. The middle layer is a set of base modules to define business objects, services, graphical widgets and workflows. The top layer is a set of standard ERP application modules such as Inventory, Accounting, Content Management etc. Developers can add their own modules or extend existing modules so the framework is highly flexible.

To give an impression of the size and strength of OFBiz, we list the following metrics: The out-of-the-box solution consists of approximately 180 000 lines of Java code and 195 000 lines of XML. The data model consists of more than 700 domain classes designed according to patterns based on industrial practice [6]. Industrial users include large companies such as British Telecom and United Airlines [7] as well as a range of small- and medium-sized companies [8]. It is a top-level project in the Apache Software Foundation and is backed by an active community. We therefore consider it a valid case study of large scale, industrial-strength development with metadata and code customizations.

Because of the size and complexity of OFBiz, we have chosen to focus on a specific subset of code customizations. OFBiz exposes a range of services to internal and external clients. These services are located by their metadata descriptions, i.e., service definitions as exemplified in listing 1.1. A concrete

service, described by a service definition, can be implemented in a variety of ways, such as SOAP, RMI, scripting languages, a custom domain-specific language for data manipulation called Minilang, or by a general-purpose language. We will focus on the last option, viz. the general-purpose language Java, since services implemented in Java are the cause of most problems. Using Java is a double-edged sword: On one hand, it gives developers great expressive power to meet their requirements; on the other hand, this is easily misused to create a whole range of new and subtle bugs.

4 Formalizing the Consistency Requirements

The OFBiz documentation does not state any consistency requirements on the relation between metadata and code customizations. The expected requirements are implicit and only enforced at runtime where a constraint violation can lead to unpredictable behaviour or system malfunction. In this section, we will formalize the *exact* consistency requirements between metadata and code customizations with respect to input and output. Later in sections 5 and 6, we will describe our analyses that are approximations of these requirements. Our formalization is based on the idea that the XML metadata files are the specification that the Java code must conform to. To formalize the consistency constraints between metadata and code, we first introduce our general formalization of metadata and code.

4.1 Formalizing Metadata

To represent a service definition from the XML metadata, we introduce the following definitions: Let Θ be the unbounded set of all possible service attributes, and let θ be an individual attribute, such as `surveyResponseId`. We will use the predicate *mandatory* to denote whether an attribute is mandatory or optional. We will use the predicates *in* and *out* to denote declared input and output attributes. Note that an attribute can be *both* input and output at the same time. Let Θ_{IN} be the bounded set of all *declared* input attributes. Let Θ_{OUT} be the bounded set of *declared* output attributes. Finally, let Θ_{REQ} be the bounded set of all declared, mandatory output attributes.

$$\Theta = \{\theta_1, \theta_2, \ldots\}$$
$$\Theta_{IN} = \{\theta | \theta \in \Theta \wedge \text{in}(\theta)\}$$
$$\Theta_{OUT} = \{\theta | \theta \in \Theta \wedge \text{out}(\theta)\}$$
$$\Theta_{REQ} = \{\theta | \theta \in \Theta_{OUT} \wedge \text{mandatory}(\theta)\}$$

4.2 Formalizing Code

Code customizations of services in OFBiz always have the following signature:

```
public static Map service
        (DispatchContext dctx, Map context)
```

Input attributes are stored in the `context` map and output attributes are typically stored in a `results` map. We will use `context` and `results` as general identifiers for the sets of input and output attributes in our formalization. Furthermore, let Σ be the set of all program traces. A program trace, $\sigma = \langle \alpha_1, \alpha_2 \ldots \alpha_n \rangle$, is a sequence of executed statements.

$$\Sigma = \{\sigma_1, \sigma_2 \ldots\}$$
$$\sigma = \langle \alpha_1, \alpha_2 \ldots \alpha_n \rangle$$

4.3 Input Requirements

Service input consists of a set of input attributes, as described in section 2. The requirement on input attributes is that only declared attributes are read in the code. As an illustration, one can think of this as similar to Java's scoping rules [9, ch.6.3]. An attribute can only be used if it is in the scope of its declaration. If an attribute is required by the service definition then it is in scope in the code.

The `context` parameter contains the input attributes so we are interested in checking every use of `context`. To use an input attribute, `x`, one must invoke the `context.get(x)`. If this method is invoked then the service definition must state that `x` is indeed an input attribute. This rule can be stated more formally as follows:

$$\forall \langle \alpha_1, \alpha_2 \ldots \alpha_n \rangle \in \Sigma. \forall \mathrm{x} \in \Theta. \forall i. \alpha_i \text{ is } \mathtt{context.get(x)} \Rightarrow \mathtt{x} \in \Theta_{IN}$$

The above constraint is violated if there is a statement α_i and a key $\mathtt{x} \notin \Theta_{IN}$ such that α_i attempts to read `x` from `context`.

4.4 Output Requirements

Service output consist of a set of output attributes. Like input attributes, output attributes are stored in a `Map` but an output map may be returned from multiple `return` statements. The requirement on output attributes is that mandatory output attributes are *definitely assigned* at the point where they are returned. In particular, one must ensure that all mandatory output attributes have definitely been assigned for each individual `return` statement. This corresponds to the definite assignment rules of Java [9, ch.16] and checking that these rules are obeyed involves many of the same intricacies, most notably in the case of `try` statements.

Output attributes are stored in a `results` map, so we are interested in checking every use of `results`. To assign a value `y` to an output attribute `x` one must invoke `results.put(x, y)`. Such an assignment can be undone by invoking the `results.remove(x)` method. If an output attribute `x` is mandatory then it must be assigned when the method returns. This rule can be stated more formally like this:

$$\forall \mathrm{x} \in \Theta_{REQ}. \forall \langle \alpha_1, \alpha_2 \ldots \alpha_n \rangle \in \Sigma. \forall i. \alpha_i \text{ is } \mathtt{return\ results} \Rightarrow$$
$$\exists j. j < i. (\alpha_j \text{ is } \mathtt{results.put(x,y)} \wedge \neg \exists k. j < k < i. \alpha_k \text{ is } \mathtt{results.remove(x)})$$

The above constraint is violated if there is a mandatory output attribute x and an execution σ with a `return` statement α_i and such that *either* there is no statement α_j that sets key x before α_i *or* there is a statement α_k between α_j and α_i that removes the key x from the map.

4.5 Further Output Attribute Checks

In addition, one might require that the service implementation only attempts to set output attributes that may be needed according to the service definition. This rule can also be stated formally as follows:

$$\forall x \in \Theta . \forall \langle \alpha_1 , \alpha_2 \ldots \alpha_n \rangle \in \Sigma . \forall i . \alpha_i \texttt{ is return results} \wedge \\ \exists j . j < i . \alpha_j \texttt{ is results.put}(x, y) \Rightarrow x \in \Theta_{OUT}$$

This constraint is violated if there is an attribute x and an execution σ with a `return` statement α_i and a key $x \notin \Theta_{OUT}$ such that α_i attempts to assign x to the returned map `results`, and x is not an output attribute.

4.6 Duality of Requirements

Interestingly, the requirement on the output in section 4.4 is the dual of the requirement on the input in section 4.3. The input requirement says that if x is used in `context.get(x)` on *some* execution path in the code then attribute x must be provided to the service according to the XML service definition. The output requirement says that if attribute x must be returned from the service according to the XML service definition then *every* execution path that leads to a `return` statement in the Java code must define x. We shall see that this duality is reflected in the analyses by the use of different *meet* operators.

5 Analysis of Service Input

In this section, we describe the individual steps performed by our analysis of service input. The analysis uses two artifacts: the service definition and the implementation code. The analysis consists of the following five steps:

1. Collect declared attributes from the XML service definition.
2. Construct a control flow graph for the Java code.
3. Perform a *reaching definitions flow analysis* on the code.
4. Construct a *def-use chain* for the code.
5. For each use of the input map, check that only declared keys are used.

5.1 Collect Declared Attributes from the XML Service Definition

The first step in the analysis is to determine which input attributes are declared. Not all service definitions are as simple as listing 1.1. Figure 2 shows the OFBiz

domain classes involved in the collection of service attributes. In addition to the service's own attributes, a service can inherit attributes from other services and optionally override characteristics of these inherited attributes. For instance, the `ClearCommerce` and `CyberSource` provider services from the `Accounting` module all inherit attributes from the general OFBiz credit card and payment processing services to enable service polymorphism. Furthermore, a service can automatically collect attributes based on the fields of a business entity through the `auto-attributes` association, shown in figure 2. For instance, the `updateAgreement` service from the `Accounting` module uses all fields of the `Agreement` business entity as input attributes. The advantage of this is that whenever the `Agreement` business entity is extended with a new field, this change is automatically reflected in the `updateAgreement` service. This can be further refined such that all primary keys are mandatory attributes and all non-primary keys are optional attributes.

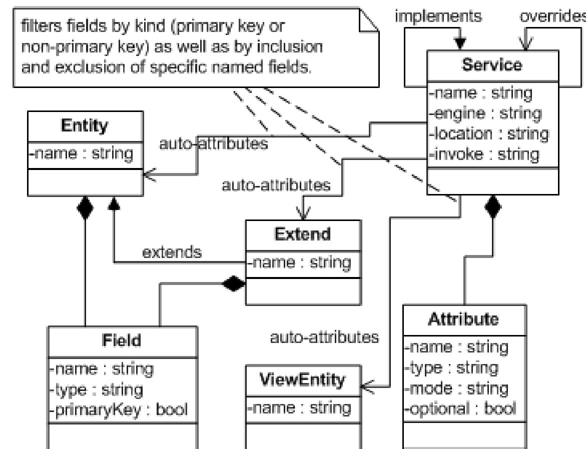

Fig. 2. The domain classes relevant for locating and collecting attributes for an OFBiz service. The collection process is described as step (1) in section 5. A service may have its own attributes, may inherit attributes from other services, and may collect automatic attributes based on fields of a set of business *entities*, *extends* and *views*.

In summary, collecting service attributes from the metadata is a non-trivial process. The collection process requires traversing the service inheritance hierarchy and checking for overridden attributes. If the service uses the `auto-attributes` association, shown in figure 2, then one must also traverse related business entities, use their fields as attributes and check whether primary or non-primary keys are filtered out and whether any named fields are included or excluded.

5.2 Construct a Control Flow Graph for the Java Code

The second step in the analysis is to construct an intraprocedural control flow graph for the service implementation to facilitate flow analysis at a later stage. The control flow graph is constructed using classical algorithms [10, ch.8.4] with graph nodes being statements. The construction process handles straightforward constructs such as conditionals and loops as well as the more complicated `try-catch-finally` construct in Java. For our prototype, described in section 7, we traverse the built-in abstract syntax tree representation of the Eclipse Java Development Tools (JDT) to build the control flow graph.

Constructing the control flow graph for the `try-catch-finally` construct requires some special considerations. The control flow must facilitate a flow analysis that can determine whether a variable is definitely assigned after a `try` statement. As the Java Language Specification [9, ch.16.2.15] shows, this is non-trivial matter since (1) `try` statements can be nested, (2) there can be several `catch` clauses, (3) the exception classes form an inheritance hierarchy that affects `catch` clause matching, and (4) one has to take any `finally` clauses into account.

The construction process proceeds in the following manner: For each statement that is in scope of one or more `try` statements, we add an edge to the next statement in the current `try` block, or if there is no such next statement, we add an edge to either the first statement in the `finally` block (if any) or else to the next statement after the `try-catch` statement. For each checked exception that the statement in question throws, we must add an edge to the corresponding `catch` clause. To do the latter, we iterate through every `catch` clause starting from the first in the innermost `try` statement to the last in the outermost `try` statement. If the innermost `try` statement does not have a relevant `catch` clause, we must add an edges to any intermediate `finally` clauses (in the case of nested `try` statements). In principle, almost every statement in the `try` block can throw an unchecked exception, but our analysis only takes checked exceptions into account. Taking unchecked exceptions into account would lead to too many edges on the graph and render the results of the final flow analysis less interesting because the final result would contain too many false positives.

5.3 Perform a Reaching Definitions Flow Analysis on the Code

The third step in our analysis is to perform a classic *reaching definitions flow analysis* [10, ch.9.2.4]. The purpose of this flow analysis is to determine which definitions reach each statement in the code. A definition is represented as a pair of a variable and its defining statement's location (its AST node). Specifically, for each statement we determine which variable definitions reach this point. This is done by solving the following equations where *entry(stmt)* is the set of definitions reaching a statement, *stmt*. The set *exit(stmt)* contains the definitions that may be exposed to the successors of *stmt* in the control flow graph. The gen_{stmt} and $kill_{stmt}$ sets are the definitions that are generated and killed by the statement *stmt*. The *init* node is the starting point of the graph [10, p.605-6].

$$exit(init) = \emptyset$$
$$exit(stmt) = gen_{\text{stmt}} \cup (entry(stmt) - kill_{\text{stmt}})$$
$$entry(stmt) = \cup_{\text{pred is a predecessor of stmt}} exit(pred)$$

The analysis is monotonic and the result is the least fixpoint of the equations. An important part of the equations is the use of union as the *meet* operator. The significance is that a definition reaches a statement if there is *some* path from a definition to the statement on the control flow that does not kill the definition.

5.4 Construct a Def-Use Chain for the Code

The control flow graph and the results of the reaching definitions analysis enable us to build *def-use* chains for every variable in the code. A def-use chain ties a definition of a variable together with the statements where the variable is being used. For the purposes of our analysis, we are specifically interested in the `defToUse` function which given a definition, `def`, returns the set of statements, `stmt`, where this definition is being used. The `use` predicate expresses whether a statement uses a variable and the `entry(stmt)` set is the previously computed set of definitions reaching *stmt* [11].

$$defToUse(def) = \{stmt | use(stmt, def) \wedge def \in entry(stmt)\}$$

5.5 For Each Use of the Input Map, Check That Only Declared Keys Are Used

The final step in our service input analysis is to check that only declared input attributes are actually being used. This is the consistency requirement on service input described in section 4.3. The analysis as implemented provides an approximation to this exact requirement, primarily because the control flow graph generates a superset of the set Σ of possible program traces. The analysis may deem a statement `context.get(x)` reachable although no actual computation could execute it.

Furthermore, the analysis checks whether the attribute is being cast to the correct type, such as when `surveyResponseId` is cast to the String type in line 5 in listing 1.2. The idiom `(C)context.get(x)` is used for casting input attributes so the analysis performs the type check simply by checking whether the declared type of `x` is assignable to type `C`.

A constraint violation is flagged as an error called *Use of undeclared input attribute*. In some cases, the key, `x`, is an expression or a variable whose value is computed by an expression at an earlier program statement. Then the analysis fails and the statement is annotated with a warning that the analysis is unable to determine whether this attribute is declared in the service definition. Using such computed keys can be considered metaprogramming on top of the OFBiz framework. An example of this practice is for instance the `updateOrRemove` service in the Content Management Module. This service can

change the structure of business entities at runtime. This practice is used in less than 10 places in our version of the framework so we consider it beyond the scope of our current analysis requirements.

6 Analysis of Service Output

In this section, we will describe the steps performed in the service output analysis. The analysis uses the service definition and the implementation code.

1. Collect declared attributes from the XML service definition.
2. Construct a control flow graph for the Java code.
3. Perform an *available map keys flow analysis* on the code.
4. For each return statement, check that each mandatory output attribute has definitely been assigned before that statement.
5. For each use of the output map, check that only declared keys are used.

6.1 Collect Declared Attributes from the XML Service Definition

The collection of output attributes is completely analogous to the collection of input attributes, described in section 5.1.

6.2 Construct a Control Flow Graph for the Java Code

The control flow graph from the previous section is reused.

6.3 Perform an Available Map Keys Flow Analysis on the Code

The third step in the analysis is to determine which keys are put into the map of output attributes during different traces of the program. The analysis is a flow analysis which shares some characteristics with the classical *available expressions flow analysis* [10, ch.9.2.6]. The purpose of the analysis is to compute the domain of each defined map for each statement in the implementation code. By domain, we here mean the set of keys that have definitely been assigned. The analysis results are represented as a pair of the AST node defining the map and a set of those map keys. Where the *available expressions flow analysis* determines whether an expression is definitely available at a given program point, our available map keys analysis determines whether a key in a given map is definitely available at a given program point. The main difference is that in our analysis map entries are treated as variables instead of merely runtime values. This difference is reflected in the `gen` and `kill` functions.

Analysis of output attributes is a bit more complicated than input attributes because an output attribute can be set in several different but equivalent ways. The most common approach is to instantiate a `HashMap` and invoke the `put(x, y)` method on that map to set the output attribute `x` to the value `y`. Another approach is to use the framework-provided method `UtilMisc.toMap` which takes a number of keys and values and returns a map containing those keys

and values as entries, i.e., a batch of `put` invocations. A third approach is to programmatically invoke another service, e.g., the `someService` service, which returns a map using the method `dctx.runSynch("someService",inputMap)`. Each of these different approaches generates a key at the statement where it occurs and plays a part in the gen_{stmt} function. Examples of the corresponding $kill_{stmt}$ function would be invocations of methods such as `clear()` or `remove(x)` on the map of output attributes.

The analysis is performed similarly to the available expressions analysis with the main difference being different `gen` and `kill` functions. Specifically, we solve the following set of equations [10, p.612]:

$$\begin{aligned} exit(init) &= \emptyset \\ exit(stmt) &= gen_{\text{stmt}} \cup (entry(stmt) - kill_{\text{stmt}}) \\ entry(stmt) &= \cap_{\text{pred is a predecessor of stmt}} exit(pred) \end{aligned}$$

An important part of the analysis is the use of intersection as the *meet* operator in these equations as opposed to union in the reaching definitions analysis in section 5.3. This reflects the duality of the consistency requirements on input and output discussed in section 4.6. The significance of using intersection here is that a map key only definitely reaches a `return` statement if every path leading to this `return` sets the map key. This means that if a map key is in the entry set of a `return` statement, the output attribute is definitely assigned at that statement. Another difference between the two analyses is that where the reaching definitions analysis starts by initializing every statement to have empty exit sets, the available map keys analysis initializes the exit set of every statement, except the starting node, to the universe of all possible keys. This is a direct consequence of using the intersection operator and in terms of execution of the analysis, it requires a pre-pass to compute the universe.

6.4 For Each Return Statement, Check That Each Mandatory Output Attribute has Definitely been Assigned Before That Statement

The fourth step in the analysis is to check that mandatory output attributes have definitely been assigned on return. This is the consistency requirement on service output described in section 4.4. A constraint violation is flagged as an error called *Missing mandatory output attribute, x.* This is because there is some path leading to this statement that does not set the key `x` in the returned map.

6.5 For Each Use of the Output Map, Check That Only Declared Keys Are Used

Finally, similar to the previous analysis in section 5.5 we check that only declared keys are used. Violations of this constraint do not cause the system to malfunction but indicate an attribute spelling error or other programmer error or misunderstanding. Setting an undeclared output attribute is somewhat similar to declaring a local variable in Java and never making use of it. It is not an error but a redundancy that indicates a possible problem in the code.

7 Prototype Implementation

Eclipse is a commonly used tool among OFBiz developers, so our implementation approach has aimed to extend Eclipse and make the previously described analyses available in that environment. Our prototype is an Eclipse plugin that provides an OFBiz model browser, as shown in figure 3, that allows developers to browse the logical structure of an OFBiz installation to manage entities and services rather than XML- and Java-files. From the browser, one can navigate to service definitions and service implementations in a single click. From the browser one can also initiate an analysis of either a single service or the entire installation.

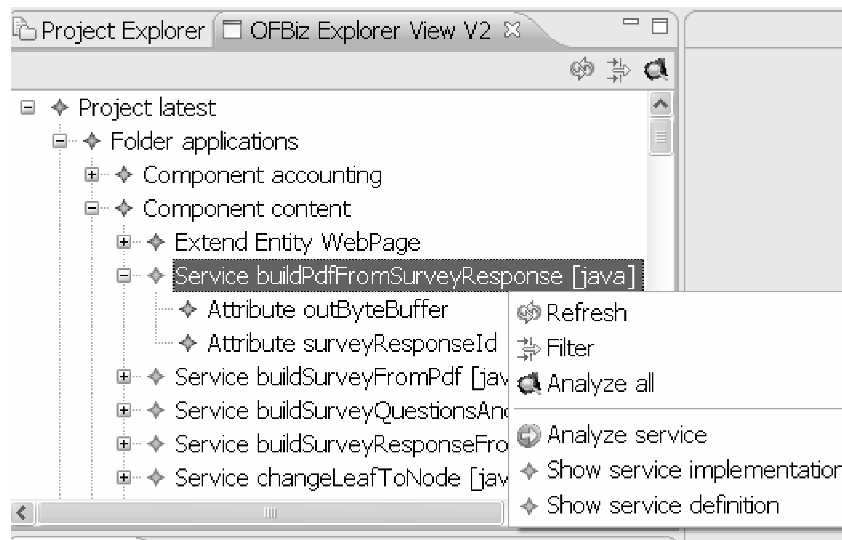

Fig. 3. Our prototype extends Eclipse with a browser for the logical model of an OFBiz installation. Here, the `buildPdfFromSurveyResponse` service in the Content Management component is selected and the two attributes of the service are shown. The context menu offers the choice of navigating to the service implementation or service definition.

The prototype uses a standard XML parser to parse and load all relevant XML files and relies on the Eclipse Java Development Tools (JDT) to parse and build abstract syntax trees for the corresponding Java code. The analyses are performed by traversing these XML- and Java-representations. The result of each analysis is reported as errors and warnings in the Problems View and marked in XML and Java editors as well, as shown in figure 4. The prototype is therefore integrated into the regular OFBiz development experience in a non-invasive way. Developers can quickly navigate from an error in the Problems View to the cause of the error in a Java editor, and repair it there.

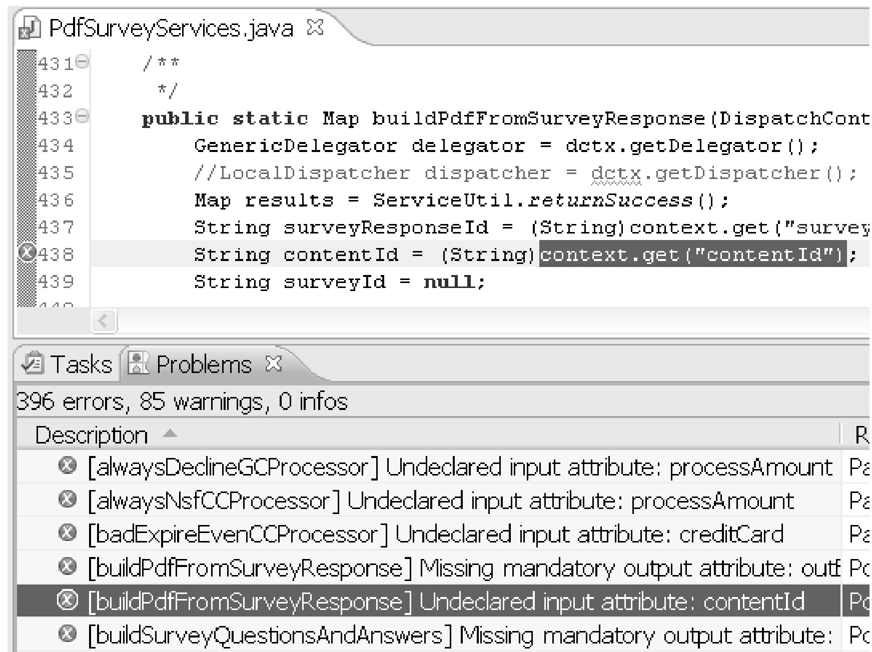

Fig. 4. Our prototype pointing out the inconsistency in listing 1.1 line 6. When an analysis is executed, the results are shown as entries in the Problems View at the bottom. When the user selects an error line in the Problems View, the Java editor is opened and the cursor is placed at the error's location in the source code.

We have released a preliminary version of the prototype to the OFBiz developer community and received very positive feedback [2, 3]. This indicates that the prototype addresses a real need and that the detected errors are considered serious. The prototype is available on the Internet along with an online Flash-demo of its capabilities [12]. In the next section, we will describe the results of applying this prototype to actual OFBiz production code.

8 Empirical Results

In order to validate our claims, we have applied our prototype on the out-of-the-box version of OFBiz (November 2007). This version contains some 2000 services of which 550 are implemented using Java code customizations. Our test setup is an Intel Core 2 CPU, 1.83 GHz, laptop with 2 GB of RAM. Running a complete analysis of the entire OFBiz installation (see section 3) took 22.3 seconds and used an average of 105 MB of heap space. In another setup with constrained memory, the complete analysis took 65.2 seconds but used only an average of 55 MB of heap space. This indicates that the prototype is fast enough to be used in a real industrial scenario.

Table 1. Overview of the 133 errors and 122 warnings detected in the out-of-the-box version of OFBiz. This version is already deployed in several industrial settings so these errors and warnings are present in *live* installations.

Severity	Type	No. of
Error	Undeclared input attribute	77
Error	Missing mandatory output attribute	56
Warning	May be missing mandatory output attribute	16
Warning	Unable to analyze expression	12
Warning	Unable to analyze complex returns	27
Warning	Unable to analyze interprocedural call	67

Our analysis found 133 errors and 122 warnings in our OFBiz installation. The tested OFBiz installation is a relatively stable version that is currently being used in several industrial settings. This means that the errors and warnings we have detected are present in several deployed OFBiz products. An overview of these errors and warnings can be found in table 1.

The two classes of errors in table 1 are the most serious problems. Use of an undeclared input attribute can potentially lead to `NullPointerException`s since reading an undeclared attribute from the map of input attributes typically returns `null`. An attempt to call a method on this attribute will therefore throw a `NullPointerException` and cause the OFBiz application to fail. A missing mandatory output attribute is quite simply a breach of the contract specified in the service definition. Clients of the service in question will expect this contract to be fulfilled. Closer examination of these errors has shown that it is often an exception handling control flow path that does not assign all mandatory output attributes. In some special cases, the analysis indicates that a mandatory output attribute *might* not be assigned. This happens when irregular use of certain framework-provided utility methods is detected. The last three classes of warnings are all caused by limitations in the analysis. If a variable is assigned the value of an expression and later used as key in the map of output attributes, it is typically part of some metaprogramming on top of the framework.

```
int j = computeFieldNo();
String key = "field" + j;
results.put(key, "someValue");
return results;
```

These parts of the code are beyond the scope of the analysis, and this is indicated by the *unable to analyze expression* warning.

If a `return` statement returns the value of an expression then the analysis issues a warning since we are unable to determine the value of that expression.

```
return isEmpty() ? new HashMap() : results;
```

Finally, the largest class of warnings are those caused by interprocedural calls that have not been incorporated into the analysis. The analysis has been adapted to handle most framework-provided utility methods. However, certain customizations introduce idiosyncratic utility methods that the analysis is unable to handle.

```
Map results = new HashMap();
otherObject.foo(results);
return results;
```

Where `otherObject.foo` is a custom method not provided by the framework and hence not included in the analysis.

9 Discussion

Several questions arise from our analysis and examination of the OFBiz framework. In this section, we will discuss four central ones. First, what are the limitations of our analysis? Second, can our approach be applied to other areas of OFBiz? Third, are the code customization described in this paper particular for OFBiz or do they appear in other frameworks as well? Fourth, is the OFBiz idiom of using maps to store input and output attributes really an internal domain-specific language, embedded in Java? If so, our input and attribute analyses are really standard compiler checks for consistency of this domain-specific language.

9.1 Limitations of the Analysis

Our analyses compute approximations of the constraints outlined in section 4. To discuss the sources of approximation, let us say that a "positive" is an actual consistency error, and a "negative" is the absence of such an error. A "true positive" is when an analysis discovers and reports an actual consistency error; a "false positive" is when an analysis reports a consistency error but there actually is none: the service will always execute without failure. Conversely, "true negative" is when an analysis reports no consistency error and there is none; a "false negative" is when an analysis reports no consistency error, but actually there is one: the service may fail.

Using an analogy from logic, we may say that an analysis without false positives is *complete* and that one without false negatives is *sound*. An analysis with neither false positives nor false negatives is exact.

For computability reasons, our analyses are necessarily incomplete and have false positives: they may report a consistency error where there is none. The main reason for this is that the control flow graph built in sections 5.2 and 6.2 is an approximation of the set Σ of actual program traces. Consider:

```
if (... complex expression, always false ...)
  context.get("thisAttributeIsNotDefined");
```

Moreover, our analyses are unsound and have false negatives: they may report no consistency error although the service may fail at runtime. Some people would consider such an analysis flawed and useless, but our point of view is that making the analysis sound (by eliminating all false negatives) would increase the number of false positives to an extent that would make the analysis uninteresting.

There are two sources of unsoundness:

(a) We perform no alias analysis [10, ch.12.4]
(b) We consider only checked exceptions when building the control flow graph for `try-catch-finally` statements

The lack of alias analysis affects the analysis of both input and output attributes. For input attributes, it means that we get a false negative in this case, where `thisAttributeIsNotDefined` is not an input attribute:

```
Map inputMapAlias = context;
inputMapAlias.get("thisAttributeIsNotDefined");
```

Our analysis will not discover that `inputMapAlias` is an alias of the input attribute may `context`, and hence will not flag the `get`-method call as a violation of the input attribute constraint.

For output attributes, the lack of alias analysis means that we get a false negative in this case:

```
Map results = new HashMap();
results.put("x", 1);
Map outputMapAlias = results;
outputMapAlias.remove("x");
return results;
```

Our analysis would not report that the output attribute `x` is missing from the `results` map.

It would be fairly easy to add an alias analysis step and hence remove this source of unsoundness, but we believe that it will only marginally affect the practical utility of the analysis, because there is no reason for service implementations to define aliases of input and output maps.

Considering only checked exceptions is the other main source of unsoundness. The construction of the control flow graph does not take unchecked (runtime) exceptions into account. Almost every Java statement may throw a runtime exception. Taking such exceptions into account would give significantly more control flow edges, which would cause a large number of false positives in the analysis without contributing any significant new useful error messages. We have therefore omitted runtime exceptions to get a simpler graph, fewer false positives, and an analysis that is overall more useful.

Finally, a further source of approximation in our analyses is that they are intraprocedural only. Hence if an input or output map is passed to a method the analysis will issue a warning, corresponding to "don't know" rather than give a positive or negative answer:

```
Map results = new HashMap();
doSomething(results);
return results;
```

Since the `doSomething` method may manipulate the entries in the `results` map, and the analysis is intraprocedural only, it cannot know the state of `results` after the method call.

The analysis *does* understand the effect of a small number of framework-provided utility methods, such as `UtilMisc.toMap`, that perform batch assignment of values to attributes. User-defined utility methods are, however, beyond the scope of our analysis.

9.2 Other Areas of Application in OFBiz

The OFBiz framework contains several other areas where our analysis may be applicable. Service implementations in Java account for only 25% of the service implementations in the entire framework. The remaining OFBiz service implementations are written in a domain-specific language called Minilang, see listing 1.3 for an example. Minilang is a simple, imperative, data manipulation language with a concrete syntax in XML. One can express operations on primitive types, Strings and OFBiz entities as well as conditionals and simple error handling.

```
1  <simple-method method-name="createWorkEffortCostCalc"
2  short-description="Create a WorkEffortCostCalc entry">
3      <make-value entity-name="WorkEffortCostCalc"
          value-name="newEntity"/>
4      <set-pk-fields map-name="parameters"
          value-name="newEntity"/>
5      <set-nonpk-fields map-name="parameters"
          value-name="newEntity"/>
6      <if-empty field-name="newEntity.fromDate">
7          <now-timestamp-to-env env-name="newEntity.fromDate"/>
8      </if-empty>
9      <create-value value-name="newEntity"/>
10 </simple-method>
```

Listing 1.3. The implementation of the `createWorkEffortCostCalc` service in the Minilang DSL. The code creates a `WorkEffortCostCalc` entity and initializes its fields from a map of input attributes called `parameters`.

Minilang is much less expressive than Java so it should be fairly easy to adapt our analyses to this language. This adaptation would involve two steps: (1) changing our parser from the Eclipse JDT parser to a standard XML parser, and (2) changing the `gen` and `kill` functions to include the syntactical constructs of Minilang. The reason that we focused on Java is that if the analysis can handle

Java with all its complexity then we claim that it is a trivial matter to extend the analysis to other implementation languages in OFBiz.

9.3 Code Customizations in Other Frameworks

An obvious question that arise from this work is whether these analyses are restricted to OFBiz. One of the main points of the paper is that exactly by adapting classic flow analyses to a framework-specific setting can we reap extra benefits. As it turns out, code customizations *are* found in other frameworks than OFBiz. Request processing in J2EE web applications provides good examples, such as servlet filters [13, ch.11] and Actions in Struts [14]. The Spring Framework [15] is an open source, J2EE-based webapplication framework with widespread industry adoption. Listing 1.4 shows an excerpt from one of the Spring sample applications. In this code we can identify the exact same patterns as in OFBiz. In line 3, an input attribute is accessed, and in lines 6 and 7 output attributes are assigned. In this case the metadata are split between Spring configuration files and the client-side HTML, and standard tools do not enforce consistency with the `handleRequest` code.

```
1 public ModelAndView handleRequest
2 (HttpServletRequest request, HttpServletResponse response)..{
3         String itemId = request.getParameter("itemId");
4         Item item = this.petStore.getItem(itemId);
5         Map model = new HashMap();
6         model.put("item", item);
7         model.put("product", item.getProduct());
8         return new ModelAndView("Item", model);
9 }
```

Listing 1.4. The `handleRequest` method from the `ViewItemController` class in the JPetStore sample application from the Spring Framework version 2.5. The code follows the same idiom as OFBiz code customizations by accepting a map, in this case the `request`, and returning another map, in this case the `model`.

Our prototype could be adapted to the Spring framework merely by changing the `gen` and `kill` functions in the available map keys analysis. The code in listing 1.4 uses `getParameter(x)` instead of OFBiz's `get(x)` to access an input attribute named `x`. Similarly, the output analysis would also only need a little tweaking. Further work, however, must be done in order to determine how easy it is to capture this kind of variability in the prototype.

9.4 An Internal Domain-Specific Language

It seems that the OFBiz style approach is common, especially in the context of Java web application frameworks. We do, however, want to further extend our

claim about this approach. The use of attributes stored in maps can be thought of as an internal domain-specific language (DSL) hosted in the Java language. Let us call this language the *Attribute DSL*. The Attribute DSL contains the following five language constructs:

Construct	**Meaning**
`java.util.Map.get(x)`	reads an attribute x
`java.util.Map.put(x, y)`	assigns value y to attribute x
`java.util.Map.remove(x)`	deletes an attribute x
`org.ofbiz.base.util.UtilMisc.toMap(...)`	batch assignment of values
`java.util.Map.clear()`	batch deletion of attributes

The Attribute DSL is weakly typed in contrast to Java. So by using this DSL within Java, we lose static guarantees. Reading an attribute is a good example of this loss; the existence of the `Customer` attribute is checked only at runtime, and the type of the attribute value is checked only at runtime. The introduction of generic types in Java 5.0 does not solve this problem since a single input map may contain attributes of type String, Customer and Integer. The most specific `java.util.Map` type that can hold these attribute is `Map<String,Object>`, which means that all attributes will have compile-time type Object.

```
Customer customer = (Customer) context.get("Customer");
```

Fortunately, OFBiz provides us with the metadata needed by our analysis to re-establish the previously lost static guarantees:

```
<attribute name="Customer" type="dk.itu.Customer" ../>
```

The existence of these metadata may explain why OFBiz developers feel confident leaving out the explicit runtime checks that one would expect in the above code.

The analyses that we have provided in the previous sections provide the static guarantees of definedness and type correctness that are missing from the Attribute DSL. The input attribute analysis is really a standard compiler check for variables being in scope when used, and the output attribute analysis is a standard compiler check for definite assignment.

10 Related Work

This work is influenced mainly by two areas of research: framework-completion and static semantic checking of weakly typed languages.

10.1 Framework-Completion

Framework-completion code is similar to code customizations in the sense that they conform to predefined interfaces and makes use of framework-provided methods. The code customizations that we have described are a bit more monolithic than framework-completion code which typically is split between multiple methods and relies heavily on framework callbacks.

Fairbanks et al. [16] have investigated framework-completion code in the Eclipse and the Java Applet framework and identified API complexity as a key problem in development. To help developers they propose *design fragments*, a kind of framework-completion recipes expressed in XML. These design fragments are similar to OFBiz service definitions in the sense that they contain extra metadata. The tool support for design fragments is of a more syntactic nature than our plugin which on the other hand relies more on semantics in the form of data flow. Other related work in this category is recent papers on framework-specific modelling languages that synthesize metadata and code in a higher-level language [17, 18].

10.2 Static Semantic Checking of Weakly Typed Languages

The idiom of storing attributes in maps has, as described in section 9, the unfortunate consequence that the type system is weakened. Our flow analyses are a way of providing static guarantees in spite of this weakened type system. There are several pieces of related work in area of weakly typed languages. Wright and Cartwright [19] have suggested the concept of *soft typing* based on work in Scheme. The main idea is to provide type information for the compiler by relying on a generalization of Hindley-Milner type inference. This work has been extended to other languages such as JavaScript [20] and Erlang [21]. Our work has a similar purpose but can be distinguished by the fact that we rely on two artifacts: metadata and code as opposed to merely code. Another related approach which relies heavily on data flow analysis is Christensen et al. [22]. Their focus is on dynamically generated string expressions such as SQL queries but the goal is similar to ours.

11 Conclusion

The use of XML and Java as a hybrid development platform for configurable enterprise systems leads to inconsistencies between metadata and code customizations. We have formalized the consistency constraints for a concrete platform, viz. OFBiz, and provided a set of dataflow analyses adapted to this platform. The analyses are implemented in the a tool that we have successfully applied to OFBiz. Using the tool, we have detected a large number of errors as well as elicited very positive feedback from the OFBiz community. Our overall contributions in this paper are:

- A formalization of the consistency constraints between metadata and code customizations in OFBiz.

– A set of framework-specific adaptions of dataflow analyses based on this formalization.
– A working implementation of these analyses in the form of an Eclipse plugin.
– An empirical validation of the tool by analyzing production code and eliciting feedback from OFBiz developers.
– A discussion of the limitations of the analyses, and the trade-off between soundness (no false negatives) and precision (only few false positives).

Acknowledgement

The authors would like to thank Kasper Østerbye, Andrzej Wąsowski and Steen Brahe for their valuable comments on an earlier draft of the paper.

References

1. The Apache Software Foundation: The Apache Open for Business Project (2007) March 8 (2007), http://ofbiz.apache.org/
2. Franck, D.: Personal correspondance (2008)
3. Chen, S.: Personal correspondance (2008)
4. Hessellund, A., Czarnecki, K., Wasowski, A.: Guided Development with Multiple Domain-Specific Languages. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 46–60. Springer, Heidelberg (2007)
5. BeanShell: Lightweight Scripting for Java (2007), http://www.beanshell.org/
6. Jones, D., Schuessler, E.: Apache OFBiz: Real-World Open Source Java Platform ERP. Presented at the 2007 JavaOne Conference (2007)
7. Chen, S.: Opening Up Enterprise Software: Why Enterprises are Adopting Open Source Applications (2006), http://www.opensourcestrategies.com/slides/
8. Various: Apache OFBiz User List (2007), http://docs.ofbiz.org/display/OFBIZ/Apache+OFBiz+User+List
9. Gosling, J., Joy, B., Steele, G.L., Bracha, G.: The Java Language Specification, 3rd edn. Sun Microsystems (2005)
10. Aho, A.W., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, & Tools. 2 edn. Pearson Education, Inc (2007)
11. Fuhrer, R.M.: Static Analysis for Java in Eclipse, Lecture slides from the tutorial Static Analysis for Java in Eclipse at the ACM SIGPLAN 2005 Programming Language Design and Implementation (PLDI), Chicago, Illinois, USA. (2005), http://www.cs.purdue.edu/homes/jv/plditut/eclipse/index.html
12. Hessellund, A.: OFBiz Explorer v2 (2007), http://www.itu.dk/people/hessellund/smartemf/ofbizexplorer.htm
13. Bodoff, S., Armstrong, E., Ball, J., Carson, D., Evans, I., Green, D., Haase, K., Jendrock, E.: J2EE Tutorial, 2nd edn. Prentice-Hall, Englewood Cliffs (2004)
14. The Apache Software Foundation: The Struts Framework (2007), http://struts.apache.org/
15. Spring Source: The Spring Framework (2007), http://www.springframework.org/
16. Fairbanks, G., Garlan, D., Scherlis, W.L.: Design fragments make using frameworks easier. In: Tarr, P.L., Cook, W.R. (eds.) OOPSLA, pp. 75–88. ACM, New York (2006)

17. Antkiewicz, M., Czarnecki, K.: Framework-specific modeling languages with round-trip engineering. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 692–706. Springer, Heidelberg (2006)
18. Antkiewicz, M., Bartolomei, T.T., Czarnecki, K.: Automatic extraction of framework-specific models from framework-based application code. In: ASE 2007: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pp. 214–223. ACM, New York (2007)
19. Wright, A.K., Cartwright, R.: A practical soft type system for scheme. In: LISP and Functional Programming, 250–262 (1994)
20. Thiemann, P.: Towards a type system for analyzing javascript programs. In: Sagiv, S. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 408–422. Springer, Heidelberg (2005)
21. Marlow, S., Wadler, P.: A practical subtyping system for erlang. In: ICFP 1997: Proceedings of the second ACM SIGPLAN international conference on Functional programming, pp. 136–149. ACM, New York (1997)
22. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise Analysis of String Expressions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 1–18. Springer, Heidelberg (2003)