Delft University of Technology
Software Engineering Research Group
Technical Report Series

Encapsulating Software Platform Logic by Aspect-Oriented Programming: A Case Study in Using Aspects for Language Portability

Lennart C. L. Kats, Eelco Visser

Report TUD-SERG-2010-019a

TUD-SERG-2010-019a

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
`http://www.se.ewi.tudelft.nl/techreports/`

For more information about the Software Engineering Research Group:
`http://www.se.ewi.tudelft.nl/`

This paper is a pre-print of:

Lennart C. L. Kats, Eelco Visser. Encapsulating Software Platform Logic by Aspect-Oriented Programming: A Case Study in Using Aspects for Language Portability. In Cristina Marinescu, Jurgen J. Vinju, editors, *Proceedings of the Tenth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2010). IEEE, 2010.*

```
@inproceedings{KatsVisser-SCAM-2010,
  title = {Encapsulating Software Platform Logic by Aspect-Oriented Programming:
           A Case Study in Using Aspects for Language Portability},
  author = {Lennart C. L. Kats and Eelco Visser},
  year = {2010},
  booktitle = {Proceedings of the Tenth IEEE International Working Conference
               on Source Code Analysis and Manipulation 2010},
  editor = {Cristina Marinescu and Jurgen Vinju},
}
```

© copyright 2010, Software Engineering Research Group, Department of Software Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology. All rights reserved. No part of this series may be reproduced in any form or by any means without prior written permission of the publisher.

Encapsulating Software Platform Logic by Aspect-Oriented Programming: A Case Study in Using Aspects for Language Portability

Lennart C. L. Kats
Software Engineering Research Group,
Delft University of Technology, Delft, The Netherlands
Email: l.c.l.kats@tudelft.nl

Eelco Visser
Software Engineering Research Group,
Delft University of Technology, Delft, The Netherlands
Email: visser@acm.org

***Abstract*—Software platforms such as the Java Virtual Machine or the CLR .NET virtual machine have their own ecosystem of a core programming language or instruction set, libraries, and developer community. Programming languages can target multiple software platforms to increase interoperability or to boost performance. Introducing a new compiler backend for a language is the first step towards targeting a new platform, translating the language to the platform's language or instruction set. Programs written in modern languages generally make extensive use of APIs, based on the runtime system of the software platform, introducing additional portability concerns. They may use APIs that are implemented by platform-specific libraries. Libraries may perform platform-specific operations, make direct native calls, or make assumptions about performance characteristics of operations or about the file system. This paper proposes to use aspect weaving to invasively adapt programs and libraries to address such portability concerns, and identifies four classes of aspects for this purpose. We evaluate this approach through a case study where we retarget the Stratego program transformation language towards the Java Virtual Machine.**

***Keywords*-programming languages, compilers, aspect-oriented programming, Stratego, Spoofax, Java**

I. Introduction

Programming languages form layers of abstraction over low-level machine code. High-level programming languages, especially those aimed at a particular domain, target a lower-level programming language and an API in that language. That is, rather than targeting a particular hardware platform, they target a *software platform*. A software platform consists of one or more programming languages, application frameworks, and libraries, and can be used on one or more hardware platforms. By targeting software platforms in high-level languages, their design and implementation can benefit from the abstractions available on such a platform [20]. Examples of software platforms are the Java and .NET platforms, LAMP[1], and C with the POSIX library. Each platform has its own ecosystem of programming languages, libraries, and developer community.

For programming language designers, targeting multiple software platforms can be appealing for many reasons. Doing so may allow the language to run on different hardware; can make integration with existing software such as the Eclipse IDE on Java possible; it may improve performance, as seen with JRuby [14] and IronPython [6] at different points in time. A new platform may also attract a different developer community. Altogether, targeting a new platform with a language can be a rewarding endeavor.

Programming languages are typically implemented using a frontend/backend architecture that aids in retargetability: a new backend can be added to generate code using a language or instruction set supported by the platform. However, this only addresses portability concerns at the bottom layer of the technology stack provided by a software platform.

Programming languages can also abstract over the layer of software (libraries) provided by a platform, in particular to provide domain-specific functionality [20]. Libraries may perform platform-specific operations, make direct native calls, or make assumptions about performance characteristics of operations or about the file system. Some libraries are only available for selected platforms. For example, if a web programming language uses the Java-based Hibernate framework in its implementation, some equivalent is required to implement the language on another platform. These platform dependencies are exposed to programs either directly through the runtime system of the language, or indirectly through standard functions or libraries written in the language. This means that programs can become tied to a particular platform, regardless of the compiler architecture.

Another area that requires attention when targeting a new platform with a language is interoperability with existing applications and libraries on the platform. Care should be taken to address platform idiosyncrasies such as event handling models and exception handling. For example, in C when some error condition arises, an application could log a message to the console and quit with a non-zero exit code. But for other platforms throwing an exception instead could increase interoperability with other applications. For example, when embedded in a GUI application, a popup could be shown.

Portability concerns at the library level can be addressed by modifying libraries and programs written in the source

[1]Linux, Apache, MySQL, and Perl/PHP/Python.

language, introducing changes to match the new platform or to abstract over the original platform. To avoid having to fork the sources and having to maintain multiple copies, *conditional compilation* can be used. Using conditional compilation, sources can be statically configured by a set of compiler flags that enable or disable logic for specific platforms. The most straightforward way of conditional compilation is to use a preprocessor. Preprocessors directly manipulate text and ignore the base language's syntax rules. This practice makes it much harder for various tools – IDEs, code analyzers, etc. – to process the code. Alternatively, true static conditional language constructs can be added, but they still complicate reasoning about the language and supporting it in tools. Moreover, using any form of conditional compilation also leads to tangling and scattering of platform logic throughout the base source code.

Instead of conditional compilation, this paper proposes to use aspect weaving [12] to address library-level portability concerns of programming languages. Using aspects, portability concerns can be expressed separately, rather than scattering them across the base source code. By using load-time aspect weaving, the base source code can be separately compiled and invasively adapted with platform-specific changes. A minimal join point model and before/around advice suffices to make such changes to libraries.

To evaluate this approach, we report on our experience in retargeting the Stratego program transformation language [3] to Java. The language has originally been compiled to C and is associated with the GNU/Linux operating system; we evaluate how aspects can be used to address portability concerns when targeting the Java platform instead. Previous experience with the Stratego interpreter for Java has shown that targeting the platform has a number of appealing applications, such as integration with compiler frontends written in Java to support transformations [9], and integration into the Eclipse IDE [10]. The present work improves the level of integration with such tools and allows Stratego programs that originally targeted the C platform – rather than specialized programs making use of Java-based libraries – to be used on Java and in Eclipse.

Stratego does not support aspect-oriented programming out of the box. Few languages do. In general, some form of aspect weaving must be added to the language for our approach to be effective. For our study, we introduced aspect weaving facilities to the language. Our results indicate that a minimal, lightweight set of aspect weaving features suffices to address portability concerns. Using load-time or run-time weaving, these features are straightforward to implement and support separate compilation.

Using aspect-oriented programming rather than conditional compilation, we can adapt the Stratego standard library without having to directly change the original code; the C and Java implementations remain truly separate. Existing Stratego programs that were designed for use on the C platform can be compiled to Java without requiring their sources to be changed. Java-specific adaptations are generally only needed in Stratego libraries, and can be expressed entirely using aspects.

This paper proposes the use of aspects to address language portability concerns. To this end, we:

- identify *four classes of aspects to address portability concerns*: glue code, migration, integration, and optimization aspects;
- describe a minimal, lightweight form of aspect weaving required to implement these aspects; and
- implemented a Stratego-to-Java compiler[2] to evaluate our approach, showing instances of all four classes of portability aspects.

We begin this paper by discussing general goals, design principles, and typical challenges faced in porting a language to multiple software platforms. We then discuss classes of aspects that can be used to meet these challenges. In Section III, we describe the Stratego language, the current compiler design, and introduce our extension of Stratego with aspects. Section IV shows portability problems in retargeting Stratego, and how these can be addressed with aspects. We discuss our results in Section V.

II. Targeting Multiple Software Platforms

When targeting a different software platform with a language, there are a number of general goals that guide the design and implementation of such an effort. First, existing applications should run on the new platform with no or minimal changes. Second, any libraries written in the language should be reused. Unlike normal applications, libraries – especially those bundled with the language – often use low-level, primitive operations. If that is the case, a number of platform-specific changes must be made that are not exposed in the API. Third, integration with other applications, frameworks, and languages on the target software platform is a key part of the retargeting effort.

Well-known design principles of portable language implementations are the use of a frontend/backend architecture and a runtime system that provides a set of primitives that abstract over the platform. These ensure that a new platform can be targeted by reusing the frontend and interface for the runtime system, while replacing the backend and implementation of the primitives.

A. Language Portability Concerns

Based on the general design principles for portable compiler language implementation, a compiler backend can be created that emits code for a particular platform, along with a runtime system that supports it. In practice, however, a new backend and runtime system do not automatically constitute language portability. Many applications, libraries,

[2]Available from http://www.strategoxt.org/Stratego/STRJ.

and sometimes the compiler itself, have been written with a particular platform in mind, making certain assumptions and showing behavior only appropriate for that particular platform. These library-level portability concerns cannot be addressed by the compiler and runtime alone, and require changes to the code base written in the language. In this section, we outline different sorts of these concerns. We revisit them in Section IV, showing concrete cases for the Stratego language.

Platform-specific libraries: High-level languages, particularly those that target a specific domain, are often implemented by providing a linguistic abstraction over high-level frameworks or libraries. These libraries are tied to a particular software platform. For example, consider the Hibernate framework for Java. There are many popular object-relational mapping (ORM) frameworks for other platforms, but their semantics differ slightly.

When a library is not supported on a particular platform, a similar, alternative library may be available that can be used instead. Using *glue code*, it may be feasible to adapt it to the same basic interface of the reference library. Glue code is code that does not directly contribute any functionality towards meeting the program's requirements, but serves solely to "glue together" different parts of code, helping programs and libraries interoperate.

Platform escapes and native calls: Escapes to code written in the host language (e.g., C) can introduce portability concerns as the code escaped to typically does not run on another platform. For other platforms, an alternative must be implemented. Ideally, code in the retargeted language – in our case study, Stratego – should be used instead. If this is not possible, the escaped code should be added to the set of primitives of the language, allowing different platforms to provide their own definitions.

Similar to escapes to platform code, native calls to other programs that run on the platform introduce a number of portability problems, as they add dependencies to external programs that may not be available on other hardware architectures or operating systems. Some platforms simply may disallow the use of native calls, or do not bundle required external programs. Rather than making direct native calls, a more portable approach is to use libraries where possible. They have an interface that is less dependent of the operating system, file system, and path configuration used for the software platform.

Interoperability and integration with platform applications: A strong motivation for targeting a particular platform can be to integrate the programs in the retargeted language with applications and libraries that run natively on the target platform (e.g., Eclipse on Java). Key to integration is a good interface for interoperability between the different languages. The public API of a generated application should be human-readable and easy to use. Advanced language features should be mapped to corresponding features on the platform. For instance, the Scala language targets the JVM and exposes advanced language features such as traits as standard Java interfaces for interoperability [15]. In addition to the basic API interface, important notions for interoperability are event models and exception handling. Events and exceptions are particularly important when embedding programs written in a language traditionally used for batch processing, such as Stratego.

Performance and stack behavior: Software platforms may use different libraries, a different language, and a different runtime system (e.g., the JVM), leading to different performance characteristics. On some platforms, performing an operation one way may be faster, while on other platforms, doing it another way is faster. High-level operations cannot be easily optimized by the compiler, as they may need to be changed at the algorithmic level.

Stack size constraints of the JVM are a notorious problem for running functional languages on the JVM, as they typically do not have explicit looping constructs but use recursion to perform loops [18]. Our case study of Stratego forms no exception in this regard. The stack consumption of recursive looping is linear to the amount of loops performed, whereas it remains constant for iterative loop constructs such as the "for" loop, typically used in Java programs. The maximum stack size cannot be changed at run-time, only when a new JVM is created, which may not be possible or desirable in all environments. A common approach to thwart stack consumption is the use of tail recursion elimination [18], but for indirect or non-tail recursion the prevailing solution is to use a non-recursive or tail-call based implementation instead.

B. Aspects to Address Language Portability Concerns

We identify four classes of aspects to address such concerns such as those listed in the previous subsection:

Glue code aspects add glue code to help compatibility with platform-specific libraries. Glue code written in the retargeted language itself is often more concise and high-level than wrappers at the platform level. Glue code aspects may be *transitional*; as libraries are ported to a platform, some additional glue code may be used to help compatibility.

Migration aspects help developers in retargeting their applications to a different platform. Typical use cases include platform escapes and native calls, which hinder portability. These aspects may either *passively* display warnings or errors for developers that use operations not (fully) supported on a platform, or they may *actively* aid the developer, redirecting such operations to alternatives that are supported on a platform but may not be fully compatible with the original operations. For example, if a language relies on Java serialization, on another platform it may implement serialization by use of an alternative API.

Integration aspects aid in interoperability and integration with other languages on the platform. For example, they can add exception handling or throwing to functions written in the retargeted language, or they can add application-specific hooks for use by other software on the platform.

Optimization aspects address run-time performance and scalability concerns. They change definitions in applications or libraries in order to achieve better performance for a particular platform, or to avoid platform restrictions such as stack size overflows resulting from uses of deep recursion on the JVM.

After introducing the Stratego language and our extension with aspects in the following section, we show concrete example use cases these of aspects in Section IV.

III. Modularity and Aspects in Stratego

In this section we first briefly introduce the Stratego language, and then present the design and implementation of an extension of Stratego with a minimal set of aspect weaving facilities. For a comprehensive description of the Stratego language, we refer the reader to [3].

Stratego is a domain-specific language for program transformation, used to build tools such as compilers, static source code analyzers, and interpreters. Together with the XT set of tools, Stratego/XT can be used for building comprehensive, stand-alone program transformation tools. The most important of these tools are the ATerm exchange format [2] and SDF/SGLR [22], both of which have been reimplemented in Java. As a bootstrapped language, the Stratego compiler is implemented in Stratego itself.

At its base, Stratego is a term rewriting language. Using a first-order term representation of (abstract syntax trees of) programs of a given language, programmers can define *rewrite rules* to transform programs. As a basic example, the following is the definition of a single rewrite rule that desugars a one-armed "if" statement with a condition e and a body stm to a two-armed "if" statement:

```
desugar-java:
  If(e, stm) → If(e, stm, Empty())
```

Whereas most term rewriting engines use a fixed (innermost) strategy for rewriting, Stratego allows the definition of custom rewriting *strategies*. Strategies are a generic description of how a rule or series of rules should be applied. They may be application-specific, or can be independent of a particular subject language and application; the Stratego standard library defines a large set of useful, generic strategies.

A. Modularity and Extensible Definitions

Stratego's module system uses hierarchical *modules* that correspond to files relative to the set of import paths. Stratego uses a flat namespace, and allows multiple definitions of rules and strategies with the same name. These can be defined within a single module or across multiple different modules. For example, one module may define multiple `desugar-java` rules related to control-flow features, while another module may define desugaring rules for other language features. All definitions for a rule or strategy are merged together, in a fashion similar to open classes or multimethods: when called, the first successful matching definition is dispatched. The traditional, whole-program compilation scheme of Stratego does not support extension of definitions that have been compiled separately.

B. Introducing Aspect-Oriented Programming to Stratego

While the Stratego language as it is offers a powerful mechanism for extending existing definitions, this mechanism is not sufficient to express the portability aspects we described in Section II-B. The mechanism cannot be used to adapt definitions in external libraries without having to "recompile the world," and can only be used to introduce new definitions that are *independent* of the existing definitions. For example, a new rule may be added that desugars the "for" statement, but the rule for the "if" statement cannot be adapted. It is also not possible to introduce pre-conditions to existing rules, add logging messages, or pre-process the input terms that are processed by rewrite rules.

In previous work, Kalleberg and Visser introduced AspectStratego [8], an extension of Stratego with a full-featured aspect language. However, since it was based on source-level aspect weaving, it cannot be used to weave into separately compiled libraries. Weaving into separately compiled libraries can only be supported by weaving into compiled code or through load-time or run-time weaving. Load-time and run-time may be the options of choice for the purpose of addressing portability concerns as they require comparatively little implementation effort.

In the present paper we introduce a new extension of Stratego based on load-time weaving, supporting basic before/after/around advice. In our extension we define a new set of modifiers that refine Stratego's definition extension mechanism and give control over definitions in separately compiled libraries. These modifiers are `extend`, which extends an external definition; `override`, which overrides an external definition; and `internal`, which indicates that a definition should be closed to extension and must not be exported to other libraries. We also introduce the `proceed` keyword, familiar to that of some more conventional aspect languages, which allows advice to return control back to the intercepted code.

As an introductory example, we can add – perhaps a cliché – logging messages to the `desugar-java` rule from the beginning of this section:

```
override desugar-java =
  log(|Info(), ["Desugaring: ", <id>]);
  proceed
```

Note that we do not syntactically separate the definition of the join point and the advice. This particular aspect specifies a join point matching any strategy or rule named `desugar-java` with zero arguments. When needed, join points can use wild cards: a pattern `desugar-*` would match

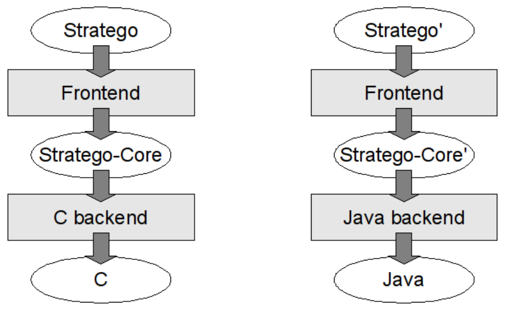

Figure 1. The basic architecture of the Stratego-to-C and Stratego-to-Java compilers, each compiling to core Stratego and then to the target language. The new Java compiler supports Stratego with aspects (Stratego′).

any rule with the prefix `desugar-`. The aspect also specifies "before" advice that prints a message with the current term (i.e., `<id>`) for every invocation of `desugar-java`. We do not distinguish between "before" and "after" advice in the language: instead, we use the standard Stratego operators to combine the definitions. This example uses the sequence operator (s_1 ; s_2), which specifies that first statement s_1 should be executed, and, if successful, then s_2 should be executed. Other operators include conditionals, and (non)deterministic choice between different strategies.

C. *Implementation of Aspects in Stratego*

To implement aspect weaving in Stratego, we must add the new syntactic constructs to the front-end, and adapt the back-end to generate code for the new features (Figure 1). We implement aspects by generating standard Java source code. Since our join point model is relatively simple, this requires only a modest extension of the compiler. In general, generating code to a language that directly supports aspects – such as AspectJ [11] – may be easier, but in this case we want to avoid introducing a dependency to the AspectJ compiler.

The Stratego frontend merges all rules and strategies with the same signature, as described in Section III-A. The Java backend then compiles each definition to a single class that inherits from the `Strategy` class. This class defines a number of overloads of the `invoke()` method with different parameters, and a `dynamicInvoke()` method for invoking strategies with an unanticipated number of parameters. Figure 2 shows the class that implements the `desugar-java` rules.

Each strategy is implemented as a *singleton class*, and has a mutable `instance` field. This field is used to invoke the strategy or to pass it as the argument to a higher-order function, such as the strategy call `topdown(desugar-java)` that applies `desugar-java` to a tree in a top-down fashion.

For strategies that are adapted by aspects, the `instance` field is assigned to the instance of the adapted strategy. For example, Figure 3 adds the logging advice from the previous subsection to the class for `desugar-java`. Note

```
public class desugar_java extends Strategy {
  public static desugar_java instance =
    new desugar_java();

  @Override
  public IStrategoTerm invoke(..., IStrategoTerm term) {
    IStrategoConstructor cons0 = term.getConstructor();
    if (cons0 == DesugarJava._consIf2) {
      Desugar the "if" construct
    } else if (...) {
      Apply other definitions of the desugar-java rule
    } else {
      return null; // rule application failed
    }
  }
}
```

Figure 2. The strategy class for `desugar-java`.

```
class desugar_java_override extends desugar_java {
  private final desugar_java proceed =
    desugar_java.instance;

  @Override
  public IStrategoTerm invoke(..., IStrategoTerm term) {
    term = log.instance.invoke(context, ...);
    if (term == null) return null;
    return proceed.invoke(context, term);
  }
}
```

Figure 3. A class overriding the `desugar-java` definition.

in particular that the `proceed` call in this definition is defined by copying the `instance` field of the original `desugar_java` class. When the library that contains the override class is initialized, it simply reassigns the existing `instance` field:

```
desugar_java.instance =
  new desugar_java_override();
```

Multiple libraries may add advice to the same definition in this fashion, following the order in which they were imported. The `proceed` field always refers to the definition that preceded the new advice, allowing multiple advice rules to be combined.

IV. Encapsulating Platform Logic using Aspects

In this section we elaborate on the concrete cases where aspects can be used to express platform logic, following the same structure as Section II-A. In total, we created seven libraries with aspect definitions to address portability concerns in different areas. Figure 4 illustrates the overall architecture and interaction between the compiler and runtime components.

A. *Platform-Specific Libraries*

On the C platform, Stratego uses the ATerm library [2] to represent terms, and the scannerless generalized-LR SGLR parser [22] for parsing. While a Java implementation of the ATerm library exists, we use a more flexible library that allows custom term library implementations based on a fixed interface, making it possible to operate on arbitrary

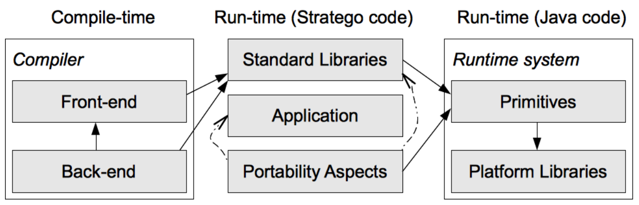

Figure 4. The role of aspects and the interaction between components of Stratego on Java. Dependencies are indicated by solid lines; weaving is indicated by a dashed line.

```
override parse-stream-pt(on-parse-error, ...) =
  read-text-from-stream;
  parse-string-pt(on-parse-error, ...)

override asfix-anno-location =
  fatal-err(|"Not implemented on this platform")
```

Figure 5. Glue code aspects for compatibility with parsing on Java.

(wrapped) Java objects [9]. It supports the same basic operations of the ATerm library, but has different performance characteristics.

To parse files, we use JSGLR, a Java port of the SGLR parser. At the time of writing, JSGLR supports most functionality of SGLR, and – for the most part – is successful in strictly following the SGLR semantics. However, JSGLR is still under development, and during the construction of the Stratego-Java compiler did not yet have the same API as the C version. To address these issues, we defined some transitional *glue code aspects* to address the incompatibilities until the JSGLR interface further evolved.

Figure 5 shows two glue aspects related to the JSGLR parser library. First, we defined an aspect that introduces a bit of glue code for parsing of streams: to read the input stream to a string we use the `read-text-from-stream` strategy, and then call the JSGLR parser to parse that string. We also added a migration aspect for the `asfix-anno-location` strategy, which adds position information annotations to parse trees, and is not (yet) supported on Java. Some applications work regardless of the annotations; they may redefine the strategy to report a warning instead.

B. Platform Escapes and Native Calls

The Good: Stratego uses a runtime system with a well-defined set of primitives, as advocated in Section II, to implement primitives in the standard library that cannot otherwise (or not efficiently) be implemented directly in Stratego. These primitives can be invoked using the `prim` language construct. For example, the standard library utility strategy `concat-strings` concatenates strings, and is implemented as follows:

```
concat-strings =
  prim("SSL_concat_strings", <id>)
```

That is, the strategy calls a primitive by the name `SSL_concat_strings` and passes it the input term `<id>` of the strategy.

Stratego has a large collection of well over a hundred of these primitives, most of which are straightforward to implement in Java. A compiler (backend) for a given software platform can recognize the `prim` constructs and translate them to function calls for that particular platform.

The Bad: In addition to the `prim` construct, Stratego also supports an `external` modifier to directly call custom, native C functions, not unlike the native methods used in a language like Java. Some Stratego applications use these as optimizations or to interface with native libraries. There are also some cases where escapes to platform code are made from the standard libraries and compiler. Because of the ad hoc nature of these functions – any application can define and use their own native functions – native function calls cannot be uniformly translated by a compiler.

As an example of an external strategy, consider the following definition that accesses a native pretty-printer (which is faster than the table-based Stratego implementation), as used in the Stratego C backend and some applications such as WebDSL [23]. This particular pretty-printer prints Java to a formatted text string:

```
external pp-java(|)
```

When Stratego runs on Java, the external C function is not available. Using a runtime check or through conditional compilation at the call sites, an appropriate alternative could be called. Using aspect weaving instead, such changes can be implemented without changing the original code – which is undesirable for third-party libraries. All aspects dealing with native function calls on Java can be collected together rather than scattered throughout the code. For the case of `pp-java`, we can simply add advice that calls the table-based pretty printer instead:

```
override pp-java =
  pp-java5-to-abox; box2text-string(|80)
```

The Ugly: Stratego programs have traditionally been based on XTC, a library for creating monolithic transformation programs by composition of smaller tools, such as a parser or a pretty-printer. XTC's function in the Stratego world has been to reconcile the philosophy of the Unix platform of "tools that do one thing, and do it well" with the need for comprehensive transformation tools. XTC maintains its own component model in a customizable location in the file system, called the XTC repository. The repository is used to store and retrieve file system paths to shared tools. Using the `xtc-command` strategy, any of these tools can be invoked with a given set of command-line options. `"-o", output]`. XTC also provides a `xtc-find` strategy that has been used to find the paths of parse and pretty-printing tables used for an application, and for locating library headers for use by the Stratego compiler.

Lately, XTC is being phased out in favor of libraries [3], which are more efficient than forking a new process for

```
override xtc-command(tool) =
  if tool ⇒ "pp-java" then
    directly pretty-print using pretty-printing library
  else
    proceed
  end

override xtc-find =
  warn-msg(|["XTC used to find non-local file ", <id>]);
  id // don't proceed with the original xtc-find

override fork(child) =
  fatal-err(|"Not supported on this platform")

override call =
  ?(program, args);
  log(|Info(), ["Calling external tool ", program]);
  prim("SSL_EXT_call", program, args) ⇒ 0
```

Figure 6. Migration aspects related to native calls and XTC usage.

```
extend pack-stratego-parse-stratego:
  (IncludeFromPath(name), includes) → ("", ast)
  where
    switch !name
      case "libstratego-lib": import-term(headerfile)
      case "libstratego-xtc": import-term(headerfile)
      case ...
    end ⇒ ast

extend pack-stratego-parse-stratego:
  (IncludeFromPath("libjava-front"), includes) → result
  where
    if not(proceed ⇒ result) then
      result := ("", <import-term(libjava-front.rtree)>)
    end

override strc-get-include-dirs =
  <get-config> "-I"
```

Figure 7. Migration aspects for XTC usage for Stratego compilation.

specific tasks. In addition to performance concerns, XTC – and direct invocations of executables in general – also hinders portability of applications that use it. On the Java platform, invoking tools in this fashion is often not possible, and using the file system for resources forms a mismatch with the light-weight deployment system of using JAR files for applications.

Figure 6 shows a number of *migration aspects* for running Stratego on the JVM. The first definition extends `xtc-command` to redirect any calls to the command-line `pp-java` tool to more portable strategies. Multiple, independent extensions like it can be added for other invoked tools, aiding in the portability of existing Stratego applications that have not yet made the transition to libraries. We also override the `xtc-find` strategy, which normally returns the absolute path of a file using the XTC repository. This is an example of a transitional migration aspect: we only display a warning and do not call `proceed`, allowing the application to continue in case the requested file simply exists in the current directory or if it is an executable on the path.

Native executables can also be directly invoked using the `call` strategy. Normally, `call` is implemented using the `fork` strategy to fork the current process, but on Java no such notion exists (although it could be simulated). Therefore, as additional migration aspects, in Figure 6 we also override `fork` to print an error, and redefine `call` using a new Java-specific primitive, printing an informational message that reminds developers they are calling a native executable.

The Stratego compiler normally uses library definitions that use XTC to find header files for imports. On Java, headers should be distributed inside the compiler's JAR file instead. They can be automatically embedded using Stratego's `import-term` construct. The `import-term` construct is also available on the C platform, but has significantly different performance characteristics, inflating compilation times and executable sizes as they are serialized to array literals in C. Using separate aspect definitions ensures that only on Java `import-term` is used to retrieve headers.

Figure 7 shows the aspects that introduce `import-term` calls to the `pack-stratego-parse-stratego`. The first aspect fetches headers of the standard libraries. These are all tied to the version of the compiler and cannot be overridden. The second aspect retrieves the headers for the Java-Front library, which may be overridden with a different version by the user. The third aspect in Figure 7 ensures that any third-party library headers are not loaded using XTC, but only from the directories specified with the "-I" command-line option.

C. Interoperability and integration with Java applications

So far, we have already shown a number of ways in which direct interoperability between compiled Stratego code and Java can be achieved. Java code is used for the implementation of the platform primitives, and can be used to implement custom term libraries. Strategy classes written in Java (or another JVM language) can be used to define new strategies with only a small boilerplate. These classes can be used to extend, override, or simply add event hooks to Stratego definitions.

Traditional Stratego programs have been batch applications such as compilers. They output a number of informational or error messages to the console, may write resulting files to disk, and then exit. When integrating such an application into interactive Java applications, such as the Eclipse IDE, a different way of presenting feedback to users is required, particularly for error reporting. This can be addressed using an *integration aspect* to improve this behavior by adapting the `fatal-err` strategy, used to report fatal errors (as seen in Figure 6). Normally, this standard library strategy is defined as follows:

```
fatal-err(|msg) =
  log(|Critical(), msg, <id>);
  <exit> 1
```

This standard definition prints an error to the standard error output, and then exits the application using `1` as the exit code. We can refine the error handling behavior by throwing an informative Java exception instead, using a primitive called `SSL_EXT_fatal_err`:

```
override fatal-err(|msg) =
  log(|Critical(), msg, <id>);
  prim("SSL_EXT_fatal_err", msg, <id>)
```

Applications such as the most recent version of the Spoofax language workbench [10] can use the thrown exception to present the user with a pop-up in case of errors, referring the user to the complete error log for more details.

Other general-purpose integration aspects can include hooks into the logging and assertion strategies of Stratego. More application-specific aspects may hook into strategies to provide feedback as a transformation runs, or to interact with the user.

D. Performance and Stack Behavior

Since a different software platform – different libraries, language, and the JVM runtime – is used for Stratego on Java, it has different performance characteristics than Stratego does on C. Using aspects, performance-critical sections of code can be replaced with new definitions that better suit the platform.

Using the excellent Java profiler, we were able to identify only a few bottlenecks in normal Stratego programs. Most could be addressed by ordinary, general compiler optimizations (e.g., caching the result of `getConstructor()` in a local variable, as seen in Figure 2). One strategy that really stood out in the profiler results was the `read-text-from-stream` strategy. It is implemented directly in Stratego by reading a stream character by character, constructing a string from the results. Still, the strategy was never a bottleneck for typical C-based Stratego applications. However, recall that we used this strategy in Figure 5, which means that it is now used for almost all parser invocations. We can override the strategy with a new definition

```
override read-text-from-stream =
  prim("SSL_EXT_read_text_from_stream")
```

where we introduce a new Java primitive that efficiently reads the stream using a block buffer instead of characters.

Stack behavior: As a functional programming language, Stratego uses recursion to express loops. On the Java platform, the stack is an expensive commodity, and using deep recursion quickly uses it up. We found that many Stratego applications, such as the compiler itself, resulted in a stack overflow exception because of this restriction.

Stratego has many strategies that rely on left-to-right traversal of terms, producing new, immutable left-to-right encoded terms. To rewrite these recursive strategies to use tail recursion would mean that additional, intermediate data would have to be maintained on the heap. The performance overhead rewriting key strategies such as `map` and `filter` in this fashion may not be acceptable. As an alternative approach, we can redefine these strategies using imperative Java code. Similar to what can be done with mixed imperative/functional programming languages such as Scala [15], selective use of imperative code in key library definitions allows for better stack behavior using mutable data structures and iterative looping.

```
override filter(s) =
  prim("SSL_EXT_filter", s | <id>)

override map(s) =
  is-list;
  all(s)

override getfirst(s) =
  is-list;
  one(where(s; ?x)); !x
```

Figure 8. Optimization aspects for stack-intensive strategies.

Figure 8 illustrates some of the *optimization aspects* that help avoid stack overflows. In the figure, we redefine `filter` using a new primitive, implemented in Java; `map` using the `all` operator for lists; and `getfirst` using the `one` operator. Internally, each Strategy is based directly on imperative Java code: `filter` uses a primitive implemented using an array and a “for” loop (not shown here for reasons of space), and the other strategies use the primitive `all`/`one` operators. These `all`/`one` operators are standard strategic programming operators (described in [3]) that apply their argument to *all* or *one* of the subterms of the current term. Since the operators are implemented directly in Java, we can use them here to avoid deep recursion on long lists. Similarly, we redefined or extended eight other key library strategies that use recursion. As a result, stack overflow conditions for virtually all applications are now avoided, without having to change the JVM's default stack size. For those applications that use custom, stack-intensive strategies, similar measures can be taken.

V. Discussion and Related Work

Our case study of Stratego for Java has shown instances of aspects of all four classes of glue code, migration, integration, and optimization aspects. We were able to neatly separate these concerns from the base source code, grouping them together by association. Using conditional compilation instead, they would be scattered throughout the existing Stratego code base.

Aspects allowed us to implement a new Stratego backend without “polluting” the existing code base with Java-related concerns. Since they can be woven into compiled code – using load-time aspect weaving – there was no need to recompile the base Stratego compiler components and libraries with every Java-related change. This resulted in a shorter development cycle.

By supporting the Java platform and deep integration with other Java applications – in part made possible by the integration aspects – this work has been an important step towards a fully fledged interactive development environment for Stratego based in the form of the Spoofax language workbench [10].

Related work: To our knowledge, there is no previous work that proposes to use aspects to address language portability concerns. However, there have been a number of recent proposals to replace conditional compilation with aspect weaving [1], [13], [16], [19]. Both Adams et al [1] and Reynolds et al [16] systematically studied different patterns of conditional compilation uses in, respectively, the Parrot virtual machine and the Linux kernel. Both studies concluded that for a large part of these uses, aspect weaving is a feasible alternative, but that in many cases preparation of the base source code was required to expose additional join points. Lohmann et al [13] did a quantitative analysis of the performance overhead of using aspects for configuration of the eCos kernel, and found the overhead to be acceptable. Likewise, they concluded that preparation of the base source code was required. C-CLR [19] is a tool that shows different views of source code, hiding disabled conditional parts. Through clone detection techniques, it allows mining of aspects. All these works have studied configuration of *systems software*, using the C language. In contrast, we studied the use of aspects for portability of high-level *programming languages*. In our case study, we did not need to prepare the base source code to expose join points or to use potentially fragile statement-level join points (as suggested by [13]), since only relatively simple aspect definitions were needed for the subset of portability concerns that could not be effectively addressed in the compiler. As such, only a modest aspect-oriented extension of Stratego was required, which meant only a small, acceptable startup cost was required for using aspects for portability.

Another area where aspects have been used for language engineering is in the construction of compilers [4], decomposing different crosscutting concerns in analysis, transformation, and generation of code. In contrast, in our approach we use aspects outside the compiler, and address platform dependencies in libraries and programs written in the compiled language.

A notably different approach to language portability is taken by the Scala compiler. It supports both the Java and .NET platforms, but its primary platform is Java: the standard Scala framework borrows types and methods from Java, and, at this point, the .NET backend has not been updated to the most recent version of the language. Instead of using conditional compilation for the Scala standard library, the .NET backend addresses the Java-based nature of Scala by redirecting a fixed set of Java-specific method calls to compatible .NET methods [17]. For example, calls to `Object.hashCode()` are redirected to their .NET equivalent, `Object.GetHashCode()`. As the two platforms are closely related, this strategy suffices for operations on standard types such as strings and objects. However, the approach reduces separation of concerns as the compiler must encode library-specific logic. In contrast, we use separate libraries of aspects. For platforms with a greater set of differences, where simple redirects do not suffice, these libraries can be used to encode further platform-specific logic. Encoding platform logic in the compiler approach is also less flexible as it cannot be used to adapt third-party libraries or applications.

There have been many proposals of intermediate languages to address compilation to multiple host languages, starting with languages such as the Universal Computer Oriented Language UNCOL [21] to more recent works such as C-- [7] and PIL [5]. These languages form an excellent complementary technique to our approach, eliminating much of the work required in implementing backends for multiple platforms. However, by themselves, they do not address portability concerns such as uses of native calls, platform-specific libraries, or platform-specific performance concerns, as discussed in Section II-A.

Aspect languages: In previous work, Kalleberg and Visser introduced AspectStratego [8], which extends Stratego with support for aspects, showing how they can be used to address concerns such as format checking, adaptable algorithms, and traceability. AspectStratego uses source-level weaving and supports a more extensive join point model than that of the present work. In contrast, for the present work we used load-time weaving, which was essential to allow weaving into compiled libraries, and allowed for a more straightforward implementation. As such, it gave a good indication of how a minimal, lightweight aspect weaving addition can be used for language portability,

Aspect languages that integrate with object-oriented programming languages, such as AspectJ [11], typically have a more elaborate join point model than the one we presented here or that of AspectStratego. They may support pointcuts for specific packages, classes, and parameter types. In contrast, aspects for domain-specific languages generally have a more restrictive join point model. In the case of Stratego, which is not object-oriented, lacks package names, and has definitions that span multiple modules, such a model arose naturally, and formed a good match with the classes of aspects shown in this paper.

Dynamic languages such as JavaScript, Smalltalk, and Ruby allow extensions and modifications of existing objects and classes with new methods that support the needs of particular applications or libraries. In our approach, all changes are performed at load time, and are statically checked; missing join points are reported by the compiler. Still, when systematically applied, a fully dynamic approach to address portability concerns as presented here is certainly feasible.

VI. Conclusion

This paper proposes to use aspect-oriented programming to address portability concerns with regard to languages that target multiple platforms. To this end, we identified four general classes of aspects to address such concerns: aspects that add glue code to platform-specific libraries, aspects that help developers migrate to a new API, aspects that help integrate with other applications on a platform, and those performing platform-specific optimizations. We showed instances of these classes in retargeting Stratego to the Java platform. In this case study, we successfully used aspects to neatly encapsulate concerns that would otherwise have spanned many different modules, in the compiler, library, and runtime system. It is our expectation that the same techniques can be used to help in separation of portability concerns for other high-level languages.

Future work with regard to aspects for portability concerns relates to software product lines: once more software platforms are targeted by languages, are there aspects that can be applied to multiple platforms? What are their dependencies on other aspects?

Acknowledgments: We would like to thank Karl Trygve Kalleberg for his work on Stratego/J that formed the basis for the compiler presented here. This research was supported by NWO/JACQUARD projects 612.063.512, *TFA: Transformations for Abstractions*, and 638.001.610, *MoDSE: Model-Driven Software Evolution*.

References

[1] B. Adams, H. Tromp, W. D. Meuter, and A. E. Hassan. Can we refactor conditional compilation into aspects? In A. Moreira and C. Schwanninger, editors, *International Conference on Aspect-Oriented Software Development (AOSD)*, 2009.

[2] M. G. J. van den Brand, H. de Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software, Practice & Experience*, 30(3):259–291, 2000.

[3] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Sci. of Comp. Programming*, 72(1-2):52–70, June 2008.

[4] O. de Moor, S. L. P. Jones, and E. V. Wyk. Aspect-oriented compilers. In K. Czarnecki and U. W. Eisenecker, editors, *Generative and Component-Based Software Engineering (GPCE 1999)*, volume 1799 of *LNCS*, pages 121–133. Springer, 1999.

[5] Z. Hemel and E. Visser. PIL: A platform independent language for retargetable DSLs. In M. van den Brand and J. Gray, editors, *Software Language Engineering (SLE 2009)*, LNCS. Springer, 2009.

[6] J. Hugunin. Dynamic languages on .NET - IronPython and beyond: IronPython 1.0 released today! http://blogs.msdn.com/hugunin/archive/2006/09/05/741605.aspx, September 2006.

[7] S. P. Jones, N. Ramsey, and F. Reig. C−−: A portable assembly language that supports garbage collection. In *Principles and Practice of Declarative Programming (PPDP 1999)*, volume 1702 of *LNCS*, pages 1–28. Springer, 1999.

[8] K. T. Kalleberg and E. Visser. Combining aspect-oriented and strategic programming. In *Workshop on Rule-Based Programming (RULE 2005)*, volume 147 of *ENTCS*, pages 5–30. Elsevier Science Publishers, 2006.

[9] K. T. Kalleberg and E. Visser. Fusing a transformation language with an open compiler. In *Workshop on Language Descriptions, Tools, and Applications (LDTA 2007)*, volume 203 of *ENTCS*, pages 21–36. Elsevier, April 2008.

[10] L. C. L. Kats and E. Visser. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In M. Rinard, editor, *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2010)*. ACM, 2010.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *European Conference on Object-Oriented Programming (ECOOP 2010)*, volume 2072 of *LNCS*, pages 327–355, 2001.

[12] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP'07)*, volume 1241 of *LNCS*, pages 220–242. Springer, 1997.

[13] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat. A quantitative analysis of aspects in the eCos kernel. *SIGOPS Oper. Syst. Rev.*, 40(4):191–204, 2006.

[14] C. Nutter. Promise and peril for alternative Ruby impls. http://blog.headius.com/2008/04/promise-and-peril-for-alternative-ruby.html, April 2008.

[15] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Press, 2008.

[16] A. Reynolds, M. E. Fiuczynski, and R. Grimm. On the feasibility of an AOSD approach to Linux kernel extensions. In *AOSD workshop on aspects, components, and patterns for infrastructure software (ACP4IS'08)*.

[17] Scala on .NET: quirks. http://www.scala-lang.org/node/169.

[18] M. Schinz and M. Odersky. Tail call elimination on the Java Virtual Machine. In *Workshop on Multi-Language Infrastructure and Interoperability (BABEL'01)*, volume 59 of *ENTCS*, pages 158–171. Elsevier, 2001.

[19] N. Singh, C. Gibbs, and Y. Coady. C-CLR: a tool for navigating highly configurable system software. In *AOSD workshop on aspects, components, and patterns for infrastructure software (ACP4IS'07)*. ACM, 2007.

[20] T. Stahl, M. Voelter, and K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[21] T. B. Steel, Jr. A first version of UNCOL. In *IRE-AIEE-ACM (Western)*, pages 371–378, New York, NY, USA, 1961. ACM.

[22] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[23] E. Visser et al. WebDSL. http://webdsl.org, 2007–2010.

TUD-SERG-2010-019a
ISSN 1872-5392