

*– I don't see how you in ten pages can do the whole thing completely wrong!*

– Barry Pekilis



# Introduction

Software is a crucial part of the modern infrastructure on which we all rely, and, therefore, it must be reliable, robust, correct and be able to evolve over time with our changing needs. Ensuring these properties for the massive amounts of software in use is considered one of the grand challenges in computer science. This social and technical challenge is often referred to as “dependable systems evolution” [Som00], “the software maintenance challenge” [Art88], “the software crisis” [DT96] and “trustworthy computing” [MdVHC03].

Better tools and techniques for processing and manipulating software are likely to be part of any solution to this challenge. Development of software processing tools and techniques is studied in the field of program transformation [PS83]. Many results from this field have proven to be highly applicable for software evolution. A frequently encountered drawback, however, is that implementations of program transformation and analysis techniques are often language-specific; they tend to be tied to the front-end or grammar they were written against, even when the underlying algorithms are general. This significantly impairs reuse of transformation code and systems.

This dissertation addresses the reuse limitation by introducing novel techniques for constructing reusable, language-independent program analyses and transformations. The proposed techniques include a versatile approach for easily plugging transformation systems into existing language infrastructures, such as compilers, and a declarative, aspect-based approach for software practitioners to express transformation programs for language families, rather than just for a single language. With these techniques in hand, the dissertation demonstrates how automatic software maintenance tasks can be increasingly expressed in a reusable manner. Case studies illustrate their applicability to encoding of architecture and design rules as executable program analyses, expressing control- and data-flow transformations, and interactive code generation of unit tests from user-written axioms.

## 1.1 Software Evolution

Software maintenance and evolution is by far the most expensive and time-consuming part of the software life-cycle [Pfl05]. The trend during the last 30 years shows that maintenance is an increasing part of the total software cost. Reports from the 1970s suggest that 60-70% of the total cost went into maintenance and evolution [ZSJG79]. In the 1980s, this figure crept closer to 70% [McK84] and, during the 1990s, it reached around 90% [Moa90, Erl00]. About 50% of the maintenance time is spent understanding the existing software [FH83].

Organisations with an investment in software are perhaps affected by this fact the most when they need to effect substantial changes. The sheer size of the code bases make radical changes and redesigns prohibitively and increasingly expensive. Ulrich [Ulr90] estimated that 120 billion lines of code was maintained in 1990. In 2000, the number was at 250 billion lines according to Sommerville [Som00]. Estimates by Müller suggest that the doubling happens around every 7 years [MWT94].

Software is becoming a limiting factor for progress in all kinds of organisations. To escape this situation, software needs to be constructed differently, and in ways which make it possible for small teams of programmers to understand, maintain and change large projects with millions of lines of code. Large parts of maintenance need to be done with (semi-) automatic software processing tools. Automation is key, but automation cannot work until the substrate being processed, the software, is easily managed by the tools. This means inventing better techniques for analysing and transforming large code bases.

## 1.2 Program Transformation

The field of program transformation is concerned with developing theories, tools and techniques for the analysis and transformation of programs. Typical applications in this field include transformation of programs to improve a certain metric such as execution speed, class cohesion or memory footprint; translation between languages, e.g. compilation, code generation and interpretation; analysis and verification of program properties such as absence of deadlocks, information leakage or buffer overflows. Each of these examples constitutes a transformation problem or a transformation task. A fuller discussion of program transformation is given in Chapter 2.

Program transformation techniques aid in the development of robust language infrastructures which in turn provide the basic components required for all forms of language processing. On top of these infrastructures, scalable analyses and transformations have been realised for many problems such as searching for code defects and security vulnerabilities. These analyses can handle multimillion line projects. How-

ever, while these analyses and transformations generally consist of algorithms and data types that are language independent, their implementation are usually specific to a given infrastructure. This makes them very difficult to reuse across different infrastructures, even for the same language. Presently, they are only accessible by a handful of specialists and have not gained widespread acceptance. This effectively reduces reuse of both knowledge and tools, and seriously lessens the promise of program transformations as an approach for dependable software evolution.

This dissertation focuses on:

- methods for constructing versatile program transformation environments which aid developers in implementing reusable, language-independent transformation programs;
- how to express transformation programs, and how to design transformation languages such that transformations can become reusable across subject languages and between transformation tasks;
- how to capture subject language constructs, and other entities found in software, using transformation functions and abstract data types in the transformation language; and
- how to manage these transformation functions and data types so that they are convenient to use by programmers of transformation programs.

This work reuses and expands upon promising techniques that encourage language independence and reuse of transformations. The paradigm of strategic programming has a central part in this dissertation.

### 1.2.1 Strategic Programming

Strategic programming [VB98, Vis99, LVV03] is a generic programming technique for processing tree- and graph-like object structures. The technique separates two concerns: object transformations and traversal schemes. Strategies are built using traversal combinators and provide complete control for expressing generic traversal schemes. These strategies are parametrised with transformations that are responsible for supplying the problem-specific transformations.

This separation is a particularly powerful approach for building reusable program transformations. The strategies can be reused across transformation problems and subject languages, whereas the transformation parameters, expressed as rewrite rules, are used to adapt the generic strategies to a particular language and problem.

Relatively few programming languages have been built with strategic programming in mind. One example of a “strategic” language is Stratego [BKVV06], a

domain-specific language for program transformation based on a sub-paradigm of strategic programming called strategic term rewriting.

In (strategic) term rewriting approaches to program transformation, programs are described as terms which in most respects may be considered analogous to trees. Using terms and rewriting allows the succinct expression of many transformation problems, but the terms are sometimes also a limitation. The choice of model used to describe programs in a given transformation system has consequences for which transformation tasks that system is best applicable to.

### 1.3 Program Models

The effectiveness and applicability of a software transformation system depends to a large extent on how its underlying program model has been formulated. The model determines which transformation tasks will be easy and which will be difficult or impossible. Particularly, the “abstractness” of the representation determines which analyses and transformations are possible – if the model is too abstract, refactoring is not possible, and if the model is too detailed, many analyses become too expensive.

Common representations include Prolog-style fact databases, relational databases, various forms of graphs, lists of tokens and concrete syntax trees. All of these are discussed in Chapter 2. One representation, which is noteworthy because it relates very closely to the representation of programs as terms, is the abstract syntax tree.

#### Abstract Syntax Trees

Abstract syntax trees (ASTs) contain the essence of programs. They are a minimal and precise form of syntax trees (sometimes called parse trees). Syntax trees are constructed by parsing the source code text. The resulting tree contains all the lexical tokens of the original source code, possibly also including whitespaces, represented as a tree according to a<sup>1</sup> subject language grammar.

For most transformation and analyses tasks, both the tokens and whitespaces are redundant. Stripping them away is desirable, for efficiency reasons. This stripping yields an AST which contains the essence of the original textual representation<sup>2</sup>.

The AST has numerous appealing advantages:

- it is a high-level, as opposed to machine-level, representation;

---

<sup>1</sup>A previous version of this manuscript erroneously used the definite article here. As Peter Mosses kindly pointed out, multiple variants (implementations) of a language grammar usually exist. Furthermore, it is desirable to keep the AST interface decoupled from the underlying grammar as much as possible, so that clients to the AST API are insulated from incidental (implementation-specific) grammar changes.

<sup>2</sup>McCarthy, the father of Lisp, is generally credited with inventing the term AST.

- ASTs capture the essence of the language;
- everything in the source code that contributes to the executed program is in the AST;
- using maximally shared, directed acyclic graphs, ASTs can be stored and exchanged very efficiently [vdBdJKO00];
- there are a number of established techniques for augmenting ASTs with extra information such as layout, line number information and traceability.

For these reasons, most of the examples in this dissertation will revolve around ASTs — since an AST captures the essence of a subject language, abstracting over languages implies abstracting over ASTs. ASTs also have their limitations. Some of these will be addressed in Chapter 7 where strategic graph rewriting is discussed. It is important to keep in mind that the techniques developed herein are not bound to just ASTs; most will work for any tree or graph-like structures which may be arbitrarily more or less abstract than ASTs.

## 1.4 Language Abstractions for Program Transformations

The strategic programming paradigm is an attractive starting point for expressing reusable, language-independent transformations. This paradigm, and in particular strategic term rewriting, provides an attractive level of genericity in the formulation of transformation programs. Certain obstacles remain, however, many of which are shared with other approaches to program transformation. These must be addressed if substantially better levels of reuse and language independence are to be achieved.

One of these limitations is the inability of transformation systems to abstract over its program model implementation. It would be attractive to separate the transformation engine logic from the program model representation. It should be complemented with a versatile technique for adapting transformation engines to external program models. This would make it possible to combine transformation engines with any software development framework that provides a suitable program model.

Another limitation is the severely restricted ability of modern transformation systems to cope with cross-cutting concerns in transformation programs. Related to this is the ability to adapt existing transformation programs to new subject languages, or to changing program models.

A final limitation, particular to strategic term rewriting, is the poor support for program models that are graph-like in nature, such as program flow graphs.

The strategic programming paradigm has been extended in this work to address the above limitations using the following abstractions:

**Program Object Model Adapters** A program object model (POM) adapter is a technique for abstracting over implementation details of the program model in a given language infrastructure. The transformation system is written against the POM adapter interface. It is a minimal interface for navigating and manipulating tree and graph structures. By supplying infrastructure-specific adapters that translate operations on this interface to operations on the internal object model, transformation engines can be freely reused across language infrastructures, e.g. across compiler front-ends. A notable feature of the technique is that the majority of the adapter code can be automatically generated by analysing the object model interface of the language infrastructure.

**Aspects** Aspects extend the strategic programming paradigm with a general approach to capturing cross-cutting concerns and deal with properties such as traceability, type checking and unanticipated extensibility. Using aspects, it becomes easier to express generic transformation algorithm skeletons and to adapt these to specific program object models and to specific subject languages.

**References** References provide an extension to the strategic term rewriting paradigm for rewriting on graph-like structures. This allows the strategic term rewriting machinery to be applied to computing on control- and data flow graphs. References provide a way to turn some global-to-local rewriting transformations into local-to-local.

It should be noted that these abstractions can be recast for other transformations languages and programming language paradigms. This will be discussed in the respective chapters.

It must also be noted that the field of software verification and validation, which is also an important direction for dependable systems evolution, largely falls outside the focus of this thesis. Software verification and validation typically uses abstract models of the underlying software. These models are partially or fully extracted from the existing software using a variety of different tools. The techniques and tools described in this dissertation can thus complement these approaches.

### 1.4.1 Extensible Languages

When expressing program transformations, one needs to handle domain abstractions with cross-cutting properties such as scoping rules, variable bindings and state propagation. The behaviour of these domain abstractions may be very complex. While manipulating domain abstractions using functions and abstract data types is possible, it is often notationally inconvenient. They frequently exhibit a cross-cutting nature which results in cross-cutting concerns in the transformation program.

In some cases, these concerns can be handled using techniques borrowed from aspect-oriented programming. By extending the transformation language with support for aspects, one can modularise some of the cross-cutting concerns arising from domain-abstractions into libraries. However, not all cross-cutting concerns are expressible in aspect-languages and many that are suffer from complicated notations. Some of the proposed abstractions, such as the ones providing graph rewriting, are therefore realised as active libraries [VG98]. Libraries in this form can interact with the compiler to provide detailed, library-specific error messages when the abstractions are misused and may also come with library-specific optimisations and notation.

Active libraries with notation extend the host transformation language with new language constructs. Each new library thus becomes a small domain-specific embedded language (DSEL). Those libraries with cross-cutting properties are termed domain-specific aspect languages (DSALs). The extensible transformation language framework called MetaStratego supports both forms of language extensions. The framework allows Stratego developers to implement their own active transformation libraries. To a certain extent, MetaStratego follows the approach to language extension described in [Vis05b].

## 1.5 Method

The method employed for arriving at each of the results in this dissertation followed a simple, four step process:

1. *Identify Problem* – A specific limitation preventing language independence or reusability was identified.
2. *Formulate Solution* – An analysis was conducted to describe the characteristics of the problem, and then a design was formulated which sought to solve it.
3. *Implement Solution* – The formulated solution was implemented as a computer program. In some cases, this led to language extensions, in other cases, it led to transformation libraries or new infrastructure.
4. *Demonstrate Applicability* – One or more prototype applications demonstrating the applicability of the implemented solution were constructed.

This process has been applied to each the proposed abstractions presented herein.

## 1.6 Contributions

The contributions of this dissertation are:

- a novel technique for plugging transformations into arbitrary language infrastructures;
- a novel extension of the strategic programming paradigm with aspects for handling cross-cutting concerns;
- demonstrating how aspects can be used to adapt strategies and rule sets after-the-fact, i.e. grey box reuse;
- a novel extension of the strategic programming paradigm for graph structures;
- the construction of a modern, interactive development environment for development of and experimentation with interactive strategic programming;
- a state-of-the-art survey of design and architectural features found in contemporary program transformation systems;
- the design and implementation of an infrastructure for an extensible program transformation language;
- a validation of the proposed techniques and abstractions through the construction of several prototypes:
  - a language extensions for alerts;
  - an interactive development environment for Stratego;
  - a compiler scripting for framework-checking; and
  - an interactive generator of unit tests from axioms of algebraic specifications.

## 1.7 Outline

This dissertation is divided into five parts, as follows.

1. *Software Transformation Systems* – provides background material from the field of program transformation. This introduction chapter is in part based on the paper *Stratego: A Programming Language for Program Transformation* [Kal06]. Chapter 2 gives a detailed discussion of the state-of-the-art in software transformation system design and architectural features, with a focus on the capabilities for language independence. In Chapter 3, the basic notions from universal algebra and term rewriting are given along with a formulation of the System S calculus for strategic term rewriting. The Stratego language is an implementation of the System S calculus.

2. *Abstractions for Language Independence* – contains the main contributions of this dissertation. Chapter 4 introduces the program object model adapter technique and shows how it allows plugging transformation systems into existing language infrastructures. This enables large-scale reuse of entire transformation environments. The chapter is based on the paper *Fusing a Transformation Language with an Open Compiler* [KV07a] written with Eelco Visser. In Chapter 5, a language extension for capturing cross-cutting concerns in strategic programming languages is introduced based on the paper *Combining Aspect-Oriented and Strategic Programming* [KV05] written with Eelco Visser. The chapter describes a flexible and declarative technique for adapting and extending general transformation algorithm skeletons to specific problems and subject languages.
3. *Supportive Abstractions for Transformations* – provides additional abstractions which augment the main abstractions proposed in the previous section. Chapter 6 introduces the Stratego programming language and MetaStratego, an extensible variant Stratego language and its compiler infrastructure. This chapter is partly based on *Stratego/XT 0.16. A Language and Toolset for Program Transformation* [BKVV07] and *Stratego/XT 0.16: Components for Transformation Systems* [BKVV06], both written with Martin Bravenboer, Rob Vermaas and Eelco Visser. The MetaStratego infrastructure forms the basis for all the language abstractions proposed in this dissertation. Chapter 7 shows an extension to Stratego that supports a particular form of graph rewriting and motivates its use by computations on control flow graphs. It is based on the paper *Strategic Graph Rewriting: Transforming and Traversing Terms with References* [KV06] written with Eelco Visser. This extension allows strategic term rewriting techniques to be applied to other program models than (syntax) trees.
4. *Case Studies* – discusses several prototypes where the abstractions from the previous parts have been tested in practise. Chapter 8 gives an application of the language extension techniques explored in this dissertation to a domain-specific aspect language for mouldable failure handling. It is based on the paper *DSAL = library+notation: Program Transformation for Domain-Specific Aspect Languages* [BK06] written with Anya Bagge, but the alert extension was first explored in *Stayin' Alert: Moulding Failure and Exceptions to Your Needs* [BDHK06] written with Anya Bagge, Valentin David and Magne Haveraen. This chapter is included to demonstrate that the language extension techniques employed in this dissertation are more generally applicable. Chapter 9 introduces an interactive development environment for (Meta)Stratego called Spoofox, based on the paper *Spoofox: An Extensible, Interactive Development Environment for Program Transformation with Stratego/XT* [KV07b] written with Eelco Visser. Parts of the Spoofox infrastructure have served as a testbed

for many of the other case studies. Chapter 10 demonstrates the applicability of the proposed abstractions with a case study demonstrating how the Stratego transformation system may easily be plugged into an existing development framework for Java. This allows library-specific analyses and transformation to be written by developers of Java frameworks and libraries. Chapter 11 shows how the transformation infrastructure and language abstraction may be applied to interactive program generation. A code generator for unit tests from axioms is presented, based on a testing methodology proposed by Magne Haverdaen.

5. *Conclusion* – contains some general reflections over language-independence as well as the concluding remarks. Chapter 12 is devoted to a summary and general discussion of the results obtained in this work. Chapter 13 discusses further work. Chapter 15 summarises. Chapter 14 contains the conclusion.

## 1.8 Summary

Dependable software evolution is one of the grand challenges in computer science. Automating maintenance tasks is one key way to tackling this challenge. Program transformation provides scalable and robust techniques for automatic maintenance, but is hindered by poor reuse and language-dependence. This dissertation claims that better reuse and language-independence can be found by abstracting over program models and by using aspects to adapt transformation algorithms to specific subject languages and program models. The rest of this dissertation serves to substantiate this claim.