Domain-Specific Program Checking*

Lukas Renggli[1], Stéphane Ducasse[2], Tudor Gîrba[3], Oscar Nierstrasz[1]

[1] Software Composition Group, University of Bern, Switzerland
scg.unibe.ch

[2] RMoD, INRIA-Lille Nord Europe, France
rmod.lille.inria.fr

[3] Sw-eng. Software Engineering GmbH, Switzerland
www.sw-eng.ch

Abstract. Lint-like program checkers are popular tools that ensure code quality by verifying compliance with best practices for a particular programming language. The proliferation of internal domain-specific languages and models, however, poses new challenges for such tools. Traditional program checkers produce many false positives and fail to accurately check constraints, best practices, common errors, possible optimizations and portability issues *particular to domain-specific languages*. We advocate the use of dedicated rules to check domain-specific practices. We demonstrate the implementation of domain-specific rules, the automatic repair of violations, and their application to two case-studies: (1) Seaside defines several internal DSLs through a creative use of the syntax of the host language; and (2) Magritte adds meta-descriptions to existing code by means of special methods. Our empirical validation demonstrates that domain-specific program checking significantly improves code quality when compared with general purpose program checking.

1 Introduction

The use of automatic program checkers to statically locate possible bugs and other problems in source code has a long history. While the first program checkers were part of the compiler, later on separate tools were written that performed more sophisticated analyses of code to detect possible problem patterns [Joh78]. The refactoring book [Fow99] made code smell detection popular, as an indicator to decide when and what to refactor.

Most modern development environments (IDEs) directly provide lint-like tools as part of their editors to warn developers about emerging problems in their source code. These checkers usually highlight offending code snippets on-the-fly and greatly enhance the quality of the written code. Contrary to a separate tool, IDEs with integrated program checkers encourage developers to write good code right from the beginning. Today's program checkers [HP04] reliably detect issues like possible bugs, portability issues, violations of coding conventions, duplicated, dead, or suboptimal code, *etc.*

* In Proceedings of the 48th International Conference on Objects, Models, Components and Patterns (TOOLS'10), LNCS 6141, pp. 213–232, Springer-Verlag, 2010.

Many software projects today use *domain-specific languages* (DSLs) to raise the expressiveness of the host language in a particular problem domain. A common approach is to derive a new pseudo-language from an existing API. This technique is known as a *Fluent Interface*, a form of an *internal domain-specific language* or *embedded language* [Fow08]. Such languages are syntactically compatible with the host language, and use the same compiler and the same runtime infrastructure.

As such DSLs often make creative use of host language features with atypical use of its syntax. This confuses traditional program checkers and results in many false positives. For example, chains of method invocations are normally considered bad practice as they expose internal implementation details and violate the Law of Demeter [Lie89]. However in internal DSLs, method chaining is a commonly applied technique to invoke a sequence of calls on the same object where each call returns the receiver object for further calls. In other words, the DSL *abstracts* from the traditional use of the host language and introduces new idioms that are meaningful in the particular problem domain.

Traditional program checkers work at the level of source code. Tools like *intensional views* [MKPW06] and *reflexion models* [MNS95,KS03] check for structural irregularities and for conformance at an architectural level. Furthermore tools like *PathFinder* [HP00] have been used to transform source code into a model and apply model checking algorithms.

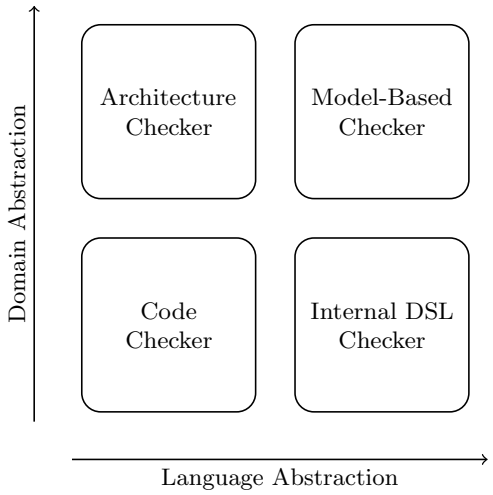

Fig. 1. Dimensions of program checking.

Figure 1 depicts the dimensions of program checking. Traditional program checkers tackle the axis of *domain abstraction* at different levels. We argue that a different set of rules is necessary as developers *abstract from the host language.* Our thesis is that standard program checking tools are not effective when it comes to detecting problems in domain-specific code. In this paper we advocate the use of dedicated program checking rules for program that know about and

check for the specific use-cases of internal domain-specific languages. As with traditional rules this can happen at the level of the source code or at a higher architectural or modeling level.

We will demonstrate two different rule-sets that each work at a different level of domain abstraction:

1. Seaside is an open-source web application framework written in Smalltalk [DLR07]. Seaside defines various internal DSLs to configure application settings, nest components, define the flow of pages, and generate XHTML. As part of his work as Seaside maintainer and as software consultants on various industrial Seaside projects, the first author developed *Slime*, a Seaside-specific program checker consisting of a set of 30 rules working at the level of the abstract syntax tree (AST). We analyze the impact of these rules on a long term evolution of Seaside itself and of applications built on top of it.

2. Magritte is a recursive metamodel integrated into the reflective metamodel of Smalltalk [RDK07]. The metamodel of an application is specified by implementing annotated methods that are automatically called by Magritte to build a representative metamodel of the system. This metamodel is then used to automate various tasks such as editor construction, data validation, and persistency. As the metamodel is part of the application source code it cannot be automatically verified. We have implemented a set of 5 rules that validate such a Magritte metamodel against its meta-metamodel.

Our approach to program checking is based on AST pattern matching. This technical aspect is not new. However, our approach builds on that and offers a way to specify declaratively domain specific rules with possible automatic transformations. Our approach uses pattern matching on the AST as supported by the refactoring engine of Smalltalk [BFJR98]. Furthermore we use Helvetia [RGN10], a framework to cleanly extend development tools of the standard Smalltalk IDE. It reuses the existing toolchain of editor, parser, compiler and debugger by leveraging the AST of the host environment. While Helvetia is applicable in a much broader context to implement and transparently embed new languages into a host language, in this paper we focus on the program analysis and transformation part of it.

The contributions of this paper are: (1) the identification of the need for DSL specific rule checking, (2) the empirical validation over a long period that such DSL specific rules offer advantages over non domain-specific ones, and (3) an infrastructure to declaratively specify domain specific rules and the optional automatic transformations of violations.

The paper is structured as follows: Section 2 introduces the different rule-sets we have implemented. We present the internal domain-specific languages addressed by our rules, and we discuss how we implemented and integrated the rules. In Section 3 we report on our experience of applying these rules on various open-source and commercial systems. Furthermore we present a user survey where we asked developers to compare domain-specific rules with traditional ones. Section 4 discusses related work and Section 5 concludes.

2 Examples of Domain-Specific Rules

In this section we demonstrate two sets of rules at different levels of abstraction: while the first set of rules (Section 2.1) works directly on the source code of web applications, the second set of rules (Section 2.2) uses a metamodel and validates it against the system. While in both cases the source code is normal Smalltalk, we focus on the domain-specific use of the language in these two contexts.

2.1 Syntactic rules for Seaside

The most prominent use of an internal DSL in Seaside is the generation of HTML. This DSL is built around a stream-like object that understands messages to create different XHTML tags. Furthermore the tag objects understand messages to add the HTML attributes to the generated markup. These attributes are specified using a chain of message sends, known in the Smalltalk jargon as a cascade[4]:

```
1 html div
2    class: 'large';
3    with: count.
4 html anchor
5    callback: [ count := count + 1 ];
6    with: 'increment'.
```

The above code creates the following HTML markup:

```
<div class="large">0</div>
<a src="/?_s=28hVYPUhdMM7mU&1">increment</a>
```

Lines 1–3 are responsible for the generation of the `div` tag with the CSS class `large` and the value of the current `count` as the contents of the tag. Lines 4–6 generate the link with the label *increment*. The `src` attribute is provided by Seaside. Clicking the link automatically evaluates the code on line 5 and redisplays the component.

This *little language* [DK97] for HTML generation is the most prominent use of a DSL in Seaside. It lets developers abstract common HTML patterns into convenient methods rather than pasting the same sequence of tags into templates every time.

As developers and users of Seaside we have observed that while the HTML generation is simple, there are a few common problems that repeatedly appear in the source code of contributors. We have collected these problems and categorized

[4] Readers unfamiliar with the syntax of Smalltalk might want to read the code examples aloud and interpret them as normal sentences: An invocation to a method named `method:with:`, using two arguments looks like: `receiver method: arg1 with: arg2`. The semicolon separates cascaded messages that are sent to the same receiver. For example, `receiver method1: arg1; method2: arg2` sends the messages `method1:` and `method2:` to `receiver`. Other syntactic elements of Smalltalk are: the dot to separate statements: `statement1. statement2`; square brackets to denote code blocks or anonymous functions: `[ statements ]`; and single quotes to delimit strings: `'a string'`. The caret `^` returns the result of the following expression.

them into 4 groups: possible bugs, non-portable code between different Smalltalk platforms/versions, bad style, and suboptimal code. Spotting such problems early in the development cycle can significantly improve the code quality, maintainability and might avoid hard to detect bugs. We offer details for a case from each group.

Possible Bugs. This group of rules detects severe problems that are most certainly serious bugs in the source code:

- The message `with:` is not last in the cascade,
- Instantiates new component while generating HTML,
- Manually invokes `renderContentOn:`,
- Uses the wrong output stream,
- Misses call to super implementation,
- Calls functionality not available while generating output, and
- Calls functionality not available within a framework callback.

As an example of such a rule we take a closer look at "The message `with:` is not last in the cascade". While in most cases it does not matter in which order the attributes of a HTML tag are specified, Seaside requires the contents of a tag be specified last. This allows Seaside to directly stream the tags to the socket, without having to build an intermediate tree of DOM nodes. In the erroneous code below the order is mixed up:

```
html div
   with: count;
   class: 'large'.
```

One might argue that the design of the DSL could avoid this ordering problem in the first place. However, in the case of Seaside we reuse the existing syntax of the host language and we cannot change and add additional validation into the compiler, otherwise this would not be an internal DSL anymore.

Slime uses a declarative internal DSL to specify its rules. Every rule is implemented as a method in the class `SlimeRuleDatabase`. Helvetia automatically collects the result of evaluating these methods to assemble a set of Slime rules. The following code snippet demonstrates the complete code necessary to implement the rule to check whether `with:` is the last message in the cascade:

```
1 SlimeRuleDatabase>>withHasToBeLastInCascade
2    ^ SlimeRule new
3       label: 'The message with: has to be last in the cascade';
4       search: (ConditionRule new
5          if: [ :context | context isHtmlGeneratingMethod ];
6          then: (TreeRule new
7             search: '`html `message with: ``@arguments';
8             condition: [ :node |
9                node parent isCascade and: [ node isLastMessage not ] ]));
```

Line 2 instantiates the rule object, line 3 assigns a label that appears in the user interface and lines 4–9 define the actual search pattern.

The precondition on line 5 asserts statically that the code artifact under test is used by Seaside to generate HTML. The `ConditionRule` object lets developers scope rules to relevant parts of the software using the reflective API of the host language. This precondition reduces the number of false positives and greatly improves the performance of the rule.

Line 6 instantiates a `TreeRule` that performs a search on the AST for occurrences of statements that follow the pattern `` `html `message with: ``@arguments ``. Search patterns are specified using a string with normal Smalltalk expressions annotated with additional meta-characters. The back-tick `` ` `` marks meta-nodes that are not required to match literally but that are variable. Table 1 gives a description of the characters following the initial back-tick.

Char	Type	Description
`#`	literal	Match a literal node like a number, boolean, string, etc.
`.`	statement	Match a statement in a sequence node.
`@`	list	When applied to a variable, match any expression. When applied to a statement, match a list of statements. When applied to a message, match a list of arguments.
`` ` ``	recurse	When a match is found recurse into the matched node.

Table 1. Meta-characters for parse-tree pattern matching.

In our example it does not matter how the variable `` `html ``, the message `` `message: `` and the arguments `` ``@arguments `` are exactly named. Furthermore, `` ``@arguments `` is an arbitrary expression that is recursively searched. If a match is found, the AST node is passed into the closure on lines 8 and 9 to verify that the matched node is not the last one of the cascade.

When the Slime rules are evaluated by Helvetia the matching AST nodes are automatically collected. Interested tools can query for these matches and reflect on their type and location in the code. The "Code Browser" depicted in Figure 2 highlights occurrences while editing code. Reporting tools can count, group and sort the issues according to severity. A more detailed description of the Helvetia rule engine can be found in our related work [RGN10].

Many of the detected problems can be automatically fixed. Providing an automatic refactoring for the above rule is a matter of adding a transformation specification:

```
10      replace: [ :node |
11         node cascade
12            remove: node;
13            addLast: node ].
```

Lines 12 and 13 remove the matched node from the cascade and add it back to the end of the sequence. After applying the transformation Helvetia automatically re-runs the search, to ensure that the transformation actually resolves the problem.

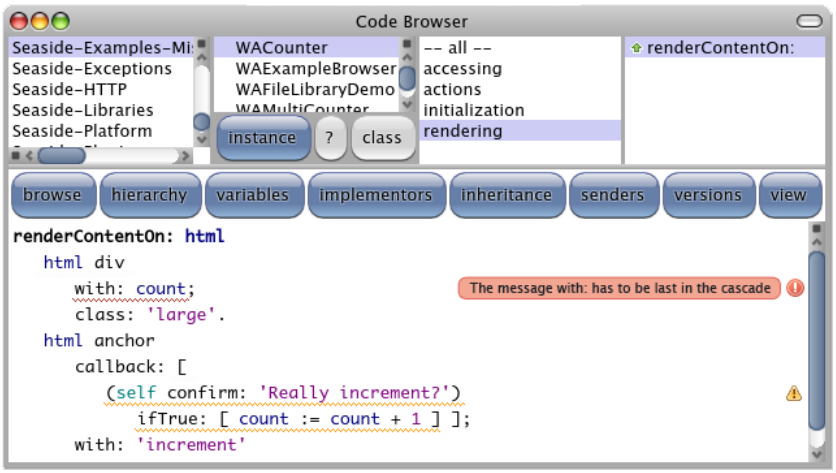

Fig. 2. Integration of domain-specific rules into the "Code Browser".

Again the tools from the IDE automatically offer the possibility to trigger such an automatic transformation. For example, when a developer right-clicks on a Slime issue in the "Code Browser" a confirmation dialog with a preview is presented before the transformation is applied. Furthermore it is possible to ignore and mark false positives, so that they do not show up again.

Bad style. These rules detect some less severe problems that might pose maintainability problems in the future but that do not cause immediate bugs. An example of such a rule is "Extract callback code to separate method". In the example below the rule proposes to extract the code within the callback into a separate method. This ensures that code related to controller functionality is kept separate from the view.

```
html anchor
   callback: [
      (self confirm: 'Really increment?')
         ifTrue: [ count := count + 1 ] ];
   with: 'increment'.
```

Other rules in this category include:

- Use of deprecated API, and
- Non-standard object initialization.

The implementation of these rules is similar to the one demonstrated in the previous section on "possible bugs".

Suboptimal Code. This set of rules suggests optimizations that can be applied to code without changing its behavior. For example, the following code triggers the rule "Unnecessary block passed to brush":

```
html div with: [ html text: count ]
```

The code could be rewritten as follows, but this triggers the rule "Unnecessary #with: sent to brush":

```
html div with: count
```

This in turn can be rewritten to the following code which is equivalent to the first version, but much shorter and more efficient as no block closure is activated:

```
html div: count
```

Non-Portable Code. While this set of rules is less important for application code, it is essential for the Seaside code base itself. The framework runs without modification on 7 different platforms (Pharo Smalltalk, Squeak Smalltalk, Cincom Smalltalk, GemStone Smalltalk, VA Smalltalk, GNU Smalltalk and Dolphin Smalltalk), which slightly differ in both the syntax and the libraries they support. To avoid that contributors using a specific platform accidentally submit code that only works on their platform we have added a number of rules that check for compatibility:

- Invalid object initialization,
- Uses curly brace arrays,
- Uses literal byte arrays,
- Uses method annotations,
- Uses non-portable class,
- Uses non-portable message,
- ANSI booleans,
- ANSI collections,
- ANSI conditionals,
- ANSI convertor,
- ANSI exceptions, and
- ANSI streams.

Code like `count asString` might not run on all platforms identically, as the convertor method `asString` is not part of the common protocol. Thus, if the code is run on a platform that does not implement `asString` the code might break or produce unexpected results.

The implementation and the automatic refactoring for this issue is particularly simple:

```
1 SlimeRuleDatabase>>nonPortableMessage
2   ^ SlimeRule new
3      label: 'Uses non-portable message';
4      search: '``@obj asString' replace: '``@obj seasideString';
5      search: '``@obj asInteger' replace: '``@obj seasideInteger'
```

Again the rule is defined in the class `SlimeRuleDatabase`. It consists of two matching patterns (line 4 and 5 respectively) and their associated transformation, so code like `count asString` will be transformed to `count seasideString`.

2.2 Magritte — code checking with a metamodel

Constraint checking is not a new domain. Classic approaches rely on constraints that are specified by the analyst [MKPW06,MNS95,KS03] and that are checked against the actual application code. In this case these rules are external to the execution of the program. Model-driven designs often rely on a metamodel to add more semantics to the code by providing transformations that are either statically (via code generation) or dynamically interpreted. These metamodels come with a set of constraints that can also be used for checking the program.

Magritte is a metamodel that is used to automate various tasks such as editor building, data validation and persistency [RDK07]. In this section we detail its use and the rules that can be derived from the constraints it imposes.

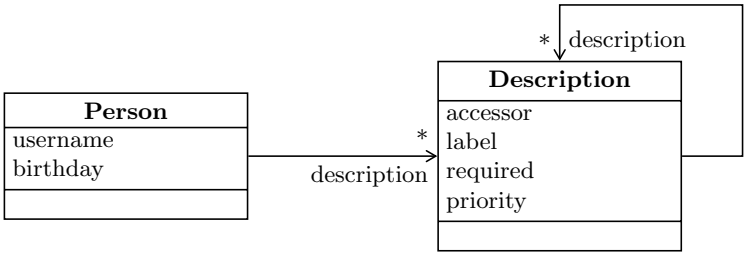

Fig. 3. The domain object `Person` with its Magritte meta-description.

On the left side of Figure 3 we see a simple domain class called `Person` with two attributes. To meta-describe a class with Magritte we need corresponding description instances. These description instances are either defined in the source-code or dynamically at run-time. The following code shows an example of how we could describe the attribute `username` in the class `Person`:

```
1 Person class>>usernameDescription
2    <description>
3    ^ StringDescription new
4       accessor: #username;
5       label: 'Username';
6       beRequired;
7       yourself
```

The method returns an attribute description of the type string (line 3), that can be accessed through the method `#username` (line 4), that has the label `'Username'` (line 5), and that is a required property (line 6). The annotation (line 2) lets Magritte know that calling the method returns a description of the receiver. Several such description methods build the metamodel of the `Person` class as visualized with the association from `Person` to `Description` in Figure 3.

Descriptions are interpreted by different services, such as form builders or persistency mappers. For example, a simple renderer that prints the label and the current values would look like this:

```
1 aPerson description do: [ :desc |
2    aStream
3       nextPutAll: (desc label);
4       nextPutAll: ': ';
5       nextPutAll: (desc toString: (desc accessor readFrom: aPerson));
6       cr ]
```

First, given an `aPerson` instance, we ask it for its description and we iterate over its individual attribute descriptions (line 1). Within the loop, we print the label (line 3), we ask the accessor of the description to return the associated attributes from `aPerson` and we transform this value to a string (line 5), so that it can be appended to the output.

We have defined five rules that check for conformance of the source code with the Magritte metamodel. The first two are defined and implemented externally to the Magritte engine:

1. Description Naming. The definitions of the attribute descriptions should relate to the accessor they describe. In our example the accessor is `username` and the method that defines the description is called `usernameDescription`. While this is not a strict requirement, it is considered good style and makes the code easier to read. The implementation points out places where this practice is neglected.

2. Missing Description. Sometimes developers fail to completely describe their classes. This rule checks all described classes of the system and compares them with the metamodel. Instance variables and accessor methods that miss a corresponding description method are reported.

The remaining three rules rely completely on the constraints already imposed by the runtime of Magritte:

3. Description Priorities. In Magritte attribute descriptions can have priorities. This is useful to have a deterministic order when elements are displayed in a user interface. This rule verifies that if a description is used to build user-interfaces then it should have valid priorities assigned to all its attribute descriptions. This rule makes use of the metamodel as well as the reflective system to detect the places where the descriptions are used.

4. Accessor Definition. The Magritte metamodel uses accessor objects to specify how the data in the model can be read and written. This rule iterates over the complete metamodel and checks the accessor object of every description against the code it is supposed to work on. The implementation of the rule is straight forward as it merely delegates to `aDescription` instance the `aClass` under scrutiny:

```
aDescription accessor canReadFromInstancesOf: aClass.
aDescription accessor canWriteToInstancesOf: aClass.
```

5. Description Definition. This rule checks if the specified metamodel can be properly instantiated and, if so, it validates the metamodel against its meta-metamodel. Magritte allows one to check any model against its metamodel, so we can validate `aPerson` against its metamodel:

```
aPerson description validate: aPerson
```

Magritte is described in itself, as depicted in Figure 3. Therefore we can use the meta-metamodel to validate the metamodel in the same way:

```
aDescription description validate: aDescription
```

The above code validates `aDescription` against the description of itself. In case of problems they are recorded by the program checker. In fact this rule is the most powerful of all rules presented here, because it can detect various kinds of different problems in the metamodel, yet it is extremely simple in the implementation as all the functionality is already present in Magritte.

We have developed a similar set of rules for FAME [KV08], a metamodeling library that is independent of the host language and that keeps the metamodels accessible and adaptable at runtime.

3 Case Studies

In this section we present three case studies: In the first two we apply Slime rules to control the code quality. The first one is Seaside itself (Section 3.1). The second one is a commercial application based on Seaside (Section 3.2). We analyze several versions of these systems and we compare the results with the number of issues detected by traditional lint rules. Then we present a survey we ran with Seaside developers concerning their experience with using Slime (Section 3.3). In the third case study we apply the Magritte rules on a large collection of open-source code (Section 3.4) and demonstrate some common issues that remained unnoticed in the code.

3.1 Seaside

Figure 4 depicts the average number of issues over various versions of Seaside. The blue line shows the number of standard smells per class (Lint), while the orange line shows the number of domain-specific smells per class (Slime). To give a feeling how the size of the code base changes in time, we also display the number of lines of code (LOC) below.

In both cases we observe a significant improvement in code quality between versions 2.7 and 2.8. At the time major parts of Seaside were refactored or rewritten to increase portability and extensibility of the code base. No changes are visible for the various 2.8 releases. Code quality as measured by the program checkers and lines of code remained constant over time.

Starting with Seaside 2.9a1 Slime was introduced in the development process. While the quality as measured by the traditional lint rules remained constant,

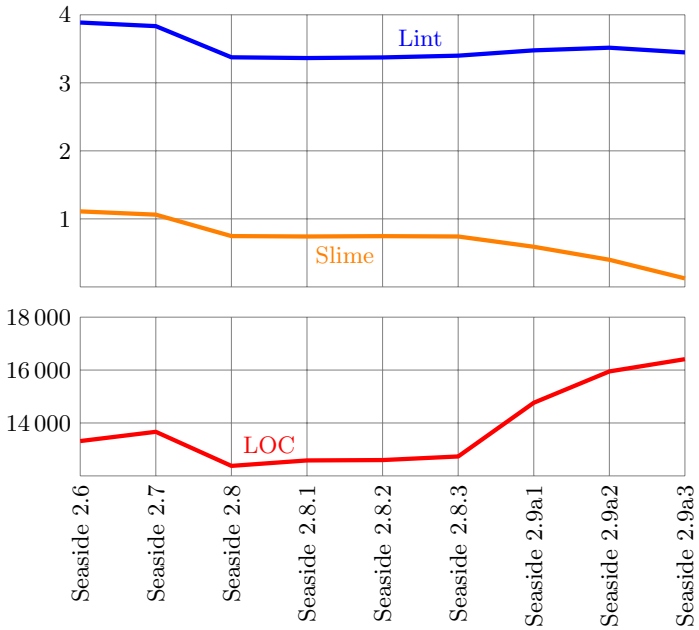

Fig. 4. Average number of Lint and Slime issues per class (above) and lines of code (below) in released Seaside versions.

guiding development by the Slime rules significantly improved the quality of the domain-specific code. This particular period shows the value in domain-specific program checking. While the Seaside code base grew significantly, the number of Slime rules could be reduced to almost zero.

Feedback we got from early adopters of Seaside 2.9 confirms that the quality of the code is notably better. Especially the portability between different Smalltalk dialects has improved. The code typically compiles and passes the tests on all platforms even-though it comes from the shared code repository.

An interesting observation is that even if the Slime smells are reduced and the quality of the code improves, the standard Lint rules continue to report a rather constant proportion of problems. This is due to the fact that the generic Lint rules address the wrong level and produce too many false positives.

We further evaluated the number of *false positives* of the remaining open issues in the last analyzed version of Seaside by manually verifying the reported issues: this is 67% (940 false positives out of 1403 issues reported) in the case of Lint, and 24% (12 false positives out of 51 issues reported) in the case of Slime. This demonstrates, that applying dedicated rules provides a better report on the quality of the software than when using the generic rules.

Due to the dynamic nature of Smalltalk and its lack of static type information it seems to be hard to further improve the quality of Slime rules. We however do see potential in future work to reduce the number of false positives by using static [PMW09] and dynamic [DGN07] type analysis.

3.2 Cmsbox

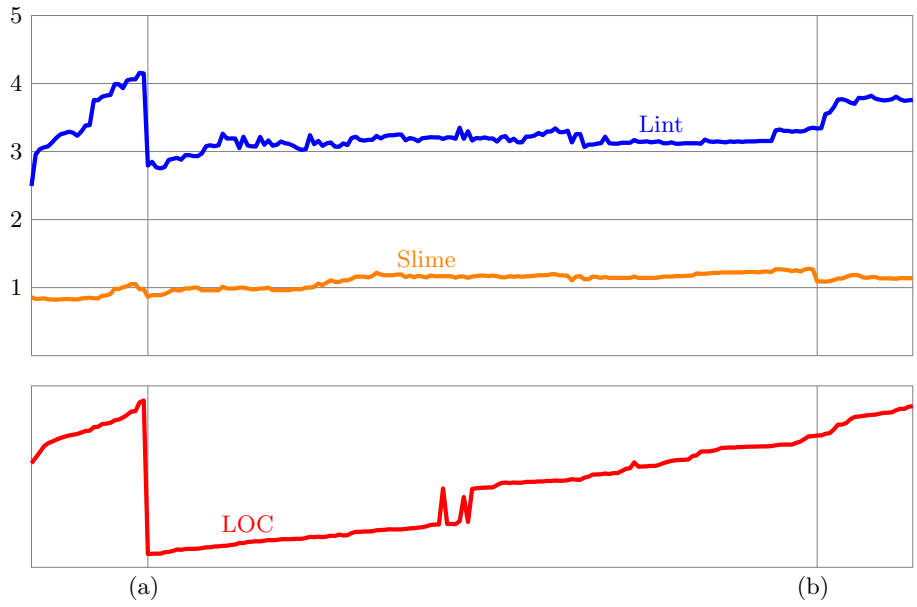

Fig. 5. Average number of Lint and Slime issues per class (above) and lines of code (below) in 220 subsequent development versions of the Cmsbox.

The Cmsbox[5] is a commercial web content management system written in Seaside. Figure 5 depicts the development of the system over three years. We are external to the development. The company gave us access to their code, but we could not correlate with their internal quality model and bug reports. Still we could deduce some interesting points: We ran the same set of Lint and Slime tests on every fifth version committed, for a total of 220 distinct versions analyzed. The number of lines of code are displayed below, though the absolute numbers have been removed to anonymize the data.

In the beginning we observe a rapid increase of detected issues. This is during the initial development phase of the project where a lot of code was added in a relatively short time. Presumably the violation of standard rules was not a concern for the developers. By contrast the number of Slime issues remained low and showed only gradual increase by comparison. This is a really interesting difference. Since the Slime rules tackle the development of the web interface which was the key part of the development effort, the result shows the benefit of using domain-specific code checking: developers focus more on domain-specific issues, rather than the general issues that can typically be resolved much more easily.

[5] `http://www.cmsbox.com/`

The abrupt drop of lint (and to some smaller extent also Slime) issues at point (a) can be explained by the removal of a big chunk of experimental or prototypical code no longer in use. Between versions (a) and (b) the code size grew more slowly, and the code quality remained relatively stable. It is worth noting that the size of the code base grew gradually, but at the same time the proportion of Slime issues stayed constant.

During the complete development of the Cmsbox the standard Lint rules were run as part of the daily builds. This explains why the average number of issues per class is lower than in the case of Seaside. At point (b) Slime rules were added and run with every build process. This accounts for the drop of Slime issues. A new development effort after (b) caused an increasing number of Lint issues. Again it is interesting to see that the better targeted Slime rules remained stable compared to the traditional ones.

Contrary to the case study with Seaside, the Slime issues do not disappear completely. On the one hand this has to do with the fact that the software is not supposed to run on different platforms, thus the rules that check for conformity on that level were not considered by the development team. On the other hand, as this is typical in an industrial setup, the developers were not able to spend a significant amount of time on the issues that were harder to fix and that did not cause immediate problems.

3.3 User Survey

We performed a user study where we asked Seaside developers to complete a survey on their Lint and Slime usage. 23 experienced Seaside developers independent from us answered our questionnaire. First, we asked them to state their use of program checkers:

1. How often do you use Slime on your Seaside code? *4 daily, 4 weekly, 8 monthly, and 7 never.*
2. How often do you use standard code critics on your Seaside code? *3 daily, 5 weekly, 7 monthly, and 8 never.*

On all answers, 16 developers (70%) are using Slime on a regular basis. We asked these developers to give their level of agreement or disagreement on the five-point Likert scale to the following statements:

3. Slime helps me to write better Seaside code: *11 agree, and 5 strongly agree.*
4. Slime is more useful than standard code critics to find problems in Seaside code: *5 neither agree nor disagree, 8 agree, and 3 strongly agree.*
5. Slime does not produce useful results, it mostly points out code that I don't consider bad: *3 strongly disagree, 10 disagree, and 3 neither agree nor disagree.*

To summarize, all developers that use Slime on a regular basis found it useful. 69% of the developers stated that Slime produces more useful results than the standard program checkers, the other 31% could not see a difference. 81% of the developers stated that Slime produces relevant results that help them to detect critical problems in their application.

We see our thesis confirmed in the two case studies and the user survey: While the general purpose Lint rules are definitely useful to be applied to any code base, they are not effective enough when used on domain-specific code. Using dedicated rules decreases the number of false positives and gives more relevant information on how to avoid bugs and improve the source code.

3.4 Magritte

In our third case study we ran the Magritte rules on a large collection of open-source code. This includes Pier[6], an application and content management system; SqueakSource, a source code management system; Conrad, a conference management system; CiteZen, a bibliography toolkit; CouchDB, a database implementation, and a large number of smaller projects that are publicly available.

In total we analyzed 70 768 lines of code in 12 305 methods belonging to 1 198 classes. 307 of these classes had Magritte meta-descriptions attached, where we found a total number of 516 Magritte related issues as listed in Table 2.

Magritte Rule	**Issues**
Description Naming	37
Description Definition	78
Description Priorities	113
Accessor Definition	120
Missing Description	168

Table 2. Number of issues in meta-described open-source code.

The most commonly observed problem is *missing descriptions*. While this is not necessarily a bug, it shows that some authors did not completely describe their domain objects. That can either happen intentionally, because they wanted to avoid the use of Magritte in certain parts of their application, or it can happen unintentionally when they forgot to update the metamodel as they added new functionality. This rule is thus helpful when reviewing code, as it identifies code that is not properly integrated with the meta-framework.

We observed also a significant number of errors in the *description definitions*. This happens when the defined metamodel does not validate against the meta-metamodel, which can be considered a serious bug. For example, we found the following description with two problems in the Pier Blog plugin:

[6] `http://www.piercms.com/`

```
1 Blog>>descriptionItemCount
2    ^ IntegerDescription new
3       label: 'Item Count';
4       accessor: #itemCount;
5       default: 0;
6       bePositive;
7       yourself
```

The first problem is that the description has no label, a required value in the meta-metamodel. The rule automatically suggests a refactoring (line 3) to add the missing label based on the name of the accessor. The second problem is the default value `0` (line 5), which does not satisfy the condition `bePositive` of the description itself (line 6).

From our positive experience with the Slime rules on the Seaside code-base, we expect a significant improvement of code quality in the realm of Magritte as these rules get adopted by the community. It is important to always keep the model and metamodel in a consistent state, which considerably improves the quality and stability of the code. With a few simple rules we can detect and fix numerous problems in the metamodel definition.

4 Related Work

There is a wide variety of tools available to find bugs and check for style issues. Rutar *et al.* give a good comparison of five bug finding tools for Java [RAF04].

PMD is a program checker that comes with a large collection of different rule-sets. Recent releases also included special rules to check for portability with the Android platform and common Java technologies such as J2EE, JSP, JUnit, *etc.* As such, PMD provides some domain-specific rule-sets and encourages developers to create new ones. In PMD, rules are expressed either as XPath queries or using Java code. In either case PMD provides a proprietary AST that is problematic to keep in sync with the latest Java releases. Furthermore reflective information that goes beyond a single file is not available. This is important when rules require more information on the context, such as the code defined in sub- and superclasses.

JavaCOP [ANMM06] is a pluggable type system for Java. JavaCop implements a declarative, rule-based language that works on the typed AST of the standard Sun Java compiler. As the rules are performed as part of the compilation process, JavaCOP can only reflect within the active compilation unit, this being a limitation of the Java compiler. While the framework is targeted towards customizable type systems, the authors present various examples where JavaCOP is used for domain-specific program checking. There is currently no integration with Java IDEs and no possibility to automatically refactor code.

Other tools such as *FindBugs* [HP04] perform their analysis on bytecode. This has the advantage of being fast, but it requires that the code compile and it completely fails to take into account the abstractions of the host language. Writing new rules is consequently very difficult (the developer needs to know how

language constructs are represented as bytecode), and targeting internal DLSs is hardly possible.

The *Smalltalk Refactoring Browser* [RBJ97] comes with over a hundred lint rules targeting common bugs and code smells in Smalltalk. While these rules perform well on traditional Smalltalk code, there is an increasing number of false positives when applied to domain-specific code. Helvetia and the domain-specific rules we presented in this paper are built on top of the same infrastructure. This provides us with excellent tools for introspection and intercession of the AST in the host system, and keeps us from needing to build our own proprietary tools to parse, query and transform source code. Helvetia adds a high-level rule system to declaratively compose the rules, and to scope and integrate them into the existing development tools.

High-level abstractions can be recovered from the structural model of the code. *Intensional Views* document structural regularities in source code and check for conformance against various versions of the system [MKPW06]. *Software reflexion models* [MNS95,KS03] extract high-level models from the source code and compare them with models the developer has specified. *ArchJava* [ACN02] is a language extension to Java that allows developers to encode architectural constraints directly into the source code. The constraints are checked at compile-time. Our approach does not use a special code model or architecture language to define the constraints. Instead our program checkers work with the standard code representation of the host language and make use of existing meta-frameworks such as Magritte or FAME. Furthermore our program checker is directly integrated with the development tools.

5 Conclusion

Our case studies revealed that rules that are targeted at a particular problem domain usually performed better and caused fewer false positives than general purpose lint rules. While more evidence is needed, these initial case studies do point out the benefits of using rules dedicated to domain-specific code over using generic ones.

As we need to accommodate new domain-specific code, the ability to add new dedicated rules is crucial. While we have demonstrated various program checking rules in the context of Seaside and Magritte, we argue that any library that uses domain-specific abstractions should come with a set of dedicated rules. Adding domain-specific rules is straightforward. Using the Helvetia framework it is possible to declaratively specify new rules and closely integrate them with the host environment. Rules are scoped to certain parts of the system using the reflective API of the host language. The existing infrastructure of the refactoring tools helped us to efficiently perform searches and transformations on the AST nodes of host system. This is the same low-level infrastructure used by the standard lint rules.

Furthermore we have shown that domain-specific rules need to be applied at different levels of abstraction. This can happen at the level of source code, as in the example of Slime; or it can focus more on model checking, where generic rules use a metamodel and the system to validate conformance, as we have shown

with Magritte. In both cases the rules were targeted at the particular domain of the respective frameworks only.

We applied the presented techniques only to internal DSLs, that by definition share the same syntax as the host language. As a generalization we envision to extend this approach to any *embedded language* that does not necessarily share the same syntax as the host language. Helvetia uses the AST of the host environment as the common representation of all executable code, thus it is always possible to run the rules at that level. Since Helvetia automatically keeps track of the source location it is possible to provide highlighting of lint issues in other languages. The challenge however will be to express the rules in terms of the embedded language. This is not only necessary to be able to offer automatic transformations, but also more convenient for rule developers as they do not need to work on two different abstraction levels.

Acknowledgments

We thank Adrian Lienhard and netstyle.ch for providing the Cmsbox case study. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 – Sept. 2010). We also thank ESUG, the European Smalltalk User Group, for its financial contribution to the presentation of this paper.

References

ACN02. Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural reasoning in ArchJava. In *Proceedings ECOOP 2002*, volume 2374 of *LNCS*, pages 334–367, Malaga, Spain, June 2002. Springer Verlag.

ANMM06. Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 57–74, New York, NY, USA, 2006. ACM Press.

BFJR98. John Brant, Brian Foote, Ralph Johnson, and Don Roberts. Wrappers to the rescue. In *Proceedings European Conference on Object Oriented Programming (ECOOP'98)*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998.

DGN07. Marcus Denker, Orla Greevy, and Oscar Nierstrasz. Supporting feature analysis with runtime annotations. In *Proceedings of the 3rd International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2007)*, pages 29–33. Technische Universiteit Delft, 2007.

DK97. Arie van Deursen and Paul Klint. Little languages: Little maintenance? In S. Kamin, editor, *First ACM-SIGPLAN Workshop on Domain-Specific Languages; DSL'97*, pages 109–127, January 1997.

DLR07. Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside: A flexible environment for building dynamic web applications. *IEEE Software*, 24(5):56–63, 2007.

Fow99. Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999.

Fow08. Martin Fowler. Domain specific languages, June 2008. http://martinfowler.com/dslwip/, Work in progress.

HP00. Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381, 2000.

HP04. David Hovemeyer and William Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92–106, 2004.

Joh78. S.C. Johnson. Lint, a C program checker. In *UNIX programmer's manual*, pages 78–1273. AT&T Bell Laboratories, 1978.

KS03. Rainer Koschke and Daniel Simon. Hierarchical reflexion models. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003)*, page 36. IEEE Computer Society, 2003.

KV08. Adrian Kuhn and Toon Verwaest. FAME, a polyglot library for metamodeling at runtime. In *Workshop on Models at Runtime*, pages 57–66, 2008.

Lie89. Karl J. Lieberherr. Formulations and benefits of the Law of Demeter. *ACM SIGPLAN Notices*, 24(3):67–78, 1989.

MKPW06. Kim Mens, Andy Kellens, Frédéric Pluquet, and Roel Wuyts. Co-evolving code and design with intensional views — a case study. *Journal of Computer Languages, Systems and Structures*, 32(2):140–156, 2006.

MNS95. Gail Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of SIGSOFT '95, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28. ACM Press, 1995.

PMW09. Frédéric Pluquet, Antoine Marot, and Roel Wuyts. Fast type reconstruction for dynamically typed programming languages. In *DLS '09: Proceedings of the 5th symposium on Dynamic languages*, pages 69–78, New York, NY, USA, 2009. ACM.

RAF04. Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A comparison of bug finding tools for Java. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 245–256, 2004.

RBJ97. Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

RDK07. Lukas Renggli, Stéphane Ducasse, and Adrian Kuhn. Magritte — a meta-driven approach to empower developers and end users. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems*, volume 4735 of *LNCS*, pages 106–120. Springer, September 2007.

RGN10. Lukas Renggli, Tudor Gîrba, and Oscar Nierstrasz. Embedding languages without breaking tools. In *ECOOP 2010: Proceedings of the 24th European Conference on Object-Oriented Programming, Maribor, Slovenia*, LNCS. Springer-Verlag, 2010. To appear.