A LANGUAGE-INDEPENDENT STATIC CHECKING SYSTEM FOR CODING CONVENTIONS

SARAH MOUNT

A thesis submitted in partial fulfilment of the requirements of the
University of Wolverhampton for the degree of Doctor of Philosophy

2013

This work or any part thereof has not previously been presented in any form to the University or to any other body whether for the purposes of assessment, publication or for any other purpose (unless otherwise indicated). Save for any express acknowledgements, references and/or bibliographies cited in the work, I confirm that the intellectual content of the work is the result of my own efforts and of no other person.

Signature: . . . .

Date: . . . .

ABSTRACT

Despite decades of research aiming to ameliorate the difficulties of creating software, programming still remains an error-prone task. Much work in Computer Science deals with the problem of specification, or *writing the right program*, rather than the complementary problem of implementation, or *writing the program right*. However, many desirable software properties (such as portability) are obtained via adherence to coding standards, and therefore fall outside the remit of formal specification and automatic verification. Moreover, code inspections and manual detection of standards violations are time consuming.

To address these issues, this thesis describes **Exstatic**, a novel framework for the static detection of coding standards violations. Unlike many other static checkers Exstatic can be used to examine code in a variety of languages, including program code, in-line documentation, markup languages and so on. This means that checkable coding standards adhered to by a particular project or institution can be handled by a single tool. Consequently, a major challenge in the design of Exstatic has been to invent a way of representing code from a variety of source languages. Therefore, this thesis describes **ICODE**, which is an intermediate language suitable for representing code from a number of different programming paradigms. To substantiate the claim that ICODE is a universal intermediate language, a proof strategy has been developed: for a number of different programming paradigms (imperative, declarative, etc.), a proof is constructed to show that semantics-preserving translation exists from an exemplar language (such as IMP or PCF) to ICODE.

The usefulness of Exstatic has been demonstrated by the implementation of a number of static analysers for different languages. This includes a checker for technical documentation written in Javadoc which validates documents against the Sun Microsystems (now Oracle) Coding Conventions and a checker for HTML pages against a site-specific standard. A third system is targeted at a variant of the Python language, written by the author, called python-csp, based on Hoare's Communicating Sequential Processes.

ACKNOWLEDGEMENTS

Most of all I owe an enormous debt of gratitude to Professor Bob Newman, for his thoughtful supervision, good friendship, and seemingly endless faith in my work, despite much evidence to the contrary. My second supervisor, Dr Rob Low, has also been a great help, especially with the work on ICODE.

Professor Alan Mycroft was an advisor in the early stages of the work. More importantly Alan graciously agreed to allow me onto the Computer Science Tripos and sparked my interest in programming languages and tools. Less formally, Dr Steve Lakin and Dr Elena Gaura both sat in on supervision meetings at Coventry. Recently Dr Chris Dennett and Dr Mohammad Hammoudeh have commented on drafts of this thesis, provided much good company and kept me (relatively) sane at work.

Work on `python-csp` was supported by Nuffield Undergraduate Research Bursary URB/37018 taken up by Sam Wilson, who contributed the Jython port of `python-csp` and work on the C implementation of channels.

My partner, John, has been endlessly supportive and tolerant of long hours spent working, which could have been spent with him. I am also grateful to my parents and to Ian Moody and David Carpenter, who have all cajoled me into finishing.

CONTENTS

5 Imperative languages in ICODE 87

6 Applying Exstatic to `python-csp` applications 99

List of Tables

LIST OF FIGURES

LISTINGS

CHAPTER 1

INTRODUCTION

Although the term software "crisis" became current in the late 1960s [29], programmers have been commenting on the need to spend a large proportion of their time debugging, rather than designing or writing code, since the early days of stored-program computers. In his memoirs, Maurice Wilkes recalls the first debugging session on the then brand new EDSAC machine:

> "*As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.*" [116]

In more recent times software has become increasingly large and complex (as hardware provides increasingly more resources) and it is still apparent that debugging is not only personally frustrating, but also costly. A 2002 NIST study [109] of the economic impact of software testing on U.S. industry found that software developers spend an average of $70 - 80\%$ of their time on testing and debugging and that the average time taken to fix a bug is 17.4 hours. The study estimated that debugging activity costs the American economy \$59.5billion, per annum and that "feasible" improvements in testing infrastructures could reduce this cost to only \$22billion (the Halting Problem prevents the total eradication of the cost). The NIST report recommends two sorts of "feasible" improvement:

- Earlier detection of bugs, moving error detection closer to the point of error introduction; and
- locating the cause of bugs more precisely and quickly.

From human factors in programming to formal methods, many areas of research in Computer Science aim to reduce the numbers of bugs in code. The *static* analysis of programs is one approach to addressing the issues raised by Tassey's NIST study [109], and has an important place in the suite of tools a programmer can use. This is partly due to the place that static analysis occupies in the production cycle (i.e. it is used during development when the programmer is still well acquainted with the code semantics) and also due to its automation (static analysis does not require extra staffing, as some forms of testing do).

Good programming practice is colloquially divided into two activities: writing the *right program*, and writing the *program right*. The former refers to program specification and work in the problem domain. The latter refers to good practice in the task of programming itself. Software errors naturally concern writing the right program, however, writing the program right also endeavours to produce code which is readable, maintainable and portable. In fact, the distinction between the two is blurred. For example, it may be a convention in a particular software project that "any lock obtained on a variable must be released within the same scope as it has been acquired". A program within the project which fails to adhere to this convention may still be semantically correct and exhibit no bugs at runtime. However, since acquiring and releasing locks in different name spaces (perhaps not even defined within the same program file on disk) is likely to be confusing for developers, violating the convention is likely to cause a fault during implementation or maintenance. Readability, maintainability and portability are qualities which rarely affect the end user but ease the job of developers, which frees their time to concentrate on program behaviour. These issues are surprisingly costly. Siy and Votta [102] have analysed code inspections and found that the majority (60%) of coding errors uncovered during inspections are errors in coding standards, style and readability, rather than errors in program behaviour:

> "... [code inspections] *improve the maintainability of the code by making the code conform to coding standards, minimising redundancies, improving language proficiency, improving safety and portability, and raising the quality of the documentation.*"

The thesis of this dissertation is that a syntax-directed static analysis system, capable of checking project-specific violations in a variety of source languages will be of benefit to software projects.

The remainder of this Chapter clarifies the phrase "writing the program right" and provides motivation for the thesis statement. The Chapter ends with a list of the research contributions made by this work and a synopsis of the rest of the thesis.

1.1 Writing the program right

Whilst much theory (e.g. [43, 44, 67, 68, 70]) and many tools exist to enable programmers to write a coherent specification and verify that their program meets it (e.g. [42, 54, 69, 103]), very few tools exist to ensure that programs are written right.

"Writing the program right" refers to issues of style, design and best practice. It might seem that these issues are trivial. Surely, a competent programmer knows about best practice and can follow a set of coding standards? Moreover, such bugs must be "shallow" and unlikely to cause serious errors, unlike the deeper bugs which are caused by semantic errors in algorithm design and caught by sophisticated program analyses – such as those described in graduate texts in static analysis [82]. Should it not be sufficient to test programs dynamically? This last criticism is important – the end user is *only* concerned with the runtime behaviour of a program, so writing the program right may seem irrelevant compared to the job of dynamically testing code. On the other hand, dynamic testing has several drawbacks. It is often difficult for testing to exercise every control-flow path of a program, making it more likely that bugs will go unnoticed. Also a dynamic test may uncover a bug in one place in the code, where in fact the bug appears in several places. Dynamic testing can be computationally expensive for some programs and practically impossible for others – GUIs, concurrent or distributed programs often need to be run as a whole in testing, and cannot be broken into convenient "units" for separate testing. Static analyses, however, can be performed on programs relating to any application domain and consider all source code in its entirety. Also, the argument given previously, that violations of coding conventions may catch bugs *before* they appear by ensuring that software is readable and, therefore, easy to maintain is one which cannot be countered by dynamic techniques.

Other evidence to support the need for adherence to coding conventions can be found both in discussions within the community of professional developers and in the literature on static checking. A brief look at the former (which is purely anecdotal) serves to give a clear understanding of the sort of problems which are encountered in industry which are the subject of this thesis.

TheDailyWTF[1] is a weblog and Internet forum where professional programmers share "war stories" about the code they maintain. Many such stories involve very obvious mistakes which are simple to check for statically, such as the canonical macro which redefines truth, or similarly the following (C) code[2]:

```
enum Bool
{
```

---

[1] http://www.thedailywtf.com/

[2] http://www.thedailywtf.com/forums/thread/80084.aspx

```
    True,
    False,
    FileNotFound
}
```

Listing 1.1: Redefining truth in a C `enum`

which one might be surprised to find in commercial, production code. However, there are many more insidious bugs which cause significant semantic errors and could be eliminated by checking for adherence to simple coding conventions. Martin Sandin gave an example[3], from a commercial PL/SQL program, on Lambda the Ultimate[4], a forum devoted to programming language research:

```
function IsValidUserLogin(user:string,
                          password:string):bool
begin
  result = select {*} from USERS
           where USER_NAME=user and PASSWORD=password;
  return not is_empty(result);
end
```

Listing 1.2: Case sensitivity in PL/SQL

The bug here springs from PL's case insensitivity. The phrase `PASSWORD=password` intuitively selects users from a database with the particular password which is a parameter to the function, but actually selects users with any password. Sandin's post said:

> "*This passed unnoticed for several months on a low-volume production system, and no harm came of it. But it is a nasty bug,*

---

[3] `http://lambda-the-ultimate.org/node/1114`

[4] `http://lambda-the-ultimate.org/`

> *sprung from case insensitivity, coding conventions, and the way humans read code. The lesson for me was that: Things that are the same should look the same.*"

One could add that developers who are accustomed to writing in modern, case sensitive languages will always be at a disadvantage when working with languages which are case insensitive, only allow short variable names, and so on. Such bugs can be eliminated by very simple coding conventions and checks. In the example above, the appropriate convention to adopt is that in any `where` clause the left hand side of a = and the right hand side should differ by more than just case.

The literature on static checking (discussed in detail in Chapter 2) contains a number of tools (such as Smatch, FindBugs [47] and Metal [32]) which check for simple invariants which can be specified by the user. Some coding standards are well known. For example, the convention that one should always put literals on the left hand side of a comparison in order to avoid assignment-in-guard errors, e.g.:

```
if (0 == x) // Literal on lhs
if (x == 0) // Literal on rhs (breaches convention).
if (x = 0)  // Assignment in guard (potential error).
```

Listing 1.3: Conventions for using `==`

This simple case is interesting as coding conventions are used here to turn a runtime error into an error which can be caught by the static analysis phases of a compiler or interpreter. By ensuring that simple comparisons in `if` statements conform to the pattern `<literal>==<name>` any typo which turns `==` into `=` will result in a compile-time error, such as this:

```
LiteralAssign.java:4: unexpected type
required: variable
found   : value
    0 = a;
    ^
1 error
```

Listing 1.4: Result of an assignment where the lvalue is a literal (Java).

Other patterns are less well known and may be specific to languages, libraries or application domains. In particular, [47] contains a discussion of a number of simple bug patterns[5] in Java. One such example involves overriding methods inherited from Java's canonical base class, `java.lang.Object`. The `equals(Object):boolean` method tests whether the argument object is equal to `this` object[6]. Hovemeyer and Pugh [47] identify a bug pattern known as "covariant equals" where programmers attempt to override the method in `Object` like this:

```
class Foo {
    public boolean equals(Foo object) {
        ...
    }
}
```

Listing 1.5: Covariant equals bug in Java

Here, the type `Foo` in the argument list should read `Object`. At runtime, code which calls the method `foo.equals(bar)` (where `foo` is an instance of

---

[5] One can think of "bug patterns" as inverse coding conventions, where the convention is "never use this pattern".

[6] Called `self` in some object oriented languages.

`Foo` and `bar` is any object) will in fact be calling the `.equals` method defined in `java.lang.Object`. In almost every case the programmer will be intending to call the method defined above in `Foo` and the resulting runtime error may not be simple to diagnose.

This bug is particularly insidious because it looks so close to the correct definition. Despite the simplicity of this bug, and the ease with which it can be (statically) checked for, Hovemeyer and Pugh still found instances of it in production code - one in the Eclipse IDE version 2.1.0, 4 in GNU Classpath version 0.06, and 13 in rt.jar (Sun's implementation of the APIs for J2SE) from Sun JDK 1.5.0, build 18. Presumably in all of these instances, the "bug" had not triggered a runtime error in any test code.

1.2 Presuppositions

The work presented in this thesis is predicated on four presuppositions, which are discussed in more detail below:

1. Writing a program that meets its specification is only part of good program implementation;
2. almost all code is written in more than one language;
3. the earlier an error is found, the cheaper it is to fix;
4. even expert programmers are prone to making "silly mistakes" and a tool for detecting these would be useful.

1.2.1 Meeting a specification is part of good programming practice

“Good” code not only meets its specification, but adheres to style guidelines (e.g. [89, 104, 111]), is idiomatic with respect to the language it’s written in and meets project-specific static invariants. This is what is meant by “writing the program right”.

The meta-level compilation project [41] has also addressed this last issue, and gives informal examples of such invariants: [32]:

- “access to variable `a` must be guarded by lock `b`”;
- “system calls must check user pointers for validity before using them”;
- “message handlers should free their buffers as quickly as possible to allow greater parallelism”.

The discussion above has also other examples from production code.

1.2.2 Computing languages

Almost all code is written in more than one language, although not all of these languages will be general-purpose or Turing complete. For example, almost all C programs have an associated `make` file, many programs have user- or developer-documentation written in a mark-up language (Texinfo, HyperText Markup Language (HTML), eXtensible HyperText Markup Language (XHTML), etc.), many come with shell scripts, many web applications are a mixture of program code and marked-up documents, and so on.

It makes sense to apply a static checker to any and all the source files in a project which have useful coding standards associated with them, in order to identify as many errors as possible. This line of reasoning implies that a checker should, ideally, be able to analyse code written in any computing language. It is not immediately clear whether this is possible, or how such a checker should be built. The majority of this thesis addresses these questions.

1.2.3 The earlier an error is found, the cheaper it is to correct

Boehm in [13] demonstrated that bugs which are found earlier in the development cycle are cheaper to correct. Maguire [62] differentiates between **one-step** and **two-step** tools and techniques for error detection. Two-step techniques, such as testing, detect errors in their first step (i.e. running the tests), then require effort from the programmer to locate the error in the original source code. More convenient one-step techniques (such as static checking and manual inspection) locate errors *in situ*, while the programmer is editing the code. This is often done by adding a static analysis system into an integrated development environment, where it can be run continuously while the programmer is typing.

1.2.4 Experts make mistakes

The fourth presupposition requires some justification. Intuitively, it would seem that programmers' expertise grows with their experience. Adelson [3] has shown experimentally that expert programmers process programs in larger "chunks" of (human) memory - that is, they abstract more concrete syntax into their mental models of code. Adelson's later work confirmed that expert programmers create "mental sets" (abstract models) of code ("what the program does") and novices create concrete sets ("how a program functions"). In [4], she describes a series of experiments where novices and experts were asked a question about code written in PPL (Polymorphic Programming Language) and a flowchart describing that code. Two groups of subjects were tested, novices and experts, who were students and lecturers from the Harvard introductory course on programming. Four sets of data were gathered:

- appropriate set conditions, immediate response;
- inappropriate set conditions, immediate response;
- appropriate set conditions, delayed response;
- inappropriate set conditions, delayed response.

In the appropriate set experiments, subjects either saw an abstract flowchart and were asked an abstract question, or they saw a concrete flowchart and were asked a concrete question. In the inappropriate set experiments, the level of abstraction of the flowcharts did not match the level of abstraction of the question. In the delayed response groups, subjects were given a 1.5 minute distracter task (solving a Rubik's cube) between seeing programs and flowcharts and answering questions.

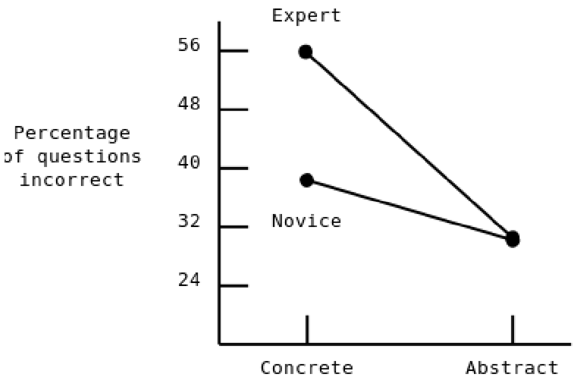


Figure 1.1: Adelson's results: appropriate set conditions for the delayed group, from [4]

The results for the appropriate set conditions, delayed response group are particularly interesting. Figure 1.1 shows that novice subjects outperformed experts subjects on the concrete questions and both groups performed equally well on the abstract questions. Adelson [4] notes that:

> *"It is striking to find novices surpassing experts; however, that fact is important only in so far as it points out what the problem representations of the novice and the expert are during program comprehension. . . .*
> *The results of these experiments do not suggest that expert programmers lose the ability to attend to the details of the program. (In the appropriate set conditions in which there is no delay, we see that the Experts are quite good at attending to detail.) Rather, they suggest that experts have learned that during comprehension of this type of program, paying attention to the abstract elements of the program is more important than pay-*

> *ing attention to the low-level details. ... This point can hold even for non-Algol and/or very high-level languages in which the distinction between what is abstract and concrete may not map onto the distinction between what the program does and how it functions.*" [4]

In the context of this work, Adelson's results confirm the hypothesis that even expert programmers may be prone to overlooking simple errors in code, since their preference is to represent programs in abstract mental sets.

1.2.5 Exstatic

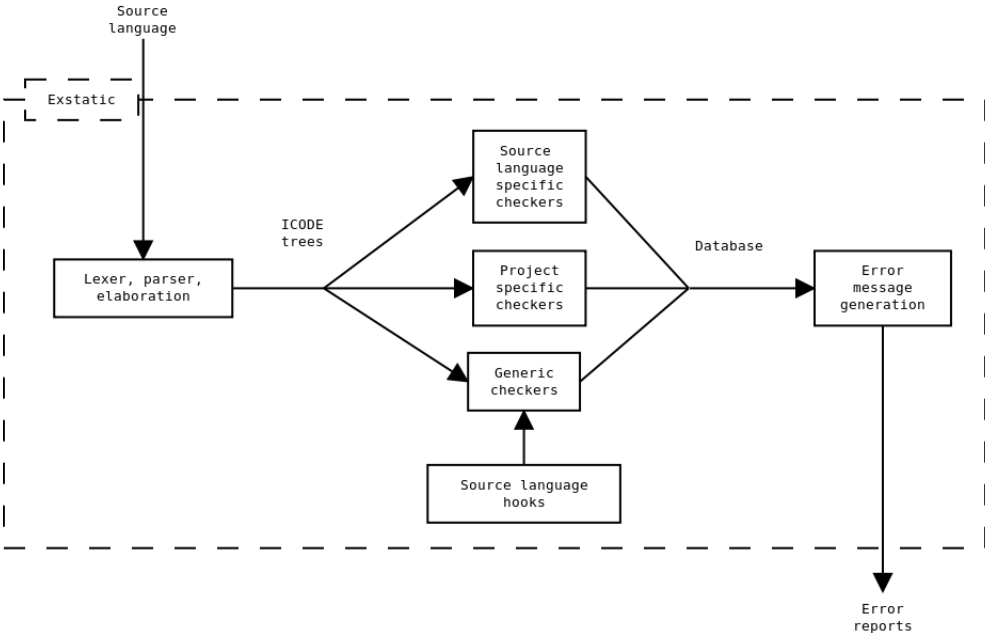


Figure 1.2: Overview of the Exstatic system

Part of the practical contribution of this work is a framework for identifying user-defined, project-specific violations of coding conventions via static analysis. Whilst some prior research has addressed this problem (e.g. [32, 47]), this work is novel in that it attempts to identify errors in many different source languages, which means that the code and supporting files for a project can conveniently be checked by the same tool. Such an approach requires a novel intermediate format, which can be used to represent code from different language paradigms.

This thesis describes a tool called **Exstatic**, in which users can translate source code into a novel intermediate language (**ICODE**) and write routines to check the resulting ICODE for errors. In the language of the thesis title, this framework is *language-independent* in the sense that it can represent source code from many different computing languages and *extensible* in the sense that it can be adapted to check for new error types. The advantage of this approach is that users can choose exactly what information in their programs is represented in the checker, which errors are detected and how such detection is implemented. This means that entire projects can be checked for various error types, including many that would otherwise be missed by traditional static checkers.

Figure 1.2 broadly describes the composition of Exstatic. Users of the tool need to write (or obtain) a parser which converts their source language(s) into the common intermediate format, ICODE. From then on the user can either take advantage of other checkers which have been written for the language, or indeed generic checkers which may be applicable to many languages, or write their own. Exstatic then provides a common framework for organising resulting error messages and warnings and presenting them to the user.

Exstatic provides the programmer with the following features:

- an intermediate representation which can be used to represent code from a disparate number of different source languages,
- facilities (in Python) to simply "walk" the intermediate representation using the widely known visitor pattern [36],
- facilities to represent violations of coding conventions in a common format, suitable for use with a continuous integration tool or similar,
- a set of simple scripts to aggregate and chart the results running Exstatic (examples can be seen in Chapter 6).

A programmer wishing to apply Exstatic to a new project, would first draw up a set of coding conventions they wish to follow, then determine whether a parser had already been written to convert the relevant source languages to ICODE. If such a parser does not exist, they would write one, either building on the tools discussed in Chapter 6, or by writing a parser from scratch. They would then look to see whether the existing checkers which run on ICODE will check the coding conventions for their own project. Many coding conventions will be useful in a variety of source languages and projects, and one advantage of using ICODE is that these static analysers will not have to be re-implemented for each given source language. If no such analyser exists then one will need to be written from scratch, as discussed, with examples, in Chapter 6.

These techniques are applicable to a range of projects, written in a variety of source languages. Exstatic is intended for use when coding conventions are relatively straight forward to test for and can be implemented over an abstract syntax tree. Complex analyses such as type checking, or techniques which require whole-program analysis, control-flow graphs, partial evaluation and so on are better suited to the more “heavy weight” frameworks found in compilers and language specific static checkers. Examples of the sorts of coding conventions that Exstatic may be used for include:

- If a class name ends in `Model` it should not import anything from `javax.swing`, `java.awt` or use `System.{in,out,err}`
- Never use `==` with a `float` or `double` type
- Every (Java) class, attribute and method should be preceded by a Javadoc comment, with the exception of “getters” and “setters”
- All HTML pages in a website should include “breadcrumb” navigation with correct hyperlinks (See Appendix C).
- Every `python-csp` process should have a “readset” and “writeset” documented and the relevant channels should be used as documented within the process (see Chapter 6).

One criticism of Exstatic may be that the user has to contribute an amount of work in order to use the tool, and if that entails writing a parser for a complex language this may be asking the user to invest more time in preparing the tool than would be saved by using it. In Chapter 6 this criticism will be explored in the context of a practical example which examines code from a real software project.

1.2.6 Thesis contributions

This thesis explores the use of static analysis to aid the programmer in adhering to project-specific coding conventions. In particular this thesis contributes:

1. the framework Exstatic, used to combine checks for violations of coding conventions in a variety of source languages;
2. an intermediate format ICODE and its concrete representation as an XML language. This thesis makes the claim that source languages from a number of popular programming paradigms can be represented in ICODE and the majority of this thesis addresses this claim;
3. a proof strategy which can be used to determine whether an exemplar language, with a given denotational semantics, can be represented in ICODE, preserving its semantics;
4. proofs that the languages PCF [88] and IMP [117] can be translated to ICODE, preserving their semantics;
5. three static analysis systems have been implemented, which demonstrate how Exstatic can be used in realistic software development sce-

narios:

(a) Exstatic has been applied to `python-csp`, an example open source project, implemented by the author, which provides message-passing concurrency to the Python programming language. Coding conventions which are exercised by Exstatic are intended to help the programmer prevent deadlocks.

(b) a system to check for violations of the Sun Microsystems (now Oracle) [86] coding conventions for Javadoc documentation.

(c) a system for checking HTML files from a specific website for adherence to the conventions of that site. For example, consistent presentation of site navigation, application of a site template, etc.

1.2.7 Synopsis

Chapter 2 is a survey of related work in the area of program correctness and static analysis. A variety of approaches to checking the validity of programs are broadly surveyed, while static analysis techniques are covered in more depth.

Chapter 3 describes ICODE, the novel intermediate format used by Exstatic. ICODE is compared with existing intermediate formats for compilers and other programming tools. The importance of ICODE in relation to the thesis statement presented on Page 3 is that ICODE can be used to represent a variety of languages from different paradigms. A general proof strategy which may be used to show that an exemplar language with a well-defined (denotational) semantics may be translated into ICODE, preserving its semantics.

Chapters 4–5 apply the proof strategy from the preceding Chapter 3 to a number of well-known languages which are exemplars of different paradigms. Chapter 4 gives a proof that the functional language PCF [88] can be translated into ICODE preserving its semantics and Chapter 5 with the imperative language IMP [117].

Chapter 6 is an account of the practical application of Exstatic, which includes a description of how a source code analyser can be built with Exstatic. At this point in the dissertation, the claims of the thesis statement relating to the generality and extensibility of Exstatic have been examined. However, it is also important to support the claim that Exstatic is useful in detecting violations of project-specific conventions in real programming projects. Chapter 6 describes a realisation of Hoare's Communicating Sequential Processes [43] as a library for the popular programming language, Python [72]. This is an open source project, designed and implemented by the author, which has some interesting coding conventions which arise from the addition of message-passing and non-deterministic selection to a language which, by design, only has support for shared-memory concurrency and coroutines. Chapter 6 describes coding conventions for `python-csp`, an instance of Exstatic that detects violations of these and describes the results of applying Exstatic to the code base of `python-csp`.

Chapter 7 contains conclusions to the work presented and an overview of directions for further work. The thesis statement is revisited and evidence to support that statement is evaluated. Preceding Chapters also contain brief summaries and conclusions.

Appendix C expands on the work discussed in Chapter 6. Two published papers [77, 76] are reproduced, which describe a checker for technical documentation written in Javadoc which validates documents against the Sun Microsystems (now Oracle) Coding Conventions, and a checker for HTML pages against a site-specific standard. The implementation of these Exstatic systems and the substantial description of them in the papers was provided by the author. The co-authors of the two papers were members of the supervisory team, or advisers.

CHAPTER 2

RELATED WORK

This Chapter gives a survey of the variety of techniques for dealing with the detection of errors in program code (Section 2.1) and static analysers in particular (Section 2.2). The Chapter concludes in Section 2.3 with a direct comparison between Exstatic and the variety of approaches surveyed here, with particular attention to checkers for violations of coding conventions, checkers where the user may write new checks and systems which can deal with more than one source language.

2.1 The variety of techniques for dealing with program correctness

The idea of automatically proving a program correct is laudable, but constrained by the Halting Problem, one corollary of which is that determining whether or not a program will exhibit any run-time errors is, in general, undecidable. However, a large number of techniques have been discovered and invented to aid the programmer in the task of generating correct code. These may be characterised as *sound* or *unsound* and *complete* or *incomplete*. Where a sound static analysis produces judgements about a program, they will always be correct. Where a complete static analysis makes a judgement,

	Sound and incomplete	Unsound and incomplete
Static techniques	Automatic or manual proof of correctness. Type and effect systems.	Static checkers such as `lint` or ESC/Java2.
Dynamic techniques	Design by Contract Assertions.	Model checking. Unit testing. Black-box testing. White-box testing.

Table 2.1: Commonly used techniques for determining the correctness of a program.

it will make a judgement on all cases relating to its analysis. Colloquially, programmers say that "a sound jury will never set a guilty person free, but a complete jury will never send an innocent person to jail".

Table 2.1 shows a small number of popular techniques characterised in this way. Some techniques in the Table have more ambiguous characters than the Table implies, for example a type checking system can be unsound, if implemented as such. Formal proof does not always provide such a strong guarantee as one might hope. Clarke's paper [22] gives an introduction to the completeness of Hoare logic and similar systems. He proves that it is impossible to give a sound and complete verification system for certain combinations of language constructs, for example those languages combining first-class procedures, recursion, static scope, global variables and internal procedures as parameters of procedure calls.

For the working programmer, other characteristics may also be important,

```
$\{n \geq 0 \wedge N = n\}$
fact=1                $\{P_1\}$
while(n>0):           $\{P_2\}$
  fact = fact * n
  n = n - 1           $\{P_3\}$
$\{fact = N!\}$

$P_1 \equiv \{n \geq 0 \wedge N = n \wedge fact = 1\}$
$P_2 \equiv \{N \geq n \geq 0 \wedge fact = \frac{N!}{n!}\}$
$P_3 \equiv \{N \geq n \geq 0 \wedge fact = \frac{N!}{(n-1)!}\}$
```

Listing 2.1: Factorial function annotated with Floyd-Hoare proof obligations.

such as ease of use, integration with IDEs and other tools, active developer support and so on. Ease of use, in particular, is difficult to quantify and yet of considerable importance in the application of these techniques. Formal methods, whether machine directed or not, are often perceived to be complex, difficult and expensive and rarely used beyond safety critical systems. This is partly due to practical constraints – unlike most products errant software does not need to be "recalled", instead updates can be made available via the World Wide Web. This makes the cost of finding errors in production software relatively cheap to fix. In contrast, in fields such as CPU design the cost of product recall is extremely high and formal methods are more commonly used. It is also likely that fewer programmers are well trained in formal methods and reading formal specifications, proof obligations (see Listing 2.1) and so on are unfamiliar to many software engineers as are tools such as model checkers [46] and theorem provers.

Static analysis tools have the advantage of not requiring any special expertise to use. These tools are generally automatic and are used in the same way a compiler is used and so are a familiar part of the programming toolchain. They do have disadvantages though. One is that static analysis tools may produce a large number of false-positive results, which will discourage the user from reviewing the results carefully. Most tools have some way to control this output, either with command line switches, configuration files or annotations in the source code, embedded in comments. Dynamic approaches, which mainly involve testing, usually require the programmer to write extra code to exercise some part of the program, with a given input, and determine whether the output is as expected, or to "step-through" the program as it runs, inspecting the stack, heap or other runtime data. One of the most systematic presentations of testing is Unit Testing, where each program is broken into a small number of units to be exercised separately (see Listing 2.2 for an example). The hope is that this systematic strategy for developing tests will increase the likelihood that the majority of the program will be exercised.

```
import unittest
import factorial

class Factorial_Test(unittest.TestCase):
    def test_zero(self):
        self.assertEquals(1, factorial(0))
    def test_pos(self):
        for datum in data :
            self.assertEquals(datum[1],
                                   factorial(datum[0]))
    def test_neg(self):
        self.assertRaises(ValueError, factorial, -100)
    def test_tyerr(self):
        self.assertRaises(ValueError,
                          factorial,
                          ["A", "list"])
if __name__ == '__main__':
    # Automatically run all test cases.
    unittest.main()
```

Listing 2.2: Unit testing for the factorial function, in Python.

Other dynamic approaches include include "trace" analysers, which check executing code for "unsafe" operations, such as buffer overruns, which may lead security violations. An approach which has gained currency in recent years is Necula's "proof carrying code" [78] where a (binary, or bytecode) program is shipped with a formal specification of its runtime behaviour. The runtime environment then checks the executing code against the proof that it has "carried" and may abort the program if it veers from its specification.

Hybrid approaches are also possible, although not widely used. For example in [18] the authors consider the problem of writing a "correct" compiler. This is a difficult problem [45] and much work (outside the scope of this thesis) has been done towards the goal of producing fully verified compilers. It may be possible to prove correct some, or even all, of the algorithms used in the compiler, but if the code itself does not implement those algorithms correctly then the proof adds no value to the user of the compiler. On the other hand, writing a direct proof of the code may be able to show that *this* individual version of the code is “correct”, but if the source or target language of the compiler were to change, if any new optimisations or analyses were added to the compiler then the proof would have to be re-written. In [118] an alternative approach was used: various test cases were written as input to the compiler and the compiler was used to provide its output in the target language. Proofs were then generated “on-the-fly” to determine that the input to the compiler and the resulting target code were semantically equivalent. This technique is robust against change to the source code of the compiler, and with a large number of test cases considerable confidence in the compiler can be gained.

2.2 Static analysers

The preceding Section has given an overview of a range of different techniques for examining the correctness of code. The remainder of this Chapter deals with static analysers in more depth.

A large number of static checkers have been written, some of which have been published in the academic literature and many which have not. For example, the programming language Python boasts three widely used checkers which are broadly similar in scope and design: PyLint [110], PyChecker [83] and PyFlakes [48] and a number of less widely used scripts (e.g. [92, 95]). Moreover, static checkers are often written in an *ad-hoc* manner, using scripting languages, UNIX utilities (`grep`, `awk`, `sed`, etc.), or other light-weight tools. In his book Code Complete [64, pp506], Steve McConnell describes such an *ad-hoc* system:

> *"Microsoft planned to include a new font technology in a release of its Windows graphical environment. Since both the font data files and the software to display the fonts were new, errors could have arisen from either the data or the software. Microsoft developers wrote several custom tools to check for errors in the data files, which improved their ability to discriminate between font data errors and software errors."* [64]

One of the purposes of this work is to provide an homogeneous framework in which such checks may be written and shared.

Several authors have observed that one can divide static checking systems into two categories: **bug-checkers** which detect possible semantic errors and **style-checkers** which detect violations of coding conventions. However, this is something of a false dichotomy as the distinction between a semantic error and a style violation can be thin and many coding patterns may be seen as both potential bugs and errors of style. For example, in some widely used imperative languages (such as C) the following code has a legal semantics (where `=` means assignment):

```
if (x = 0) { ... }
```

Many checkers would flag this as a potentially problematic piece of code. On the one hand the code seems erroneous, as if the author intended to write a comparison `if (x == 0) { ... }` and, of course, if an assignment expression returns the value just assigned and zero values are "false" then the `if` block will never be executed, making this an "unreachable code" error. On the other hand, the code above may be legal and portable, but the `if` statement has a side-effect which may be confusing to readers and violates the "one statement per line" convention which is widely believed to be an important part of producing self-documenting code[1].

Although the work of this thesis is intended to contribute largely to the latter category of style-checkers, this Section is not structured by this particular distinction between static checkers. Instead the focus of the Section is on the question of how to build a static checker which is not only pragmatically helpful (in the sense of finding bugs) but also adaptable, so that users can build in checks for custom bug patterns or violations of coding standards, which are the subject of this thesis. The first two following Subsections deal with the traditional classes of static checkers whose design is based around that of other tools: compilers and theorem provers. The third Section discusses tools which are designed to be extensible. These are generally light-weight (meaning they don't employ sophisticated analyses, such as interprocedural analysis) and based around simple models of interaction (such as finite state automata, OOP design patterns, and so on).

---

[1] A third point of view would argue that this is an issue of poor syntax design on the part of the language implementer and that `:=` is a preferably way of denoting assignment compared to overloading `=` which most people would recognise as an equality comparator from school-level mathematics.

2.2.1 Compiler-like tools

This section examines static checkers which are designed in certain ways like compilers – they are monolithic, difficult for novices to extend and can perform some (algorithmically) sophisticated analyses on source- or bytecode.

2.2.1.1 `lint`

`lint` [25, 52] is one of the earliest and perhaps still the best-known static checker and now has many variants on different platforms and environments. It was built by Johnson, in 1978, as a UNIX[2] utility, intended to complement his portable compiler [53], for the C programming language [56]. Since memory and CPU time were at a premium, Johnson split-off some of the static analyses that would ordinarily be conducted by the compiler into a separate tool. This meant that compilation would be fast and efficient, concentrating only on code generation; whereas `lint` could afford to be unsound and incomplete, since its analyses were not part of a mission critical compilation process. As hardware has become cheaper and faster, later compilers, such as `gcc`[3], have integrated the role of `lint` back into the compilation process.

Johnson also intended that users could separate their implementation cycle into two parts: firstly, they would concentrate on the architectural design of their programs, and produce a tested and debugged executable; secondly they would run `lint` on their code and use its output to retrofit

---

[2] UNIX is a trademark of the X/Open Company Ltd.

[3] http://www.gnu.org/gcc/

portability and (perhaps) good style.

`lint` is also notable for introducing the notion of annotations, embedded in comments, used to prevent the checker from producing spurious errors. For example:

```
/* NOTREACHED */
```

can be used to flag code which is (intentionally) never reached and so suppress a false-positive warning about unreachable code.

2.2.1.2 PREfix

PREfix [17] is a static checker for C and C++ programs, sold by Intrinsa Inc. PREfix extracts models of individual functions via **simulation**, which involves traversing the call-graph of the program (from the leaves to the root) and using a virtual machine to trace execution paths within functions and determine the effect of each operator or function call. By examining the memory of the virtual machine, program errors can be detected. Function models can be used to determine the effect of function calls, during simulation. Models for basic operating system functions are provided with PREfix.

This technique was based on the approach of the SELECT checker [14], which traced all paths through functions, constructing predicate models for each path and using these to generate test cases and perform formal verification. In contrast to SELECT, one of the strategies of PREfix is to only examine those control-flow paths which are possible; thus reducing simulation time and the volume of spurious errors. This also differs from `lint`'s approach. Johnson [52] says:

> "*. . . information about. . . unused variables and functions can occasionally serve to discover bugs; if a function does a necessary job, and is never called, something is wrong!*" [52]

The authors of PREfix used the Purify [50] debugger as a use model, which they claim is "striking in its ability to find obscure errors quickly in a large mass of code".

The authors were motivated to make this design decision by their belief that the software industry is resistant to the sort of methodological change required to adopt specification or annotation checkers, such as those described below. In addition to this, PREfix's usability is improved by a set of tools used to store warnings in an SQL database, display them along with contextual information in a web browser, filter, order and summarise warnings, and so on.

PREfix is also notable for being one of the few checkers reviewed whose authors provide a detailed quantitative assessment of its abilities. The following is a brief summary of their results:

- The time to parse and simulate code was typically between two and five times the build time. The most striking example presented was the code for Mozilla[4], which took ten hours to parse and simulate 540,613 lines of code.

---

[4] Mozilla is an Open Source version of the Netscape web browser, available at http://www.mozilla.org/

- The number of warnings PREfix produced (some of which are duplicate) range from 0.2 to 10 warnings/KLOC in commercial code and from 0.5 warnings/KLOC to 5 warnings/KLOC in Open Source code is interesting. It may give some weight to the idea that Open Source code benefits from being read by many developers with diverse backgrounds and experience [91]. On the other hand, this may suggest that the errors found by PREfix can be found by human developers, given enough time or staff.
- The number of false-positive warnings generated by PREfix varied between 10% and approximately 25% in the Open Source code tested, with similar results reported for commercial code.

2.2.1.3 Clean++

Clean++ is a checker for the **C++ Constraint Expression Language** (CCEL) [30]. Users write constraints which can be embedded into their code or stored separately. The system parses C++ source, and stores the results of this analysis in a database. CCEL constraints are then converted to database queries. If a query has a non-null result a constraint has been violated and is reported back to the user.

2.2.2 Static analysers based on theorem proving

The Curry-Howard isomorphism implies that theorem proving and compilation are complementary activities – one can consider the successful compilation of a program to be a formal proof that there exists at least one legal translation of the source program into object code. However, theorem provers aim to be generic reasoning tools and have a reputation for being esoteric and difficult to use [12]. Several static checking systems have, however, made use of an underlying theorem prover to automate reasoning over programs. Users may annotate source code with either pre- and postconditions (as in ESC) or with a domain specific language for program annotation (as in Splint/LCLint) which aids the generation of theorems and may suppress false-positive warnings.

2.2.2.1 ESC

The **Extended Static Checking** [26] project has produced checkers for Modula-3 and Java code, which make use of a purpose-built automatic theorem prover, Simplify [79] which acts on a simple specification language.

Users add specifications to functions, which consist of a precondition, a list of variables the function may modify and a postcondition. Verification conditions are generated from these and the theorem prover is applied. This approach is similar to Meyers' **design by contract** [66] for object-oriented languages (such as Eiffel [65]), where methods contract to ensure that pre- and postconditions will hold.

In the Java version of ESC, the source program is translated into an intermediate representation, based on Dijkstra's guarded commands [80] before verification conditions are generated. In the latest, Haskell, version of

ESC, Haskell itself is used as the language with which to specify pre- and postconditions.

2.2.2.2 Splint

Splint (formerly called LCLint) [33, 34] is a static checking tool for ANSI C, based on Larch [39] formal specification technology. Programmers may use Splint as a fully automatic tool (using the `-weak` switch) or as a specification checker (by providing LCL specifications, separate to their code), but it is mainly used as an annotation checker, with embedded annotations in the style of `lint`. Various command-line switches can be used to control which error types are reported on. The authors advocate the use of Splint both in implementing a new program and maintaining an old one. In the latter case, Splint can initially be run with the `-weak` switch and stronger checks can be added incrementally. The Splint manual [35] recommends that the following properties should be checked in order:

1. stricter type checking;
2. concrete access to abstract types;
3. use-before-definition, unreachable code, etc;
4. null dereferences;
5. macros;
6. memory management;
7. aliasing;

8. function interfaces;
9. buffer sizes;

In addition to these and other built-in checking routines, users may define their own checks and program annotations, using a custom language. This enables users to define annotations and specify how they change during program execution.

2.2.2.3 Aspect

The Aspect analyser [51] detects bugs in CLU [59] procedures. Programmers write partial specifications in a custom annotation language, which relate "aspects" of a result object to "aspects" of argument objects. If a result object is calculated without making use of its specified dependencies, Aspect reports an error.

Internally, Aspect represents procedures as flows from aspects in the pre-state to aspects in the post-state. The flow specification is the minimum required flow and the view specification is an partial ordering on aspects, which is used to determine which aspects in the pre-state may be substituted for one another. Both flow and view specifications are provided by the user. Bugs are reported when the actual flow does not include the minimum flow. Importantly, every error reported is guaranteed to be a genuine bug (up to the correctness of specifications).

The author claims that Aspects checking time is typically linear in the length of programs and in the worst case quadratic in the number of aspects.

2.2.2.4 Meta-level compilation

Meta-level compilation [19] employs state machine based analyses of applicative structures, where each analysis is defined by a compiler extension written in a high-level state machine definition language, **metal** [32]. These extensions are then loaded into an extensible compiler, **xg++**.

2.2.3 Light-weight, extensible tools

Several tools now exist which are designed to be easily extensible by users. This is perhaps motivated by the rising popularity of modern OOP languages, where "everything" is an object and the internals of the compiler and runtime are exposed, thus allowing a meta-programming approach to the design of static checkers. Many such systems use the Visitor pattern [36] to iterate over a parse tree representation of the source- or bytecode representation of the program.

2.2.3.1 Ctool

Tom Lord's *Ctool* (pronounced *cool*) [60] is a C parser coupled with a Scheme interpreter, with which users can construct static checkers. The built-in Scheme interpreter includes facilities that ease the job of manipulating syntax trees. The only interface Ctool provides is a read-eval-print loop (REPL), which reads Scheme expressions, evaluates them and prints a result. Practically, this mix of very low-level (imperative) programming and very high-level (functional) programming is interesting, and comes from the MIT tradition, where operating systems in both C and Lisp were once very popular. However, it seems unlikely that practical, particularly commercial programmers, will be familiar with both imperative and functional schools of thought[5].

2.2.4 Style checkers

CheckStyle [16], FindBugs [47] and PMD [2] are similar in that they check Java source (CheckStyle, PMD) code and bytecode (FindBugs) for stylistic and semantic errors and are extensible via the Visitor pattern [36]. All three can be embedded in commonly used IDEs such as Eclipse. Unlike compiler-like tools, they do not perform "deep" analyses of code, but instead look for deviations from common programming practice and in spite of these have been successful in finding many bugs in mature code bases such as GNU classpath and JBoss [47]. These simple analyses may still uncover significant bugs. The previous Chapter described how Hovemeyer and Pugh still found one instance of the "covariant equals" bug in the Eclipse IDE version 2.1.0, 4 in GNU Classpath version 0.06, and 13 in rt.jar (Sun's implementation of the APIs for J2SE) from Sun JDK 1.5.0, build 18.

---

[5] This may however change as an increasing number of object-oriented languages are incorporating ideas from functional programming, particularly Ruby and Python.

2.2.5 Checkers which consider more than one source language

There are a small number of tools similar to Exstatic, in the sense of being able to check a number of different source languages. These fall into two categories: some [49, 115] are syntax-directed and come with a number of parsers which convert source languages into a common intermediate format and others [100] are lexical. Syntax-directed systems such as RATS [49] are written by companies or small open source teams who define the intermediate format of the tool and write parsers which target it. Exstatic takes a slightly different approach and provides an XML schema for the intermediate format, which means that users of the system are able to write new parsers themselves. This makes it simple to check for bugs in systems which the developer of Exstatic has not catered for. Lexical tools such as [95] are intended to check each line of code, essentially with regular expressions, or YASCA [100], have a plug-in structure which makes it possible to use syntax-directed tools and have the output presented in a homogeneous manner. Exstatic is able to do this too, in Chapter 6 Exstatic is shown to interface to the pylint checker. pylint is called and its output is formatted in the common format Exstatic uses for error messages and the user is presented with error reports from both tools at once.

2.3 Conclusions

The work described in this thesis is concerned with checking for violations of user-specified conventions. Users will have to interact directly with the Exstatic framework and so it is important for pragmatic reasons that users find themselves dealing with a tool that is similar to other widely used development systems. Exstatic therefore errs away from using formal methods directly (although users may choose to implement static analyses which are based on formal methods). Instead, Exstatic makes use of technologies that are similar to common programming tasks, and therefore familiar to many developers. The intermediate representation is concretely represented as an XML tree which can be "walked" over in the same way as any other XML document. External tools and checkers can be called and their output integrated with the output of user-written checks from Exstatic.

More generally, the novelty of the work presented in this thesis comes from the following features:

- several tools are available which check code in a variety of source languages, Exstatic is unusual in allowing the user to write parsers for new languages themselves;
- other tools have intermediate formats which can represent code written in a number of source languages (Chapter 3 has a detailed discussion of this), but this work goes further in proving that languages from many commonly used programming paradigms can be translated to ICODE,

in a manner which preserves their semantics.

CHAPTER 3

ICODE: AN INTERMEDIATE LANGUAGE

The previous Chapter surveyed related work in bug-finding techniques, and static analysis in particular. The Chapter ended with a summary of the attributes of Exstatic which make it novel, the most important of which is an intermediate format (ICODE) which can describe languages from a variety of programming paradigms and a proof that the format will preserve the semantics of a number of example languages in those paradigms. This Chapter begins with a survey of other intermediate formats which are "universal" in various senses and proceeds to describe ICODE and the proof strategy which is used to support the claim that ICODE is a universal intermediate format for static analysis.

3.1 Related work: Universal languages

Universal languages have a long history dating back to discussions on UNCOL in the 1950s and 60s [23, 105, 106, 107, 108]. This original dialogue concerned the pace at which new computers and languages were being adopted. The desire was to allow programmers to write code in programming languages appropriate to the problem domain to be executed on any machine architecture. The difficulty of compiler writing was to be ameliorated by providing a universal intermediate format which would be independent of both source and target languages (thus for n languages and m machines $m > 1$, $n + m$ compilers would be required, rather than $n.m$). A common intermediate format was never defined or developed and the UNCOL project was hampered by its own ambition. To succeed, the UNCOL team would have had to invent a universal character set as well as to be among the first to invent or adopt now-common compiler techniques such as bootstrapping [61].

Since the UNCOL project ended several more attempts at universal intermediate languages have been made. The text that follows contains a discussion of these successors to UNCOL and a comparison between these formats and ICODE. As will be seen, there is a major difference in intention: UNCOL was explicitly intended to enable compilation to target machine code, whereas ICODE is intended to enable (often simple) semantic analyses. This leads to a difference in level of abstraction. We also discuss other forms of universal language, in particular the use of systems programming languages as intermediate formats and universal languages in other contexts.

3.1.1 Universal languages as intermediate compiler formats

There exists a plethora of intermediate languages for compilers, many of which claim to be able to represent code in more than one language. Well known examples (apart from UNCOL) include OCODE [93] and INTCODE

[94] from the BCPL toolchain, P-code [81], Janus [40] and Microsoft's Common Intermediate Language (CIL) [31, 38]. Brandis [15] gives an overview of the design space of ILs used by optimising compilers, including different representations for control-flow and data-flow.

CIL [31] is remarkable for being a fully-implemented universal IL which is contemporaneously used by a community of developers who work in a wide variety of languages. Translations exist from C♯, COBOL, C, FORTRAN, Eiffel, Oberon and others. CIL differs considerably from ICODE because it is specifically designed to facilitate the compilation of efficient code. Consequently, CIL contains a number of features specifically aimed at code optimisation (for example, a tail-call instruction to aid tail-call optimisation in functional languages). CIL has a large instruction set (over 200 instructions) and is intended to be human-readable. A short example follows, and should make sense to anyone with a familiarity with object oriented programming and assembler:

```
.method public static void Main() cil managed {
    .entrypoint
    .maxstack 8
    lstr "Hello World!"
    call void [mscorlib]System.Console::WriteLine(string)
    ret
}
```

As the example suggests, CIL is at a similar level of abstraction to JVM code [58], although it differs considerably by providing facilities for polymorphic typing.

ICODE is similar to the **Stanford University Intermediate Format** (SUIF), which is part of a compiler infrastructure project, also called SUIF [5]. SUIF (the IL) is an inheritance hierarchy of tree node types, which are extensible via annotations. ICODE differs from SUIF in two respects. Firstly, ICODE is simpler in that it has far fewer type constructors. Consequently, ICODE is less prescriptive about the syntax of various constructs and avoids duplication (for example, SUIF has separate constructors for `if` statements and `select` expressions). Also, ICODE doesn't have primitive support for fundamental types (for example, SUIF has `ArrayReferenceExpressions`s and `MultiDimArrayExpressions`), these are to be dealt with in ICODE annotations. Secondly, SUIF is an object hierarchy, which limits the languages it can be represented in (without "kludges") to object-oriented languages. ICODE has an XML representation and can be generated or manipulated by any language which has facilities for parsing and representing XML.

3.1.2 General purpose programming languages as universal languages

An alternative to using intermediate languages for compilation, static checking or code distribution is to use a general purpose programming language (such as Lisp or C) or their runtime environments (such as the JVM [58]). One advantage of this approach (for portability) is that popular languages are often well supported on a wide variety of platforms. This means that compiling a given source language for multiple platforms merely requires cross-compilers for the chosen intermediate language. However, if the semantics of the source language and the "intermediate" language are vastly different, or the source language contains features missing in the intermediate language, compiler writing can be difficult. For this reason, this approach is usually implemented with C as the intermediate language. Since C provides a number of features (such as weak typing and user-defined pointers) it is tractable to use C to emulate language features that are available in higher-level languages.

As an intermediate language for the static checking of coding conventions, general purpose programming languages are inconvenient. Usually such languages have large grammars (in terms of the number of productions available) and so writing a checking algorithm to traverse such a representation becomes long-winded and time consuming for the user. Secondly, since some language constructs may be emulated in the intermediate language (for example, if C is chosen it does not have exceptions or objects) then the user must trust that the translation to intermediate code is faithful and exact. Lastly, it must be clear to the user how source language constructs map to the intermediate language. If some constructs are emulated then this may be non-trivial.

3.1.3 Universal languages in other contexts

In other contexts the word "universal" may have connotations outside the realm of static analysis and compilation, but several such languages have been proposed. UML [1] is a diagrammatic modeling language used to specify and describe (object oriented) programs at a number of levels of abstraction. For example, *use-case* diagrams describe what a program should do from the point of view of a user, whereas *class diagrams* should describe what types should be contained in each class. UML is universal in the sense that it is largely independent of programming language although stubs can be compiled from UML specifications. It is interesting to note that although UML is “universal” in a sense, its use can tightly constrain the later choice of implementation language. For example, a design which uses multiple inheritance can not be directly implemented in Java.

ANDF [61, 84] is a language- and machine-neutral distribution format aimed at portability, which has similarities to compiler intermediate languages. However, ANDF never became popular, and TDF was its first and only candidate distribution format.

Ryu and Ramsey [97] describe a debugger which is retargettable to source languages, with `lcc` and MiniJava as example targets. Universality (in this sense) is achieved by implementing a compile time support library which compiler writers can use to interface their intermediate structures with the debugger, without having to redesign the compiler.

3.1.4 Comparison with ICODE

Since ICODE is not intended to be included in a compiler which generates target code it need not be as concrete as those intermediate languages described above. Moreover, ICODE is designed *without* knowing what sort of static analysis a user might want to implement for it. Therefore, it is desirable to be able to encode any sort of program text (including comments and whitespace) in ICODE. For this reason, the design of ICODE is such that it provides a range of syntactic structures (such as arithmetic expressions or name spaces) which can be used to represent a number of different semantic structures in various languages and requires the user to annotate the program representation to distinguish between similar structures (such as the various sorts of name space that appear in a given language).

3.2 The design of ICODE

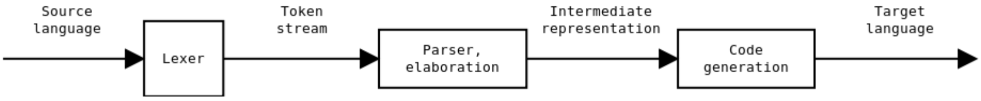


Figure 3.1: Coarse-grained modularisation of a compiler

Intermediate languages (ILs) are typically used in compilers and compiler-like applications, where large amounts of data need to be stored in memory, manipulated and converted into some other format for output. ILs themselves are usually tree-like data types which represent some abstract syntax of a simple language (not necessarily the source or target language of the application). Using an IL has three primary advantages:

1. ILs provide a natural point at which to break the system into modules

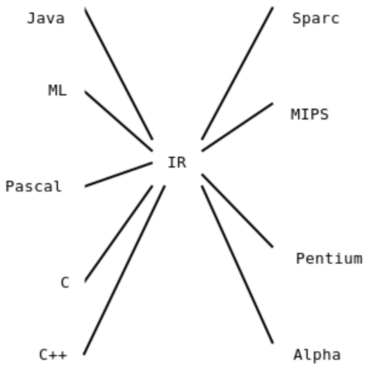


Figure 3.2: Compilers for four languages and four target machines, without an intermediate language (left) and with an IL (right), from [10].

(see Figure 3.1);

2. the IL is typically chosen in such a way that writing algorithms to manipulate instances of it is easier than writing programs to manipulate the raw data that the IL represents;

3. the application becomes more portable - new front-ends can be written to convert input data into the intermediate representation and new backends can be written to convert IL phrases into different output formats. Moreover, for N source languages and M target-languages $M > 1$, only $N + M$ (rather than NM) systems need to be constructed (see Figure 3.2).

3.2.1 Intermediate representations in compilers

In designing a new intermediate language for static checking, it is useful to start by examining ILs used in compilers, as the two systems are often es-

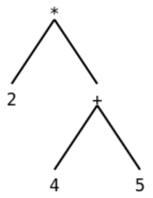


Figure 3.3: Abstract syntax tree of the expression `2 * (4 + 5)`

sentially the same, up to optimisation and code generation (See Chapter 2). Compiler writers have experimented with a number of intermediate representations at a variety of levels of abstraction, Chapter 5 of Brandis' PhD thesis [15] contains a good summary. At the highest level of abstraction are **abstract syntax trees**, which directly copy the syntax of the source language into a tree structure. For example, Figure 3.3 shows an AST representation of the following expression:

```
2*(4+5)
```

At the other end of the spectrum, some compilers (particularly those that employ aggressive optimisations) use a sub-machine code level representation. This is often annotated with details of register and stack allocations. For example, the gcc compiler uses a representation called **register transfer level** (RTL) code. Listing 3.1 shows gcc's internal representation (before optimisation) of the following C program:

```
void main() {
  2*(4+5);
}
```

```
;; Function main
```

```
(note 2 0 6 NOTE_INSN_DELETED)
(insn 6 2 8 (parallel[
            (set (reg/f:SI 7 esp)
                (and:SI (reg/f:SI 7 esp)
                    (const_int -16 [0xfffffff0])))
            (clobber (reg:CC 17 flags))
        ] ) -1 (nil)
    (nil))
(insn 8 6 10 (set (reg:SI 59)
        (const_int 0 [0x0])) -1 (nil)
    (expr_list:REG_EQUAL (const_int 0 [0x0])
        (nil)))
(insn 10 8 12 (parallel[
            (set (reg/f:SI 7 esp)
                (minus:SI (reg/f:SI 7 esp)
                    (reg:SI 59)))
            (clobber (reg:CC 17 flags))
        ] ) -1 (nil)
    (nil))
(insn 12 10 3 (set (reg/f:SI 60)
        (reg/f:SI 55 virtual-stack-dynamic)) -1 (nil)
    (nil))
(note 3 12 4 NOTE_INSN_FUNCTION_BEG)
(note 4 3 13 NOTE_INSN_DELETED)
(note 13 4 14 NOTE_INSN_DELETED)
(note 14 13 16 NOTE_INSN_DELETED)
(note 16 14 21 NOTE_INSN_FUNCTION_END)
(insn 21 16 22 (clobber (reg/i:SI 0 eax)) -1 (nil)
    (nil))
(insn 22 21 18 (clobber (reg:SI 58)) -1 (nil)
    (nil))
(code_label 18 22 20 1 "" "" [0 uses])
(insn 20 18 23 (set (reg/i:SI 0 eax)
        (reg:SI 58)) -1 (nil)
    (nil))
```

```
(insn 23 20 0 (use (reg/i:SI 0 eax)) -1 (nil)
    (nil))
```

Listing 3.1: RTL representation of a simple C program

This variety of intermediate representations provides a wide choice for writers of static analysis systems. Any choice is semantics-preserving and some may be more time- or space-efficient than others.

3.3 ICODE

The most important design decision to make about an IL is to decide what its level of abstraction should be. This will be somewhere along a spectrum which extends from abstract syntax trees (most abstract) to sub-machine level virtual machine code (most concrete). The needs of Exstatic differ quite dramatically from those of compilers, and it's worth noting where these differences occur before making this decision:

- ICODE not only has to represent languages which are compiled to machine code, but all computing languages. Therefore, a fixed set of tree nodes or type constructors, suitable for a particular paradigm or paradigms of computing language is undesirable;
- ICODE is not intended to be optimised or compiled to machine code. Therefore, there is little reason for ICODE to be significantly less abstract than the source code which it represents. One argument for ICODE to be a more concrete representation of a program than the

original source code might be that there are many well known semantic analysis algorithms which operate on CFGs, etc. However, since CFGs and other graphs could be computed from a high-level representation of a program, this argument is not very compelling;

- since Exstatic is only intended to detect errors in code, it should be able to represent as much semantic information as possible. Ideally, it should be possible to accurately decompile an ICODE representation which would be useful when reporting an error to the user.

```
datatype icode = EPSILON (* EMPTY expression. *)

    | Val of { v      : SL.value, (* Values. *)
               annote : SL.annote list}

    | Arith of { e1     : icode,   (* Arithmetic. *)
                 e2     : icode,
                 aop    : SL.aop,
                 annote : SL.annote list }

    | Bool of { e1     : icode,    (* Booleans. *)
                e2     : icode,
                bop    : SL.bop,
                annote : SL.annote list }

    | Prim of { e1     : icode,    (* Primitive exprs. *)
                e2     : icode,
                pop    : SL.pop,
                annote : SL.annote list }

    | Assign of { lvalue : icode, (* Assignment. *)
                  rvalue : icode,
                  annote : SL.annote list }

    | Call of { name   : string,  (* Fn Calls, etc. *)
                args   : icode list,
                annote : SL.annote list }

    (* Selection (guarded commands). *)
    | Select of { guards : (icode * icode * icode) list,
                  annote : SL.annote list }

    (* Iteration (guarded commands). *)
    | Iterate of { guards : (icode * icode) list,
                   annote : SL.annote list }

    |  Name of { n      : string, (* Names. *)
                 annote : SL.annote list }

    (* Name spaces. *)
    | NameSpace of { name   : string,
                     space  : icode list,
                     annote : SL.annote list }

    (* Parameterized name space. *)
    | ParamNameSpace of  { name   : string,
                           args   : icode list,
                           space  : icode list,
                           annote : SL.annote list }
```

Listing 3.2: An SML definition of ICODE. Note that `SL` is an SML *structure* containing declarations of datatypes, pertaining to an individual source language.

For these reasons, ICODE is as abstract as possible. ICODE nodes are essentially generic versions of items which might be found in an AST and every node is extensible via user defined annotations. Since many static checking algorithms are specific to either the dataflow or control flow aspects of a program, ICODE is intended to separate these constructs by providing different type constructors for dataflow and control flow expressions and statements. In particular, this is intended to simplify the process of writing static checking routines that can be applied to many different source languages (such as a check for unreachable code).

3.3.1 ICODE as an SML datatype

ICODE itself is a tree-like datatype which could be described diagrammatically or in pseudocode. In Listing 3.2, ICODE is written as an SML datatype, to be consistent with later descriptions in Chapters 4 and 5 – any other general purpose programming language would serve equally well. Note also that this definition aims at clarity rather than, say, efficiency. In a production implementation it may desirable to use functional arrays or imperative arrays rather than lists.

Note that in SML, *`(* text here... *)`* is a multi-line comment, `datatype` introduces a new type, `|` is used to separate different constructors of the new type and

```
  Person of {name : string, age : int}
```

describes a type constructor `Person`, which is a record consisting of two fields: `name` (of type `string`) and `age` (of type `int`). A value constructed with the `Person` constructor might look like this:

```
Person{name="Tony Blair", age=50}
```

The construct `SL.something` means that `something` is some object (a value, exception, type, function or structure) in a separate structure (or module) called `SL`. In the definition of the `icode` datatype, `SL` is the structure which holds types and other program objects relevant to a specific source language. In particular, this includes enumeration types for operators and keywords and types for ICODE annotations.

3.3.2 ICODE nodes

Each node of an ICODE tree can represent many different source language constructs. Below is a list of ICODE type constructors and the sorts of object they are intended to represent.

`EPSILON` (after ϵ, used in regular expressions to stand for "the empty string") is an empty expression or statement, which may be a `SKIP` or `NOP` statement in the source language or may be a place-holder. For example a unary operator expression in a source language may map to a binary operator expression in its ICODE representation, where one operand is an `EPSILON`.

`Val` represents literal values such as strings, integers, floating point numbers, etc.

`Arith` represents arithmetic expressions, e.g. expressions with operators such as `+`, `-`, `/`, `mod`, etc. `Arith` expressions are binary and consist of a two operands, `e1` and `e2`, which are ICODE nodes and an operator, `aop`, whose type is distinct for each source language.

`Bool` represents boolean expressions, e.g. expressions with operators such as `&&`, `||`, `!`, etc. `Bool` expressions are binary and consist of a two operands, `e1` and `e2`, which are ICODE nodes and an operator, `bop`, whose type is distinct for each source language.

`Prim` represents expressions with operators which are primitive in the source language, but not arithmetic or boolean. Examples include `car` and `cdr` from Lisp, `return` and `import` from Java, `raise` and `::` (`cons` for lists) from ML, `register` and `sizeof` in C, etc. `Prim` expressions are binary and consist of a two operands, `e1` and `e2`, which are ICODE nodes and an operator, `pop`, whose type is distinct for each source language.

`Assign` represents assignments which may be side-effecting (in imperative or impure declarative languages) or not. `Assign` nodes consist of an *lvalue*, called `lvalue` and an *rvalue*, called `rvalue`.

`Call` represents calls to execute code in other parts of the program. For example, `goto` statements, function, procedure or method calls, raised (thrown) exceptions, etc. `Call`s consist of a `name`, of type `string`, which is the name of the `Call`'s destination and a list of arguments, `args`, which are ICODE nodes.

`Select` represents choice, such as `if`, `switch`, `|` (pattern-matching), etc. A

`Select` statement is intended to be a guarded statement (very loosely based on Dijkstra's guarded command language [27]) and may involve some refactoring of the original sour code. In an ICODE `Select` each guard is a triple. The first element in the triple should be a boolean guard (an ICODE `Bool` expression), the second should be the statement or expression that is executed if the guard evaluates to *true* and the third is should be the statement or expression that is executed if the guard evaluates to *false*. This way, it is easy to translate `if-then-else` statements to a single guard and it is also possible to translate multi-branch selection (such as `switch` statements in C and Java), by making the third guard in each triple an `EPSILON` node. The single guard for an `if-then-else` statement is a more obvious translation, which is easy to decompile into the original source code for error reporting.

`Iterate` represents iteration constructs such as `for`, `while`, `do-while`, `REPEAT`, `LOOP`, etc. Iteration is represented in ICODE by guarded commands and `Iterate` nodes consist of a list of guard-statement pairs.

`Name` represents names of program objects, such as modules, classes, methods, functions, variables, etc.

`NameSpace` represents name spaces. In block-structured languages these will be blocks of various sorts (including functions, methods, classes, modules, `synchronized` blocks in Java, etc.). In declarative languages, `let-in` expressions and modules may be `NameSpace`s. `NameSpace` consist of a name (of type `string`) which may be the empty string if the namespace is anonymous, and a list of ICODE nodes which are within the scope of the namespace.

`ParamNameSpace` represents name spaces that are parameterised (i.e. that have arguments). These include procedures, methods, functions, macro definitions, etc. `ParamNameSpace` are exactly the same as `NameSpace`, with the addition of a list of arguments, `args`.

3.3.3 Example

As an example of an ICODE representation of a program, consider the following Java code:

```
public class Hello {
  public static void main(String args[]) {
    System.out.println("Hello");
  }
}
```

Java classes and methods can be represented as ICODE `Namespaces` and `ParamNameSpace`, respectively. The call to `System.out.println` is an ICODE call and the string literal `"Hello"` is a `Val`. Below is a possible translation of the above class into ICODE. The meaning of annotations should be obvious (as most are clearly named enumeration types). We assume that the code has been type checked before an ICODE representation has been generated and provide annotations for types. In a more complete example annotations might also hold line and character numbers relating to the original source code, to aid in producing clearer error reports. In this document, annotations are omitted if they are empty.

```
NameSpace {
  name="Hello",
  space={[ParamNameSpace{
    name="main",
    args=[Name{n="args",
          annote=[Java.Type(
                    Java.array(
                      Java.lang.String))]}],
    space=[Call{name="System.out.println",
               args=[Val{v=Java.string("Hello"),
               annote=[Java.Type(Java.lang.String)]}]}]
   annote=[Java.METHOD,
           Java.PUBLIC,
           Java.STATIC,
           Java.retType(Java.VOID)]}],
  annote=[Java.CLASS, Java.PUBLIC]}
```

3.4 ICODE implementation

ICODE has been defined above in SML, which is extremely convenient for the proofs given in Chapters 4 and 5. However, it may be that other users will not find SML convenient and prefer to implement Exstatic parsers or static checkers in some other language. To facilitate this, ICODE can be expressed in *XML* (eXtensible Markup Language), which is a language that allows developers to define bespoke markup languages.

```
<namespace name="Hello">
  <space>
    <paramnamespace name="main">
      <args>
        <name n="args">
        <annote>
          <java:type>
            <java:array><java:string/></java:array>
          </type>
        </annote>
      </args>
      <space>
        <call name="System.out.println">
          <args>
            <val>
              <v>
                <java:string>
                  hello
                </java:string>
              </v>
            </val>
            <annote>
              <java:type><java:string/></java:type>
            </annote>
          </args>
        </call>
        <annote>
          <java:method/>
          <java:public/>
          <java:static/>
          <java:rettype><java:void/></java:rettype>
        </annote>
      </space>
    </paramnamespace>
  </space>
  <annote>
    <java:class/>
    <java:public/>
  </annote>
</namespace>
```

Listing 3.3: An ICODE-XML representation of a short Java program

XML representations of ICODE tend to be very verbose, for example, Listing 3.3 shows a possible ICODE-XML representation of the Java program discussed above. However, XML is not intended to be read by humans and its biggest advantage is that most high-level languages have libraries for parsing and generating XML.

3.4.1 Document validation

XML has two mechanisms defining extensions: *document type definitions* (DTDs) and *schemas*. Schemas define a type discipline for XML documents, based on a rich set of primitive types and constructors. DTDs are less strict and merely define element types and their attributes. However, it is more common to find DTD validators than schema validators in the widely available XML-libraries for high-level languages. To this end, both a DTD and a Schema have been written for ICODE.

3.4.2 Nomenclature

Note that, as stated in Section 3.3, the ICODE representation of a given language, L, will be called ICODE$_L$. The XML representation of ICODE will be called ICODE-XML. If, for some L, ICODE$_L$ is expressed in XML then it should be called ICODE$_L$-XML.

3.5 Universality of ICODE: methodology

In order to substantiate the claim that ICODE is *universal*, it is necessary to first define the word. In terms of compiler writing and systems programming, the term “universal intermediate language” usually refers to UNiversal Computer Oriented Language (UNCOL) - a universal systems language discussed in the late 1950s and early 1960s [107, 108]. The intention was that applications could be written in high-level *program oriented languages* (POLs) and translated into UNCOL, systems programming could be carried out it UNCOL, and a series of compilers would translate UNCOL code into machine languages. This would mean that $m + n$ compilers (rather than $m\ n$) would be needed to execute code in m languages on n machines. Several implementations of UNCOLs were posited [23, 106], although none were implemented.

ICODE is claimed to be universal in a slightly different sense. ICODE representations of programs will never be compiled into an executable and no programming is intended to be carried out in ICODE. In order to compile some language, L, into ICODE, some data structures must first be added to the intermediate format to create ICODE$_L$. Therefore, ICODE can afford to be a considerably higher-level representation than any other known to the author (see Chapter 3).

Thesis 3.1. ICODE is *universal* in the sense that for any computing language, L, there exists a representation ICODE$_L$ for which $[\![L]\!] = [\![\mathrm{ICODE}_L]\!]$. ($[\![E]\!]$ are known as *Scott brackets* and refer to the meaning, or denotation of some abstract syntax E).

This is a thesis, rather than a theorem, as it is impossible to enumerate every L and prove that a suitable representation exists for each. Instead, we choose a small number of representative, Turing-complete languages, with features common to many commercially used systems, and show that for those specific languages a translation to ICODE exists.

3.5.1 Substantiating the claim of universality

In order to substantiate the claim in Thesis 3.1, we instantiate and prove the following theorem for two individual computing languages:

Theorem 3.1. For a given computing language, L, there exists a representation ICODE_L for which $[\![L]\!] = [\![\text{ICODE}_L]\!]$.

The two example languages are chosen to be small enough such that proofs are tractable, but large enough to exhibit the main features of the programming paradigms they represent. These are:

- Typed PCF, a functional language defined by Plotkin [88];
- IMP, an IMPerative language, defined in [117], and
- PROLOG, a logic-programming language, as defined by Allison's semantics [6] is discussed in outline, as are mixed-paradigm languages.

3.5.2 Proof strategy

The following method is applied to each language:

1. A semantic domain (or smash sum of domains) is defined as a pure SML datatype.

2. The denotational semantics of the language is written as a function, `interp_L:L→domain`, in a pure subset of the language SML.
3. A function, `translate:L→ICODE`, is written that translates the language, L, into ICODE$_L$. Datatypes for annotations and operators that define ICODE$_L$ are also written.
4. The denotational semantics of ICODE$_L$ is written as a function, `interp_icode_L:ICODE→domain`, in a pure subset of the language SML.
5. The theorem: `interp_L` = `translate` ∘ `interp_icode_L` is proven by structural induction of the structure of `L`.

The description of a denotational semantics as a program is due to Allison [7, 8]. Allison used the language Pascal to describe his semantics. A pure subset of SML is used here, as it has the advantage of referential transparency, which greatly simplifies step five.

The advantages of using this technique in the context of static checking are distinct from the reasons a language designer might give for using an executable semantics. Firstly, these proofs not only support the claim of Thesis 3.1, they also serve to guide anyone wishing to write an Exstatic front-end for a real language. Using an executable semantics, with a "real" implementation of various ICODE$_L$ datatypes is a clearer guide than writing proofs by hand. Where a language supports several programming paradigms (Lisp, Java, C++, SML, etc.), constructs from several of these proofs can be picked out and used orthogonally. Secondly, the use of a compiler to type-check each semantic function ensures that descriptions are largely error-free. Thirdly, this approach is congruent with one of the main theses of this work: that static checking can and should be applied outside the realm of application programming.

3.6 Conclusions

This Chapter has surveyed a number of intermediate languages and formats which claim to be "universal" and has described ICODE, the intermediate representation used in Exstatic. Section 3.5.2 described how the claim that ICODE is a universal intermediate format for static checking will be substantiated. The following four Chapters apply the proof strategy to source languages in various paradigms which each have a well understood denotational semantics.

CHAPTER 4

FUNCTIONAL LANGUAGES IN ICODE

Chapter 3 described a strategy to show how a proof can be constructed to demonstrate that a given source language, with a denotational semantics, can be translated into ICODE in such a way that its semantics is preserved. This Chapter presents such a proof for the functional language Typed PCF [88]. This contributes to the claim in the original thesis for this work that Exstatic is capable of analysing code in a number of different source languages.

4.1 Functional languages

In contrast to imperative languages, which model computation as a more abstract version of the operation of a CPU, functional languages, such as SML, Caml[1], OCaml, Hope, Miranda and Haskell derive from Church's λ-calculus [20, 21]. Historically, functional languages have had the reputation of being esoteric, in the sense that they have been much studied in academia, but little used in industry, with the exception of some niche areas such as the financial sector and telecommunications. However, the reality is somewhat

---

[1] SML (Standard ML) and Caml are different dialects of the same language - that is, they have different syntaxes for essentially the same semantics. Here, we used SML for the SML syntax of ML and ML as the family of languages that includes SML and Caml. Note that OCaml is Caml with object-oriented extensions.

different, as a large number of domain-specific languages are functional and many of these have become market leaders in their fields. For example, Peyton Jones et. al. [55] note that the programming language which is built into Microsoft's Excel spreadsheet is a functional language, perhaps the most widely used functional language in the world. Equally, the functional language Erlang is used to develop the firmware of mobile phone and XSLT for performing transformations on XML documents.

Functional programs are said to be **referentially transparent**, meaning that they do not have state (mutable variables) and can be treated like an equivalent piece of mathematical reasoning. For example the following piece of simple reasoning relies on function composition and variable substitution (assuming the keyword `let` here has the same semantics as in a pure subset of Standard ML):

$$\text{Let } x \;=\; 1,\, f(y) = x + y,\, g(z) = z \times 3 \text{ and } h(w) = f \circ g(w)$$

$$\text{Then, } h(w) = 3 \times (w + 1)$$

If the above is translated into an imperative language then the conclusion no longer holds, because the value of x may be updated by code in any thread which has x in its scope. In a functional language, however, such substitutions may be made by a compiler (or interpreter) because of the property of referential transparency. This makes formal reasoning about functional languages relatively simple and allows programs to be built up by composition – an approach popularised by Backus in his Turing award lecture

[11] where he discussed an "algebra of programs". It can also be argued that this paradigm is more usable for the programmer, partly because it leverages understanding from high-school mathematics and also because a functional program need only describe what needs to be computed, rather than the exact steps that must be taken to perform a computation.

Apart from referential transparency, functional languages are often characterised by a number of features:

- Functions are first-class objects and can therefore be passed to and returned from other functions;
- the primary (compound) data structure is usually the list; and
- repetition is achieved by recursion. This is usually optimised by techniques such as tail-call optimisation and admits reasoning by (structural) induction proofs on programs.

Beyond the scope of this Chapter, functional programming has contributed many other concepts to Computer Science, including higher-order functions, currying, monads and arrows (from Category theory), call-by-need evaluation, continuations and Hindley-Milner type inference.

4.2 Translating functional languages into ICODE

Before discussing how functional programs may be translated into ICODE, it is useful to consider an ICODE representation of λ-expressions. These have two language constructs: abstraction and application. The abstraction $\lambda\, x.x$ is the identity function, which will return any value applied to it, for example $(\lambda\, x.x)g$ applies the value g to the abstraction $\lambda\, x.x$ and the whole expression evaluates to g. In the expression $\lambda\, x.x$, the name x is bound. Therefore, it might seem reasonable to represent λ-abstractions by ICODE name-spaces or parameterised name-spaces, which contain the expressions bound by the abstraction. The identifier between the λ and the period could be a parameter. However, λ-expressions are themselves unnamed, so every λ-expression would be an anonymous name-space and it doesn't seem natural to create a name-space which does not itself introduce a new name[2]. Instead, `Prim` expressions can represent both abstraction and application. For example, the two λ-expressions above would be written as (note that empty annotation lists are omitted):

```
Prim{
  e1=Name{n="x"},
  e2=Name{n="x"},
  pop=SL.LAMBDA
}
```

and:

```
Prim{
  e1=Prim{
       e1=Name{n="x"},
       e2=Name{n="x"},
       pop=SL.LAMBDA
```

---

[2] Although it is sometimes desirable to do this in block-structured languages, where an anonymous block may be used to constrain the scope of names inside it.

```
    },
  e2=Name{n="g"},
  pop=SL.APPLY
}
```

Trees of expressions can be annotated with lists of free and bound names, if necessary.

Consider the following factorial function, written in SML:

```
fun fact 0 f = f
  | fact n f = fact (n-1) (f*n)
```

This shows what appears to be a function with two arguments, which pattern-matches on the first of these. In fact, SML functions are a syntactic sugar for a name, bound to a λ-expression, which has precisely one argument. The following is a de-sugared version of the factorial function:

```
val rec fact =
    fn n => fn f => case n of 0 => f
                            | _ => fact (n-1) (f*n)
```

The keyword `rec` indicates that the name fact appears on the right-hand side of the equals sign and SML uses `fn` to stand for λ - the identity function would be written as follows:

```
fn x => x
```

An ICODE version of the SML sugared version of the `fact` function is shown in Listing 4.1. Since SML functions have names, it now makes sense to represent them as name-spaces. The `n` and `f` could then be parameters to that name-space, but this seems to obscure the semantics of the function and hides the pattern-match on `n`. Instead, a representation is used that is much closer to the de-sugared version of the function that relates to our implementation of λ-expressions. The names of arguments are bound by a `Prim` expression which appears in the same place as the SML `case` statement.

Similarly, the function call to `fact` with argument `(n-1)` returns a function of type `int`→`int`, to which `(f*n)` is applied. Rather than obscure the semantics of this statement by representing it as parameterised name, it is written as a `Prim` expression.

4.3 PCF and its semantics

PCF (Programming Computable Functions) is a simple, typed functional language which has been studied extensively by theoreticians, especially in connection with the relationship between operational and denotational semantics. PCF was introduced by Dana Scott, *circa* 1969 (eventually published in 1993 as [99]) as part of LCF, a “Logic of Computable Functions”, which was the inspiration for the ML family of languages. Later, Plotkin studied PCF as a distinct programming language, in [88]. There are various versions of PCF which differ in minor ways. The version used here derives from Andrew Pitts’ Cambridge lecture notes [87].

PCF has three types, `nat` (natural numbers), `bool` (booleans) and $\tau \to \tau$ functions:

```
datatype tau = NAT | BOOL | Fn of tau * tau
```

```
<namespace name="fact">
  <space>
    <paramnamespace name="sml:match">
      <args>
        <val>0<annote><sml:int/></annote>
        <name n="f">
      </args>
      <space><name n="f"></space>
      <annote><sml:match/></annote>
    </paramnamespace>
    <paramnamespace name="sml:match"/>
      <args><name n="n"/><name n="f"/></args>
      <space>
        <call>
          <name>fact</name>
          <args>
            <arith>
              <aop><sml:int-minus/>
              <e1><name>n</name></e1>
              <e2><val>1<annote><sml:int/></annote></val>
              </e2>
            </arith>
            <arith>
              <aop><sml:int-times/>
              <e1><name>f</name></e1>
              <e2><name>n</name></e2>
            </arith>
          </args>
        </call>
      </space>
      <annote><sml:match/></annote>
    </paramnamespace>
  </space>
  <annote><sml:fun/></annote>
</namespace>
```

Listing 4.1: ML factorial function in ICODE-XML

PCF has three literals: `NOUGHT`, `TRUE`, and `FALSE` and several constructors. `Var` constructs variable names, represented here by SML strings. `Succ` and `Pred` are successor and predecessor functions, respectively. Note that there is no syntactic restriction which prevents `Succ` and `Pred` being applied to non-`nat`s. `Lambda`, `Apply` and `Fix` are λ-abstraction, application and the Y combinator, from λ-calculus. Lastly, `IfElse` represents choice. The SML datatype representing the PCF language is as follows:

```
datatype pcf = NOUGHT
             | TRUE | FALSE
             | Succ of pcf | Pred of pcf
             | Zero of pcf
             | IfElse of (pcf * pcf * pcf)
             | Var of string
             | Lambda of (string * tau * pcf)
             | Apply of (pcf * pcf)
             | Fix of pcf
```

4.3.1 Semantic domain

Before describing the semantics of PCF, we first need to describe a domain, $[\![\tau]\!]$, by induction on the structure of τ and a function which will find the value of a bound variable, in a given environment:

```
datatype domain = Nat of int
                | Bool of bool
                | Fun of domain -> domain
                | BOTTOM

fun env ((x, v)::rho) s = if x = s then v else env rho s
  | env []            s = BOTTOM
```

4.3.2 PCF semantics

Now that preliminary definitions have been dealt with, a semantics for PCF can be written. Section 3.5.2 prescribes that this should be a function, `interp_pcf:pcf`→`domain`. Note that an interpretation for `Fix` expressions is not given in the code. Since PCF is typed, the semantics of `Fix` is given by:

$$[\![\texttt{Fix}(M : \sigma)]\!]\rho = \mathit{fix}_\sigma([\![M]\!]\rho)$$

where fix_σ is the function which assigns least fixed points to continuous functions of type σ. Note that `Fix(M)` is dependent on the type of `M` and so is $[\![\texttt{Fix}]\!]$. This is why $[\![\texttt{Fix}]\!]$ does not appear in the code below: a different case for every possible type of `M` would have to be written, and because PCF types include functions, there are an infinite number of these.

The following is an abbreviated listing of the `interp_pcf` function. A full listing can be found in Appendix D:

```
fun interp_pcf rho (NOUGHT) = Nat 0
  | interp_pcf rho (TRUE)   = Bool true
  | interp_pcf rho (FALSE)  = Bool false
  | interp_pcf rho (Succ e) = (case interp_pcf rho e of
                                   Nat n => Nat (n + 1)
                                     | _ => BOTTOM)
  | interp_pcf rho (Pred e)  = (case interp_pcf rho e of
                                   Nat n => Nat (n - 1)
                                     | _ => BOTTOM)
  | interp_pcf rho (Zero e)  =
      (case interp_pcf rho e of Nat n
             => if n = 0 then Bool true else Bool false
```

```
                                | _ => BOTTOM)
  | ...
  | interp_pcf rho (Fix M)   = (* Omitted. *)
```

4.4 Translating PCF into ICODE$_{\text{PCF}}$

Translating PCF into ICODE$_{\text{PCF}}$ is done in three steps. Firstly, a structure is written to define all the PCF specific datatypes that will be used in ICODE$_{\text{PCF}}$:

```
structure PCF = struct
  (* Values in icode.Val expressions. *)
  datatype value = NOUGHT | TRUE | FALSE
  (* Operators in icode.Arith expressions. *)
  datatype aop = SUCC | PRED
  (* Operators in icode.Bool expressions. *)
  datatype bop = ZERO
  (* Operators in icode.Prim expressions. *)
  datatype eop = LAMBDA | APPLY | FIX
  (* Annotations. *)
  datatype annote = NAT | BOOL | Fn of annote * annote
end
```

Secondly, a function is defined which translates PCF types into ICODE$_{\text{PCF}}$ annotations:

```
fun tau2ann NAT           = PCF.NAT
  | tau2ann BOOL          = PCF.BOOL
  | tau2ann (Fn(t1, t2)) = PCF.Fn(tau2ann t1, tau2ann t2)
```

Next, the `translate:pcf→icode_pcf` function is defined (a full listing can be found in Appendix D:

```
fun translate (NOUGHT)    = Val{v=PCF.NOUGHT, annote=[]}
  | translate (TRUE)      = Val{v=PCF.TRUE,   annote=[]}
  | translate (FALSE)     = Val{v=PCF.FALSE,  annote=[]}
  | translate (Succ pcf) = Arith{e1=(translate pcf),
                                  e2=EPSILON,
                                  aop=PCF.SUCC,
                                  annote=[]}
  | ...
  | translate (Fix pcf)  = Prim{e1=(translate pcf),
                                 e2=EPSILON,
                                 pop=PCF.FIX,
                                 annote=[]}
```

4.5 The semantics of ICODE$_{\mathrm{PCF}}$

Lastly, the semantics of ICODE$_{\mathrm{PCF}}$ needs to be defined. Section 3.5.2 prescribes that this should be a function, `interp_icode_pcf:ICODE→domain`. Note that, as above, an interpretation for `Fix` expressions is not given in the code and a full listing of `interp_icode_pcf` can be found in Appendix D:

```
fun interp_icode_pcf rho (Val{v=value, annote=a}) =
    (case value of
         PCF.NOUGHT => Nat 0
       | PCF.TRUE   => Bool true
       | PCF.FALSE  => Bool false)
   | interp_icode_pcf rho (Arith{e1=M,
                                 e2=EPSILON,
                                 aop=oper,
                                 annote=an}) =
    (case oper of PCF.SUCC =>
        (case interp_icode_pcf rho M of Nat n =>
           Nat (n + 1)
```

$$\begin{tikzcd}
\mathrm{PCF} \arrow[r, "\mathrm{interp\_pcf}"] \arrow[dr, "{[\![-]\!]:PCF\to\mathbb{D}_\perp}"'] & \mathrm{ICODE}_{\mathrm{PCF}} \arrow[d, dotted, "{[\![-]\!]:\mathrm{ICODE}_{\mathrm{PCF}}\to\mathbb{D}_\perp}"] \\
 & \mathbb{D}_\perp
\end{tikzcd}$$

Figure 4.1: `interp_pcf` = `translate` ∘ `interp_icode_pcf`

```
         | _ => BOTTOM)
      | PCF.PRED => (case interp_icode_pcf rho M of
                       Nat n => Nat (n - 1)
                       | _ => BOTTOM))
| ...
(* Wild-card. *)
| interp_icode_pcf rho _ = BOTTOM
```

4.6 Sketch of a proof that `interp_pcf` = `translate`∘`interp_icode_pcf`

Theorem 4.1. `interp_pcf` = `translate` ∘ `interp_icode_pcf`

Proof. The proof of theorem 4.1 proceeds by structural induction on the `pcf` datatype. The base cases are the enumerations and the inductive steps are the type constructors.

The proof is straight forward and repetitive, and therefore not reproduced here in full. Two cases suffice to give the flavour of the proof (the latter being the case for `Fix`, which is omitted from code listings). Line numbers refer to the code listed in Appendix D.

Case `NOUGHT`:

$[\![\mathtt{NOUGHT}]\!]\rho = \mathtt{Nat}(0)$

$[\![\mathtt{translate}(\mathtt{NOUGHT})]\!]\rho = [\![\mathtt{Val}(\mathtt{v} = \mathtt{PCF.NOUGHT}\ldots)]\!]\rho)$

$[\![\mathtt{Val}(\mathtt{v} = \mathtt{PCF.NOUGHT}\ldots)]\!]\rho = \mathtt{Nat}(0)$

Case `Fix`:

The inductive hypothesis is that there exists some p : `pcf` for which Theorem 4.1 is true.

$[\![\texttt{Fix}(p:\sigma)]\!]\rho = \mathit{fix}_\sigma([\![p]\!])$

where fix_σ returns the least fixed point of continuous functions of type σ.

$[\![\texttt{translate}(\texttt{Fix}(p))]\!]\rho = [\![\texttt{Prim}\{\texttt{e1} = (\texttt{translate}\,p), ..., \texttt{pop} = \texttt{PCF.FIX}, ...\}]\!]\rho$

$[\![\texttt{Prim}\{\texttt{e1} = (\texttt{translate}\,p), ..., \texttt{pop} = \texttt{PCF.FIX}, ...\}]\!]\rho = \mathit{fix}_\sigma([\![(\texttt{translate}\,p)]\!]\rho)$ (for $p : \sigma$)

Therefore, $[\![\texttt{Fix}(p:\sigma)]\!]\rho = \mathit{fix}_\sigma([\![(\texttt{translate}\,p)]\!]\rho)$ by the inductive hypothesis.

□

4.7 Mixed-paradigm languages

This Chapter has contributed a proof that the simple functional language PCF can be translated into ICODE, preserving its semantics. It was argued in Section 4.1 that functional languages are important in practice (rather than in their intellectual contribution to Computer Science) because a number of domain-specific languages are used in industry in, for instance, spreadsheets, database access and XML processing. However, many modern general-purpose programming languages mix features of imperative and functional languages. These mixed-paradigm languages are popular, not least because they give programmers a choice in the approach which they take in constructing a particular piece of code. For example, OCaml is an functional languages which includes facilities to manipulate state and to construct classes and objects. Unlike OCaml which is usually regarded as an impure functional language, Python[3] is usually said to be an imperative and object-

---

[3] http://www.python.org

oriented language with functional features. As an example of how a single language can be used to produce functional and imperative code which is idiomatic in both paradigms, consider the following excerpts of set ADTs, which were written by the author for pedagogical use [101, 72]. The following code is functional in nature. Sets are modelled by lists which (by convention) are immutable. A number of functions are provided to perform set operations and these make good use of higher-order functions (such as `filter`) and λ-expressions:

```
empty = [] # The empty set

def ismem(e, set):
    """Returns True if e appears in set, False otherwise.
    pre::
      not set or type(set) == type([])
    post::
      __return__ == e in set
    """
    return bool(filter(lambda x: x == e, set))

def addmem(e, set):
    """Adds element e to set and returns the new set.
    pre::
      not set or type(set) == type([])
    post::
      ismem(e, __return__)
    """
    if not ismem(e, set):
        return [e] + set
    else:
        return set

```

```
def intersect(set1, set2):
    """Returns the intersection of set1 and set2.
    That is, every element that is in both set1 and set2.
    pre::
  type(set1) == type(set2) == type([])
    post::
      forall(__return__,
          lambda e: ismem(e, set1) and ismem(e, set2))
    """
    return filter(lambda x: ismem(x, set2), set1)

def difference(set1, set2):
    """
    Returns the differnce of set1 and set2.
    That is, every element that is in set1
    and not in set2.
    pre::
       type(set1) == type(set2) == type([])
    post::
       forall(__return__,
        lambda e: ismem(e, set1) and not ismem(e, set2))
    """
    return filter(lambda x: not ismem(x, set2), set1)

...
```

Listing 4.2: An excerpt of a functional set ADT in Python

By contrast, the next excerpt is object-oriented. A class is constructed to represent sets and object methods are written to perform set operations. Higher-order functions are not used and iteration is preferred to recursion:

```
class Set:
    """An abstract data type for sets.
    Sets are modelled as lists.

    inv::
        type(self.set) == type([])
    """

    empty = [] # The empty set.

    def __init__(self):
        self.set = []
        return

    def ismem(self, e):
        """Returns True if e appears in self.set,
        and False otherwise.
        pre::
            True
        post::
            __return__ == e in self.set
        """
        return e in self.set

    def addmem(self, e):
        """Adds element e to self.set.
        pre::
            True
        post::
            ismem(e, self.set)
        """
        if not self.ismem(e):
            self.set.append(e)
        return

    def intersect(self, set):
        """Returns the intersection of self.set and set.
        That is, every element that is in both self.set
        and set.
        pre::
            type(set) = type(Set.empty)
        post::
            forall(__return__,
              lambda e: (ismem(e, self.set) and
                    ismem(e, set)))
        """
        res = Set()
        for i in self.set:
            if set.ismem(i):
                res.addmem(i)
        return res

    def difference(self, set):
        """
        Returns the difference of self.set and set.
        That is, every element that is in self.set
        and not in set.
        pre::
            type(set) == type(Set.empty)
        post::
            forall(__return__,
                lambda e: (ismem(e, self.set) and
                      not ismem(e, set)))
        """
        res = Set()
        for i in self.set:
            if not set.ismem(i):
                res.addmem(i)
        return res

...
```

Listing 4.3: An excerpt of an object-oriented set ADT in Python

Although this thesis does not present a separate proof that ICODE can reasonably represent mixed-paradigm languages, this Chapter and Chapter 5 together handle the major concepts that can be found in such languages. This is reasonable because mixed-paradigm languages offer orthogonal syntax for constructs which originate from different paradigms. For example, in the above code, Python keywords and functions such as `lambda`, `map` and `filter` are used which are completely orthogonal to the imperative constructs offered by Python. This offers the programming a set of declarative language features which hide any imperative behaviour which may be implemented in the Python Virtual Machine. Equally, Standard ML offers facilities for variable update and pointer manipulation (`:=`,`ref`, `!`, etc) which are orthogonal to its facilities for pure functional programming. Exstatic makes the assumption that these features can be treated in isolation, as that is how they are described in the language references for the relevant high level languages.

CHAPTER 5

IMPERATIVE LANGUAGES IN ICODE

Chapter 3 described a strategy to show how a proof can be constructed to demonstrate that a given source language, with a denotational semantics, can be translated into ICODE in such a way that its semantics is preserved. This Chapter presents such a proof for the imperative language IMP [117]. This contributes to the original claim in the thesis for this work that Exstatic is capable of analysing code in a number of different source languages.

5.1 Imperative languages

Although the theoretical roots of imperative languages lie in Turing, they derive practically from a perceived need to abstract the details of CPU operations and machine code. Nearly all[1] CPUs execute machine code which operates in an imperative style: a linear sequence of instructions is used to manipulate data which exists in a volatile store. In the early days of stored program computers programmers would write directly in machine code, then assembly languages which had direct translations to machine code, then increasingly higher level languages, such as COBOL, BCPL, C, C++ and so on.

---

[1] The exceptions being esoteric hardware such as the SKIM machine.

Although imperative languages are characterised by their ability to represent and manipulate program state (an abstraction of values held in volatile storage) and the sequential composition of commands, their other major contribution is the subroutine, invented as the "closed subroutine" by Wilkes, Gill and Wheeler, working on the EDSAC. Subroutines and cognate constructs (procedures, methods, coroutines and so on) introduced concepts such as call and return, parameter passing and evaluation, local variables and scope, code reuse and modularity. It can be argued that this is a usable representation of computation for human programmers because people are used to following sequences of instructions in, for example, recipes, musical scores and travel directions.

5.2 Translating imperative languages into ICODE

Imperative languages, such as C, C++, Java, Pascal and Modula have very natural ICODE representations.

- Blocks, classes, procedures, methods, and so on become name-spaces and parameterised name-spaces, respectively.
- `for`, `while`, `do-while`, `LOOP`, `repeat`, etc. become `Iterate` nodes.
- `if`, . . . `ifelse`, `switch`, etc. become `Select` nodes.
- Assignments (`=`, `:=`, . . .) become `Assign` nodes.

- Boolean statements (with operators `==`, `&&`, `||`, . . .) and comparisons (with operators `<`, `>=`, . . .) become `Bool` nodes.

- `return`, `break`, raised/thrown exceptions, etc. become `Call` expressions. Procedure or methods calls become `Call` expressions with parameterised names. This is valid because most imperative and OO languages do not allow curried functions (although Smalltalk is an exception).

As an example, consider the following factorial function in Java:

```java
package example;
final class Example {
  public static final int fact(int n) {
    int f = 1;
    while(n>1) {
      f *= n;
      n--;
    }
    return f;
  }
}
```

The translation of this program into ICODE is given in Listing 5.1. Note that names of methods and classes are fully qualified. The `example` package does not qualify as a `NameSpace`, since other classes (which may or may not be examined by Exstatic) may begin with the same declaration. We assume that type checking has taken place, and some of the ICODE nodes are annotated accordingly.

```
NameSpace{
  name="Example",
  space=[ParamNameSpace{
     name="example.Example.fact",
     args=[Name{n="n", annote=[Java.Type(Java.INT)]}],
     space=[assign{lvalue=Name{n="f"},
                         rvalue=Val{v=1},
             annote=[Java.Type(Java.INT)]},
      Iterate{
        guards=[(Bool{e1=Name{n="n"},
                e2=Val={v=1},
                bop=Java.GT},
           NameSpace{name="",
             space=[Arith{e1=Name{n="f"},
              e2=Name{n="n"},
              aop=Java.TIMES_EQ,
              annote=[]},
              Arith{e1=Name{n="n"},
              e2=EPSILON,
              aop=Java.POST_DEC}],
             annote=[Java.BLOCK]}],
        annote=[Java.WHILE]},
      Prim{e1=Name{n="f"},
           e2=EPSILON,
           pop=Java.RETURN,
           annote=[Java.Type(Java.INT)]}],
     annote=[Java.Method{"example.Example.fact"),
       Java.retType(Java.INT),
       Java.PUBLIC, Java.FINAL, Java.STATIC]}],
  annote=[Java.FINAL, Java.Class("example.Example")]}
```

Listing 5.1: Java factorial function in ICODE

5.3 IMP and its semantics

IMP is a simple, untyped, IMPerative language, due to Winskel [117]. IMP has three syntactic groups: *(i)* arithmetic expressions, *(ii)* boolean expressions and *(iii)* side-effecting commands.

Arithmetic expressions comprise integer numbers, string variables, additions, subtractions and multiplications.

Boolean expressions consist of the literals `TRUE` and `FALSE`, the arithmetic relations, equal-to and less-than and the logical operators, `And`, `Or` and `Not`.

Commands (which have side-effects) comprise `SKIP`, which has no effect, assignment (of arithmetic expressions to strings), sequencing, choice and while-loops.

The SML representation of IMP is as follows:

```
datatype aexp = Num  of int
              | Var  of string
              | Plus of aexp * aexp
              | Sub  of aexp * aexp
              | Mult of aexp * aexp
and bexp = TRUE
         | FALSE
         | Eq  of aexp * aexp
         | Lt  of aexp * aexp
         | Not of bexp
         | And of bexp * bexp
         | Or  of bexp * bexp
```

```
and imp = SKIP
        | Assign of string * aexp
        | Sequ   of imp * imp
        | IfElse of bexp * imp * imp
        | While  of bexp * imp
```

5.3.1 Semantic domain

Before describing the semantics of IMP, we first need to describe a domain, by induction on the structure of `imp` and a function which will find the value of a bound variable, in a given environment:

```
datatype domain = Int of int
                | Bool of bool
                | BOTTOM
fun state ((x, v)::rho) s =
    if x = s then v else state rho s
  | state []            s = BOTTOM
```

5.3.2 Semantics

Now that preliminary definitions have been dealt with, a semantics for PCF can be written. Section 3.5.2 prescribes that this should be a function, `interp_imp:imp→domain`. Because of the syntactic structure of IMP, this function is split up into three mutually recursive functions. The following is an abbreviated listing of the `interp_imp` function. A full listing can be found in Appendix E:

```
fun interp_imp rho (SKIP)             = rho
  | interp_imp rho (Assign(s, a))     =
    (s, interp_imp_aexp rho a)::rho
  | interp_imp rho (Sequ(c1, c2))      =
    interp_imp (interp_imp rho c1) c2
  | interp_imp rho (IfElse(b, c1, c2)) =
    if (interp_imp_bexp rho b) = Bool true
    then interp_imp rho c1
    else interp_imp rho c2
  | interp_imp rho (While(b, c))       =
    if (interp_imp_bexp rho b) = Bool true
    then interp_imp (interp_imp rho c) (While(b, c))
    else rho
(* Interpret arithmetic expressions. *)
and interp_imp_aexp rho (Num i)        = Int i
  | interp_imp_aexp rho (Var s)        = state rho s
  | ...
(* Interpret boolean expressions. *)
and interp_imp_bexp rho (TRUE)        = Bool true
  | interp_imp_bexp rho (FALSE)       = Bool false
  | ...
```

5.4 Translating IMP into ICODE$_{\text{IMP}}$

The first step in translating IMP into ICODE$_{\text{IMP}}$ is to define a structure to hold all the IMP specific datatypes that will be used in ICODE$_{\text{IMP}}$. Note that unlike the analogous structure for PCF (see Appendix E) no types are needed for operators in primitive expressions or annotations.

```
structure IMP =
struct
(* Values in icode.Val expressions. *)
datatype value = TRUE | FALSE | Num of int
(* Operators in icode.Arith expressions. *)
datatype aop    = PLUS | SUB | MULT
(* Operators in icode.Bool expressions. *)
```

```
datatype bop    = EQ | LT | NOT | AND | OR
end (* structure IMP *)
```

Next, the `translate:imp→icode_imp` function is defined (a full listing can be found in Appendix E:

```
fun translate (SKIP)              = EPSILON
  | translate (Assign(s,a))       =
    Assign{lvalue=Name{n=s, annote=[]},
           rvalue=(trans_aexp a),
           annote=[]}
  | translate (Sequ(c1, c2))      =
    NameSpace{name="",
              space=((translate c1)::(translate c2)::[]),
              annote=[]}
  | ...
(* Translate boolean expressions. *)
and trans_bexp (TRUE)        = Val{v=TRUE,  annote=[]} =
  | trans_bexp (FALSE)       = Val{v=FALSE, annote=[]}
  | trans_bexp (Eq(a1, a2))  = Bool{e1=(trans_aexp a1),
                                    e2=(trans_aexp a2),
                                    bop=EQ,
                                    annote=[]
  | ...
(* Translate arithmetic expressions. *)
and trans_aexp (Num i)        = Val{v=(Num i), annote=[]}
  | trans_aexp (Var s)        = Name{n=s, annote=[]}
  | trans_aexp (Plus(a1, a2)) = Arith{e1=(trans_aexp a1),
                                      e2=(trans_aexp a2),
                                      aop=PLUS,
                                      annote=[]}
| ...
```

5.5 The semantics of ICODE$_{\mathrm{IMP}}$

Lastly, the semantics of ICODE$_{\mathrm{IMP}}$ needs to be defined. Section 3.5.2 prescribes that this should be a function, `interp_icode_imp:ICODE→domain`. As above, a full listing of `interp_icode_imp` can be found in Appendix E:

```
fun interp_icode_imp rho (EPSILON) = rho
  | interp_icode_imp rho (Assign{lvalue=l,
                                 rvalue=r,
                                 annote=an}) =
    (case l of Name{n=s, annote=a} =>
       (s, interp_icode_aexp rho r)::rho)
  | ...
  | interp_icode_aexp rho (Name{n=x, annote=a}) =
    state rho x
  | interp_icode_aexp rho (Arith{e1=a1,
                                 e2=a2,
                                 aop=oper,
                                 annote=an}) =
    (case oper of
         IMP.PLUS => (case interp_icode_aexp rho a1 of
            Int i => (case interp_icode_aexp rho a2 of
                                 Int j => Int (i + j)))
       | IMP.SUB  => ...
and interp_icode_bexp rho (Val{v=value, annote=an}) =
    (case value of
         IMP.TRUE  => Bool true
       | IMP.FALSE => Bool false
       | IMP.Num i => BOTTOM)
  | interp_icode_bexp rho (Bool{e1=b1,
                                e2=b2,
                                bop=oper,
                                annote=an}) =
    (case oper of
         IMP.EQ  => (case interp_icode_bexp rho b1 of
```

$$\begin{tikzcd}
\mathrm{IMP} \arrow[r, "\mathrm{interp\_imp}"] \arrow[dr, "\llbracket - \rrbracket : IMP \to \mathbb{D}_\bot"'] & \mathrm{ICODE}_{\mathrm{IMP}} \arrow[d, dotted, "\llbracket - \rrbracket : \mathrm{ICODE}_{\mathrm{IMP}} \to \mathbb{D}_\bot"] \\
 & \mathbb{D}_\bot
\end{tikzcd}$$

Figure 5.1: `interp_imp` = `translate` ∘ `interp_icode_imp`

```
            Int i => (case interp_icode_bexp rho b2 of
                      Int j => if i = j then Bool true
                                     else Bool false))
      | IMP.LT  => ...
```

5.6 Sketch of a proof that `interp_imp` = `translate`∘`interp_icode_imp`

Theorem 5.1. `interp_imp` = `translate` ∘ `interp_icode_imp`

Proof. The proof of Theorem 5.1 can be split into three parts, corresponding to the structure of IMP's syntax (see Section 5.3):

1. proof that: `interp_imp_aexp` = `trans_aexp` ∘ `interp_icode_aexp`,
2. proof that: `interp_imp_bexp` = `trans_bexp`∘`interp_icode_bexp` and
3. proof that: `interp_imp` = `translate` ∘ `interp_icode_imp`

As with the proof of Theorem 4.1, we proceed by structural induction on the structure of `aexp`, `bexp` and `imp`. Base cases are enumerations and inductive steps are type constructors.

The proof is straightforward and repetitive, and therefore not reproduced here in full. Two cases suffice to give the flavour of the proof. Line numbers refer to the code listed in Appendix E. (Note that ρ stands for the program state.)

Case $\texttt{Num}(i)$ (part of the $\texttt{aexp}$ type):

$\llbracket\texttt{Num}(i)\rrbracket\rho = \texttt{Int}(i)$
$\llbracket\texttt{translate}(\texttt{Num}(i))\rrbracket\rho = \llbracket\texttt{Val}(\texttt{v} = \texttt{IMP.NUM}\, i ...)\rrbracket\rho$
$\llbracket\texttt{Val}(\texttt{v} = \texttt{IMP.NUM}\, i ...)\rrbracket\rho = \texttt{Int}(i)$

Case $\texttt{While}(b, c)$ (part of the $\texttt{imp}$ type):

$$\llbracket\texttt{While}(b, c)\rrbracket\rho = \begin{cases} \rho & \text{if } \llbracket b\rrbracket\rho = \texttt{Bool}\,\texttt{false} \\ \llbracket\texttt{While}(b, i)\rrbracket\rho' & \text{if } \llbracket b\rrbracket\rho = \texttt{Bool}\,\texttt{true} \end{cases}$$

where $\rho' = \llbracket c\rrbracket\rho$
$\llbracket\texttt{translate}(\texttt{While}(b, c))\rrbracket\rho = \llbracket\texttt{Iterate}(\texttt{guards} = [(g_1, g_2)]\rrbracket\rho$
where $g_1 = \texttt{trans\_bexp}(b)$ and $g_2 = \texttt{translate}(c)$

$$\llbracket\texttt{Iterate}(\texttt{guards} = [(g_1, g_2)]\rrbracket\rho = \begin{cases} \rho \\ \qquad\qquad \text{if } \llbracket g_1\rrbracket\rho = \texttt{Bool}\,\texttt{false} \\ \llbracket\texttt{Iterate}(\texttt{guards} = [(g_1, g_2)]\rrbracket\rho' \\ \qquad\qquad \text{if } \llbracket g_1\rrbracket\rho = \texttt{Bool}\,\texttt{true} \end{cases}$$

where $\rho' = \llbracket g_2\rrbracket\rho$

$\square$

CHAPTER 6

APPLYING EXSTATIC TO `python-csp` APPLICATIONS

The thesis statement in Chapter 1 states that "The thesis of this dissertation is that a syntax-directed static analysis system, capable of checking project-specific violations in a variety of source languages will be of benefit to software projects". The previous Chapters have dealt with the first part of the statement, that such a system should be capable of dealing with a variety of source languages. This Chapter deals with the second part of the statement – that such a system is of *benefit to software projects*.

In the following, an embedded domain specific language (DSL), `python-csp` is introduced, which implements process-oriented concurrency and parallelism, based on Communicating Sequential Processes [43]. This DSL has been implemented by the author (as part of a separate project [71, 74, 75]) in the host language Python [73]. Existing static analysis systems and coding conventions for Python are described in Section 6.2, followed by the implementation of an Exstatic checker for `python-csp` and lessons learned from the experience of constructing it.

6.1 `python-csp`

The design philosophy behind `python-csp` is to keep the syntax of the library as "Pythonic" and familiar to Python programmers as possible, while providing the high-level abstractions found in CSP. In particular, two things distinguish this library from others such as JCSP [114] and PyCSP [9]. Where languages such as Java have strong typing and sophisticated control over encapsulation, Python has a dynamic type system, often using so-called "duck typing" (which means that an object is said to implement a particular type if it shares enough data and operations with the type to be used in the same context as the type). Where an author of a Java library might expect users to rely on the compiler to warn of semantic errors in the type-checking phase, Python libraries tend to trust the user to manage their own encapsulation and use run-time type checking. Although Python is a dynamically typed language, the language is helpful in that few, if any, type coercions are implicit.

CSP [43] contains three fundamental concepts that are implemented directly in `python-csp`: processes, (synchronous) channel communication and non-deterministic choice. `python-csp` provides two ways in which the user may create and use these CSP object types: one method where the user explicitly creates instances of types defined in the library and calls the methods of those types to make use of them; and another where users may use syntactic sugar implemented by overriding the Python built in infix operators.

Operator overloading has been designed to be as close to the original CSP syntax as possible and is as follows:

Syntax	**Meaning**	**CSP equivalent**
$P;\ Q$	Sequential composition of processes	$P\,;\,Q$
$P//[Q]$	Parallel composition of processes	$P \mathbin{\|\|} Q$
$c_1 \mid c_2$	Non-deterministic choice	$c_1 \sqcap c_2$
$n * A$	Repetition	$n \bullet A$
$A * n$	Repetition	$n \bullet A$
$\mathit{Skip}()$	Skip guard, always ready to synchronise	Skip

where:

- n is an integer;
- P and Q are processes;
- A is a non-deterministic choice (or ALT); and
- c_1 and c_2 are channels.

In fact `Skip` is *both* a process which does no work and terminates and a guard which is always ready to synchronise.

Further details of `python-csp` are not relevant to this thesis and a more complete discussion of the features of the library can be found in [73]. Before the discussion of coding conventions for `python-csp` below, it is helpful to see an example of the coding style encouraged by `python-csp`. Listing 6.1 shows a solution to the Sleeping Barber problem posed by Dijkstra [28]. Parts of the program are composed as independent processes, which are run together in parallel by the starting process `main`, and communicate via channels. The program is not written entirely from scratch, it makes use of built in processes (`Printer`) and synchronisation primitives (`Timer` and `BoundedQueue`). The documentation also gives additional hints as to how the programmer intended channels to be used.

```
import random

from csp.csp import *
from csp.builtins import Printer
from csp.guards import Timer
from csp.queue import BoundedQueue

@process
def generate_customers(out_chan, printer):
    """
    readset =
    writeset = out_chan, printer
    """
    customers = ['Michael Palin', 'John Cleese',
                 'Terry Jones', 'Terry Gilliam',
                 'Graham Chapman']
    while True:
        python = random.choice(customers)
        printer.write('%s needs a good shave!' % python)
        out_chan.write(python)

@process
def barber(door, printer):
    """
    readset = door
    writeset = printer
    """
    timer = Timer()
    while True:
        printer.write('Barber is sleeping.')
        customer = door.read()
        printer.write('Barber is awake to shave  %s.'
                      % customer)
        timer.sleep(random.random() * 5)

@process
def main(max_chairs):
    """
    readset =
    writeset =
    """
    door_in, door_out = Channel(), Channel()
    printer = Channel()
    Par(generate_customers(door_in, printer),
        BoundedQueue(door_in, door_out, max_chairs),
        barber(door_out, printer)).start()
```

Listing 6.1: `python-csp` solution to Dijkstra's Sleeping Barber problem [28]

6.2 Related work: Existing standards and lint tools for Python

There are a number of existing coding standards and static analysis systems that can be applied to `python-csp` code. Major changes to the Python language and standard Python and canonical informational resources such as coding standards are reviewed and published as "Python Enhancement

Proposals" or PEPs. PEP8 [113] is the "Style Guide for Python Code" which lists general conventions for Python code. The PEPs also contain style guides for documentation [37] and C code [112] (where C is coupled with Python via a foreign-function interface). Much of PEP8 is devoted to the readability of code, particularly the use of whitespace (spaces, are preferred to tabs, and so on) and naming conventions on the basis that "code is read much more often than it is written". A small number of recommendations relate to general good practice in code, such as "Don't compare boolean values to `True` or `False` using ==".

6.2.1 Static checkers against the PEP8 guidelines

A number of static checkers exist which are able to check for conformance with some or all of the PEP8 guidelines:

`pep8.py` [95] checks whether or not code conforms to the PEP8 coding standards [113]. The `pep8.py` script is an extensible, lexical checker in which new checks are implemented as functions known as plugins. Each plugin may apply to a logical (i.e. tokenised) or physical line of code. The script makes use of the Python `tokenizer` in the standard library.

tabnanny is a script which is included in the Python standard library and can be used to detect whitespace related problems, such as mixing tabs and spaces, inconsistent indentation and so on. Again, this is a lexical checker and does not make use of the AST.

PythonTidy [92] transforms the input code into a program which conforms to PEP8. This is a syntax-directed checker, which uses the standard Python `compiler` module to generate an AST, then transforms the AST into a similar tree where each node is augmented with new methods. Checks for PEP8 conformance and tree rewriting algorithms are then applied to the new syntax tree.

6.2.2 Lint-like tools

Several more generic tools are available to check more general coding standards in Python:

pylint [110] is perhaps the most popular and sophisticated of Python lint tools. pylint includes checks for PEP8 violations and many idiomatic aspects of Python programming, including failure to override an "abstract" method in a subclass (where "abstract" is indicated by raising a `NotImplementedError` exception).

PyFlakes [48] reports two major causes of error in Python code: names which are used before they are defined (or used and not defined) and names which are redefined without having been used. PyFlakes works by visiting AST nodes and keeping track of the scope in which each name is used or defined via stacks. It has a very simple implementation and exists mainly to compensate for some basic checks which are not performed by the Python interpreter.

6.2.3 General `python-csp` coding standards

The tools described above can catch a variety of software defects, or violations of coding standards, but cannot help to address particular errors which relate specifically to `python-csp`. For example, deadlocks can occur when two processes offer to read from a channel and no process is offering to write to the channel. In statically typed languages this error is prevented by making available "reading" and "writing" ends of a channel, so that each process is passed the channel "end" that it requires. This is not something which is checkable in Python, where every object is of type `instance`, and it would not by idiomatically Pythonic to use channel "ends", rather than channels themselves. Ideally a static checker for `python-csp` would be able to check for deadlocks relating to the incorrect use of channels with minimum intrusion for the programmer.

The list below comprises of common errors and convention violations that can be usefully checked by a tool such as Exstatic. The following Chapter will discuss the implementation of an Exstatic extension to do this.

- The `@process` decorator should only be applied to functions (not methods). This should discourage programmers from attempting to share state between "processes" derived from methods which belong to the same object.
- Any channel passed to a process which is both written to and read from should raise a warning.

- Processes should only read or write to / from channels which have been passed to them via their argument list. This is to encourage programmers to think of processes as "owning" channels and to prevent incorrect behaviour which could occur, for example, by having several readers reading from a channel when only one was intended.
- All channels passed to a process should be read from or written to at least once in the process body.

6.2.4 Conclusion

The existing tools reviewed above have a number of advantages. They enforce widely accepted standards and can detect a number of diverse defects. However, these tools do not address problems which can occur when using `python-csp` specifically. `python-csp` implements a message-passing style of concurrency, existing static analysis systems address the style of concurrency which is used in the Python standard library - i.e. a shared memory model.

6.3 Coding conventions and analysis of `python-csp` code

Before writing an Exstatic instance to check `python-csp` code, a clear set of coding conventions are needed. Table 6.1 describes a set of such conventions. Many of these are related to *readsets* and *writesets*. These are metadata within comments against `python-csp` processes which describe which channels within the scope of the process are intended to be read from or written

Code	Message
I001	Function is a CSP process or server process
W001	Channel in both readset and writeset.
W002	No readset given in documentation.
W003	No writeset given in documentation.
E001	Process / forever decorator wraps a method, not a function.
E002	Channel in readset is not a formal parameter to this process.
E003	Channel in writeset is not a formal parameter to this process.
E004	Channel appears in documented readset but not read from in function body.
E005	Channel is read from in function body but does not appear in documented readset
E006	Channel appears in documented writeset but not written to in function body.
E007	Channel is written to in function body but does not appear in documented writeset
E008	Imported thread, threading or multiprocessing libraries

Table 6.1: CSPlint errors, warnings and information messages. Codes beginning with *I* denote information, code beginning with *W* warnings and *E* errors.

to (see the example in Listing 6.2). In a statically compiled language such as Java, a CSP library such as JCSP [114] may use the type system to prevent some deadlocks. Processes can be passed a read "end" or write "end" of a channel, in much the same way as the UNIX API provides read and write "ends" of pipes. This way, a programmer is prevented from writing down a channel which is only intended for reading, which could cause a deadlock. In a dynamically typed language, such as Python, deadlocks such as this cannot be detected at compile time by a type checker. Indeed, culturally, Python programmers have an expectation that APIs work in a forgiving manner, i.e. that methods and functions can be callable by client code and if called wrongly should throw an appropriate exception. Therefore, to be more "Pythonic", `python-csp` allows client code to pass whole channel objects to other processes, allowing any process to read and write to any channel within its scope. The usual Python practice of throwing exceptions is problematic in this context, as a deadlock will not terminate in order for an exception to be thrown! Readsets and writesets are a compromise solution to this problem - programmers can document how they expect channels to be used, much like pre- and post-conditions and invariants may be documented, but the library does not itself complain about differences between the documentation and implementation.

```
@process
def example(chan1, chan2):
    """
    readset = chan1
    writeset = chan2
    """
    r = chan1.read()
    chan2.write('foobar')
```

Listing 6.2: Readset / writeset example

To ameliorate this lack on built in type analysis, and stay true to the culture of Python programming, an instance of Exstatic called *csplint* has been implemented to catch violations of the readset / writeset convention and other errors.

6.4 Implementing a checker for `python-csp`

The implementation of csplint simply uses the visitor pattern [36] to walk a Python AST (generated by the Python standard library) and emit an ICODE version of the AST. The ICODE representation of `python-csp` code is then walked to detect violations of the conventions listed in Table 6.1. No complex analysis (such as the solution of dataflow equations) took place, and standard software engineering practices [36] were used. This is a straightforward process, as Python (like many modern programming languages) already provides APIs to access Python parse trees. This means that the first part of the implementation of csplint is straightforward –

```
import ast
import pyicode

class Ast2IcodeVisitor(ast.NodeTransformer):
    """
    AST Visitor which creates an ICODE translation
    of the AST, stored in its ICODE attribute.
    """
    def __init__(self):
        super(AST2ICODEVisitor, self)

    def visit_Assign(self, node):
        self.emit(pyicode.Assign(lvalue=...,
                                 rvalue=...,
                                 annotations={}))
        ast.generic_visit(self, node)
        return

  ...
```

Listing 6.3: Implementing a parser for csplint

Listing 6.3 shows a fragment of the class responsible for converting Python source code to ICODE, using the `pyicode` library which is part of the current (Python) implementation of Exstatic. This class works by traversing a Python abstract syntax tree (obtained directly from calling the Python parser) and emitting ICODE nodes which are constructed from the information contained in each Python node. If desirable, the visitor methods here can keep as much information as is available from the Python parser and store that information in the `annote` (metadata) lists. This might include information such as the file name that the original source code came from, the line number of each expression or statement, and so on.

Once an ICODE tree has been constructed, any available ICODE checker can be applied to it. The examples here have all been written in Python, but if it happened that useful checkers had been written in, say, Java, the ICODE tree could be converted to a flat XML representation (using an ICODE visitor which comes with `pyicode`) and the Java checkers could be applied directly to that XML representation.

An ICODE checker is then much like the parser above. A visitor is written which visits each node and computes whatever is necessary to check for violations of the coding standards concerned.

```
class ProcessChecker(visitor.ICODEVisitor):

    ...

    def is_process(self, annotations):
        """Determine whether or not the current function
        is a CSP process.
        """
        for annote in annotations:
            if ((annote[name] == 'process' and
                 annote[type] == 'decorator'):
                return True
        return False

    def visitFunction(self, node):
        """Visit function definition.
        """
        # If this function definition is not a
        # CSP process, ignore it.
        if (node.annotations is None or
            !self.is_process(node.annotations)):
            return

        # Store useful information about this process.
        self.current_process = node.annotation[name]
        self.current_process_lineno = \
                             node.annotation[lineno]

        # 'I001':'Function is a CSP process process'
        exstatic.cspwarnings.create_error(
                         self.filename,
                         self.current_process_lineno,
                         self.current_process,
                         'I001')

        # 'W004':'@process applied to method (rather
        # than function)'
        if 'self' in node.argnames:
            exstatic.cspwarnings.create_error(
                             self.filename,
                             self.current_process_lineno,
                             self.current_process,
                             'W004')

        return
```

Listing 6.4: Implementing a checker for csplint

Listing 6.4 shows a fragment of the part of csplint which checks that `@process` is applied to a function, not a method. These are items I001 and W004 in Table 6.1. The Exstatic libraries are used to create errors and warning messages, and when the available checkers are run over `python-csp` code Exstatic will automatically collate and present the relevant messages to the user. Listing 6.5 shows how to achieve this in code, using the built in Python parser to generate the initial abstract syntax tree.

```
checkers = [csp.lint.channels.ChannelChecker,
            csp.lint.processes.ProcessChecker]
```

```

def run(filename, excluded=[]):
    exstatic.cspwarnings.reset_errors()
    for checker in checkers:
        lint = checker(filename)
        compiler.walk(compiler.parseFile(filename),
                      lint,
                      walker=lint,
                      verbose=5)
    exstatic.cspwarnings.print_errors(excluded=excluded)
    return
```

Listing 6.5: Running Exstatic checkers over Python code

6.4.1 Building checkers for other languages

The focus of this Chapter is on the specific example of building and using csplint on an open source project. Appendix C contains two published papers which describe checkers for inline documentation (Javadoc) and a markup language (HTML). These were implemented in a similar manner to the example above. However, in the case of csplint the Python parser could be used to provide an abstract syntax tree of Python source code. In the case of HTML and Javadoc such convenient tools are not built into the Python standard library, so parsers needed to be built from scratch. In the case of Javadoc this can be done very simply with a regular expression library. HTML, however, is a context free language, and in that case a top-down parser was written, and from that ICODE trees were generated and the static analyses were implemented in the style above.

6.5 Results

An obvious target for csplint is the `python-csp` code base itself. csplint has been applied to changeset `bc4243a38823` on the “default” and “new_tests” branches of the version control repository[1]. The results from the two branches were aggregated, results which simply provided information (error code I001 in Table 6.1) were discarded, along with any duplicate reports.

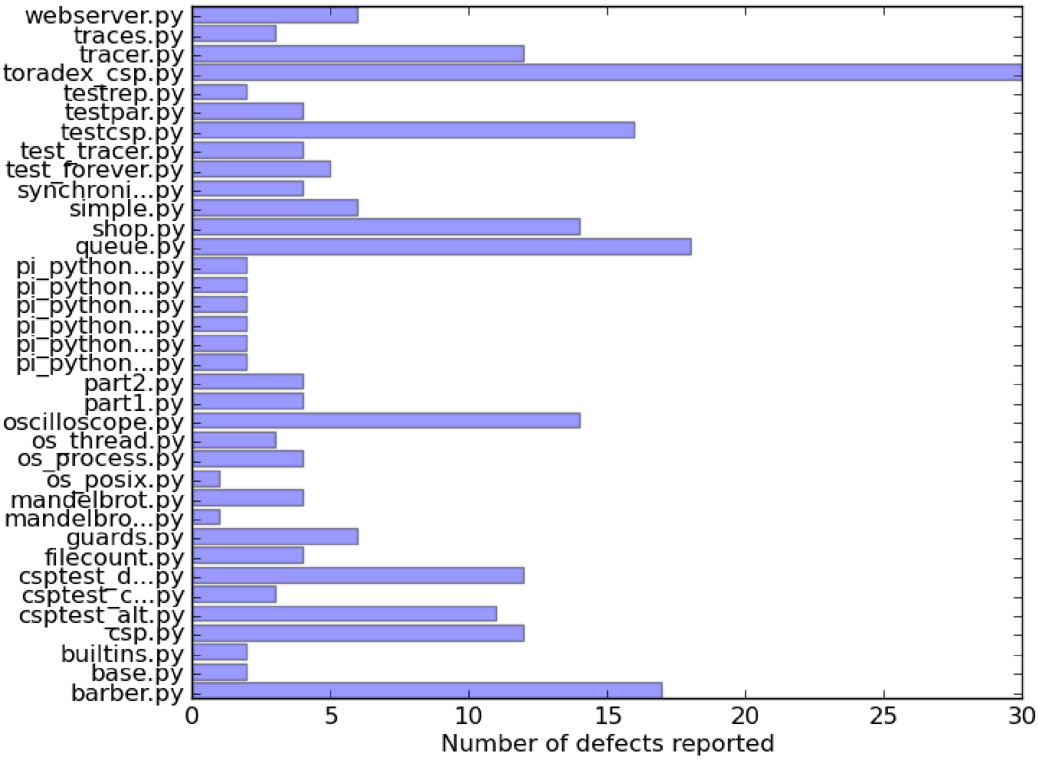


Figure 6.1: Number of CSP related defects found by csplint in each file of the python-csp project

---

[1] http://code.google.com/p/python-csp/

csplint found 4 distinct warning types in 22 files (159 warnings in total) and 7 distinct error types in 27 files (81 errors in total) on this changeset. Figure 6.1 shows the errors and warnings detected per file. This makes plain a number of problems which would have been time consuming to check by hand. Many errors are the result of code which was written before the readset / writeset convention was written. In particular, the file `toradex_csp.py` is an implementation of a library to provide access to the Toradex Oak series of USB sensors. Each type of sensor has a distinct process written for it, which will communicate sensor data along a channel. In changeset `bc4243a38823` the documentation for this module had not been updated to include details of its channel usage.

Figure 6.2 shows the number of errors per error type. This shows a number of important errors which, again would have been time consuming to detect by eye. Error E008 is an excellent example of this. Users of the `python-csp` library are not intended to use shared-memory concurrency libraries from the Python standard library with `python-csp`. These forms of concurrency will interact with `python-csp` processes in unpredictable ways which can easily cause intractable errors. However, csplint has detected 36 errors of this type. These belong to three categories:

1. 17 violations of the coding convention within the core library. This is intentional as these modules contain the definitions of CSP processes and channels. In normal use, errors from these files would be excluded from being reported by csplint.

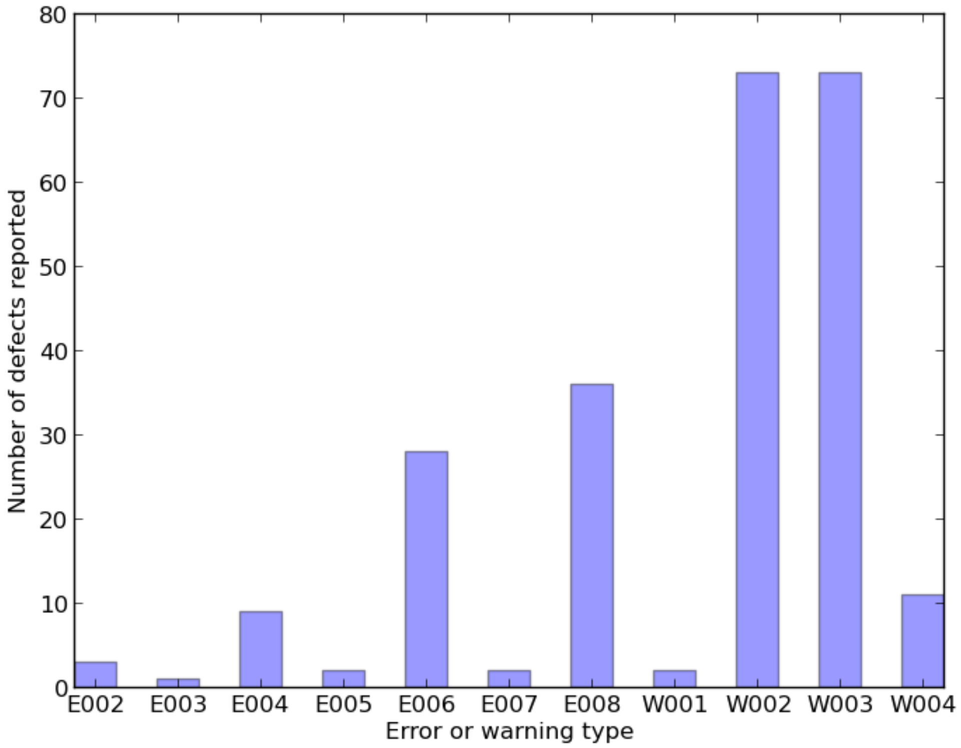


Figure 6.2: Number of CSP related defects found by csplint in the python-csp project

2. 12 violations where the shared memory libraries were used for a safe purpose, such as determining the number of cores available on the current hardware. This suggests that similar facilities should be added as a new feature of `python-csp`.

3. 7 violations which were written by an open source contributor who was new to the project and had not been clearly appraised of its coding conventions. These are real bugs which include unsafe mixing of concurrency libraries and deadlocks.

6.6 Conclusions

This Chapter has considered whether Exstatic can be of benefit to real software projects. An instance of Exstatic called csplint was written by the author to check for violations of coding conventions in an embedded DSL. The conventions addressed problems which could not be detected by the type system of the language or by existing static analysis systems. Since the implementation of csplint relied only on standard software engineering patterns and techniques Exstatic was a tractable and efficient analysis system to use and in this case all analyses were both sound and complete. A number of real bugs were discovered, as well as some indications of new features the library could implement. In a medium sized project (`python-csp` contains around 6.8kLOCs of Python code) these defects, scattered over just under 30 files, would be time consuming to detect by eye. Although the errors caught here are violations of idioms relating to how the `python-csp` library is intended to be used, those idioms are designed to prevent serious semantic bugs such as deadlock.

CHAPTER 7

CONCLUSIONS

The thesis statement in Chapter 1 stated:

> *"a syntax-directed static analysis system, capable of checking project-specific violations in a variety of source languages will be of benefit to software projects."*

This statement firstly claims that a syntax-directed static analysis system can be built which will be able to represent source code from a variety of different languages.

This part of the thesis statement can be broken down further still. Firstly there is the idea that syntax-directed static analysis is useful at all, then secondly there is the claim that it is possible to apply such a system to a variety of source languages. It is important to understand the meaning of the term "syntax-directed", which (in this thesis) is contrasted with the style of static analysis often found in a compiler, or those which rely on formal methods such as automated theorem proving (as used by [96]) or model checking (as in [24]).

The distinction between much of the related work, and this thesis lies in the fact that Meta-Level Compilation, FindBugs and other tools typically apply to code in one programming language (perhaps in the case of

FindBugs, any source language which compiles to JVM bytecode). Those few tools which do address more than one programming language have not produced formal proofs that they can perform translations into their intermediate representations without losing information.

The thesis statement above is directed specifically at detecting defects in a variety of source languages, which may be general purpose programming languages, or domain specific languages (such as the syntax of Javadoc comments [85]) or "little" languages, such as those used by software builders like Make, Scons, Bazaar, and so on. This "universality" of syntax-directed static analysis was addressed by a direct analysis of ICODE, the intermediate language introduced in Chapter 3. Chapters 4, and 5 presented proofs that archetypal functional, logic and imperative languages can be translated into ICODE, in a manner which preserves the (denotational) semantics of the source language. Since ICODE is aimed at representing code from a variety of source languages, it is important that authors of checking systems can translate programs to ICODE in a convenient manner, rather than, say, writing a custom parser for every language of interest. Practically, this is likely to involve the use of APIs and tools designed for the language in question and written in that language. Convenient tools which come with ICODE (for example tools to output a graph of an ICODE structure, or present a list of possible defects to the user) have been written in Python, which may or may not be the language of choice for any given project. To work around this, the usual expression of an ICODE structure is in XML, which is a markup language with parsers and code generators written for most common general-purpose programming languages.

It is not, however, enough to show that a tool is useful in theory, it must also be applicable to real software projects. Chapter 6 presents csplint, an instance of Exstatic which for `python-csp` – a realisation of Hoare's CSP [43] in the language Python. The checker dealt with an issue in the Python type system, that is, how to detect errant calls to channel `read()`s and `write()`s which might lead to deadlock. In a statically typed language such as Java or OCCAM-π, these errors would be detected at compile-time by a type checker. Python does not have such a system (as standard). To compensate, a little language was created to define the "readset" and "writeset" of the channels passed into a new process as arguments. Actual calls to `read()`s and `write()`s within the body of the process are checked against the documented invariants.

Implicit in this discussion is the idea that "ordinary" programmers will be able to use the system presented in this thesis, and write new checking routines for it. By "ordinary" here we mean programmers with no special expertise in compiler construction, semantics or computability. Chapter 6 demonstrated that it is possible to produce static analyses with ICODE which are idiomatic in the language in which they are written and use common design patterns [36] to achieve their goals.

7.1 Further work

The framework presented in this thesis leaves scope for further work in a number of directions. The central motivation of the thesis is that simple checks for violations of coding conventions can assist programmers in finding errors in code early in the development cycle. Like all source code analysis systems, the examples presented here use the source code itself to detect errors. However, software projects often contain a wealth of additional information, which is available at "compile time" and can be drawn upon to assist in detecting violations of coding standards. For example, the FixCache and BugCache algorithms [57, 90, 98] make use of version control history and issue trackers to predict which locations in the source code are likely to contain errors. This strategy is based on the idea that errors tend to occur in clusters, either being committed to the repository at approximately the same time, or occurring in the same files or scopes. A potentially useful addition to existing Exstatic systems, such as those described in Chapter 6, would include information from version control history, bug trackers, wikis and in-line documentation to determine whether adding this information makes it possible to detect more violations of coding conventions than using the source code alone. Since documentation and meta-data is not necessarily written in a strict style, or in a language amenable to simple parsing, more sophisticated data mining techniques than the ones presented here may have to be employed.

Assessing how useful Exstatic could be when applied to a live project is a difficult task. The approach taken in this thesis has been to provide an exemplar project and show that a number of violations of coding standards can be detected by an Exstatic checker (further examples are provided in Appendix C). This addresses the question of whether Exstatic can find errors, but not the question of how Exstatic can compare to other methods, such as unit testing. One particularly interesting question is whether Exstatic can usefully detect errors earlier in the development cycle than other techniques. Live projects with strong unit testing disciplines (as in *test-driven development* and similar methods) can be used to examine this question further. A strategy for investigating this research question might work as follows:

1. collect, either by hand or automatically, a set of simple coding conventions from the current set of unit tests, or project documentation.
2. determine when each coding convention was introduced to the software. A straight forward way to do this would be to assume that the date at which a unit test is added to the code base is the same date on which the coding convention started to be applied to the project.
3. write a set of Exstatic analysers which exercise the conventions discovered in step 1.
4. for each changeset in the version control history automatically run both the Exstatic checks and the unit tests, and record results of each. Ignore unit test errors (rather than test *failures*), as these indicate that the

code which was exercised by the tests either had not yet been added to version control, or had been refactored in some future changeset.

5. for each violation of the coding standards from step 1. determine whether the Exstatic analysers from step 3. were able to find violations of the standards earlier or later in the version control history of the project than the unit tests.

In the context of `python-csp`, the work presented here touches on, but does not exploit the large amount of literature on applying formal methods to process-oriented programs. In particular, [63] presents a set of design patterns for producing deadlock-free CSP style code. It may be possible to detect adherence to these statically (i.e. via abstract syntax tree of a `python-csp` program). This may prove more useful than instrumenting the Python interpreter, as if a deadlock occurs in a program it may prevent the static analyser from terminating.

BIBLIOGRAPHY

[1] ISO/IEC 19501:2005 information technology – open distributed programming – Unified Modelling Language (UML) version 1.4.2.

[2] PMD. `http://pmd.sourceforge.net/`. [Accessed 1 March 2013].

[3] B. Adelson. Problem solving and the development of abstract categories in programming languages. *Memory and Cognition*, 9:422–433, 1981.

[4] B. Adelson. When novices surpass experts: How the difficulty of a task may increase with expertise. *Journal of Experimental Psychology: Learning, Memory and Cognition*, 10(3):483–495, July 1984. Reprinted in Human Factors in Software Development (2nd ed.); Bill Curtis (ed.). 1985. IEEE Computer Society Press: Washington, DC. 55-67.

[5] Gerald Aigner, Amer Diwan, David L. Heine, Monica S. Lam, David L. Moore, Brian R. Murphy, and Constantine Sapuntzakis. An overview of the SUIF2 compiler infrastructure. Technical report, Stanford University, 2000.

[6] Lloyd Allison. An executable prolog semantics. *Algol Bulletin*, (50):10–18, December 1983.

[7] Lloyd Allison. Programming denotational semantics. *The Computer Journal*, 26(2):164–174, 1983.

[8] Lloyd Allison. Programming denotational semantics II. *The Computer Journal*, 28(5):480–486, 1985.

[9] Otto J. Anshus, John Markus Bjrndalen, and Brian Vinter. PyCSP - Communicating Sequential Processes for Python. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 229–248, July 2007.

[10] Andrew Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[11] John Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[12] N. Benton. Machine obstructed proof: How many months can it take to verify 30 assembly instructions? In *ACM SIGPLAN Workshop on Mechanizing Metatheory*. ACM, 2006.

[13] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[14] R.S. Boyer, B. Elspas, and K.N. Levitt. SELECT - a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245, apr 1975.

[15] Marc M. Brandis. *Optimizing Compilers for Structured Programming Languages*. PhD thesis, ETH Zürich, 1995. `ftp://ftp.inf.ethz.ch/`

pub/publications/dissertations/th11024.ps [Accessed 1 March 2013].

[16] Oliver Burn. The Checkstyle project. http://checkstyle.sourceforge.net/ [Accessed 1 March 2013].

[17] W. Bush, J. Pincus, and D. Siela. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30(7):775–802, June 2000.

[18] Laurian M. Chirica and David F. Martin. An approach to compiler correctness. In *Proceedings of the international conference on Reliable software*, pages 96–103, New York, NY, USA, 1975. ACM.

[19] Andy Chou, Benjamin Chelf, Dawson Engler, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. *Operating Systems Design and Implementation*, 2000.

[20] Alonzo Church. A set of postulates for the foundations of logic. *Annals of Mathematics*, 2(33):346–366, 1932. and 2(34):839-846, 1933.

[21] Alonzo Church. *The Calculi of Lambda Conversions.* Princeton University Press, Princeton, 1941.

[22] E. M. Clarke. The characterization problem for hoare logics. In *Proc. of a discussion meeting of the Royal Society of London on Mathematical logic and programming languages*, pages 89–106, Upper Saddle River, NJ, USA, 1985. Prentice-Hall, Inc.

[23] M.E. Conway. Proposal for an UNCOL. *Communications of the ACM*, 1(10):5–8, 1958.

[24] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *In Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448. ACM Press, 2000.

[25] Ian F. Darwin. *Checking C Programs With lint.* O'Reilly & Associates, October 1998.

[26] David Detlefs, K. Rustan, M. Leino, Greg Nelson, and James B. Saxe. Extended Static Checking. Technical Report 159, Compaq Systems Research Centre, Palo Alto, CA, December 1998.

[27] Edsgar W. Dijkstra. *A Discipline of Programming.* Prentice-Hall Series in Automatic Computing. Prentice-Hall, 1976.

[28] Edsger W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[29] Edsger W. Dijkstra. On the interplay between mathematics and programming. In *Program Construction, International Summer School*, pages 35–46, London, UK, 1979. Springer-Verlag.

[30] Carolyn K. Duby, Scott Meyers, and Steven P. Reiss. CCEL: A metalanguage for C++. In *USENIX C++ Conference Proceedings*, August 1992.

[31] ECMA International. Common Language Infrastructure (CLI). Technical Report TG3, June 2012. `http://ecma-international.org/publications/standards/Ecma-335.html` [Accessed 1 March 2013].

[32] Dawson Engler and Andy Chou. Metal: A language and system for building lightweight, system-specific software checkers, analyzers and optimizers. Available upon request: acc@cs.stanford.edu, 2000.

[33] David Evans. Using specifications to check source code. Technical Report MIT/LCS/TR-628, MIT Laboratory for Computer Science, June 1994.

[34] David Evans, John Guttag, Jim Horning, and Yang Meng Tan. LCLint: a tool for using specifications to check code. In *SIGSOFT Symposium on the Foundations of Software Engineering*, December 1994.

[35] David Evans and David Larochelle. *Splint User's Manual Version 3.1.1*. University of Virginia, Department of Computer Science, April 2003. `http://www.splint.org/manual/manual.html` [Accessed 1 March 2013].

[36] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, October 1994.

[37] David Goodger and Guido van Rossum. PEP 257: docstring conventions. `http://www.python.org/dev/peps/pep-0257/`, May 2001. [Accessed 1 March 2013].

[38] Andrew D. Gordon and Don Syme. Typing a multi-language intermediate code. Technical Report MSR-TR-2000-106, Microsoft Research, 2000.

[39] John Guttag, S. J. with Garland, James Horning, K. D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Varlag, 1993.

[40] Bruce K. Haddon and William M. Waite. Experience with the universal intermediate language Janus. *Software Practice & Experience*, 8(5):606–616, September 1978.

[41] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Programming Language Design and Implementation (PLDI)*. ACM SIGPLAN, 2002.

[42] I. J. Hayes, editor. *Specification Case Studies.* Prentice-Hall, London, 1987.

[43] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, New Jersey, 1985.

[44] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.

[45] C.A.R. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.

[46] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

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

[48] Divmod Inc. PyFlakes. `http://launchpad.net/pyflakes` [Accessed 1 March 2013].

[49] Fortify Software Inc. RATS: Rough Auditing Tool for Security. `http://code.google.com/p/rough-auditing-tool-for-security`. [Accessed 1 March 2013].

[50] Pure Software Inc. Purify user's guide, 1994.

[51] Daniel Jackson. Aspect: A formal specification language for detecting bugs. Technical Report MIT/LCS/TR-543, MIT Laboratory for Computer Science, June 1992.

[52] Stephen C. Johnson. lint, a C program checker. Unix Programmer's Manual, AT&T Bell Laboratories, 1978.

[53] Stephen C. Johnson. A portable compiler: Theory and practice. In *Proceedings of the Fifth ACM Symposium on Principles of Programming Languages*, pages 97–104, January 1978.

[54] C.B. Jones. *Systematic Software Development Using VDM.* Prentice Hall International, 1986.

[55] Simon Peyton Jones, Alan Blackwell, and Margaret Burnett. A user-centred approach to functions in Excel. In *In Proceedings of the 2003 ACM International Conference on Functional Programming (ICFP'03)*, Uppsala, August 2003.

[56] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, 2 edition, 1988.

[57] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, page 489498, Washington, DC, USA, 2007. IEEE Computer Society.

[58] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.

[59] B. Liskov, R. Atkinson, T. Boom, E. Moss, J. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual*, 1981.

[60] Tom Lord. Application specific static code checking for C programs: Ctool. in ˜Twaddle: A Digital Zine (version 1.0), August 1997.

[61] Stavros Macrakis. From UNCOL to ANDF: Progress in standard intermediate languages. Technical report, Open Software Foundation, January 1992.

[62] Steve Maguire. *Writing Solid Code.* Microsoft Press, 1993.

[63] J.M.R. Martin. *The Design and Construction of Deadlock-Free Concurrent Systems.* PhD thesis, University of Buckingham, UK, 1996.

[64] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction.* Microsoft Press, 1993.

[65] Bertrand Meyer. Eiffel: A language for software engineering. Technical Report TR-CS-85-19, Uiversity of California, Santa Barbara, 1985.

[66] Bertrand Meyer. Design by contract. Technical Report TR-EI-12/CO, Interactive Software Engineering Inc., 1986.

[67] Robin Milner. *Communication and Concurrency.* Prentice Hall International, 1989.

[68] Robin Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.

[69] C. C. Morgan. *Programming from Specifications.* Prentice Hall International, second edition, 1994.

[70] C. C. Morgan, K. A. Robinson, and P. H. B. Gardiner. On the refinement calculus. Technical report, Programming Research Group, 1988.

[71] S. Mount, R.M. Newman, E. Gaura, and J. Kemp. Sensor: an algorithmic simulator for wireless sensor networks. In *In Proceedings of Eurosensors 20*, volume II, pages 400–411, Gothenburg, Sweden, 2006.

[72] S. Mount, J. Shuttleworth, and R. Winder. *Python for the Rookies.* Thomson Learning (now Cengage Learning), 2008. ISBN 1844807010.

[73] Sarah Mount, Mohammad Hammoudeh, Sam Wilson, and Robert M. Newman. CSP as a domain specific language embedded in python and jython. In Herman Roebbers Peter Welch, editor, *Communicating Process Architectures 2009.* IOS Press, 2009.

[74] S.N.I. Mount, R.M. Newman, and E.I. Gaura. A simulation tool for system services in ad-hoc wireless sensor networks. In *In Proceedings of NSTI Nanotechnology Conference and Trade Show (Nanotech'05)*, volume 3, pages 423–426, Anaheim, California, USA, May 2005.

[75] S.N.I. Mount, R.M. Newman, E.I. Gaura, and J.R. Kemp. Sensor: an algorithmic simulator for wireless sensor networks. In *Proceedings of Eurosensors*, pages 400–411, Gothenburg, Sweden, 2006.

[76] S.N.I. Mount, R.M. Newman, and R.J. Low. Checking marked-up documentation for site-specific standards. In *In Proceedings of the 23nd Annual International Conference on Design of Communication*, pages 76–79, Coventry, UK, 2005. ACM Press.

[77] S.N.I. Mount, R.M. Newman, R.J. Low, and A. Mycroft. Exstatic: A generic static checker applied to documentation systems. In *In Proceedings of the 22nd Annual International Conference on Design of Communication*, pages 52–57. ACM Press, 2004.

[78] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. *SIGPLAN Not.*, 39(4):612–625, 2004.

[79] Greg Nelson. *Techniques for Program Verification.* PhD thesis, Stanford University, 1980.

[80] Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, 1989.

[81] P.A. Nelson. A comparison of PASCAL intermediate languages. *SIGPLAN Notices*, 18(8):208–213, 1979.

[82] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer, 1st edition, 1999.

[83] Neil Norwitz. pychecker. `http://pychecker.sourceforge.net`. [Accessed 1 March 2013].

[84] Open Software Foundation. Architecture neutral distribution format: A white paper. Technical report, November 1990.

[85] Oracle Corporation. How to write doc comments for the Javadoc tool. `http://www.oracle.com/technetwork/java/j2se/documentation/index-137868.html`. [Accessed 1 March 2013].

[86] Oracle Corporation. Code conventions for the Java programming language. `http://www.oracle.com/technetwork/java/codeconv-138413.html`, 1995-1999. [Accessed 1 March 2013].

[87] Andrew M. Pitts. Denotational semantics. Lecture Notes, University of Cambridge, 1999.

[88] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, 1977.

[89] Federico Mena Quintero, Miguel de Icaza, and Morten Welinder. GNOME programming guidelines. The Free Software Foundation, 1999. `ftp://ftp.gnome.org/pub/GNOME/teams/docs/devel/guides/programming-guidelines/`.

[90] Foyzur Rahman, Daryl Posnett, Abram Hindle, Earl Barr, and Premkumar Devanbu. BugCache for inspections: hit or miss? In

Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ESEC/FSE '11, page 322331, New York, NY, USA, 2011. ACM.

[91] Eric S. Raymond. *The Cathedral and the Bazaar*. O'Reilly & Associates, October 1999.

[92] Chuck Rhode. PyTidy. `https://pypi.python.org/pypi/PythonTidy/`. [Accessed 1 March 2013].

[93] Martin Richards. The portability of the BCPL compiler. *Software Practice and Experience*, 1(2):135–146, 1971.

[94] Martin Richards. Bootstrapping the BCPL compiler using INTCODE. In *Machine Oriented Higher Level Languages*, pages 253–264, Amsterdam, North Holland, 1974.

[95] Johann C. Rocholl. pep8.py. `http://pep8.readthedocs.org`. [Accessed 1 March 2013].

[96] K. Rustan, M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user manual. Technical Report 2000-002, Compaq Systems Research Center, October 2000.

[97] Sukyoung Ryu and Norman Ramsey. Source-level debugging for multiple languages with modest programming effort. In *14th International Conference on Compiler Construction*, pages 10–26, 2005.

[98] Caitlin Sadowski, Chris Lewis, Zhongpeng Lin, Xiaoyan Zhu, and E. James Whitehead,Jr. An empirical analysis of the FixCache algorithm. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, page 219222, New York, NY, USA, 2011. ACM.

[99] Dana Scott. A type-theoretic alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121:411–440, 199.

[100] Michael V. Scovetta. YASCA. `http://www.scovetta.com/yasca/documentation.html`. [Accessed 1 March 2013].

[101] James Shuttleworth and Sarah Mount. An introduction to python programming. Department of Creative Computing Lecture Notes, University of Coventry, 2005.

[102] Harvey Siy and Lawernce Votta. Does the modern code inspection have value? In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, 2001.

[103] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International, 1989.

[104] Richard Stallman. GNU coding standards. `http://www.gnu.org/prep/standards.html`, October 2001. [Accessed 1 March 2013].

[105] T.B. Steel. UNCOL: Universal computer oriented language revisited. *Datamation*, 6(1):18–20, 1960.

[106] T.B. Steel. A first version of UNCOL. In *Proceedings of the Western Joint IRE-AIEE-ACM Computer Conference*, pages 371–378, Los Angeles, California, 1961.

[107] T.B. Steel. UNCOL: The myth and the fact. *Annual Review in Automatic Programming*, 2:325–344, 1961.

[108] J. Strong, J. Wegstein, A. Tritter, J. Olsztyn, O. Mock, and T.B. Steel. The problem of programming communication with changing machines: A proposed solution: Report of the share ad-hoc committee on universal languages. *Communications of the ACM*, 1(8):12–18, 1958. and 1:9,9-15.

[109] G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical Report Planning Report 02-3, Prepared by RTI for the National Institute of Standards and Technology (NIST), May 2002. `http://www.nist.gov/director/planning/report02-3.pdf` [Accessed 1 March 2013].

[110] Sylvain Thénault. pylint. `http://www.pylint.org`. [Accessed 1 March 2013].

[111] Linus Torvalds. Linux kernel coding style. `http://www.kernel.org/doc/Documentation/CodingStyle/` [Accessed 1 March 2013], 2000.

[112] Guido van Rossum. PEP 7: style guide for C code. `http://www.python.org/dev/peps/pep-0007/`, July 2001. [Accessed 1 March 2013].

[113] Guido van Rossum and Barry Warsaw. PEP 8: style guide for Python code. `http://www.python.org/dev/peps/pep-0008/`, November 2006.

[114] P. H. Welch. Process oriented design for Java: concurrency for all. pages 51–57. CSREA Press, 2000.

[115] David Wheeler. `http://www.dwheeler.com/flawfinder/`. [Accessed 1 March 2013].

[116] Maurice Wilkes. *Memoirs of a Computer Pioneer*. MIT Press, 1985.

[117] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

[118] Anna Zaks and Amir Pnueli. Covac: Compiler validation by program analysis of the cross-product. In *FM '08: Proceedings of the 15th international symposium on Formal Methods*, pages 35–51, Berlin, Heidelberg, 2008. Springer-Verlag.

APPENDIX A

FULL CODE LISTINGS FROM CHAPTER 4

```
structure PCF =
struct

(* Values in icode.Val expressions. *)
datatype value = NOUGHT | TRUE | FALSE

(* Operators in icode.Arith expressions. *)
datatype aop    = SUCC | PRED

(* Operators in icode.Bool expressions. *)
datatype bop    = ZERO

(* Operators in icode.Prim expressions. *)
datatype eop    = LAMBDA | APPLY | FIX

(* Annotations - types. *)
datatype annote = NAT | BOOL | Fn of annote * annote

end (* structure PCF *)

structure ICODE-PCF =
struct

(* PCF types and expressions. *)
datatype tau = NAT | BOOL | Fn of tau * tau
datatype pcf = NOUGHT | TRUE | FALSE
```

```
       | Succ of pcf | Pred of pcf
       | Zero of pcf
       | IfElse of (pcf * pcf * pcf)
       | Var of string
       | Lambda of (string * tau * pcf)
       | Apply of (pcf * pcf)
       | Fix of pcf

(* The semantic domain of PCF (as a smash sum of domains). *)
datatype domain = Nat of int
    | Bool of bool
    | Fun of domain -> domain
    | BOTTOM

(*
 * Lookup the value bound to a variable in an environment.
 * val env : string * domain list -> string -> domain
 *)
fun env ((x, v)::rho) s = if x = s then v else env rho s
  | env []            s = BOTTOM

(*
 * Interpret a PCF expression.
 * val interp_pcf : string * domain list -> pcf -> domain
 *)
fun interp_pcf rho (NOUGHT)             = Nat 0
  | interp_pcf rho (TRUE)               = Bool true
  | interp_pcf rho (FALSE)              = Bool false
  | interp_pcf rho (Succ e)             =
    (case interp_pcf rho e of Nat n => Nat (n + 1)
              | _ => BOTTOM)
  | interp_pcf rho (Pred e)             =
    (case interp_pcf rho e of Nat n => Nat (n - 1)
                                | _ => BOTTOM)
  | interp_pcf rho (Zero e)             =
     (case interp_pcf rho e of Nat n =>
        if n = 0 then Bool true else Bool false
         | _ => BOTTOM)
  | interp_pcf rho (IfElse(M1, M2, M3)) =
    (case interp_pcf rho M1 of Bool b =>
       if b then interp_pcf rho M2
       else interp_pcf rho M3
        | _ => BOTTOM)
  | interp_pcf rho (Var x)              = env rho x
  | interp_pcf rho (Lambda(s, t, M))    =
    Fun (fn v => interp_pcf ((s,v)::rho) M)
  | interp_pcf rho (Apply(M1, M2))      =
    (case interp_pcf rho M1 of Fun ff => ff (interp_pcf rho M2)
                                  | _ => BOTTOM)
  | interp_pcf rho (Fix M)              = (* Omitted. *)

(*
 * Convert PCF types into ICODE annotations.
 * val tau2ann : tau -> PCF.annote
 *)
fun tau2ann NAT          = PCF.NAT
  | tau2ann BOOL         = PCF.BOOL
  | tau2ann (Fn(t1, t2)) = PCF.Fn(tau2ann t1, tau2ann t2)

(*
 * Translate PCF expressions into ICODE.
 * val translate : pcf -> icode.icode
 *)
fun translate (NOUGHT)           =
    Val{v=PCF.NOUGHT, annote=[]}
  | translate (TRUE)             =
    Val{v=PCF.TRUE,   annote=[]}
  | translate (FALSE)            =
    Val{v=PCF.FALSE,  annote=[]}
  | translate (Succ pcf)         =
    Arith{e1=(translate pcf), e2=EPSILON,
          aop=PCF.SUCC, annote=[]}
  | translate (Pred pcf)          =
    Arith{e1=(translate pcf), e2=EPSILON,
          aop=PCF.PRED, annote=[]}
  | translate (Zero pcf)          =
    Bool{e1=(translate pcf), e2=EPSILON,
         bop=PCF.ZERO,annote=[]}
  | translate (IfElse(p1,p2,p3)) =
    Select{guards=[(translate p1, translate p2,
                    translate p3)],
           annote=[]}
  | translate (Var s)             =
    Name{n=s, annote=[]}
  | translate (Lambda(s,t,p))     =
    Prim{e1=Name{n=s, annote=[]},
         e2=(translate p), pop=PCF.LAMBDA,
         annote=[tau2ann t]}
  | translate (Apply(p1,p2))      =
    Prim{e1=(translate p1), e2=(translate p2),
         pop=PCF.APPLY, annote=[]}
  | translate (Fix pcf)           =
    Prim{e1=(translate pcf), e2=EPSILON,
         pop=PCF.FIX, annote=[]}

(*
 * Interpret an ICODE_PCF expression.
 * val interp_icode_pcf :
 *      string * pcf.domain list -> icode.icode
 *                               -> pcf.domain
 *)
fun interp_icode_pcf rho (Val{v=value, annote=a}) =
    (case value of
   PCF.NOUGHT => Nat 0
      | PCF.TRUE   => Bool true
```

```
     | PCF.FALSE  => Bool false)
(* Succ or Pred. *)
| interp_icode_pcf rho (Arith{e1=M,
      e2=EPSILON,
      aop=oper,
      annote=an}) =
  (case oper of PCF.SUCC =>
     (case interp_icode_pcf rho M of
   Nat n => Nat (n + 1)
           | _ => BOTTOM)
   | PCF.PRED => (case interp_icode_pcf rho M of
       Nat n => Nat (n - 1)
                     | _ => BOTTOM))
(* Zero. *)
| interp_icode_pcf rho (Bool{e1=M,
           e2=EPSILON,
           bop=PCF.ZERO,
           annote=a}) =
  (case interp_icode_pcf rho M of
 Nat n => if n = 0 then Bool true else Bool false
     | _     => BOTTOM)
(* Variables. *)
| interp_icode_pcf rho (Name{n=x, annote=a}) =
  env rho x
(* Lambda, Application, Fix-point combinators. *)
| interp_icode_pcf rho (Prim{e1=M1, e2=M2, pop=oper, annote=a}) =
  (case oper of
 PCF.LAMBDA => (case M1 of
        Name{n=s, annote=a} =>
        Fun(fn v => interp_icode_pcf((s, v)::rho) M2)
      | _ => BOTTOM)
     | PCF.APPLY  => (case interp_icode_pcf rho M1 of
        Fun ff => ff (interp_icode_pcf rho M2)
      | _      => BOTTOM)
     | PCF.FIX    => (* Omitted. *))
  (* If-then-else. *)
  | interp_icode_pcf rho (Select{guards=[(M1, M2, M3)], annote=a}) =
    (case interp_icode_pcf rho M1 of
   Bool b => if b then interp_icode_pcf rho M2
       else interp_icode_pcf rho M3
       | _      => BOTTOM)
  (* Wild-card. *)
  | interp_icode_pcf rho _ = BOTTOM

end (* structure ICODE-PCF *)
```

APPENDIX B

FULL CODE LISTINGS FROM CHAPTER 5

```
structure IMP =
struct

(* Values in icode.Val expressions. *)
datatype value = TRUE | FALSE | Num of int

(* Operators in icode.Arith expressions. *)
datatype aop    = PLUS | SUB | MULT

(* Operators in icode.Bool expressions. *)
datatype bop    = EQ | LT | NOT | AND | OR

(* Operators in icode.Prim expressions. *)
datatype pop    = EMPTY_POP

(* Annotations - types. *)
datatype annote = EMPTY_ANNOTE

end (* structure IMP *)

structure ICODE-IMP =
struct

(* IMP commands. *)
datatype aexp = Num  of int
        | Var  of string
        | Plus of aexp * aexp
        | Sub  of aexp * aexp
        | Mult of aexp * aexp
and bexp = TRUE
         | FALSE
   | Eq  of aexp * aexp
   | Lt  of aexp * aexp
   | Not of bexp
   | And of bexp * bexp
   | Or  of bexp * bexp
and imp = SKIP
        | Assign of string * aexp
  | Sequ   of imp * imp
  | IfElse of bexp * imp * imp
  | While  of bexp * imp

(* The semantic domain of IMP as a smash sum of domains. *)
datatype domain = Int of int
    | Bool of bool
    | BOTTOM

(*
 * Lookup the value of a variable in a state.
 * val state : string * domain list -> string -> domain
 *)
fun state ((x, v)::rho) s = if x = s then v else state rho s
  | state []            s = BOTTOM


(*
 * Interpret an IMP command.
 * val interp_imp   : string * domain list -> imp
                     -> string * domain list
 * val interp_aexp : string * domain list -> aexp
                     -> domain
 * val interp_bexp : string * domain list -> bexp
                     -> domain
 *)
fun interp_imp rho (SKIP)                 = rho
  | interp_imp rho (Assign(s, a))         =
    (s, interp_imp_aexp rho a)::rho
  | interp_imp rho (Sequ(c1, c2))         =
    interp_imp (interp_imp rho c1) c2
  | interp_imp rho (IfElse(b, c1, c2)) =
    if (interp_imp_bexp rho b) = Bool true
    then interp_imp rho c1
    else interp_imp rho c2
  | interp_imp rho (While(b, c))          =
    if (interp_imp_bexp rho b) = Bool true
    then interp_imp (interp_imp rho c) (While(b, c))
    else rho
(* Interpret arithmetic expressions. *)
and interp_imp_aexp rho (Num i)           = Int i
  | interp_imp_aexp rho (Var s)           = state rho s
  | interp_imp_aexp rho (Plus(a1, a2)) =
    (case interp_imp_aexp rho a1 of
   Int i => (case interp_imp_aexp rho a2 of
           Int j => Int (i + j)))
  | interp_imp_aexp rho (Sub(a1, a2))   =
    (case interp_imp_aexp rho a1 of
   Int i => (case interp_imp_aexp rho a2 of
           Int j => Int (i - j)))
  | interp_imp_aexp rho (Mult(a1, a2)) =
    (case interp_imp_aexp rho a1 of
   Int i => (case interp_imp_aexp rho a2 of
           Int j => Int (i * j)))
(* Interpret boolean expressions. *)
and interp_imp_bexp rho (TRUE)          = Bool true
  | interp_imp_bexp rho (FALSE)         = Bool false
  | interp_imp_bexp rho (Eq(a1, a2))    =
    (case interp_imp_aexp rho a1 of
   Int i => (case interp_imp_aexp rho a2 of
            Int j => if i = j then Bool true else Bool false))
  | interp_imp_bexp rho (Lt(a1, a2))  =
    (case interp_imp_aexp rho a1 of
   Int i => (case interp_imp_aexp rho a2 of
            Int j => if i < j then Bool true else Bool false))
  | interp_imp_bexp rho (Not b)       =
    if (interp_imp_bexp rho b) = Bool false
    then Bool true
    else Bool false
  | interp_imp_bexp rho (And(b1, b2)) =
    if (interp_imp_bexp rho b1) = Bool true andalso
       (interp_imp_bexp rho b2) = Bool true
    then Bool true
    else Bool false
  | interp_imp_bexp rho (Or(b1, b2))  =
    if (interp_imp_bexp rho b1) = Bool true orelse
       (interp_imp_bexp rho b2) = Bool true
    then Bool true
    else Bool false


(*
 * Translate IMP expressions into ICODE.
 * val translate  : imp  -> icode
 * val trans_bexp : bexp -> icode
 * val trans_aexp : aexp -> icode
 *)
fun translate (SKIP)              = EPSILON
  | translate (Assign(s,a))       =
    Assign{lvalue=Name{n=s, annote=[]},
           rvalue=(trans_aexp a), annote=[]}
  | translate (Sequ(c1, c2))      =
    NameSpace{name="",
        space=((translate c1)::(translate c2)::[]),
        annote=[]}
  | translate (IfElse(b, c1, c2)) =
    Select{guards=[(trans_bexp b, translate c1,
                    translate c2)], annote=[]}
  | translate (While(b, c))        =
    Iterate{guards=[(trans_bexp b, translate c)], annote=[]}
and trans_bexp (TRUE)        = Val{v=IMP.TRUE,  annote=[]}
  | trans_bexp (FALSE)       = Val{v=IMP.FALSE, annote=[]}
  | trans_bexp (Eq(a1, a2))  = Bool{e1=(trans_aexp a1),
            e2=(trans_aexp a2),
            bop=IMP.EQ,
            annote=[]}
  | trans_bexp (Lt(a1, a2))  = Bool{e1=(trans_aexp a1),
            e2=(trans_aexp a2),
            bop=IMP.LT,
            annote=[]}
  | trans_bexp (Not b)       = Bool{e1=(trans_bexp b),
            e2=EPSILON,
            bop=IMP.NOT,
            annote=[]}
  | trans_bexp (And(b1, b2)) = Bool{e1=(trans_bexp b1),
            e2=(trans_bexp b2),
            bop=IMP.AND,
            annote=[]}
  | trans_bexp (Or(b1, b2))  = Bool{e1=(trans_bexp b1),
            e2=(trans_bexp b2),
            bop=IMP.OR,
            annote=[]}
and trans_aexp (Num i)        =
    Val{v=(IMP.Num i), annote=[]}
  | trans_aexp (Var s)        = Name{n=s, annote=[]}
  | trans_aexp (Plus(a1, a2)) = Arith{e1=(trans_aexp a1),
              e2=(trans_aexp a2),
              aop=IMP.PLUS,
              annote=[]}
  | trans_aexp (Sub(a1, a2))  = Arith{e1=(trans_aexp a1),
              e2=(trans_aexp a2),
              aop=IMP.SUB,
              annote=[]}
  | trans_aexp (Mult(a1, a2)) = Arith{e1=(trans_aexp a1),
              e2=(trans_aexp a2),
              aop=IMP.MULT,

(*
 * Interpret an ICODE_IMP expression.
 * val interp_icode_imp  : string * domain list -> icode
 *                         -> string * domain list
 * val interp_icode_aexp : string * domain list -> icode
 *                         -> domain
 * val interp_icode_bexp : string * domain list -> icode
 *                         -> domain
 *)
fun interp_icode_imp rho (EPSILON) = rho
  (* Assignment. *)
  | interp_icode_imp rho (Assign{lvalue=l,
         rvalue=r,
         annote=an}) =
    (case l of
       Name{n=s, annote=a} => (s, interp_icode_aexp rho r)::rho)
  (* If-then-else. *)
  | interp_icode_imp rho (Select{guards=((g1, g2, g3)::gs),
         annote=an}) =
    if (interp_icode_bexp rho g1) =
      Bool true then interp_icode_imp rho g2
    else interp_icode_imp rho g3
  (* While loops. *)
  | interp_icode_imp rho (Iterate{guards=gs,
          annote=an}) =
    (case gs of
    ((g1, g2)::ls) =>
    if (interp_icode_bexp rho g1) = Bool true
    then interp_icode_imp (interp_icode_imp rho g2)
              (Iterate{guards=gs, annote=an})
    else rho)
  (* Sequences. *)
  | interp_icode_imp rho (NameSpace{name=n,
          space=(i1::i2::[]),
          annote=an}) =
    interp_icode_imp (interp_icode_imp rho i1) i2
  (* Wild-card. *)
  | interp_icode_imp rho _ = rho
and interp_icode_aexp rho (Val{v=value, annote=an}) =
    (case value of
    IMP.TRUE  => BOTTOM
        | IMP.FALSE => BOTTOM
        | IMP.Num i => Int i)
  (* Variables. *)
  | interp_icode_aexp rho (Name{n=x, annote=a}) =
    state rho x
  (* Aexp expressions. *)
  | interp_icode_aexp rho (Arith{e1=a1,
        e2=a2,
        aop=oper,
        annote=an}) =
    (case oper of
    IMP.PLUS => (case interp_icode_aexp rho a1 of
         Int i => (case interp_icode_aexp rho a2 of
           Int j => Int (i + j)))
        | IMP.SUB  => (case interp_icode_aexp rho a1 of
         Int i => (case interp_icode_aexp rho a2 of
           Int j => Int (i - j)))
        | IMP.MULT => (case interp_icode_aexp rho a1 of
         Int i => (case interp_icode_aexp rho a2 of
           Int j => Int (i * j))))
  (* Wild-card. *)
  | interp_icode_aexp rho _ = BOTTOM
and interp_icode_bexp rho (Val{v=value, annote=an}) =
    (case value of
   IMP.TRUE  => Bool true
       | IMP.FALSE => Bool false
       | IMP.Num i => BOTTOM)
  (* Bexp expressions. *)
  | interp_icode_bexp rho (Bool{e1=b1,
        e2=b2,
        bop=oper,
        annote=an}) =
    (case oper of
   IMP.EQ  => (case interp_icode_bexp rho b1 of
       Int i => (case interp_icode_bexp rho b2 of
               Int j => if i = j then Bool true
            else Bool false))
       | IMP.LT  => (case interp_icode_bexp rho b1 of
       Int i => (case interp_icode_bexp rho b2 of
               Int j => if i < j then Bool true
            else Bool false))
       | IMP.NOT =>
         if (interp_icode_bexp rho b1) = Bool false
   then Bool true
   else Bool false
       | IMP.AND =>
         if (interp_icode_bexp rho b1) = Bool true andalso
       (interp_icode_bexp rho b1) = Bool true
   then Bool true
   else Bool false
       | IMP.OR  =>
         if (interp_icode_bexp rho b1) = Bool true orelse
       (interp_icode_bexp rho b1) = Bool true
   then Bool true
   else Bool false)
  (* Wild-card. *)
  | interp_icode_bexp rho _ = BOTTOM

end (* structure ICODE-IMP *)
```

APPENDIX C

ADDITIONAL EVIDENCE

Chapter 6 describes how static checkers can be built with Exstatic. The principle example in that Chapter is the application of Exstatic to Python programs which make use of a Communicating Sequential Processes [43] library, written by the author. This supports the thesis statement that Exstatic is “capable of checking project-specific violations in a variety of source languages” and “of benefit to software projects”.

However, other Exstatic analysers have been built and the results obtained from these systems have been published in [77, 76]. The systems described in these papers and the substantial description of them in the papers was provided by the author. The co-authors of the two papers were members of the supervisory team, or advisers.

[77] describes an Exstatic instance which examines Javadoc comments for violations of the Java Coding Conventions and [76] checks HTML markup for violations site-specific usability standards.

C.1 Bibliography

S.N.I. Mount, R.M. Newman, R.J. Low, and A. Mycroft. Exstatic: A generic static checker applied to documentation systems. In *In Proceedings of the 22nd Annual International Conference on Design of Communication*, pages 52–57. ACM Press, 2004

S.N.I. Mount, R.M. Newman, and R.J. Low. Checking marked-up documentation for site-specific standards. In *In Proceedings of the 23nd Annual International Conference on Design of Communication*, pages 76–79, Coventry, UK, 2005. ACM Press

Exstatic - A Generic Static Checker Applied To Documentation Systems

S.N.I. Mount, R.M. Newman, R.J. Low
Cogent Computing
Coventry University
Coventry, UK
{s.mount, r.m.newman, r.low}@coventry.ac.uk

A. Mycroft
Computer Laboratory
University of Cambridge
Cambridge, UK
Alan.Mycroft@cl.cam.ac.uk

ABSTRACT

Exstatic is a generic static checker developed by the author to address many of the practical problems in program development. Static checking provides a valuable means for automating time consuming checks not only concerned with program correctness (writing the right program), but also to do with style (writing the program right). Previous static checkers have been closely coupled with compilation systems, and therefore tend to be applicable to the code itself and not to all of the textual information (such as makefiles, comments, documentation sources) surrounding the code. The generic nature of Exstatic allows it to overcome these boundaries, and indeed it can be applied to any medium for which there is a formally definable syntax and (to an extent) semantics. Exstatic can therefore be used to increase the productivity and quality of documentation of programs, checking for such things as adherence to house style, consistency with the program being documented and self consistency. This paper describes the design and use of Exstatic, with particular reference to its use in documentation systems.

Categories and Subject Descriptors

D.2.4 [**Software**]: Software/Program Verification ; D.2.7 [**Software**]: Distribution, Maintenance, and Enhancement—*Documentation*

General Terms

Documentation, Standardization

Keywords

Exstatic, static checking, standards, Javadoc, docstrings

1. INTRODUCTION

In the early 1980s Knuth published his work on Literate Programming [11], which was an attempt to ease the human understating of computer programs. The literate programmer views the program as an extended essay, interspersed with code. Knuth's rationale is that explaining an algorithm to a human requires a clarity of thought that is often abandoned when "explaining" the same algorithm to a compiler (via its formal description in a programming language). By writing an essay, the literate programmer not only explains his or her thinking through the essay, but produces clearer and more maintainable source code as well. Knuth found (to his surprise) that this method often resulted in not just clearer programs, but better programs. Moreover, Knuth was keen to emphasise that in his experience literate programming is a more exhilarating and enjoyable process than, say, structured programming, which was fashionable at the time.

Knuth's method has not achieved the widespread adoption he hoped for. Perhaps this is due to a belief that large-scale developer documentation is prohibitively time-consuming. Lethbridge et al. [13] have carried out a study which seems to support this hypothesis:

> Our results indicate that software engineers create and use simple yet powerful documentation, and tend to ignore complex and time-consuming documentation.

However, Siy and Votta [16] have analysed code inspections and found that the majority (60%) of coding "errors" uncovered during inspections are errors in coding standards, style and readability, rather than errors in program behaviour:

> . . . [code inspections] improve the maintainability of the code by making the code conform to coding standards, minimising redundancies, improving language proficiency, improving safety and portability, and raising the quality of the documentation.

This result has several interesting implications:

- Producing readable, maintainable and well-documented code is clearly important enough to developers that it warrants a significant investment of time and effort;
- developers prefer documentation to be simple and concise, yet powerful;

SIGDOC'04, October 10–13, 2004, Memphis, Tennessee, USA.

- existing tools are not able to adequately detect errors in coding standards and improve the quality of documentation, etc. at compile-time. If such tools were adequate, we presume that less time would be spent on such issues during manual code inspections.

So, developers seem to consider complex documentation to be too time-consuming to produce *and* they consider documentation in general, to be sufficiently important to discuss during code inspections. It seems that whilst most modern programs are not literate in Knuth's sense, they aren't necessarily illiterate either. The need for high quality developer documentation is taken much more seriously than it has been in the past, as is evidenced by the plethora of tools that exist to enhance and prettify in-code documentation. Most modern programming languages have some sort of built-in tool to convert in-code documentation (usually called *docstrings*) to a human-readable format such as HTML. Perl has POD (Pretty Ordinary Documentation), Python has pydoc and Java has Javadoc.

Brooks [2] describes *self-documenting code*, which takes much of Knuth's argument on board, but is more concise and retains program code as the core output of the programming process:

> The solution, I think, is to merge the files, to incorporate the documentation in the source program. This is at once a powerful incentive toward proper maintenance, and an insurance that the documentation will always be handy to the program user. Such programs are called *self-documenting.*

1.1 This paper

This paper describes the use of an extensible static checker as tool support for self-documenting code. Many programming teams create style-guides or coding standards to ensure a certain amount of homogeneity is guaranteed between programmers. These style guides often include minimum standards for documentation. Many of these standards are trivial to check automatically (for example that certain documentation elements must occur in a particular order) and we present an example of a checker for Javadoc.

2. EXSTATIC: AN EXTENSIBLE, LANGUAGE-INDEPENDENT STATIC CHECKER

Exstatic is a language-independent, extensible static checker, developed by the first author. The distinguishing feature of Exstatic is that it is a whole-project checker. Almost all non-trivial pieces of software contain not only program code but documentation, data on the development of the program, data for internationalisation, scripts to compile, build, configure and install the software, and so on. Because Exstatic is independent of any particular language, we claim it can be used to check many different languages for adherence to standards, and can therefore be used to examine whole projects.

2.1 Motivation for language-independent checking

Colloquially, good programming practice is divided into two activities:

- *writing the right program*, and
- *writing the program right.*

The former refers to program specification and work in the problem domain. The latter refers to good practice in the task of programming itself.

Whilst much theory and many tools exist to enable programmers to write a coherent specification and verify that their program meets it very few tools exist to ensure that programs are written right.

Boehm [1] demonstrated that the earlier an error is found, the cheaper it is to fix . Maguire [14] differentiates between one-step and two-step tools and techniques for error detection. Two-step techniques (such as testing) detect errors in their first step, then require effort to locate the error in code. More convenient one-step techniques (such as static checking and manual inspection) locate errors in situ. To ensure that many errors are caught, one-step techniques should be adjunct to two-step methods.

We have already mentioned Siy and Votta's [16] work which found that a large proportion of time in code inspections is spent on adherence to coding standards. We suggest that this time could be better spent examining the correctness of programs (*writing the right program*) and that much of *writing the program right* could be automatically checked by a tool such as Exstatic.

2.2 Motivation for applying Exstatic to documentation systems

Writing a small system to check for standards adherence in one programming language's documentation system is a relatively straightforward task. Why shouldn't programmers write bespoke systems to do this? We argue that the strength of Exstatic's contribution to the documentation cycle is in it's ability to promote the reuse of checking routines.

Exstatic uses a novel intermediate language called *ICODE*, whose usual formal representation is in XML. ICODE is designed to be very high-level and may be used to represent information which would normally be thrown away during lexing or parsing by a compiler (such as in-code documentation).

When a new language is integrated into the Exstatic system, a lexer and parser must be written to translate code in the new language into ICODE. Once this is done, any appropriate Exstatic checking routines can be performed on the new language's ICODE representation.

For example, Javadoc has the following @tags which document certain features of the Java source code:

- `@author`,
- `@deprecated`,
- `{@docRoot}`,
- `@exception`,
- `{@inheritDoc`,
- `{@link}`,
- `{@linkplain}`,
- `@param`,
- `@return`,

- `@see`,
- `@serial`,
- `@serialData`,
- `@serialField`,
- `@since`,
- `@throws`,
- `{@value}`,
- `@version`.

Pydoc (for the Python programming language) doesn't have anything equivalent to @tags. Instead, Python source files may contain a number of special variables which document particular code attributes:

- `__author__`,
- `__version__`,
- `__date__` and
- `__credits__`.

Suppose you had already written some Exstatic modules to check for standards adherence in Javadoc comments (called, say, ex-javadoc), and wanted to augment these Python variables with analogs of Javadoc's @tags. You would have to modify the pydoc Python module to ensure that it's output would include your new variables (or tags, depending on how you choose to represent them). You may then wish to impose the same codings standards on your new version of pydoc as you do in Javadoc. In order to use Exstatic to do this you would have to rewrite the ex-javadoc lexer and parser to cope with Python and it's documentation system. This would be a straightforward task, as docstrings have a very simple syntax. Once this has been done, the checking routines applied to ICODE-Javadoc could be applied to ICODE-pydoc.

The alternative would be to find or write an entire new tool to check pydoc, or make changes to an existing tool for checking Python code (such as PyChecker).

Also, the Exstatic system does not restrict the programming language that must be used to write parsers (from any language to ICODE) or checking algorithms. ICODE nodes are stored in XML format and any language that can read, output and manipulate XML may be used. In fact, it isn't even necessary to write a parser for a language, X, in the same language that checking algorithms for X are written in!

3. CHECKING JAVADOC DOCSTRINGS WITH EXSTATIC

To substantiate our claims for Exstatic, we present an example of it's use with Javadoc docstrings. We describe the translation of Javadoc to ICODE and a subset of checks for standards adherence that have been implemented.

3.1 Translating Javadoc to ICODE

Javadoc comments begin with the token `/**` and end with the token `*/`. The comments themselves are written in HTML and may contain a number of tags. These are used to control the final layout of the document. For example `@see` tags are used to generate hyperlinks to another part of the program documentation. The following is an example of a Javadoc comment from a tautology checker written as a teaching aid for an undergraduate coursework. Notice that the first line of the comment contains a succinct summary of the program construction being commented (in this case a class). This is used by Javadoc in it's summary of program components.

```
/** Top level class of the tautology checker.
 *
 * Contains a command-line interface to the rest
 * of the checker and the tautology checking
 * algorithm.
 *
 * The {@link #main(String[]) main} method calls
 * {@link checker.parser.Parser#parse() parse} to
 * parse the contents of a file given on the
 * command line and {@link #check() check} to
 * tautology check the resulting abstract syntax
 * tree.<p>
 *
 * To run the tautology checker (on a Linux platform),
 * open an <code>xterm</code>, and type:<br>
 * <code>java checker.Checker $filename</code><br>
 * where <code>$filename</code> is a file containing
 * the proposition you want to check. Note that the
 * <code>checker/</code> directory must be in your
 * <code>CLASSPATH</code>.<p>
 *
 * The grammar of the input language is as follows:
 * <br>
 * Prop ::= TT<br>
 * &nbsp;&nbsp;| FF<br>
 * &nbsp;&nbsp;| &lt;alpha&gt;&lt;alphanum&gt;*<br>
 * &nbsp;&nbsp;| ( Prop )<br>
 * &nbsp;&nbsp;| NOT Prop<br>
 * &nbsp;&nbsp;| Prop AND Prop<br>
 * &nbsp;&nbsp;| Prop OR Prop<br>
 * &nbsp;&nbsp;| Prop =&gt; Prop<br>
 * &nbsp;&nbsp;| Prop &lt;=&gt; Prop<p>
 *
 * Where &lt;alpha&gt; is an alphabetic characher
 * and &lt;alphanum&gt; is an alpha-numeric character.
 *
 * @author  Sarah Mount
 * @version 1.0
 *
 * @see checker.parser.Parser
 * @see checker.ast.Formula
 */
```

ICODE is a very abstract intermediate representation which resembles abstract syntax trees. Because of its high level of abstraction, several source code structures in the same language will often map to a single ICODE node type.

In the case of Javadoc, we make use of only two ICODE node types: `<DOC>` elements (to hold docstrings) and `<VAL>` elements (to hold Javadoc and HTML tags). If we were checking program code for errors, `<VAL>` would be used to represent numerical and alphanumeric literals. This overloading keeps the ICODE DOM very simple. It is made possible by the `<ANNOTE>` tag, which can occur inside any other ICODE element and contain data specific to the domain language being checked.

Rather than present the algorithm for converting Javadoc to ICODE, we simply present (part of) the above comment in it's ICODE form. The translation of other tags and Javadoc elements is straightforward and can be inferred from this example. It is worth mentioning that our algorithm examines code around Javadoc tags and attempts to determine (among other things) the name and type of program component being commented (see the last `ANNOTE` tag in the example). For clarity, this example has been reformatted. In reality, all formatting, whitespace, asterisks, etc. remains intact.

```
<DOC>
Top level class of the tautology checker.

Contains a command-line interface to the rest
of the checker and the tautology checking
algorithm.

The
<VAL>
<V>#main(String[])</V>
<ANNOTE>tag=link text=main</ANNOTE>
</VAL>
method calls

<VAL v="checker.parser.Parser#parse()">
<ANNOTE>tag=link text= parse</ANNOTE>
</VAL>
to parse the contents of a file given on the
...

To run the tautology checker (on a Linux platform),
open an
<VAL>
<V></V>
<ANNOTE>HTML=code text=xterm</ANNOTE>
</VAL>
, and type:
<VAL>
<V></V>
<ANNOTE>HTML=br</ANNOTE>
</VAL>
<VAL>
<V></V>
<ANNOTE>HTML=code
text=java checker.Checker $filename</ANNOTE>
</VAL>
<VAL>
<V></V>
<ANNOTE>HTML=br</ANNOTE>
</VAL>
...
<ANNOTE>
CLASS=checker.Checker
</ANNOTE>
</DOC>
```

3.2 Checking Javadoc

It may be cynical to say it, but one significant check is simply for the presence of a sufficient number of Javadoc comments. Leslie [12] writes:

> A little browsing on the Internet quickly reveals that Javadoc is often run with source files that contain no or extremely limited explanatory comments. The output may be difficult to use, but it is arguably a lot better than nothing. A little trial and error (or perhaps a lot!) based on names, types, signatures, and inheritance trees, and the tenacious application developer can start to figure out what the documentation failed to include.

However, a number of more interesting checks have been implemented, based on coding standards from Sun Microsystems [10, 9]. These include:

- Missing and extraneous tags. For example, an `@author` tag may only be used before a class or interface definition. Every method with formal parameters must have a `@param` tag for each parameter.
- Tags used out of order. Tags should be used in the following order [9]:
    - `@author`
    - `@version`
    - `@param`
    - `@return`
    - `@exception` or `throws`
    - `@see`
    - `@since`
    - `@serial`
    - `@deprecated`
- The presence of illegal characters. These are usually “curly” quotes which are inserted by some IDEs and editors but cannot be read by others.
- Incorrect use of HTML tags. For example `<code><em>FooBar</code></em>`.
- The use of parentheses to mark out methods inside docstrings. For example: `The add() method....`
- High density of inline links inside comments. This does not include the use of the `@see` tag.
- Warnings about using the word “field”. This can be turned off with a command line switch. The word field is ambiguous in object-oriented programming and can refer to a class variable or a field in a GUI.

Results from the checking routines are collated into a database and displayed to the user in HTML. They are categorised for ease of navigation and given a level of severity.

3.3 Not all errors can be caught

Unfortunately, there are some errors and standards violations which are impossible to catch. The obvious example of this is errors in natural language. However, a more subtle example are docstrings which do not correctly match the code the describe. For example:

```
/** Reports on whether a flag is set.
 *
 * @return <code>true</code> if
 * <code>flag</code> is set and
 * <code>false</code> otherwise.
 */
boolean isFlagSet() {
    if(flag) return false;
    return true;
}
```

In this trivial example the error is obvious, although it may still be overlooked.[1] In more complicated code fragments, this sort of error can be particularly pernicious. One company we are aware of (which will remain nameless!) saw an developer overwrite an entire client database because of inaccurate documentation. An (in-house) database management tool required various command-line flags. Unfortunately another developer has exchanged the documentation for the "insert" and "overwrite" flags, causing disaster. Happily, backups were available.

Clearly, a checker which could heuristically evaluate the likelihood of code mismatching its documentation would be extremely useful!

4. RELATED WORK

There are a reasonably large number of mature static checking systems currently available, including ESC (for Java) [5], Splint (for C code) [6, 7] and Meta-level compilation (also for C code) [8].

The obvious difference between Exstatic and these systems is that Exstatic is able to check code in more than one language. Equally as important, most static checkers are limited in their ability to *add* new checking routines. Splint has facilities for this and Meta-level compilation has a domain specific language called *metal* in which users can write new checks. However, this is still rare, as most static checkers are modelled after compilers and hide their internals.

Also, these static checkers are primarily designed to find coding errors which lead to incorrect program behaviour, inefficiency, portability problems and so on. Exstatic, on the other hand, is targeted directly at ensuring that coding standards are met.

Despite this, the need to check documentation is becoming apparent, and two projects in particular are able to check Javadoc docstrings for errors and standards violations.

Sun Microsystems has brought out it's Doc Check Doclet [?], which works in conjunction with the Javadoc system. Doc Check checks for various errors, including many implemented in this work, and gives template suggestions for bug fixes, which can be cut and pasted into code.

Checkstyle [3] is a checker for the Java programming language. Users are provided with an interface to an abstract syntax tree representation of their code (and comments). They can then write their own checkers (in Java) to inspect and report on their code.

5. CONCLUSIONS AND FUTURE DIRECTIONS

We have argued that checking for standards violations in docstrings could save time during later stages of the software engineering cycle. We have presented Exstatic, which is a language-independent, extensible static checking system and described its application to Javadoc comments.

Documentation is a rich area of problems for static checking to solve. One area of interest is project-specific standards for marked-up documents. For example, rules such as "breadcrumbs must appear at the top of each page" or "every `img` tag must have an `alt` attribute". Some of these can already be checked by language-specific checkers, such as the W3C HTML [4]. Exstatic could be useful here, firstly for implementing project-specific checks and secondly for migration issues. If, for example, a web developer wanted to move an HTML website over to XHTML, the same checking routines could be applied to the new website (although a parser would have to be written to translator XHTML to ICODE).

Documentation systems are beginning to incorporate increasingly sophisticated information which can be useful in static checking. Splint [6, 7] allows users to embed directions in code. Several languages allow elements of Meyers' design by contract [15] to be placed in docstrings. Python has a module called doctest, which allows users to cut and paste interpreter sessions into docstrings. Interpreter directives are run by the doctest module and if results differ from those in the comment this is reported to the user. Although this last example isn't *static* checking, it seems that in-code comments have a lot more to offer than has traditionally been exploited. The advantage of Exstatic is that it can be used to apply checks designed for one language to another, as in the example of incorporating Javadoc-style tags into Python in Section 2.2.

6. REFERENCES

[1] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[2] F. Brooks. *The Mythical Man-Month*. Addison-Wesley, 1995.

[3] O. Burn. The Checkstyle project. http://checkstyle.sourceforge.net/.

[4] W.W.W. Consortium. http://validator.w3.org/.

[5] D. Detlefs, K. Rustan, M. Leino, G. Nelson, and J. B. Saxe. Extended Static Checking. Technical Report 159, Compaq Systems Research Centre, Palo Alto, CA, Dec. 1998.

[6] D. Evans. Annotation-assisted lightweight static checking. In *The First International Workshop on Automated Program Analysis, Testing and Verification*, Feb. 2000.

[7] D. Evans and D. Larochelle. *Splint User's Manual Version 3.0.1.* University of Virginia, Department of Computer Science, Jan. 2002. http://www.splint.org/manual/manual.html.

[8] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static

---

[1] Confusion could be avoided if the code was replaced with the single line `return flag;`.

analyses. In *Programming Language Design and Implementation (PLDI)*. ACM SIGPLAN, 2002.

[9] S. M. Inc. How to write doc comments for the javadoc tool. http://java.sun.com/j2se/javadoc/writingdoccomments/

[10] S. M. Inc. Requirements for writing Java API specifications. http://java.sun.com/j2se/javadoc/writingapispecs/

[11] D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984.

[12] D. M. Leslie. Using javadoc and XML to produce API reference documentation. In *Proceedings of the 20th annual international conference on Computer documentation*, pages 104–109. ACM Press, 2002.

[13] T. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: The state of the practice. *IEEE Software*, 2003.

[14] S. Maguire. *Writing Solid Code*. Microsoft Press, 1993.

[15] D. Mandrioli and B. Meyer, editors. *Advances in Object-Oriented Software Engineering*, chapter Design by Contract, pages 1–50. Prentice Hall, 1991.

[16] H. Siy and L. Votta. Does the modern code inspection have value? In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, 2001.

Checking Marked-Up Documentation for Adherence to Site-Specific Standards

S.N.I. Mount, R.M. Newman, R.J. Low
Cogent Computing
Coventry University
Coventry, UK
{s.mount, r.m.newman, r.low}@coventry.ac.uk

ABSTRACT

Marked-up text (e.g. HTML and XML) is the format of choice for the delivery of end-user information in pervasive environments. Consistent style and structure of a set of pages can greatly aid their usability and this paper presents a tool called *Exstatic* to automatically check that a hyperdocument follows site-specific conventions. We describe an example of such an analysis for a live document written in HTML 4.01.

Categories and Subject Descriptors

D.2.4 [**Software**]: Software/Program Verification ; D.2.7 [**Software**]: Distribution, Maintenance, and Enhancement—*Documentation*

General Terms

Documentation, Standardization

Keywords

Exstatic, static checking, standards, HTML

1. INTRODUCTION

Marked-up documentation is the format of choice for the delivery of end-user information in pervasive environments. Mobile phones and PDAs use WAP (Wireless Access Protocol) or wireless Internet protocols to gather marked-up web pages. Even wireless sensor nets can exchange data in XML [2]. As we move towards a world where pervasive computing devices are available to the consumer, small devices proliferate, which need to exchange data. As simplistic example, it is currently possible to transfer mobile phone contacts from a phone to a PDA. Mark-up languages (and XML in particular) are often proposed as a useful way of making data available to users or to applications running on heterogeneous platforms.

It is generally accepted that the usability of a hypermedia document is greatly increased by the *consistency* of its style and presentation. This means that the aesthetic of a document should be consistent and that its structure and navigation options should be consistent. Although some tools exist to check documents for adherence to industry standards (see Section 1.1), document designers do not usually have access to an automated way of checking adherence to house-styles.

Exstatic is an extensible, language-independent static checker, designed primarily for use with programming languages and systems. Here, we show that Exstatic can be equally well applied to marked-up text, which might be the user manual of a PC application, but could equally be online content gathered from a pervasive environment. Since Exstatic is extensible and can be used to check code in many different languages, we note that one of the advantages of our work is that the same checking routines can be applied to documents written in different mark-up languages.

1.1 Background and related work

Much work has been done on ensuring that websites meet W3C standards for HTML, and checkers such as W3CValidator [3] report on standards compliance. Other checkers such as WebXACT [4] check for compliance against a set of extra constraints, which ensure that sites meet accessibility guidelines and are legible to text-only browsers and speech readers. This includes such standards as:

- Frames should have a `NOFRAMES` section;
- all elements should be operable without a mouse;
- grouped links should have a link at the start, allowing the user to bypass the group; and
- alternative text should be provided for all image map hot-spots.

However, these checkers do not check for site-specific standards such as "breadcrumbs must come before an `H1` tag", which are the focus of this paper.

Also, work has been done on ensuring the project-specific correctness of hypermedia in the design cycle, including German's HadeZ [5] and Newman's work on high reliability documentation systems [8, 9]. German's work, in particular, makes use of a very sophisticated model of the various levels of abstraction which must be specified by the designer. For example, linking between pages is treated entirely independently of content and style. Unfortunately, German only

---

SIGDOC'05, September 21–23, 2005, Coventry, United Kingdom.

describes methods for developing a formal specification for hypermedia, with no way of generating conforming marked-up text or even verifying that a real document meets it's specification.

This work is related to ours, but addresses a different stage in the design-implementation cycle. Researchers such as German and Newman are interested in ways of formally specifying documentation, *before* it is written, in the hope of eradicating certain sorts of errors, such as allowing the reader to access information out of order. In our work, we hope to address some of the same document-specific concerns, in a manner which complements work done on the specification and design cycle of document preparation. Rather than write a formal specification and derive a document from it (which, we believe, is rarely done in practice), we write a set of checking routines to detect the presence of certain errors, then run this custom static checker (or set of static checkers) over the document, during or after its implementation.

1.2 This paper

This paper is structured as follows: Section 2 describes the Exstatic system in more detail. Section 3 describes how HTML 4.01 can be checked with the Exstatic checking system. Section 4 introduces an example document structure for a document which is publicly available over the World Wide Web. A site-specific set of standards for that site are given. Section 5 concludes.

2. EXSTATIC: AN EXTENSIBLE, LANGUAGE-INDEPENDENT STATIC CHECKER

Exstatic is a language-independent, extensible static checker, developed by the first author. The distinguishing feature of Exstatic is that it is a whole-project checker. Almost all non-trivial pieces of software contain not only program code but documentation, data on the development of the program, data for internationalisation, scripts to compile, build, configure and install the software, and so on. Because Exstatic is independent of any particular language, we claim it can be used to check many different languages for adherence to standards, and can therefore be used to examine whole projects.

Colloquially, good programming practice is divided into two activities:

- *writing the right program*, and
- *writing the program right*.

The former refers to program specification and work in the problem domain. The latter refers to good practice in the task of programming itself.

Whilst much theory and many tools exist to enable programmers to write a coherent specification and verify that their program meets it very few tools exist to ensure that programs are written right.

Boehm [1] demonstrated that the earlier an error is found, the cheaper it is to fix . Maguire [6] differentiates between one-step and two-step tools and techniques for error detection. Two-step techniques (such as testing) detect errors in their first step, then require effort to locate the error in code. More convenient one-step techniques (such as static checking and manual inspection) locate errors in situ. To ensure that many errors are caught, one-step techniques should be adjunct to two-step methods.

Siy and Votta's [10] work found that a large proportion of time in code inspections is spent on adherence to coding standards. We suggest that this time could be better spent examining the correctness of programs (*writing the right program*) and that much of *writing the program right* could be automatically checked by a tool such as Exstatic.

3. STATIC ANALYSIS OF HTML 4.01

There are a plethora of languages designed to format and style text, including HTML, TeX, LaTeX, SGML, troff and DocBook. Many of these are used in large project and are hence subject to project-wide coding standards. For example, troff is used to generate *NIX man pages, DocBook is used to generate manuals, tutorials and other user documentation and HTML is used in millions of multi-page websites all over the world.

The advantage here of using Exstatic, is that markup-languages tend to be very similar, so the same structures in ICODE can be used to represent similar features of different languages. For example, there are many variants of HTML. If, say we create an Exstatic checker for a particular website (as we describe below), then update that website from, say, HTML 4.01 to XHTML, we could continue to use our Exstatic checker without having to re-write any of our checking routines. Of course, a new parser for XHTML would need to be re-written, then this could be used for any XHTML document.

In this section, we consider HTML 4.01 as an example of a markup-language, subject to a set of custom coding standards.

3.1 Translating HTML 4.01 to ICODE

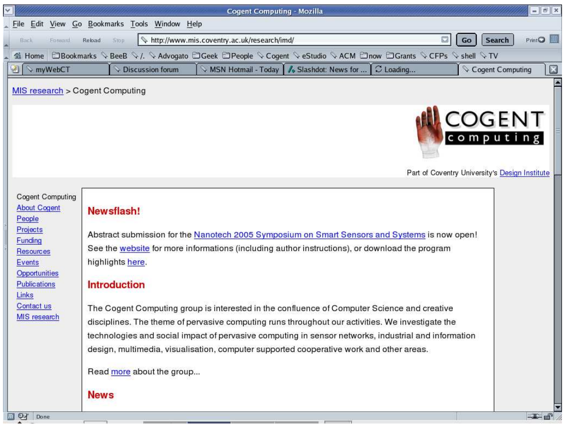
In order to check HTML documents for standards violations, we first need to write a parser which converts HTML to ICODE-XML, the intermediate language of Exstatic. This is very similar to the Javadoc example, given in [7].

We use the <`DOC`> tag do denote the beginning of a document and a <`VAL`> tag for each HTML tag. The <`ANNOTE`> tags to encode tag types and parameters. We have separate translations for start and end tags. For example, <`strong`> and <`/strong`> translate to
<`VAL`><`ANNOTE`>`START-STRONG`<`/ANNOTE`><`/VAL`> and
<`VAL`><`ANNOTE`>`END_STRONG`<`/ANNOTE`><`/VAL`>.

There is one problem with parsing HTML that is beyond the scope of this paper, which is that much HTML is written badly and probably doesn't sensibly parse. Many HTML-generators produce start-tags without matching end-tags, allow browser-specific extensions, etc. The website we're checking has been written by-hand and is standards-compliant (which we're consequently not checking for). Although it is useful to check for parsing-errors, we believe that site-specific standards are where Exstatic can make a novel contribution to document checking, and concentrate on that aspect of the work.

4. EXAMPLE DOCUMENT STRUCTURE: THE COGENT COMPUTING WEBSITE

As an example, we use the live website of the research


Cogent Computing - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://www.mis.coventry.ac.uk/research/imd/
Go
Search
Print
Home Bookmarks BeeB /. Advogato Geek People Cogent eStudio ACM now Grants CFPs shell TV
myWebCT
Discussion forum
MSN Hotmail - Today
Slashdot: News for ...
Loading...
Cogent Computing
COGENT computing
Done

MIS research > Cogent Computing

Part of Coventry University's Design Institute

Cogent Computing
About Cogent
People
Projects
Funding
Resources
Events
Opportunities
Publications
Links
Contact us
MIS research

Newsflash!

Abstract submission for the Nanotech 2005 Symposium on Smart Sensors and Systems is now open! See the website for more informations (including author instructions), or download the program highlights here.

Introduction

The Cogent Computing group is interested in the confluence of Computer Science and creative disciplines. The theme of pervasive computing runs throughout our activities. We investigate the technologies and social impact of pervasive computing in sensor networks, industrial and information design, multimedia, visualisation, computer supported cooperative work and other areas.

Read more about the group...

News

Figure 1: Front page of the Cogent Computing website. The breadcrumbs at the top of the page reflect the directory structure of the site. The banner should be identical on every page, as should the layout. The navigation bar on the left hand side represents the directory structure and page titles of the site.

group to which the authors belong. The site has the following directory structure, which is reflected in the navigation bar and breadcrumbs (see Figure 1):

- index.html
- about.html
- . . .
- images/
  - cogent.png
  - group-photo.png
  - . . .
- members/
  - bob/
    - bob.html
    - publications.html
    - . . .
  - elena/
    - elena.html
    - publications.html
    - . . .
  - . . .
- . . .

4.1 Coding standards for the Cogent Computing website

Each page in the website implements the template in Figure 2. From the template, we can generate a list of coding conventions that must be adhered to. The following are a representative sample of these:

- Each page should contain the meta-data listed in the template.
- Each page should have an identical banner.
- Each page should have an identical footer.
- Each page should begin with breadcrumbs representing the position of each file in the directory structure.
- Each page should have a navigation bar on the left of the main text which . . .
- No `<h1>` tags should be used.
- ...

In practice, it is surprising how many such errors can be found, even when using a simple template for a site such as this. To date, the most common errors we have found have been:

- Links to pages which have been removed;
- incomplete or erroneous breadcrumbs;
- erroneous links in the navigation bar.

5. CONCLUSIONS

We have presented *Exstatic*, an extensible, language-independent static checker. We have applied Exstatic to a set of live web pages and determined their compliance to site-specific design guidelines. Since Exstatic is language-independent, we can apply the checks we have written to documents defined in other mark-up languages.

6. REFERENCES

[1] B. W. Boehm. *Software Engineering Economics.* Prentice-Hall, 1981.

[2] M. Botts. Sensor model language (SensorML) fo in-situ and remote sensors. Technical report, Open Geospatial Consortium Inc., Nov. 2004. Version 1.0Beta.

[3] W. W. W. Consortium. http://validator.w3.org/.

[4] W. Corporation. http://webxact.watchfire.com/.

[5] D. M. German. *HadeZ: a Framework for the Specification and Verification of Hypermedia Applications.* PhD thesis, University of Victoria, 2000.

[6] S. Maguire. *Writing Solid Code.* Microsoft Press, 1993.

[7] S. Mount, R. Newman, R. Low, and A. Mycroft. Exstatic: a generic static checker applied to documentation systems. In *Proceedings of the 22nd annual international conference on Design of communication*, pages 52–57. ACM Press, 2004.

[8] R. Newman. *A Visual Design Method and Its Application to High Reliability Hypermedia Systems.* PhD thesis, Coventry University, 1998.

[9] R. Newman. Designing hypermedia documents for safety critical applications. In *Proceedings of the First IEEE Conference on Informations Technology Codes and Computing (ITCC 2000)*, Las Vegas, Nevada, USA, Mar. 2000. IEEE Computer Press.

[10] H. Siy and L. Votta. Does the modern code inspection have value? In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, 2001.

Figure 2: A template for web pages in the Cogent Computing website.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3c.org/TR/1999/REC-html401-19991224/loose.dtd">
<HTML lang=en><HEAD><TITLE>Cogent Computing: THIS PAGE</TITLE>
<META http-equiv=Content-Type content="text/html; charset=utf-8">
<META content="Coventry University UK: Cogent Computing Research Group"
  name=description>
<META content=University,UK,Informatics,research,media,design,computing
  name=keywords>
<LINK title=style href="cogentstyle.css" type=text/css rel=stylesheet>
<BODY><A href="http://www.mis.coventry.ac.uk/research/research.html">MIS
  research</A> &gt; Cogent Computing %gt; THIS PAGE
<P>
<DIV class=header style="TEXT-ALIGN: right">
<IMG height=100  alt="Cogent Computing Logo" src="images/cogent.png" width=247>
<P style="FONT-SIZE: small; TEXT-ALIGN: right">Part of Coventry University's
  <A href="http://www.coventry.ac.uk/cms/jsp/polopoly.jsp?d=844">Design
  Institute</A>
</DIV>
<P>
<TABLE width="90%">
  <TBODY>
  <TR><!-- Navigation bar -->
      <TD vAlign=top width=120>
      <DIV class=navleft>Cogent Computing
      <BR><A href="http://www..../research/imd/about.html">About Cogent</A>
      <BR><A href="http://www..../research/imd/people.html">People</A>
      <BR><A href="http://www..../research/imd/projects.html">Projects</A>
      ...
      <BR><A href="http://www..../research/research.html">MIS research</A>
      </DIV>
      </TD>
      <TD vAlign=top><!-- Page Content -->
      <DIV class=content>
      <H2>THIS PAGE</H2>
      <!-- Page contents placed here... -->
      </DIV></TD>
  </TR>
  </TBODY>
</TABLE>
<!-- Validation -->
<P>
<HR>
<A href="http://jigsaw.w3.org/css-validator/">
  <IMG style="BORDER-TOP-WIDTH: 0px; BORDER-LEFT-WIDTH: 0px;
    BORDER-BOTTOM-WIDTH: 0px; WIDTH: 88px; HEIGHT: 31px;
    BORDER-RIGHT-WIDTH: 0px"
    alt="Valid CSS!" src="Cogent Computing_files/vcss.gif">
</A>
<A href="http://validator.w3.org/check/referer">
  <IMG height=31 alt="Valid HTML 4.01!"
    src="Cogent Computing_files/valid-html401" width=88 border=0>
</A>
</BODY>
</HTML>
```

Appendix D

Full code listings from Chapter 4

```
structure PCF =
struct

(* Values in icode.Val expressions. *)
datatype value = NOUGHT | TRUE | FALSE

(* Operators in icode.Arith expressions. *)
datatype aop    = SUCC | PRED

(* Operators in icode.Bool expressions. *)
datatype bop    = ZERO

(* Operators in icode.Prim expressions. *)
datatype eop    = LAMBDA | APPLY | FIX

(* Annotations - types. *)
datatype annote = NAT | BOOL | Fn of annote * annote

end (* structure PCF *)

structure ICODE-PCF =
struct

(* PCF types and expressions. *)
datatype tau = NAT | BOOL | Fn of tau * tau
datatype pcf = NOUGHT | TRUE | FALSE
```

```
       | Succ of pcf | Pred of pcf
       | Zero of pcf
       | IfElse of (pcf * pcf * pcf)
       | Var of string
       | Lambda of (string * tau * pcf)
       | Apply of (pcf * pcf)
       | Fix of pcf

(* The semantic domain of PCF (as a smash sum of domains). *)
datatype domain = Nat of int
    | Bool of bool
    | Fun of domain -> domain
    | BOTTOM

(*
 * Lookup the value bound to a variable in an environment.
 * val env : string * domain list -> string -> domain
 *)
fun env ((x, v)::rho) s = if x = s then v else env rho s
  | env []            s = BOTTOM

(*
 * Interpret a PCF expression.
 * val interp_pcf : string * domain list -> pcf -> domain
 *)
fun interp_pcf rho (NOUGHT)             = Nat 0
  | interp_pcf rho (TRUE)               = Bool true
  | interp_pcf rho (FALSE)              = Bool false
  | interp_pcf rho (Succ e)             =
    (case interp_pcf rho e of Nat n => Nat (n + 1)
              | _ => BOTTOM)
  | interp_pcf rho (Pred e)             =
    (case interp_pcf rho e of Nat n => Nat (n - 1)
                                | _ => BOTTOM)
  | interp_pcf rho (Zero e)             =
     (case interp_pcf rho e of Nat n =>
        if n = 0 then Bool true else Bool false
         | _ => BOTTOM)
  | interp_pcf rho (IfElse(M1, M2, M3)) =
    (case interp_pcf rho M1 of Bool b =>
       if b then interp_pcf rho M2
       else interp_pcf rho M3
        | _ => BOTTOM)
  | interp_pcf rho (Var x)              = env rho x
  | interp_pcf rho (Lambda(s, t, M))    =
    Fun (fn v => interp_pcf ((s,v)::rho) M)
  | interp_pcf rho (Apply(M1, M2))      =
    (case interp_pcf rho M1 of Fun ff => ff (interp_pcf rho M2)
                                  | _ => BOTTOM)
  | interp_pcf rho (Fix M)              = (* Omitted. *)

(*
 * Convert PCF types into ICODE annotations.
 * val tau2ann : tau -> PCF.annote
 *)
fun tau2ann NAT          = PCF.NAT
  | tau2ann BOOL         = PCF.BOOL
  | tau2ann (Fn(t1, t2)) = PCF.Fn(tau2ann t1, tau2ann t2)

(*
 * Translate PCF expressions into ICODE.
 * val translate : pcf -> icode.icode
 *)
fun translate (NOUGHT)           =
    Val{v=PCF.NOUGHT, annote=[]}
  | translate (TRUE)             =
    Val{v=PCF.TRUE,   annote=[]}
  | translate (FALSE)            =
    Val{v=PCF.FALSE,  annote=[]}
  | translate (Succ pcf)         =
    Arith{e1=(translate pcf), e2=EPSILON,
          aop=PCF.SUCC, annote=[]}
  | translate (Pred pcf)          =
    Arith{e1=(translate pcf), e2=EPSILON,
          aop=PCF.PRED, annote=[]}
  | translate (Zero pcf)          =
    Bool{e1=(translate pcf), e2=EPSILON,
         bop=PCF.ZERO,annote=[]}
  | translate (IfElse(p1,p2,p3)) =
    Select{guards=[(translate p1, translate p2,
                    translate p3)],
           annote=[]}
  | translate (Var s)             =
    Name{n=s, annote=[]}
  | translate (Lambda(s,t,p))     =
    Prim{e1=Name{n=s, annote=[]},
         e2=(translate p), pop=PCF.LAMBDA,
         annote=[tau2ann t]}
  | translate (Apply(p1,p2))      =
    Prim{e1=(translate p1), e2=(translate p2),
         pop=PCF.APPLY, annote=[]}
  | translate (Fix pcf)           =
    Prim{e1=(translate pcf), e2=EPSILON,
         pop=PCF.FIX, annote=[]}

(*
 * Interpret an ICODE_PCF expression.
 * val interp_icode_pcf :
 *      string * pcf.domain list -> icode.icode
 *                               -> pcf.domain
 *)
fun interp_icode_pcf rho (Val{v=value, annote=a}) =
    (case value of
   PCF.NOUGHT => Nat 0
       | PCF.TRUE   => Bool true
```

```
     | PCF.FALSE  => Bool false)
(* Succ or Pred. *)
| interp_icode_pcf rho (Arith{e1=M,
      e2=EPSILON,
      aop=oper,
      annote=an}) =
  (case oper of PCF.SUCC =>
     (case interp_icode_pcf rho M of
   Nat n => Nat (n + 1)
           | _ => BOTTOM)
   | PCF.PRED => (case interp_icode_pcf rho M of
       Nat n => Nat (n - 1)
                     | _ => BOTTOM))
(* Zero. *)
| interp_icode_pcf rho (Bool{e1=M,
           e2=EPSILON,
           bop=PCF.ZERO,
           annote=a}) =
  (case interp_icode_pcf rho M of
 Nat n => if n = 0 then Bool true else Bool false
     | _     => BOTTOM)
(* Variables. *)
| interp_icode_pcf rho (Name{n=x, annote=a}) =
  env rho x
(* Lambda, Application, Fix-point combinators. *)
| interp_icode_pcf rho (Prim{e1=M1, e2=M2, pop=oper, annote=a}) =
  (case oper of
 PCF.LAMBDA => (case M1 of
        Name{n=s, annote=a} =>
        Fun(fn v => interp_icode_pcf((s, v)::rho) M2)
      | _ => BOTTOM)
     | PCF.APPLY  => (case interp_icode_pcf rho M1 of
        Fun ff => ff (interp_icode_pcf rho M2)
      | _      => BOTTOM)
     | PCF.FIX    => (* Omitted. *))
  (* If-then-else. *)
  | interp_icode_pcf rho (Select{guards=[(M1, M2, M3)], annote=a}) =
    (case interp_icode_pcf rho M1 of
   Bool b => if b then interp_icode_pcf rho M2
       else interp_icode_pcf rho M3
       | _      => BOTTOM)
  (* Wild-card. *)
  | interp_icode_pcf rho _ = BOTTOM

end (* structure ICODE-PCF *)
```

APPENDIX E

FULL CODE LISTINGS FROM CHAPTER 5

```
structure IMP =
struct

(* Values in icode.Val expressions. *)
datatype value = TRUE | FALSE | Num of int

(* Operators in icode.Arith expressions. *)
datatype aop    = PLUS | SUB | MULT

(* Operators in icode.Bool expressions. *)
datatype bop    = EQ | LT | NOT | AND | OR

(* Operators in icode.Prim expressions. *)
datatype pop    = EMPTY_POP

(* Annotations - types. *)
datatype annote = EMPTY_ANNOTE

end (* structure IMP *)

structure ICODE-IMP =
struct

(* IMP commands. *)
datatype aexp = Num  of int
        | Var  of string
        | Plus of aexp * aexp
        | Sub  of aexp * aexp
        | Mult of aexp * aexp
and bexp = TRUE
         | FALSE
   | Eq  of aexp * aexp
   | Lt  of aexp * aexp
   | Not of bexp
   | And of bexp * bexp
   | Or  of bexp * bexp
and imp = SKIP
        | Assign of string * aexp
  | Sequ   of imp * imp
  | IfElse of bexp * imp * imp
  | While  of bexp * imp

(* The semantic domain of IMP as a smash sum of domains. *)
datatype domain = Int of int
    | Bool of bool
    | BOTTOM

(*
 * Lookup the value of a variable in a state.
 * val state : string * domain list -> string -> domain
 *)
fun state ((x, v)::rho) s = if x = s then v else state rho s
  | state []            s = BOTTOM


(*
 * Interpret an IMP command.
 * val interp_imp  : string * domain list -> imp
                     -> string * domain list
 * val interp_aexp : string * domain list -> aexp
                     -> domain
 * val interp_bexp : string * domain list -> bexp
                     -> domain
 *)
fun interp_imp rho (SKIP)                    = rho
  | interp_imp rho (Assign(s, a))            =
    (s, interp_imp_aexp rho a)::rho
  | interp_imp rho (Sequ(c1, c2))            =
    interp_imp (interp_imp rho c1) c2
  | interp_imp rho (IfElse(b, c1, c2)) =
    if (interp_imp_bexp rho b) = Bool true
    then interp_imp rho c1
    else interp_imp rho c2
  | interp_imp rho (While(b, c))             =
    if (interp_imp_bexp rho b) = Bool true
    then interp_imp (interp_imp rho c) (While(b, c))
    else rho
(* Interpret arithmetic expressions. *)
and interp_imp_aexp rho (Num i)          = Int i
  | interp_imp_aexp rho (Var s)          = state rho s
  | interp_imp_aexp rho (Plus(a1, a2)) =
    (case interp_imp_aexp rho a1 of
   Int i => (case interp_imp_aexp rho a2 of
           Int j => Int (i + j)))
  | interp_imp_aexp rho (Sub(a1, a2))   =
    (case interp_imp_aexp rho a1 of
   Int i => (case interp_imp_aexp rho a2 of
           Int j => Int (i - j)))
  | interp_imp_aexp rho (Mult(a1, a2)) =
    (case interp_imp_aexp rho a1 of
   Int i => (case interp_imp_aexp rho a2 of
           Int j => Int (i * j)))
(* Interpret boolean expressions. *)
and interp_imp_bexp rho (TRUE)          = Bool true
  | interp_imp_bexp rho (FALSE)         = Bool false
  | interp_imp_bexp rho (Eq(a1, a2))    =
    (case interp_imp_aexp rho a1 of
   Int i => (case interp_imp_aexp rho a2 of
            Int j => if i = j then Bool true else Bool false))
  | interp_imp_bexp rho (Lt(a1, a2))  =
    (case interp_imp_aexp rho a1 of
   Int i => (case interp_imp_aexp rho a2 of
            Int j => if i < j then Bool true else Bool false))
  | interp_imp_bexp rho (Not b)       =
    if (interp_imp_bexp rho b) = Bool false
    then Bool true
    else Bool false
  | interp_imp_bexp rho (And(b1, b2)) =
    if (interp_imp_bexp rho b1) = Bool true andalso
       (interp_imp_bexp rho b2) = Bool true
    then Bool true
    else Bool false
  | interp_imp_bexp rho (Or(b1, b2))  =
    if (interp_imp_bexp rho b1) = Bool true orelse
       (interp_imp_bexp rho b2) = Bool true
    then Bool true
    else Bool false


(*
 * Translate IMP expressions into ICODE.
 * val translate  : imp  -> icode
 * val trans_bexp : bexp -> icode
 * val trans_aexp : aexp -> icode
 *)
fun translate (SKIP)              = EPSILON
  | translate (Assign(s,a))       =
    Assign{lvalue=Name{n=s, annote=[]},
           rvalue=(trans_aexp a), annote=[]}
  | translate (Sequ(c1, c2))      =
    NameSpace{name="",
        space=((translate c1)::(translate c2)::[]),
        annote=[]}
  | translate (IfElse(b, c1, c2)) =
    Select{guards=[(trans_bexp b, translate c1,
                    translate c2)], annote=[]}
  | translate (While(b, c))       =
    Iterate{guards=[(trans_bexp b, translate c)], annote=[]}
and trans_bexp (TRUE)        = Val{v=IMP.TRUE,  annote=[]}
  | trans_bexp (FALSE)       = Val{v=IMP.FALSE, annote=[]}
  | trans_bexp (Eq(a1, a2))  = Bool{e1=(trans_aexp a1),
            e2=(trans_aexp a2),
            bop=IMP.EQ,
            annote=[]}
  | trans_bexp (Lt(a1, a2))  = Bool{e1=(trans_aexp a1),
            e2=(trans_aexp a2),
            bop=IMP.LT,
            annote=[]}
  | trans_bexp (Not b)       = Bool{e1=(trans_bexp b),
            e2=EPSILON,
            bop=IMP.NOT,
            annote=[]}
  | trans_bexp (And(b1, b2)) = Bool{e1=(trans_bexp b1),
            e2=(trans_bexp b2),
            bop=IMP.AND,
            annote=[]}
  | trans_bexp (Or(b1, b2))  = Bool{e1=(trans_bexp b1),
            e2=(trans_bexp b2),
            bop=IMP.OR,
            annote=[]}
and trans_aexp (Num i)        =
    Val{v=(IMP.Num i), annote=[]}
  | trans_aexp (Var s)        = Name{n=s, annote=[]}
  | trans_aexp (Plus(a1, a2)) = Arith{e1=(trans_aexp a1),
              e2=(trans_aexp a2),
              aop=IMP.PLUS,
              annote=[]}
  | trans_aexp (Sub(a1, a2))  = Arith{e1=(trans_aexp a1),
              e2=(trans_aexp a2),
              aop=IMP.SUB,
              annote=[]}
  | trans_aexp (Mult(a1, a2)) = Arith{e1=(trans_aexp a1),
              e2=(trans_aexp a2),
              aop=IMP.MULT,

(*
 * Interpret an ICODE_IMP expression.
 * val interp_icode_imp  : string * domain list -> icode
 *                         -> string * domain list
 * val interp_icode_aexp : string * domain list -> icode
 *                         -> domain
 * val interp_icode_bexp : string * domain list -> icode
 *                         -> domain
 *)
fun interp_icode_imp rho (EPSILON) = rho
  (* Assignment. *)
  | interp_icode_imp rho (Assign{lvalue=l,
         rvalue=r,
         annote=an}) =
    (case l of
       Name{n=s, annote=a} => (s, interp_icode_aexp rho r)::rho)
  (* If-then-else. *)
  | interp_icode_imp rho (Select{guards=((g1, g2, g3)::gs),
         annote=an}) =
    if (interp_icode_bexp rho g1) =
      Bool true then interp_icode_imp rho g2
    else interp_icode_imp rho g3
  (* While loops. *)
  | interp_icode_imp rho (Iterate{guards=gs,
          annote=an}) =
    (case gs of
    ((g1, g2)::ls) =>
    if (interp_icode_bexp rho g1) = Bool true
    then interp_icode_imp (interp_icode_imp rho g2)
              (Iterate{guards=gs, annote=an})
    else rho)
  (* Sequences. *)
  | interp_icode_imp rho (NameSpace{name=n,
            space=(i1::i2::[]),
            annote=an}) =
    interp_icode_imp (interp_icode_imp rho i1) i2
  (* Wild-card. *)
  | interp_icode_imp rho _ = rho
and interp_icode_aexp rho (Val{v=value, annote=an}) =
    (case value of
    IMP.TRUE  => BOTTOM
        | IMP.FALSE => BOTTOM
        | IMP.Num i => Int i)
  (* Variables. *)
  | interp_icode_aexp rho (Name{n=x, annote=a}) =
    state rho x
  (* Aexp expressions. *)
  | interp_icode_aexp rho (Arith{e1=a1,
         e2=a2,
         aop=oper,
         annote=an}) =
    (case oper of
    IMP.PLUS => (case interp_icode_aexp rho a1 of
         Int i => (case interp_icode_aexp rho a2 of
           Int j => Int (i + j)))
        | IMP.SUB  => (case interp_icode_aexp rho a1 of
         Int i => (case interp_icode_aexp rho a2 of
           Int j => Int (i - j)))
        | IMP.MULT => (case interp_icode_aexp rho a1 of
         Int i => (case interp_icode_aexp rho a2 of
           Int j => Int (i * j))))
  (* Wild-card. *)
  | interp_icode_aexp rho _ = BOTTOM
and interp_icode_bexp rho (Val{v=value, annote=an}) =
    (case value of
   IMP.TRUE  => Bool true
       | IMP.FALSE => Bool false
       | IMP.Num i => BOTTOM)
  (* Bexp expressions. *)
  | interp_icode_bexp rho (Bool{e1=b1,
        e2=b2,
        bop=oper,
        annote=an}) =
    (case oper of
   IMP.EQ  => (case interp_icode_bexp rho b1 of
       Int i => (case interp_icode_bexp rho b2 of
               Int j => if i = j then Bool true
            else Bool false))
       | IMP.LT  => (case interp_icode_bexp rho b1 of
       Int i => (case interp_icode_bexp rho b2 of
               Int j => if i < j then Bool true
            else Bool false))
       | IMP.NOT =>
         if (interp_icode_bexp rho b1) = Bool false
   then Bool true
   else Bool false
       | IMP.AND =>
         if (interp_icode_bexp rho b1) = Bool true andalso
       (interp_icode_bexp rho b1) = Bool true
   then Bool true
   else Bool false
       | IMP.OR  =>
         if (interp_icode_bexp rho b1) = Bool true orelse
       (interp_icode_bexp rho b1) = Bool true
   then Bool true
   else Bool false)
  (* Wild-card. *)
  | interp_icode_bexp rho _ = BOTTOM

end (* structure ICODE-IMP *)
```