Improving Perimeter Security with Security-oriented Program Transformations

Munawar Hafiz, Ralph E. Johnson
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{mhafiz,johnson}@cs.uiuc.edu

Abstract

A security-oriented program transformation maps programs to security-augmented programs, i.e. it introduces a security solution to make programs more secure. Our previous work defined security-oriented program transformations [6], introduced a catalog of program transformations [8], and showed how program transformations could be applied to systematically eradicate various types of data injection attacks [9]. This paper shows how security-oriented program transformations could be used to improve the security of a system's perimeter by introducing authentication, authorization and input validation components. The program transformation examples in this paper are JAVA specific, but the transformations could be implemented to use other authentication and authorization frameworks.

1 Introduction

Security is architectural; it is a property of the entire system, not one part of it [3]. Security cannot be added to a system by adding a module, but it can be added in other ways. In particular, it is possible to improve the security of a system by applying program transformations.

There are many kinds of program transformations. Compilers transfer programs in source form to equivalent programs for a particular machine language. Refactorings [4] are source to source transformations that intend to change the structure of programs but not their behavior. The transformations in this paper include both source to source and binary to binary transformations. They improve the security of systems, which means that they do not preserve all types of behavior. They preserve the expected behavior, but change a system's response to security attacks.

A software system is often thought of as a collection of components. In that case, "adding on" to a software system means adding a new component. However, a software system can also be thought of as a sequence of program transformations. For example, consider a sequence of versions in a version control system. The difference between any two adjacent versions is a program transformation. Some version control systems, such as cvs, store the differences between versions instead of storing the versions directly. These systems produce a version by applying a series of program transformations. Our focus is not on these very specific program transformations.

We are interested in automated, general purpose program transformations that remove security threats from programs. This paper presents examples of security-oriented program transformations that could be applied to systematically introduce authentication, authorization and input validation components to existing systems.

Improving the security at the system perimeter is the first line of defense. Many security exploits could be avoided if proper authentication, authorization and input validation is done at the system perimeter. Thus the program transformations not only solve authentication problems, but have long term benefits in improving the security of systems.

We have three contributions. We show that it is possible to think about adding security in terms of general purpose program transformations. Second, we show that we could improve perimeter security with transformations that make structural changes to programs without requiring deep understanding of the program behavior; these transformations could be thought of as refactorings with the additional ability of changing program behavior. Finally, we show how Java Authentication and Authorization Service (JAAS) API is used by these program transformations.

The next section introduces security-oriented program transformations. Then we describe the security-oriented program transformations that improve perimeter security. Finally, we discuss how general purpose tools can be built to automate the program transformations.

2 Security-oriented Program Transformation

A program transformation is a function that maps programs to programs. A *security-oriented program transfor-*

mation [6] maps programs to security-augmented programs, i.e. it introduces a security solution to make programs more secure. A developer has to apply multiple program transformations on a program to have defense in depth [14]. Composing multiple security solutions at different layers hardens a program.

We have compiled a list of forty two candidate security solutions [8] that can be described as security-oriented program transformations. The list is created by surveying vulnerability trend reports [1] to identify relevant security problems, and using our comprehensive catalog of security patterns [7] to identify solutions for those problems.

Our security-oriented program transformations make structural changes that do not depend on detailed understanding of application logic. This property of security-oriented program transformations aligns them with refactorings [4]. On the other hand, our program transformations are not behavior preserving as refactorings are. The concept of behavior preservation should mean that a program preserves the correct behavior and fixes the incorrect behavior. Security-oriented program transformations are behavior preserving (or perhaps good path behavior preserving) according to the more liberal definition.

Security-oriented program transformations are semi-automatic. It takes parameters, usually in the form of a manual specification, that describe what structural changes can bring about a desired behavioral change (but preserve good path behavior). At the minimum, a transformation has to know which part of a program it will work on. For example, a *Partitioning* transformation takes a program and distribute its tasks into multiple processes. Such a transformation at least needs to know which program is to be transformed. Often program transformations take more parameters, i.e. a specification describing how the output should be. The amount of specification varies with the context. An architectural transformation such as a *Partitioning* transformation requires a lot of specification, e.g. task distribution, interfaces of partitions, privilege levels, inter-process communication mechanism etc.

Security-oriented program transformations could be automated because they separate policy from mechanism. A transformation makes the structural changes, while the behavioral specification is manually provided by the developer. A developer's specification is the *policy*. The tools implement the *mechanism*. With this separation, program transformations can automate new security protections without encoding deep understanding of program behavior in the transformation.

All of our program transformations separate policy from mechanism in this way. For each program transformation, we describe what a developer has to specify and what the tool does to apply the changes.

3 Improving Perimeter Security with Program Transformations

In this section, we will describe how a security-oriented program transformation could be applied to improve perimeter security of an existing software. Each program transformation is platform and language specific; we assume that the transformations are applied on a Java program on Windows platform.

We will use a running example to illustrate the effect of transformations. Suppose, an imaginary developer Alice has a Java program that connects to a hostname and a port via a socket. It connects to two hostname/port pairs and uses the data received from one connection to query an SQL database and the data received from another connection to query an LDAP database. Alice designed the program for the employees of her company; everybody was trusted, everybody had the same rights, hence there was no need for authenticating users. Now, Alice has to retrofit authentication, authorization, and input validation, because her company is planning to let external users use the program.

Let us follow Alice's thought process. The first thing to consider is where to add the new authentication, authorization and input validation features. Since, her program has two separate methods for socket connection (henceforward, we will refer to them as the two access points), she could apply the *Add Authentication Enforcer*, *Add Authorization Enforcer* and *Add Perimeter Filter* transformations in the two places of the program. But applying program transformations in this manner might open more chances of errors. For example, if one method is patched later, the programmer writing the patch needs to remember to fix the other method too. This can be avoided, if Alice minimizes the access points of the program to one place by applying the *Single Access Point* transformation. This transformation is usually followed by a *Policy Enforcement Point* transformation. A policy enforcement point provides a single check point for authentication, authorization and input validation. Alice could then apply *Add Authentication Enforcer*, *Add Authorization Enforcer* and *Add Perimeter Filter* transformations. The perimeter filter applies global input validation policies, but many other validation policies are input-specific. Hence Alice could apply the *Decorated Filter* transformation on several places of the program.

We will describe these six program transformations—*Single Access Point*, *Policy Enforcement Point*, *Add Authentication Enforcer*, *Add Authorization Enforcer*, *Add Perimeter Filter* and *Decorated Filter* transformation. These program transformations are applied in this sequence. We will describe the transformations in the following sections and refer back to the running example to show the impact of applying the security-oriented program transformations. We will follow Martin Fowler's format of describing refactor-

ings [4] to describe the transformations. First we will state the problem solved by a transformation and the solution. The motivation and mechanics section details the problem and the solution correspondingly. Finally, the example section describes the changes that are applied to the running example.

4 *Single Access Point* Transformation

You have a system where there are multiple ways for entering the application. You want to secure the system from outside intrusion.

Minimize the number of access points to a system.

4.1 Motivation

Having multiple access points is bad because a developer has to secure all the access points. Besides, authentication, authorization and input validation components have to be included at all access points. When a developer changes one of these components, he might fail to apply the change to all access points. Keeping a small number of access points avoids this vulnerability.

Merging multiple access points creates a gatekeeper component where authentication, authorization and input validation components could be added. An example of a gatekeeper component is the SMTP service component (*smap* and *smapd*) of TIS Firewall Toolkit (FWTK), that acts as a gatekeeper for an MTA running in the background. The goal of *smap* and *smapd* is to preserve the functionality of *sendmail*, while preventing an arbitrary user from communicating directly with *sendmail*.

4.2 Mechanics

A developer specifies the access points.

The transformation makes a Façade [5] in the object-oriented world or introduces a wrapper component that has the same API.

4.3 Example

The two socket connections are in separate methods in two separate classes. Alice selects these methods as the target of a *Single Access Point* transformation. The transformation pulls up the methods [4] to a new class. However, the methods to be pulled up may not share a common superclass. In that case, a program transformation might create a common superclass, or give each of them a component and create a common superclass for those components. The new class is the unified system access point. For the subsequent transformations, this will act as the base.

5 *Policy Enforcement Point* Transformation

You want to centralize authentication, authorization and input validation at the system entry point.

Create a policy enforcement point which acts as a Mediator between access points and authentication, authorization and input validation components

5.1 Motivation

When a system has minimal number of access points, then authentication, authorization and input validation components could be added. However, a component should act like a Mediator [5] between the access points and the components to be added.

5.2 Mechanics

A developer specifies the system entry point and the authentication, authorization and input validation policies.

The transformation creates a policy enforcement component and components for authentication, authorization and input validation. It delegates incoming requests to secure base action component.

5.3 Example

In this step, Alice selects the `AccessPoint` class created in the previous step and applies a *Policy Enforcement Point* transformation. Figure 1 shows the classes added by the transformation.

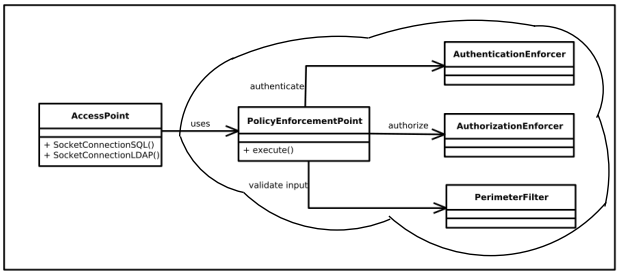

Figure 1. Classes added by a *Policy Enforcement Point* transformation

The `PolicyEnforcementPoint` class delegates authentication request to an `AuthenticationEnforcer` class. This class is a placeholder and could be initialized to return a negative value by default. This class is the starting point of the next transformation in our list that creates an username/password-based authentication component. Similarly, default classes are added for an authorization component and a perimeter filter component.

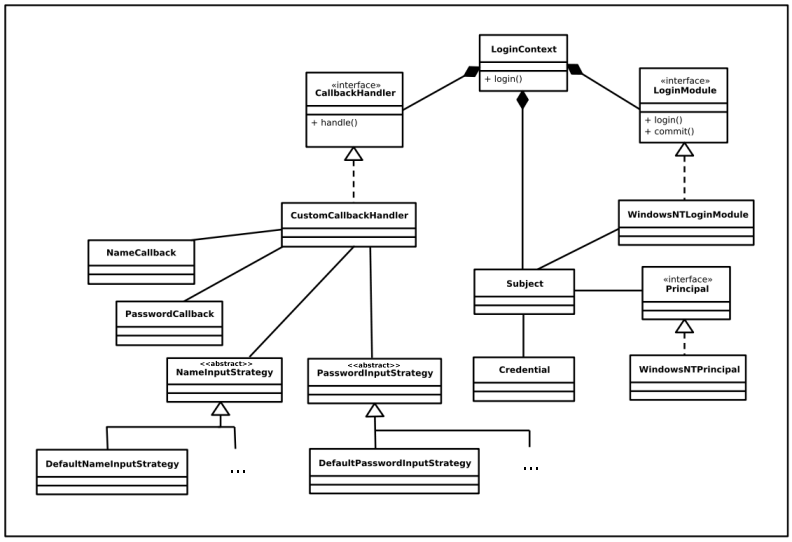

Figure 2. Classes added by an *Add Authentication Enforcer* transformation

6 *Add Authentication Enforcer* Transformation

You want to authenticate a user at the system entry point.

Create an authentication enforcer component which authenticates a user and creates a subject to represent the user.

6.1 Motivation

During authentication, applications transfer user credentials to verify the identity requesting access to a particular resource. After successful authentication, a subject is created with appropriate principals and credentials. The authentication mechanism could be different for different contexts, e.g. username-password based authentication, client certificate based authentication, or kerberos based authentication. Irrespective of the mechanism, the authentication logic should be concentrated in a single component that could be created based on user specification.

6.2 Mechanics

A developer specifies the insertion point of an authentication component, source of input data, authentication mechanism, and outcome of a successful or a failed authentication.

The transformation composes classes of a security framework (JAAS, .NET etc) to create a component with the specified authentication mechanism, and delegates authentication requests from the insertion point.

6.3 Example

Suppose Alice wants to introduce an authentication component which authenticates using username and password.

Alice replaces the `AuthenticationEnforcer` with a group of classes that perform username/password based authentication (see Figure 2). The *Add Authentication Enforcer* transformation composes the classes of the JAAS framework to provide a default authentication scheme. In our example, Alice specifies authentication type (username/password), source of user inputs, authentication knowledge base (username/password store), and the outcome of a successful authentication (which principals and credentials are added to the subject). The `AuthenticationEnforcer` creates a new `LoginContext` instance and calls its `login` method. This creates a new `Subject` representing the authentication requester and passes it to a `LoginModule` implementation. For this example, JAAS framework provides a `WindowsNTLoginModule`. It implements two methods—the `login` method decides on the authentication, and the `commit` method populates the subject with appropriate principals and credentials when the authentication is successful. User inputs are collected through the implementers of the `CallbackHandler` interface. The program transformation introduces a default strategy which will most likely be modified. Alice could apply an *Add Strategy* [10] refactoring and add a custom implementation later. Similarly, the authentication algorithm in a customized implementation of the `LoginModule` interface could be a Strategy [5] that developers could extend. The *Add Authentication Enforcer* transformation provides a default implementation only; Alice then customizes to fit the application specific requirements.

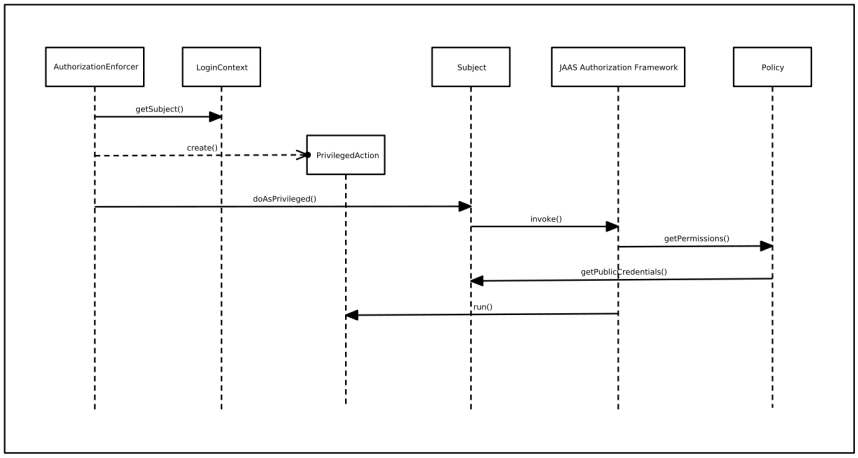

Figure 3. Steps of authorization introduced by the *Add Authorization Enforcer* transformation

7 *Add Authorization Enforcer* Transformation

You want to introduce the functionality of an access controller.

Add an authorization enforcer component that uses the permissions of principals to determine whether a subject is allowed to act upon a resource.

7.1 Motivation

The purpose of authenticating a user is to establish its identity so that its actions can be controlled. The subject created in the end of the authentication represents a user. It is embellished with principals, each of which has permissions associated. During authorization, a component matches the permission requested with the permissions retrieved from a subject's principals. Concentrating authentication logic in one component makes it easy to maintain access control code.

Before enforcing authorization, users in the system must be authenticated and must be described by a subject. A subject must be associated with an access control context, i.e. every action happens as if a subject is running it. At the same time, a set of security policies must be defined in order to determine whether access is allowed or not. These are all taken care of by the *Add Authentication Enforcer* transformation in the previous step. The *Add Authorization Enforcer* transformation creates a component to check the permissions and provide the decision.

7.2 Mechanics

A developer specifies how to get an authenticated subject, the set of policies of a system, the action that needs to be authorized, and the authorization strategy.

The transformation composes classes of a security framework (JAAS, Java 2 Security API, .NET etc) to create a component with the specified authorization strategy, and wraps the action so that an access control decision is made before the action is performed.

7.3 Example

Authorization decision for Java based systems could be made by using JAAS principal based policy files or using the permission hierarchy of Java 2 security API. Let us suppose that Alice wants to use the JAAS based authorization strategy.

Alice replaces the `AuthorizationEnforcer` created by the *Policy Enforcement Point* transformation with a group of classes that perform JAAS based authorization. Figure 3 shows the steps of authorization and the classes involved. In our example, Alice specifies the login context (`LoginContext` class created in the previous step), authorization strategy (JAAS principal based authorization), the set of policies (location of policy file to be used by the JAAS framework), and the action that needs to be secured (a portion of the code). The *Add Authorization Enforcer* transformation replaces the `AuthorizationEnforcer` class with a new instance. This `AuthorizationEnforcer` retrieves the `Subject` from the `LoginContext`, creates an instance of `PrivilegedAction` and moves the portion of code to secure, and invokes the `doAsPrivileged` method of the `Subject` to perform the `PrivilegedAction`. The `doAsPrivileged` method uses JAAS framework to make the authorization decision and runs the `PrivilegedAction` if authorization is successful. The JAAS framework classes use the policy specified by Alice to make the decision. The transformation creates a default implementation based on user specification; a developer might have to customize further to meet application requirements.

8 *Add Perimeter Filter* Transformation

You want to scan and check incoming data for malicious content before it is used in the system.

Add a policy enforcement point at the system entry point to validate data.

8.1 Motivation

For every application, there are some input validation policies that are common for all inputs. Writing checks for these policies for every input variable duplicates a lot of code. One way of eliminating duplication is to move the checks at the application entry point inside a policy enforcement point component.

However, there are policies that are either input-specific or require knowledge of the business logic; these policies cannot be factored out.

8.2 Mechanics

A developer specifies where to insert a perimeter filter and what policies to apply.

The transformation creates a policy validation component [13] that contains the filters implementing input validation policies. The program transformation adds a request at the entry point to delegate input validation to the secure base action component which then passes the input through the filters to validate the input.

8.3 Example

In this step, Alice applies the *Add Perimeter Filter* transformation to extend the default implementation of `PerimeterFilter` class. She selects a list of input validation policies and the program transformation adds filters that implement the policies to the `PerimeterFilter` class. They can be added as a `Collection`, or by embellishing the `PerimeterFilter` class with a Decorator [5].

9 *Decorated Filter* Transformation

You want to apply multiple policies to an input variable.

Validate an input variable by decorating it with a series of filters implementing the policies.

9.1 Motivation

Programmers determine a comfort zone [12] of inputs and write checks to prevent inputs outside the zone. Mistakes in writing checks is very common. In the first week of September 2008, Bugtraq lists 34 SQL injection and 21 XSS attack instances that could be solved by more stringent input validation.

Replacing manual checking with automated tools increases programmer efficiency because they can concentrate on policies rather than the mechanism of implementing checks.

9.2 Mechanics

A developer specifies the target input variable and the policies to be applied.

The program transformation adds the policies to an input variable by using Decorators [5]. Figure 4 describes how a string variable is decorated with policies that remove SQL injection attack vectors. The string input variable becomes encapsulated in the abstract `AbstractStringContainer` class. Its concrete instance `UnsafeString` becomes the target of input validation policies.

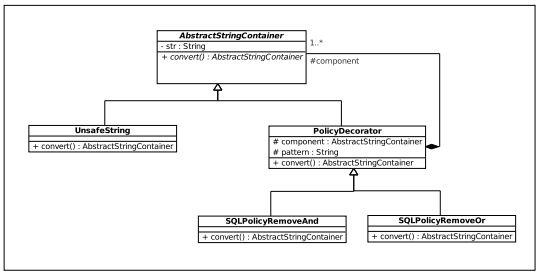

Figure 4. Class diagram describing policies to apply on an unsafe string

9.3 Example

Some policies are input specific. Alice applies these policies on the particular instances of variables. She chooses the filters from a list of implemented policies for each input variable. The policies for SQL injection and LDAP injection are different. The program transformation adds the policies to an input variable by using Decorators [5]. Figure 4 describes how a string variable is decorated with policies that remove SQL injection attack vectors. The string input variable becomes encapsulated in the abstract `AbstractStringContainer` class. Its concrete instance `UnsafeString` becomes the target of input validation policies.

10 Implementation Issues

We envision tools for security-oriented program transformations, available as part of commercial IDEs. They could

be built using the same underlying structure as refactoring tools. None of the security-oriented program transformations described in this paper require deep analysis of the program being transformed. They share many structural similarities with refactorings and can probably be implemented similarly.

The tools for security-oriented program transformations to add authentication and authorization components automatically generate a default implementation based on a developer's specification. A developer would have to customize the implementation to fit the application specific requirements. It is important to balance the amount of specification required to generate the default implementation so that too much behavioral specification is not required. An implementation with minimal specification would be easier to generate but it would require further customization. Nevertheless, using a tool to generate a default implementation eliminates programming errors and non-standard use of authentication and authorization API.

Possibly the easiest way to implement some of our transformations in tools is to use aspect-orientation [11]. Logging and weaving in method calls to new components are the poster children of AOP, so our transformations such as the *Add ** transformations can be implemented with AOP relatively easily. However, there are other transformations for which AOP does not offer immediate solutions. Simple scripts or transformations based on abstract syntax tree manipulation are other implementation alternatives for such tools. All these transformations can probably be implemented by a code transformation language such as TXL [2] and Stratego/XT [15], or a framework for implementing refactorings.

All the program transformations are platform dependent, at least to the extent that they depend upon the language of the program being transformed, and perhaps on the operating system or the libraries. Frameworks for refactoring are also platform dependent, as are most of the languages for writing program transformations.

It is unreasonable to expect researchers to build such tools for every platform. Building a tool for a single application is usually not cost effective, so it is unreasonable to expect application developers to do it. This will need to be done by platform and tool vendors, who can amortize the cost over many users.

11 Conclusion

This paper shows how to systematically apply multiple program transformations to secure a system's perimeter. However, it is impossible to ensure that a system is secured by these six transformations; in fact it is impossible to ensure that any number of program transformations will secure a system. Nevertheless, security-oriented program transformations allow developers to think about security systematically and in a more structured manner.

Acknowledgement

Thanks to Paul Adamczyk, Farhana Ashraf, Nicholas Chen, David Garlan, Carl Gunter, Sam Kamin, Darko Marinov and Roger Whitney for providing valuable feedback. We also thank US Department of Energy for their financial support under the grant number DE-FG02-06ER25752.

References

[1] S. Christey and R. Martin. Vulnerability type distributions in CVE, version 1.1. `http://cwe.mitre.org/documents/vuln-trends/index.html`, May 2007.

[2] J. Cordy, C. Halpern-Hamu, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. In *ICCL*, pages 280–285. IEEE, 1988.

[3] A. Eden and R. Kazman. Architecture, design, implementation. In *Proceedings of the 25th International Conference on Software Engineering (ICSE-03)*, pages 149–159, Piscataway, NJ, May 3–10 2003. IEEE Computer Society.

[4] M. Fowler. *Refactoring: Improving The Design of Existing Code*. Addison-Wesley, Jun 1999.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[6] M. Hafiz. Security oriented program transformations (Or how to add security on demand). In *OOPSLA '08: Companion to the 23rd annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, New York, NY, USA, 2008. ACM.

[7] M. Hafiz, P. Adamczyk, and R. Johnson. Organizing security patterns. *IEEE Software*, 24(4):52–60, Jul/Aug 2007.

[8] M. Hafiz, P. Adamczyk, and R. Johnson. A catalog of security-oriented program transformations. In *Submitted to ECOOP 2009*, 2009.

[9] M. Hafiz, P. Adamczyk, and R. Johnson. Systematically eradicating data injection attacks using security-oriented program transformations. In *Proceedings of the International Symposium on Engineering Secure Software and Systems (ESSoS-09)*, Feb. 4–6 2009.

[10] J. Kerievsky. *Refactoring to Patterns*. Addison Wesley, 2004.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. L. Chris Maeda, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programmin (ECOOP) 1997*, pages 220–242, 1997.

[12] M. Rinard. Living in the comfort zone. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 611–622, 2007.

[13] C. Steel, R. Nagappan, and R. Lai. *Core Security Patterns : Best Practices and Strategies for J2EE(TM), Web Services, and Identity Management*. Prentice Hall PTR, Oct 2005.

[14] J. Viega and G. McGraw. *Building Secure Software: How to Avoid Security Problems The Right Way*. Addison-Wesley, 2002.

[15] E. Visser. Stratego: A language for program transformation based on rewriting strategies. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *LNCS*, pages 357–361. Springer-Verlag, May 2001.