School of Mathematical and Computer Sciences
Heriot-Watt University
Honours Degree in Computer Science

CS4 Dissertation:

Automatic Generation of Algorithmic Properties (AutoGAP)

Tommy Ingulfsen
email: ceeti@macs.hw.ac.uk

Supervisor: Andrew Ireland

June 2003

DECLARATION

I,

....., confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: ..... Date:

Acknowledgements

The author would like to thank the supervisor, Dr. Andrew Ireland for his ability to motivate, his patient review of all written documents, his willingness to support whenever problems arose and for providing office space in the Dependable Systems Group laboratory. Also, the second reader, Professor Fairouz Kamareddine, deserves credit for her independent reviews of the deliverables and her helpful comments. Research Associate Bill James Ellis is given a special mention. Throughout the year he has been most helpful and patient whenever implementation problems arose, which happened frequently. His expertise on Prolog, Latex, Unix and formal verification in general has been invaluable. He also contributed with writing part of the software, and on numerous occasions showed us other examples of his own code. Praxis Critical Systems Ltd. kindly provided the full SPARK programming language grammar, without which this project would have been much more difficult. Finally, we thank Ben Gorry for reviewing this document.

Abstract

We develop a framework for heuristic code-level analysis in support of program proof. Our hypothesis is that this will increase proof automation. We see this framework as part of a larger research project, and the system that we engineer will be integrated with the rest of the project software.

The framework is based on the general concept of algorithmic patterns. An algorithmic pattern is a pattern of data type manipulation that is common to a family of algorithms, e.g. array based searching and sorting algorithms. We develop seven different heuristics and test them on non-trivial sorting and searching algorithms.

Heuristic code-level analysis focuses on two kinds of properties, logical properties, which directly support proof, and meta-data, which supports the search for proof.

The system we create performs a bottom-up style code analysis that complements the existing top-down analysis that most contemporary proof software uses. This will provide additional leverage in developing program proofs.

Contents

Chapter 1

Introduction

Safety-critical software systems are systems whose failure can cause catastrophic injury or loss of life. Examples of such software include applications for air traffic control, fly-by-wire software, computer programs for ABS brakes on cars and nuclear power station control systems.

Because of the risks involved, safety-critical software is subject to much more stringent standards than other computer systems, for example the UK MoD Defence Standard 00-55 [19]. Typically formal methods are used in the requirements specification phase to express the desired workings of the system in mathematical terms. In the coding and testing phase, the most stringent approaches demand formal verification of code. Such verification involves mathematical proof, and is very time consuming and demands significant expertise if carried out by hand. Computer assistance is essential when conducting mathematical proof within the context of formal verification. This is true whether we are reasoning about specification and/or program code.

Except in special cases, fully automatic verification is not possible. The focus of this project is in supporting verification at the level of program code. Often the verifier is able to handle most of the code itself, but for the remaining 10%-20%, say, the user must actively participate. One reason why verifiers do not manage to verify all code is that they are unable to

derive from the code all the necessary information required for the proof to go through, so most user interaction involves the user supplying the missing knowledge. Almost all verifiers use purely mechanical methods for finding code information, whereas users tend to use heuristics developed through experience when they find the missing knowledge.

Little research has been carried out on how one can automate the heuristics that users apply in this process. In this dissertation we investigate how heuristics can be encoded within a verifier so that some of the knowledge supplied by the user in traditional verifiers is automatically discovered. This will lead to less user interaction, thus increasing productivity since users are able to spend less time on the proof level and more on other tasks.

In this dissertation we propose and prototype a framework for extracting such information from source code. This information will complement the knowledge that the verifier derives on its own, and thus enable the verifier to carry out more of the verification task without human help. We implement our methods within an extendible software framework, so that further experiments and prototyping can take place using our software.

We call our framework AUTOGAP - *Automatic Generation of Algorithmic Properties*. AUTOGAP allows us to represent common patterns of algorithms and associate algorithmic properties with these patterns. An algorithmic pattern is a general pattern that often occurs in programs. For example, programs that frequently manipulate arrays provide good examples of algorithmic patterns. This is because most code tends to treat arrays in the same way, using loops to operate on them. Algorithmic properties, or algorithmic patterns, identify such frequently used manners of computation, expressing them in a general way so that we can reuse them for different algorithms. Two kinds of properties are considered, conventional logical properties that support program proofs, and properties that guide the search for program proofs. We refer to these two kinds of properties as *logical properties* and

meta-data respectively.

AUTOGAP forms part of the NUSPADE project. The aim of the NUSPADE project is to increase the level of automation for one particular verifier, called SPADE. The programming language that NUSPADE and AUTOGAP are developed for is called SPARK. SPARK is an industrial strength high integrity language.

Some background knowledge of formal methods in general and of AUTOGAP in particular is needed to understand this dissertation. The necessary background is given in the next chapter, together with an outline of the AUTOGAP proposal. We then develop the requirements for the AUTOGAP software in chapter 3. There is a separate chapter 4 on system design, where AUTOGAP is decomposed into its constituent parts. After elaborating the design we describe in chapter 5 how the AUTOGAP software was implemented, so that the necessary maintenance can be carried out. Finally, in chapters 6 and 7, we present the test results and discuss the achievements we have made.

Chapter 2

Background

As mentioned in chapter 1, AUTOGAP can be placed within formal verification, which is part of the larger area of computing known as formal methods. In this part of the document we aim to provide the reader with the necessary background knowledge to understand AUTOGAP. First we give a general introduction to formal methods, with a focus on formal verification. Then we elaborate on the theory behind AUTOGAP, giving examples of what algorithmic patterns are and how they can be used in verification.

Finally we place AUTOGAP within the research project NUSPADE. This will enable us to clearly explain the problem we are trying to solve with AUTOGAP, and the contribution we have made to the NUSPADE project.

2.1 Formal methods

Formal methods can be divided into formal specification and formal verification. Common to both of these areas is that they are mostly applied to safety-critical software, where correctness is particularly important. Below we give a brief introduction to formal methods. The interested reader should consult [11], [6] and [16] for a more complete overview of the area.

Formal specification uses mathematics to describe the software system

and its properties. Therefore, a formal specification is much more precise and unambiguous than a specification written in English. This is a great advantage, because it avoids misunderstandings between engineers and customers due to English usage and also because mathematics allows us to reason about the specification in a rigorous way.

Formal verification goes one step beyond formal specification; it is used to analyse a system for desired properties and to prove correctness. There are two main approaches to formal verification, *model checking* and *theorem proving*.

In model checking we build a finite model of a system and check via verification that some desired property holds in the model. This check is performed as an exhaustive state space search, guaranteed to terminate since the model is finite. Model checking is a powerful technique which can be carried out by computers in a matter of minutes or hours, but may take days, depending on the size of the model.

Theorem proving for verification is about proving logical properties of programs using a formal logic. This involves combining the axioms and rules of the logic in order to justify a conjecture. Theorem proving is more powerful than model checking, but in general, requires significant user interaction.

2.1.1 Formal specification

The following table summarises of some important techniques for specifying software systems.

Abstraction level	**Purpose**	**Language**
High	Modelling	Z, LOTOS
	Modelling with refinement	B, VDM
Low	Code-level specification	SPARK, ESC/Java

Table 2.1: Summary of formal specification techniques.

There is a hierarchy of techniques that depends on the abstraction level employed. At the highest level we have modelling techniques like Z and LOTOS, which model the entire software system in terms of logic.

Modelling with refinement encompasses methods that not only model the system but also refines the model from a higher to a lower level of abstraction. Using the B-Method and VDM we start by creating a specification of the required functionality. The design then proceeds by refining the specification to a lower level again and again until the specification is defined at such a low level that it is ready to be implemented. Code generation is suitable for these methods since the refinement process can be automated.

At the lowest level in the hierarchy of specification techniques, we find methods that work at the code level. In contrast to the other methods, SPARK and ESC/Java do not attempt to model the entire software system but instead concentrate on the code itself. The programmer is required to supply extra *annotations*, special comments in the code, that contain information on how the program should behave. Given the annotations, code level specification techniques perform *static analysis* of the source code, which means that the code is analysed at compile-time. The program is translated into a set of *verification conditions (VCs)*, which expresses the code using logic formulae. Proving the VCs formally guarantees the correctness of the code with respect to the specification. Proving the VCs is achieved via a theorem prover, a computer program that carries out proof. Automated reasoning is the research area for this kind of verification. An accessible survey of the field is provided in [5]. Examples of annotations include pre- and postconditions and loop invariants. The most ambitious systems, like SPARK, use annotations both for error checking and for formal verification. ESC/Java does not to aim verify code, but only carries out the static analysis to find errors. Code level specification techniques provide the static analyser as a separate program, so that the user can use an ordinary

compiler to generate the executable code. See [18] for an introduction to ESC/Java. SPARK is treated separately later.

2.1.2 Formal verification of program code

When using a theorem prover to verify that software is correct, we must supply some extra information within the code by adding annotations. Most importantly, we give *pre-* and *postconditions* for our programs. Usually we want to carry out a proof of the *partial correctness* of the program. Partial correctness means that if the program is called with the right input, then, assuming that the program terminates, the program will give the right output. The input is specified by the precondition and the output is specified by the postcondition. Proving then involves showing that the postconditions - the final state of the program - are correct with respect to the preconditions - the initial state of the program. Also, we supply careful statements of what is done within the program. Such statements are called *assertions*. The point at which we insert an assertion is called a *checkpoint*. Attaching an assertion to a checkpoint means that we attach a statement about what we believe to be true at this point of execution [10]. A particularly important class of assertions are *loop invariants*. A loop invariant describes what is true within loops. Automatic loop invariant generation is undecidable in general.

Once we have added this extra information to the code, we "hand over" the code to the verification software. Now, the code is first translated into a language that the theorem prover can understand. It is in this translation process that our assertions are used. The result of the translation is a a set of *verification conditions* (VCs). A verification condition is a mathematical conjecture, and consists of some hypotheses followed by a conclusion, expressed in a mathematical language. Now, the task of the theorem prover is to prove that the conclusions of the VCs follow from the hypotheses. Thus,

the problem of program proof has been reduced to proving that the VC conclusion is valid.

What we have described so far is the traditional method of software verification. But there is a problem with this VC-oriented approach.

Despite all the extra information we put into the code, the verification package is often unable to carry out the entire proof on its own. Sometimes it simply gets stuck and needs a user to help out, demanding human interaction. Often this involves the user giving new facts about the code that the verification software did not spot during the translation from programming language code to VCs. The verification package will not go back to the code itself once the translation is complete, which is a weakness. By focusing only on the VCs, without looking more at the code, the verification software is effectively "tying one hand behind its back". We believe that extracting more information from the code, in the form of meta-data, will enable more proofs to go through without human interaction.

2.2 Exploring algorithmic patterns

As mentioned in the introduction, algorithmic patterns express general, re-occurring, structures of code. Programmers use algorithmic patterns all the time, usually unconsciously, when they reuse particular code sequences. For example, arrays are often initialised using loops, an act that appears completely natural to programmers. Algorithmic patterns are akin to design patterns, but describe software at a much lower level. If we can pick out some patterns that can be useful for proof, and search for them in code, we will be able to deduce additional knowledge about how a program works. This type of knowledge is not manifested in VCs.

Algorithmic patterns describe general code properties. The meta-data and logical properties are instances of algorithmic patterns for particular programs. We want to use algorithmic patterns to drive the search for such

information. The additional logical properties are often crucial to obtaining a proof while meta-data is used to guide search. Let us reinforce the explanation with an illustrative example.

2.2.1 Algorithmic patterns at work

The following code snippet, written in the SPARK language, should illustrate what AUTOGAP does. Note that **Flag'First** and **Flag'Last** refers to the subscripts of the first and last elements of the array **Flag** respectively. Incidentally, the code presents a solution to a famous programming problem, the *Polish Flag problem* [1]. See [7], [14] and [1] for descriptions of this problem.

[1] The Polish Flag problem is sometimes called the two-colour problem, and is a simplification of the Dutch Flag problem, which deals with three colours

```
type Colour is (Red, White);

procedure Partition_Section(Flag: in out ArrayOfColours)
--# pre (for all I in IndexRange => (Flag(I)=Red or Flag(I)=White));
--# post for some P in Integer range Flag'First..(Flag'Last+1) =>
--#         ((for all Q in Integer range Flag'First..(P-1) => (Flag(Q) = Red)) and
--#         (for all R in Integer range P..Flag'Last => (Flag(R) = White)));

is
    subtype JustBiggerRange is Integer range Flag'First .. Flag'Last+1;
    I: JustBiggerRange;
    J: JustBiggerRange;
    T: Colour;
begin
    I:=Flag'First;
    J:=Flag'Last+1;

    loop
      --# assert true;
        exit when I=J;
        if Flag(I)=Red then
            I:=I+1;
        else
            J:=J-1;
            T:=Flag(I);
            Flag(I):=Flag(J);
            Flag(J):=T;
        end if;
    end loop;
end Partition_Section;
```

Figure 2.1: Polish flag problem

This is a primitive sorting algorithm that takes an array of colours and sorts it so that all the Red elements precede all the White elements. This outcome is expressed by the postcondition.

Variables `I` and `J` play an important role in the algorithm, because they are *index variables*. An index variable is used to index arrays. The idea behind the use of `I` and `J` is that they refer to two distinct partitions of the array. All elements in the lower part are red while all the elements in the upper part are white. The lower part is indexed by `Flag'First..I-1` while the upper part is indexed by `J..Flag'Last`. This is conveyed in picture (b)

of figure 2.2.

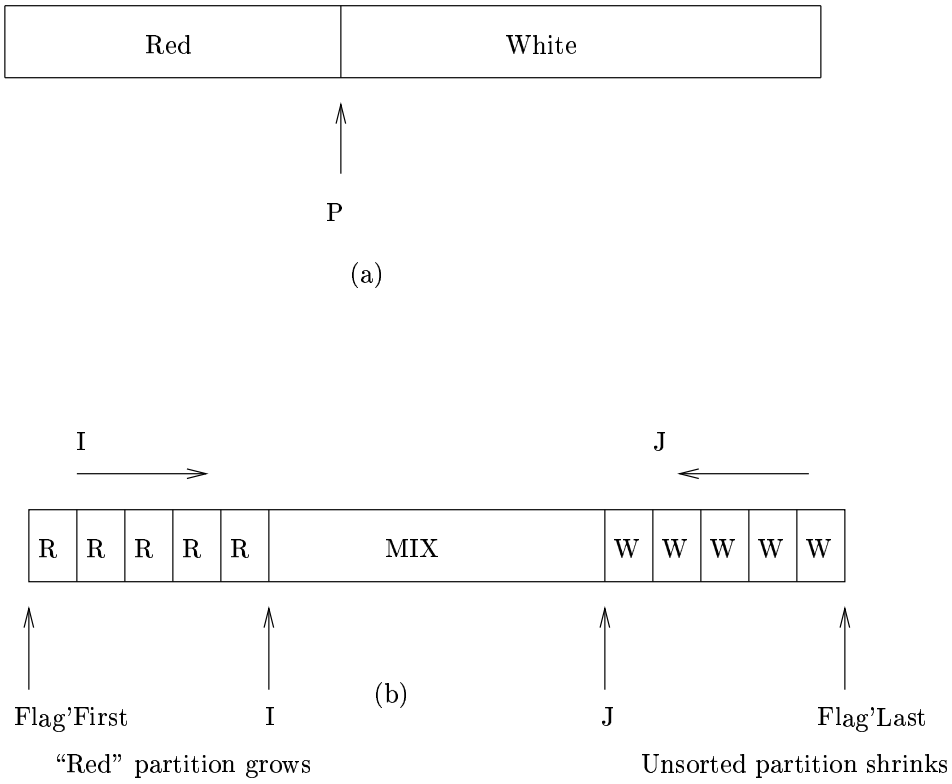

Figure 2.2: (a) illustrates the postcondition. When sorting finishes there are two partitions. All the Red elements precede the White ones. (b) shows three partitions in the array for the Polish Flag problem during execution. The algorithm places all the Red elements before the White ones. `I` indexes the sorted part of the array while `J` indexes the unsorted part. In the middle there is a mix of colours that have not yet been sorted. (b) captures a loop invariant for the program.

Note also the user supplied loop invariant - `--# assert true`. This invariant is inadequate for the partial correctness proof of this program. Picture (b) in figure 2.2 shows the necessary invariant, which can be written as a SPARK annotation like this:

```
--# assert  I<=J and
--#        (for all Q in Integer range Flag'First..(I-1) => (Flag(Q)=Red)) and
--#        (for all R in Integer range J..Flag'Last => (Flag(R)=White));
```

The assertion involves three conjuncts, and AUTOGAP seeks to generate properties like `I <= J` directly and automatically from the code. AUTOGAP also aims to provide meta-data on the code that will guide a proof planner in generating the second and third conjuncts during the course of a proof. Note that the last two conjuncts of this invariant represent a weakening of the given postcondition. AUTOGAP will not generate such properties directly, but the discovery of meta-data will aid the proof planner in deducing this knowledge. The trick involves finding the instantiation for `P` that will provide a suitable invariant. Note that any instantiations for `P` of the correct type are possible - the problem is in selecting the right instantiation. The key insight is in identifying that the variables `I` and `J` define the upper and lower bounds on array partitions for which the desired postcondition property holds. Let us study how we can discover such meta-data and properties like `I <= J` automatically.

2.2.2 Searching for algorithmic patterns

When the above program starts, it initialises `I` to be smaller than `J`. Inside the loop, `I` is incremented while `J` is decremented. These operations are the only ones performed on the loop control variables themselves, so we may say that `I` is *monotonically increasing* while `J` is *monotonically decreasing*. These statements about how the loop control variables change during execution can be represented as meta-data. In identifying the monotonic nature of `I` and `J`, we used two algorithmic patterns; we looked for variables that only increase or decrease.

In this particular example, elements `0` up to `I` and `Flag'Last` down to `J` denote partitions with only `Red` or `White` elements. This observation, or what we call meta-data, about the code strongly suggests that `P` should be

instantiated to be `I` in the first conjunct of the postcondition and to be `J` in the second conjunct.

Analysing the code further, we see that before the loop is entered it is true that `I < J`. At the start of the loop, where the present invariant is inserted, all we can say about the relationship between `I` and `J` is that `I <= J` - which is part of the loop invariant.

Since `I` increases while `J` decreases we can identify at least two different partitions of the array, as shown in figure 2.2. For this example, we are thus able to deduce that the variable `I` is the upper bound on a partition of the array while `J` is a lower bound. So the meta-data represents aspects of picture (b) in figure 2.2.

2.2.3 Two kinds of algorithmic properties

When describing algorithmic properties it is useful to introduce two overall classes. These correspond to information that can be used directly to generate VCs and information that cannot be used in VC-generation, but can still be useful.

The first kind of algorithmic properties are called *logical properties*. Such properties contribute directly to VC-generation. The most important example of this class of properties are loop invariants, such as `I <= J` in the example above.

The second kind of algorithmic properties we call *meta-data*. Such properties are more abstract than logical properties. They do not enable VC-generation and are therefore not directly useful to the theorem prover. Meta-data are properties such as variables being monotonically increasing, array partitions changing (as in the example above) and other properties that are not directly translatable into VCs.

2.3 The SPARK Language

In order to find algorithmic properties, AUTOGAP will carry out static analysis of source code, as roughly lined out in the previous example. We focus on analysis of programs written in SPARK, a high-integrity programming language developed by Praxis Critical Systems Ltd. [2] [2]

SPARK is a subset of Ada, augmented with an annotation language. The annotations are written as special Ada comments, which allows users to apply any Ada compiler to generate executable code. We used SPARK for the Polish Flag example, where the pre- and postconditions are examples of annotations. SPARK excludes many unsafe Ada constructs in order to provide a language that is suitable for safety-critical systems development. Recursion and dynamic memory allocation are examples of features that have been removed, since they provide a risk of the program crashing by running out of memory. Developing safety-critical software using SPARK involves coding in SPARK, applying a static analyser called the Examiner to the code in order to generate verification conditions, and proving these verification conditions using the SPADE theorem prover. The example of section 2.2.1 is written in SPARK.

2.4 The NUSPADE Project

The NUSPADE project is a collaboration between the Dependable Systems Group and Praxis Critical Systems Ltd [3]. The primary focus of this research is to apply recent advances in proof planning and automatic deduction to industrial strength problems. Praxis Critical Systems provide a vehicle for this, the SPARK programming language and its associated toolset. [13] and [4] provide good explanations of proof planning.

The SPADE theorem prover is automated, and is able to carry out parts

[2] The SPARK website is http://www.sparkada.com/
[3] See http://www.cee.hw.ac.uk/~air/clamspark/

of proofs without any help from human users. The remaining parts, however, require human interaction at a sophisticated level. This human engagement is time consuming and therefore a bottleneck in the overall proof effort. In NUSPADE we seek to reduce this bottleneck by improving SPADE in such a way that users need contribute less in the proof process. The *research hypothesis* of the NUSPADE project is that:

By applying proof planning techniques to improve the SPADE theorem prover, significantly less human interaction will be required. This will speed up the proof process.

SPADE takes VCs as input and uses a set of rewrite rules to attempt a proof. SPADE's task can be viewed as searching a space of several hundred rewrite rules to find applicable rules. The proof is then carried out using the matching rewrite rules. Thus SPADE is a low-level tool. With proof planning we attempt to bring the verification task to a higher abstraction level by introducing an additional level of control above SPADE, the SPADE-PP proof planner. This proof planner sits on top of SPADE, taking the VCs as input and producing a tactic. A tactic is a kind of "program" that specifies the rewrite rules that SPADE should apply. Hence the search of the rule space is much constrained. Together, SPADE and SPADE-PP make up the NUSPADE software, as shown in figure 2.3.

2.5 AUTOGAP - Automatic Generation of Algorithmic Properties

AUTOGAP will work in support of NUSPADE software. The task of AUTOGAP is to analyse source code in order to extract algorithmic properties that will enable the NUSPADE theorem prover to carry out more proofs without help

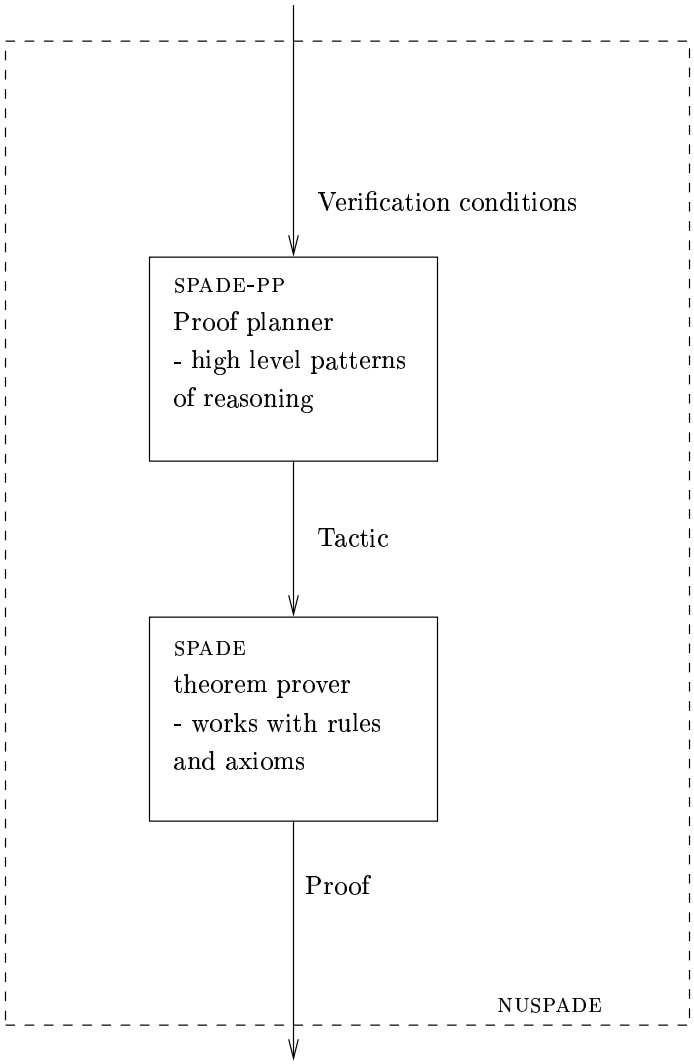

Figure 2.3: With proof planning we extend theorem proving by providing an extra layer of control on top of the SPADE theorem prover. A main outcome of the NUSPADE project is the NUSPADE software system, an augmented theorem prover for SPARK with a proof planner.

from humans, partly alleviating the disadvantages of the VC approach.

2.5.1 The task of AUTOGAP

The AUTOGAP system will perform an extra level of source code analysis to add to the knowledge presented to the theorem prover as VCs. This added knowledge is algorithmic properties, deduced by search. The hypothesis that motivates us to build AUTOGAP is:

Knowledge of the algorithmic patterns that occur within a program can significantly increase automation of program proof.

2.5.2 The relationship between AUTOGAP and the NUSPADE system

AUTOGAP can be viewed both as an additional system within NUSPADE and as stand-alone software. Figure 2.4 gives a simple overview of the interaction between the SPADE-PP proof planner and AUTOGAP.

AUTOGAP will be applied to a file of SPARK, analyse the code and produce algorithmic properties for that program. AUTOGAP works as a stand-alone system at the moment, because SPADE-PP is not yet complete. This means that for the moment the output of AUTOGAP is made available to the user only. In the future it is intended that AUTOGAP is integrated within NUSPADE, so that SPADE-PP can use AUTOGAP's results in order to simplify the proof task.

AUTOGAP's contribution to the NUSPADE project is to provide an extra level of code analysis within an extendible framework. The analysis encodes heuristics that are akin to those humans use when interacting with verification packages and so AUTOGAP will give NUSPADE an extra "edge" since few other verification systems use such methods. The logical properties that

AUTOGAP generates are concrete enough to be translated into SPARK annotations and thus used by SPARK to generate improved VCs. These VCs will be easier to prove automatically than the original ones since they contain more information. The more abstract meta-data can be used by NUSPADE to constrain the search for applicable proof rules. AUTOGAP is extendible so that experiments with other heuristics than those presently used can be conducted within the AUTOGAP framework. We will revisit this feature of our system in chapter 3.

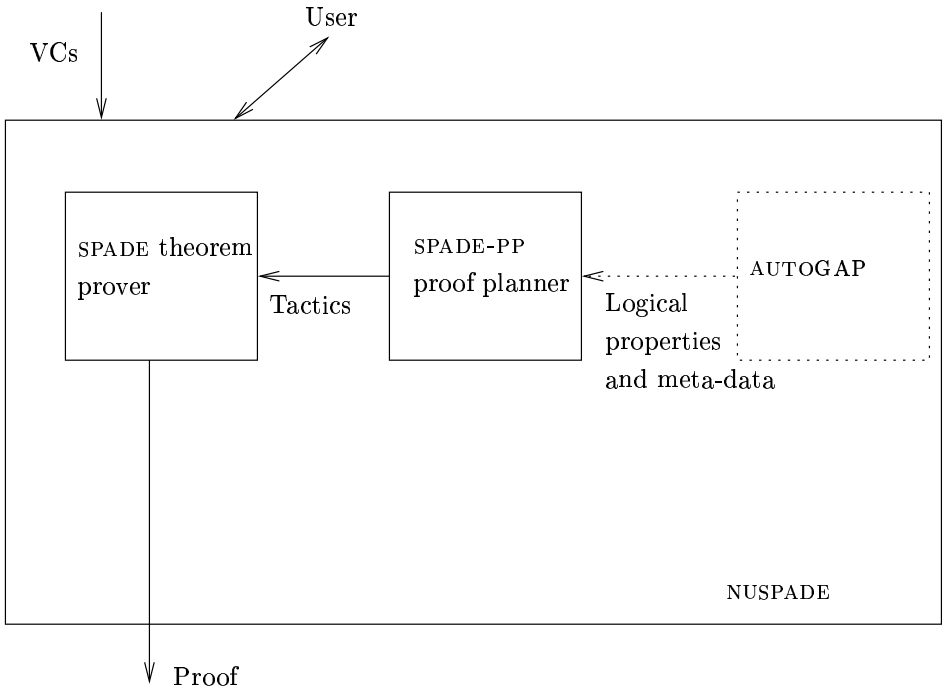

Figure 2.4: AUTOGAP and the NUSPADE system. AUTOGAP provides an extra input to the SPADE-PP proof planner.

2.5.3 AUTOGAP explained

SPADE-PP's use of the algorithmic properties generated by AUTOGAP is an example of a *heuristic approach* to program verification, where heuristic

techniques are employed to obtain more knowledge about the program. This approach was introduced by Zohar Manna and Shmuel M. Katz in [17]. Heuristics are like rules of thumb, where we employ a mixture of science and knowledge gained from experience to analyse the code. This is how users help the theorem prover carry out a proof that it is unable to complete alone, using their experience and understanding of the proof process to find the missing pieces of information that the prover needs. Note that there is no guarantee of success when using a heuristic approach. However, we do have empirical evidence for the effectiveness of such an approach, since users apply this successfully when they interact with theorem provers.

The paper by Manna and Katz was influential, but the techniques they described were never implemented in a mechanised form. The properties they focused on are still mostly supplied to the theorem prover by human users. In building AUTOGAP we take inspiration from their ideas. Note that AUTOGAP is not intended to be a "complete" tool, but a "proof of concept", showing that heuristic techniques can be mechanised. Also, it is important to mention that our approach is novel in the sense that it includes meta-data. [17] only mentions the more concrete logical properties.

AUTOGAP will attempt to translate the more concrete properties, the logical properties, into SPARK annotations. In SPARK annotations supply the extra information needed to produce VCs. Meta-data, which cannot be translated into annotations and therefore have no connection with VCs, will be taken up by the SPADE-PP proof planner instead.

Top-down and bottom-up methods for code analysis

In [17] Katz and Manna distinguish between two general approaches to obtaining algorithmic properties, *top-down* and *bottom-up methods*.

In the top-down approach we expect to be given information about what is true at various points in the program. This information should be given

by the programmer in the form of assertions. For example, in generating a loop invariant we would combine assertions that are true at the beginning and end of the loop. An example of a top-down method is VC-generation, which in SPARK requires annotations in order to work.

Using the bottom-up approach we analyse the code itself in order to identify algorithmic properties, as briefly sketched in the example of section 2.2. We do not assume that any extra assertions are supplied by the user. The AUTOGAP system will identify algorithmic patterns using the bottom-up approach, as a complement to the top-down approach presently used for VC-generation. We believe that by using bottom-up analysis methods in addition to the existing top-down VC-generation, NUSPADE will be able to carry out more proofs on its own than if just one of the approaches were used. As Katz and Manna put it in [17], for simple programs "it is generally clear that the top-down approach is the natural method to use. However, ... for real (nontrivial) programs ... bottom-up techniques were found indispensible." An additional advantage of our bottom-up approach is that by automatically generating annotations, we alleviate the user from some of the responsibilities of adding annotations to the code. In [18] adding annotations is identified as a significant bottleneck because many programmers are unfamiliar with adding them.

Chapter 3

Software requirements

3.1 Introduction

Here we discuss in more detail what the requirements of AUTOGAP are. Firstly, we state the overall objectives of the software and then we consider the system at a lower level. While this chapter takes the form of an informal discussion, note that Appendix D contains a more detailed listing of individual requirements.

3.2 Overall objectives

AUTOGAP should analyse SPARK code and produce logical properties and meta-data. This will provide an extra level of code analysis, which will be useful for verification of that source code. The analysis should be based on a number of *heuristics*. These heuristics, or *algorithmic patterns*, are divided into two classes, logical properties and meta-data. Logical properties are directly relevant for VC-generation in SPARK because they can be expressed as annotations in the code. Meta-data will help constrain the proof search. Furthermore, AUTOGAP should be designed and implemented such that it may be integrated into NUSPADE at a later time (when the imple-

mentation status of NUSPADE allows this). Also, AUTOGAP should provide an extendible framework so that it is possible to add further heuristics in the future. AUTOGAP is envisaged as a fully automatic software tool used in batch mode. Note that AUTOGAP is not intended to be a complete tool, but a prototype to show how heuristics may be encoded and automated.

3.3 System requirements

Here we look at the different features of AUTOGAP in turn, and give the resulting requirements.

3.3.1 Algorithmic patterns

We will focus the analysis on non-trivial programs that have loops and perform computations on arrays. These are the sorts of programs that are most interesting from a verification perspective, since they can be hard to prove and often require human interaction for the proof to succeed. Therefore, the algorithmic patterns employed by AUTOGAP focus on arrays and loops. Let us define the patterns that AUTOGAP will use. The definitions given below are fairly short, due to space constraints, but a fuller version appears in Appendix D together with an explanation of the syntax for the output of each heuristic.

Initial values It is useful to know the initial value of variables. The initial value of a variable is the first value it is assigned. However, if the variable is assigned to for the first time after it has been referenced in a test or another assignment, no initial value will be recorded. An example of such a case is variables that are input to a procedure, and therefore initialised outside that procedure.

Monotonic variables Some variables are monotonically decreasing or increasing within a loop. This means that they consistently increase or

decrease, and never oscillate in relative value.

Counter variables These are variables that appear in a loop exit condition and have their values changed by assignment in that loop.

Index variables Index variables appear as an index into an array.

Array partitions Within a loop there is a partition in an array. This heuristic will name the array, the start and end of the partition and the loop within which the partition exists.

Bounds We ascertain what the minimum or maximum values of a variable are.

Loop invariants A loop invariant is a statement about some of the variables within a loop that is true at all times.

Of these seven heuristics, six are meta-data that identifies properties of variables and arrays. The loop invariants pattern is a logical property. In SPARK loop invariants can be expressed as annotations within the code. Note that these heuristics are just a sample of many possible ones and that more than one definition of a heuristic is possible.

3.3.2 Use and integration with NUSPADE

We are seeking to increase the automation of program proof, so AUTOGAP should be a fully automatic tool. Since AUTOGAP is intended for use by programmers and scientists a simple, quick interface is preferable.

We therefore require the AUTOGAP software to run in batch mode, with no intervention from the user. The ideal way of invoking AUTOGAP would be a single command followed by the name of the file we want to analyse.

NUSPADE will not be ready for integration with AUTOGAP until after this dissertation is finished. However, AUTOGAP should be implemented in such a way that integration can happen seamlessly. The best way of ensuring

this is to have AUTOGAP as a program on its own that can be invoked from within NUSPADE, and to let the result of AUTOGAP be written to a file that is readable by NUSPADE. Since AUTOGAP is meant to be a prototype, we also require it to be extendible, so that it is easy to add new heuristics. By implementing the AUTOGAP system in a kind of library fashion, AUTOGAP will not be a monolithic program but one that can be used as a framework for prototyping and experimenting.

3.3.3 Analysis

In order to analyse SPARK source code it will be necessary to parse SPARK, so AUTOGAP should contain a parser. To keep the size of AUTOGAP down the parser should apply to a subset of SPARK rather than the entire language. The entire subset appears in Appendix G. It excludes administrative constructs like packages but still contains enough constructs to enable coding of interesting programs. Also, we will assume that the user has applied the Examiner (the SPARK static analyser) to the source code before submitting it to AUTOGAP. This is advantageous because the Examiner contains a full SPARK parser, which means that no error checking will be needed in the AUTOGAP parser.

After parsing, the analysis itself should take place on an internal representation of the code. This representation should be a graph, the most obvious and useful structure for analysing computer programs. This analysis should look for instances of the algorithmic patterns given above.

3.3.4 AUTOGAP output

The results of AUTOGAP should be made available to both the user and for future use within NUSPADE. The easiest way of achieving both these goals is to have AUTOGAP produce some new files as its output rather than displaying meta-data and logical properties on the screen. Two files are

called for; one that holds the results in a format accessible to NUSPADE and one which relates the results to the code of the analysed program. When possible, logical properties should be integrated into the original code as annotations. This will give the user feedback in the relevant place in the code and enable SPARK VC-generation to proceed with added annotations. The only logical property that can be translated into annotations is the loop invariants algorithmic pattern.

Chapter 4

Design

Now that we have shown how AUTOGAP exists in context with NUSPADE and given the requirements we can design the AUTOGAP software. Let us start by considering a large picture, figure 4.1.

AUTOGAP takes SPARK code, parses it and then analyses it. The output (meta-data and logical properties) is made available to the NUSPADE proof planner and theorem prover. Meta-data is presented directly to NUSPADE while logical properties are combined with the original code in order to create new annotations. Using the SPARK VC generator, the new code is translated into verification conditions, which are presented to NUSPADE.

In this chapter we give a detailed explanation of the design of AUTOGAP. First we elaborate some high-level considerations, which govern how the low-level design is created. Then we move on to giving the design itself.

The design is described in terms of Data Flow Diagrams (DFDs), figures and high-level pseudocode where applicable. See [20] for a good introduction to the DFD notation.

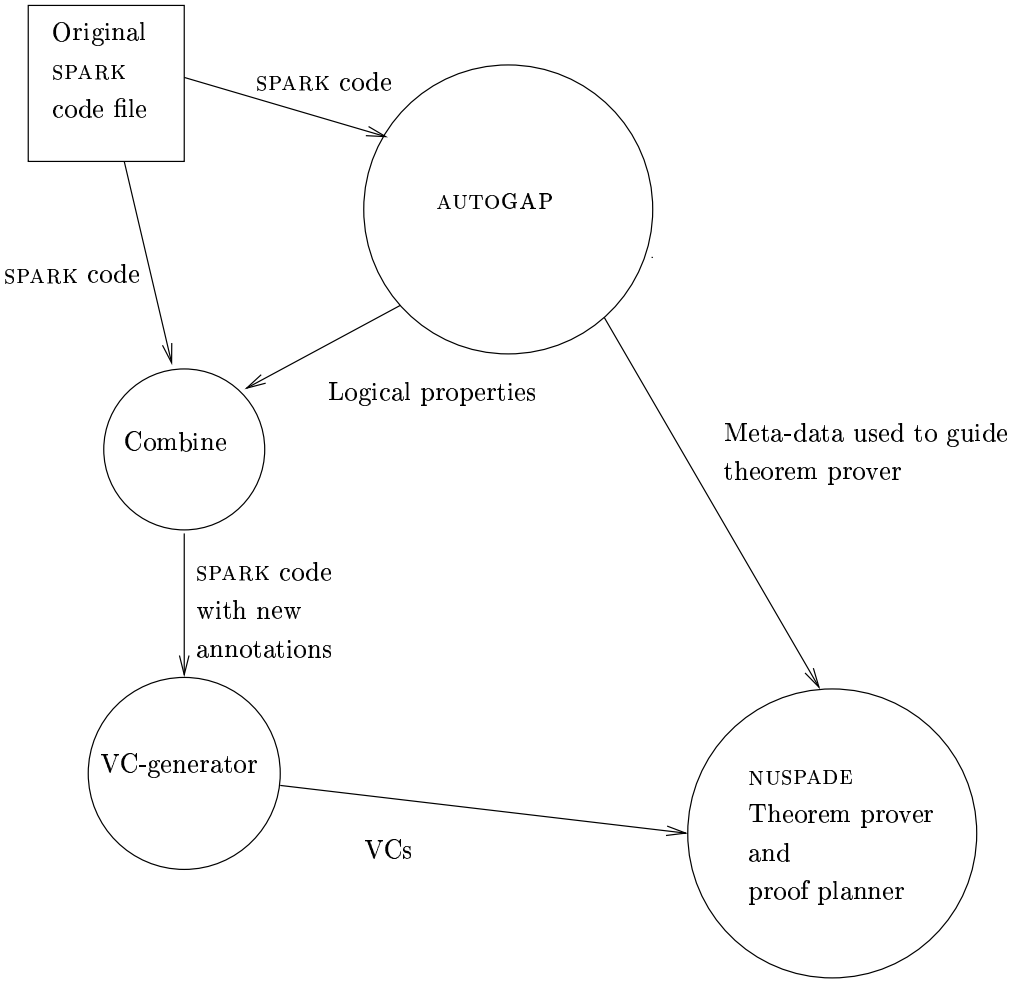

Figure 4.1: AUTOGAP takes SPARK code and produces Meta-data for guiding the theorem prover and Logical properties that can be used to create new annotations and combined with the original code.

4.1 High-level design considerations

At the highest level there some important factors that play a decisive role in how AUTOGAP is designed.

Firstly, there is a need to translate the SPARK source code into some internal representation. This internal representation must lend itself well to deriving algorithmic properties. It would be very difficult to use the SPARK code directly for this purpose. Therefore we build a *parser*, or some other preliminary processing system to carry out the transformation from source code to the internal representation.

The process of extracting algorithmic properties from SPARK code is a kind of static analysis, and a common internal representation for static analysis is the graph. Graphs allow us to represent control structures in an intuitive way. The basic graphs are shown in figure 4.2. By choosing a graph as our internal representation we can derive algorithmic properties by traversing the graph, collecting information about the program semantics and structure.

Also, we cannot afford to lose any information about the semantics of the program as we perform the translation into a graph. So any intermediate representations as well as the graph must preserve the meaning of the SPARK source code.

Finally, the design should adhere to good software engineering practice and be modular, so that the components of the system depend as little as possible on each other.

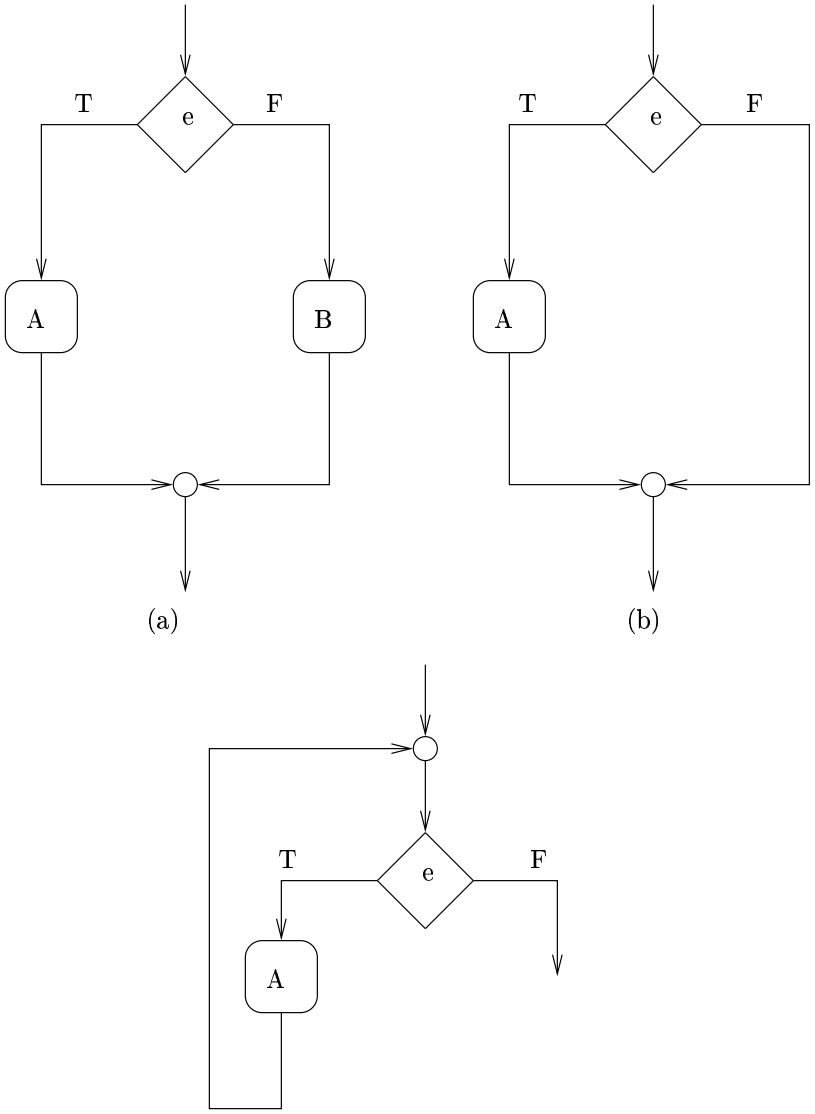

Figure 4.2: Control-flow graphs of program statements. (a) **if** e **then** A **else** B. (b) **if** e **then** A. (c) **while** e **do** A. From [3].

Having identified these high-level factors, we now have a rough layout of the AUTOGAP design. Let us first explore the translation process in greater depth, before we move on to the algorithmic properties generation.

4.2 Translating SPARK code

First, we create the top level design for AUTOGAP. Then we look closely at the part of the design that translates SPARK into our internal representation (a graph).

The top-level DFD (level 0) is shown in figure 4.3.

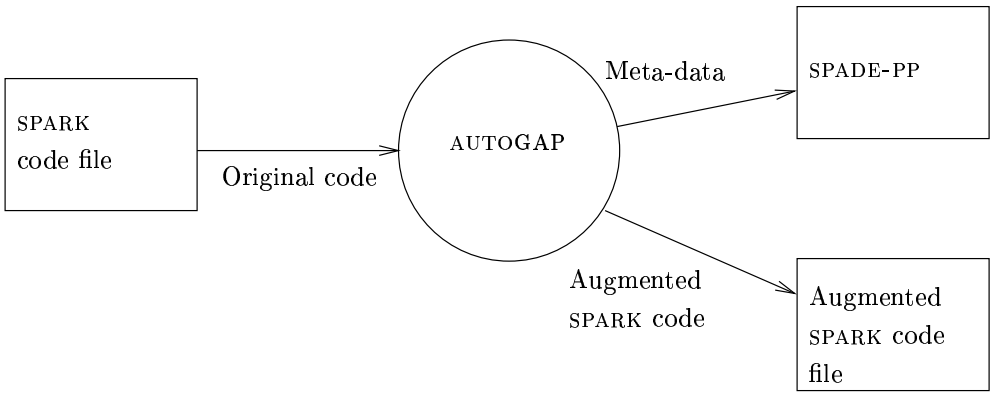

Figure 4.3: **DFD-0.** Level 0 Data Flow Diagram showing the basic inputs and outputs of AUTOGAP.

The level 0 DFD describes the entire AUTOGAP system, showing the most basic flow of data. Figure 4.3 conveys that there is one source of data for AUTOGAP, the SPARK code, and two results, Meta-data and Augmented SPARK, with new annotations. Both outputs are made available to SPADE-PP. We must now decompose this high-level view of the system into components that are described in sufficient detail to be implemented.

As noted above, we need to translate SPARK into a graph and then analyse the graph in order to generate the required logical properties and

meta-data. We propose to divide this task into two subsystems. The first subsystem will take the SPARK code and translate it into a representation from which a graph can be derived. The second subsystem's task will be to create the graph and analyse it, producing meta-data. We call these modules Preliminary Processing and the Kernel respectively. Thus, the level 1 DFD is shown in figure 4.4.

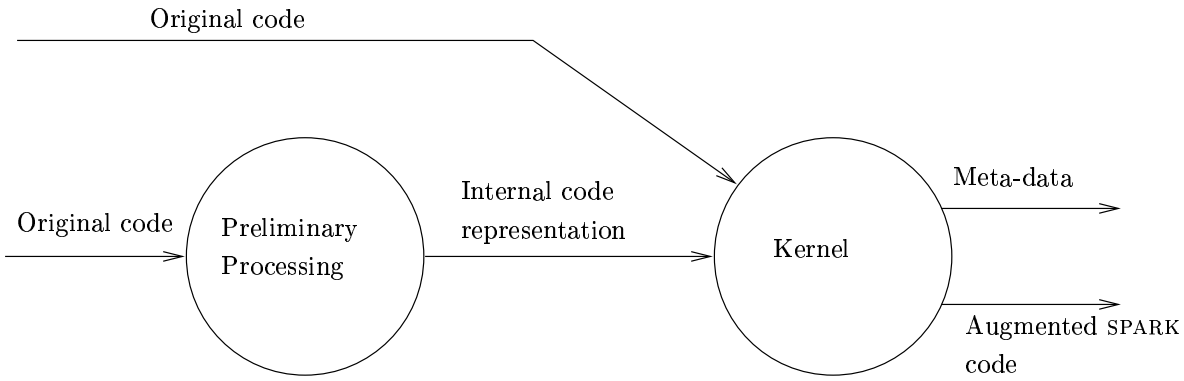

Figure 4.4: **DFD-1.** Level 1 Data Flow Diagram. AUTOGAP is broken down into two subsystems, one that does some preliminary processing on the source code and one that generates the algorithmic properties.

Another way of designing AUTOGAP would be to keep all the translation within Preliminary Processing (including the graph building) and have the meta-data generation as an entirely separate component in the Kernel. However, as we shall see later, the graph and the meta-data generation are so tightly related that they fit naturally together.

Having found two subsystems within AUTOGAP, we expand on the Preliminary Processing module. As explained above, we wish to obtain a new representation of the SPARK source for analysis. This mapping from code to internal representation is called parsing. Hence, the Preliminary Processing module is actually a kind of parser. The reason we do not name this module "Parser" instead is that some more computations are needed that

an ordinary parser cannot be expected to perform. A regular parser for a compiler takes source code and produces e.g. a parse tree as an intermediate representation, which is useful if one wants to generate machine code. But we need to carry out a kind of analysis that demands a higher level internal representation. We therefore split the Preliminary Processing module into two further subsystems; the LW-SPARK-PARSER (Light-Weight SPARK Parser) and SUBPROG-SPIDER (Subprogram Spider). This is shown in figure 4.5, a level 2 DFD.

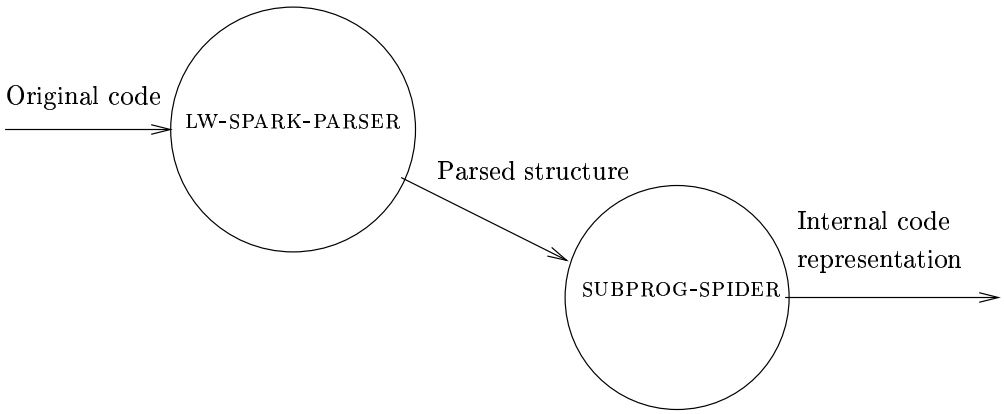

Figure 4.5: **DFD-1-1.** Level 2 Data Flow Diagram. Expands the Preliminary Processing process. Two subsystems emerge, one that deals with basic parsing and one that creates an intermediate representation.

The LW-SPARK-PARSER carries out an ordinary parse of the source code. Its output is a Parsed structure, essentially a textual version of the Parse tree that it builds as it tries to match the code statements to the grammar. By nature, a parse tree is a very low level construct that gives a lot of detail about how the source code is built up of grammatical statements. In order to obtain a more compact representation, which will be easier to convert into the graph structures exemplified in figure 4.2, we create SUBPROG-SPIDER. The task of SUBPROG-SPIDER is to take the Parsed structure and turn it into

a higher level structure that will be used to generate the graphs [1]. In all, we would like the Preliminary Processing module to produce simple output that is readily convertible into graph structures.

Let us decompose the Preliminary Processing module further, down to a level where it is ready to be implemented. The level 3 DFD that shows the components of LW-SPARK-PARSER is figure 4.6.

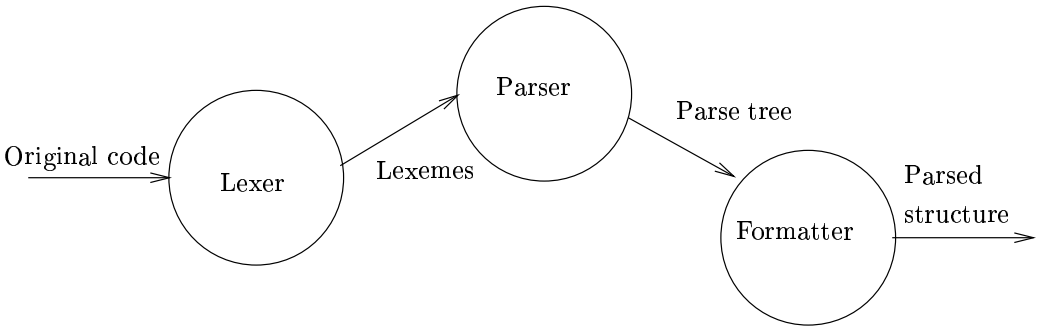

Figure 4.6: **DFD-1-1-1.** Level 3 Data Flow Diagram. Breaks the LW-SPARK-PARSER process from DFD-1-1 up into subsystems along the lines of a traditional parser. Process Formatter carries out rudimentary formatting of the Parse tree for translation into a more useful representation.

Three subsystems are found. The first stage of parsing is carried out by the Lexer. The Lexer's task is to read in the SPARK code from file and divide program statements up into the basic constituent parts, like identifiers and numbers. These *Lexemes* are fed into the Parser. The parser matches the sequence of Lexemes against a grammar for the SPARK programming language, building a Parse tree. If the entire program parses, the complete Parse tree is output to the Formatter. Here, the Parse tree is traversed and a textual representation of the Parse tree is generated, the Parsed structure. This structure is the input to SUBPROG-SPIDER.

Lexemes are defined in terms of regular expressions, like those commonly

[1] SUBPROG-SPIDER is implemented entirely by Research Associate Bill J. Ellis. This is explained in the "Implementation" section later.

used in Unix. The Lexer operates by looking at the source program and matching it against these regular expressions When a regular expression is recognised, the Lexer looks at its corresponding Lexeme definition and passes that Lexeme on to the parser, as shown in figure 4.7.

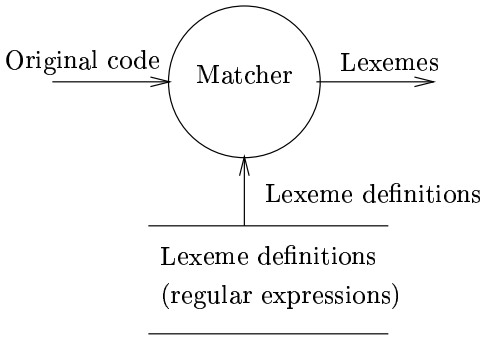

Figure 4.7: **DFD-1-1-1-1**. Level 4 Data Flow Diagram. Expands the Lexer process from DFD-1-1-1. Using Lexeme definitions in the form of regular expressions, the Lexer translates the source program into Lexemes.

The Parser matches the Lexeme sequence against a SPARK Grammar. This Grammar is defined in terms of the Lexemes. Whenever a particular Grammar rule is satisfied, a node in the Parse tree is generated and linked to the other nodes by the Parse tree builder. The organisation of the Parser is shown in figure 4.8.

The Formatter can be described as a Parse tree traverser. It moves over the parse tree, using the structure of the tree and a set of rules for how to transform the nodes of the tree into text. The Formatter's structure is shown in figure 4.9.

Having decomposed the LW-SPARK-PARSER, we now turn to SUBPROG-SPIDER. The decomposition of SUBPROG-SPIDER is shown in figure 4.10.

This is a description that is readily implemented, so we will not decom-

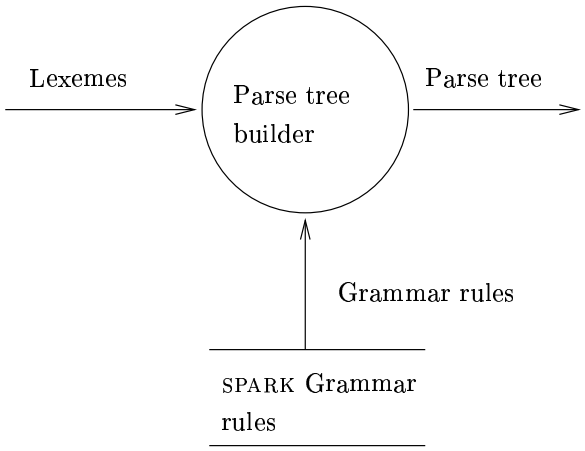

Figure 4.8: **DFD-1-1-1-2**. Level 4 Data Flow Diagram. Expands the Parser process from DFD-1-1-1. Matching the Lexeme sequence against the Grammar rules, the Parser builds a Parse tree.

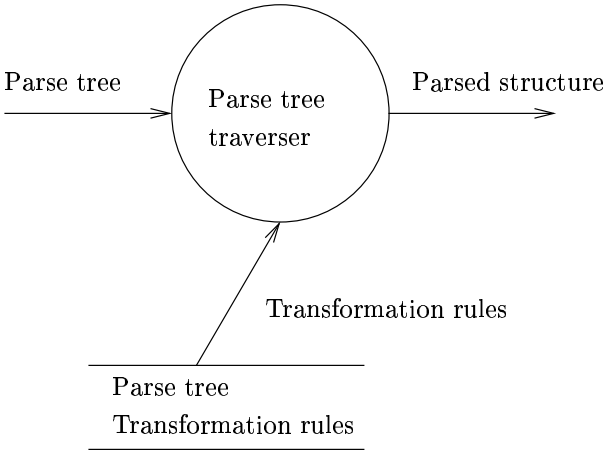

Figure 4.9: **DFD-1-1-1-3**. Level 4 Data Flow Diagram. Expands the Formatter process from DFD-1-1-1. Matching the structure of the parse tree against a set of transformation rules, the Formatter creates the Parsed structure.

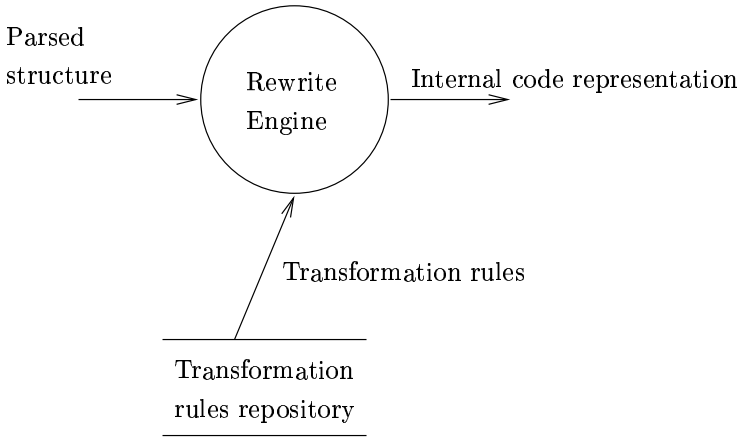

Figure 4.10: **DFD-1-1-2**. Level 3 Data Flow Diagram. Expands the SUBPROG-SPIDER process from DFD-1-1, revealing two subsystems - a set of rules and an "engine" for applying them.

pose it further. The core of the SUBPROG-SPIDER module is the Rewrite Engine. Using a set of transformation rules, the Rewrite Engine takes the Parsed structure from the Formatter and alters it. The result is an Internal code representation. Thus we now have two intermediate representations, resulting from the decomposition of the Preliminary Processing module.

So far, we have only elaborated on the actual processing within Preliminary Processing. Before designing the Kernel, we need to give more details on the intermediate representations.

The Parse tree, the Parsed structure and the Internal code representation

The Parse tree is made up of a number of nodes and edges. There is one node per grammar rule, and each node can have any number of children from zero to six. The nodes contain all the grammatical information about the SPARK code in a record. This record is required to hold:

1. *The type of the node.* This is the name of the grammar rule that the

node corresponds to, for example `expression`.

2. *The Lexeme.* The Lexeme is the actual string found in the source code.

3. *Links to children.* Clearly, the leaves have no children. Likewise, the root must have some children. The number of children for a node is determined by how many composite rules the node's grammar rule is made up of.

4. *Additional information.* Some "administrative" information is useful in the Formatter process. This data does not correspond to the source code itself but will aid the final tree traversal. Additional information includes a count of the number of children for a node, any special formatting flags (some nodes have a non-standard text format), and a count of how many times a node has been visited.

.

The actual meta-data generation is best carried out in Prolog, since Prolog is ideal for symbolic and logic processing. Therefore the Parsed structure and the Internal code representation are best represented in a form that is easily processed in Prolog.

We envisage the parsed structure as a "pretty-printed" version of the Parse tree, in the form of a Prolog term. Anticipating the implementation, SUBPROG-SPIDER will be coded in Prolog, since the Prolog DCG formalism provides a very powerful language interpretation mechanism. The Parsed structure will contain all the grammatical information encoded in the Parse tree.

The SUBPROG-SPIDER output, the Internal code representation, should be a simpler, more direct representation of the code than the Parsed structure. Because it directly reflects the grammatical structure of the SPARK source, the Parsed structure is large and unwieldy. Also, it contains much

information that is not interesting for deriving meta-data and logical properties. When we look for algorithmic patterns in a program, we do not care how identifiers and reserved words are combined to make up a grammatical construct. Rather, we want to analyse the *semantics* of the program. So the Internal code representation need be more compact than the Parsed structure while not changing the program meaning. The SUBPROG-SPIDER output is formed by several Prolog terms that are easily translatable into a graph. The terms should keep information on

- *The data types used.* Any defined data types should be present in the SUBPROG-SPIDER output, with their name and attributes (default values, range limits and so on).
- *The variables used.* All the variables that appear in the source code should be represented, preferably separate from the rest of the source code.
- *The code.* The code sequence should be encoded as a list of standard statement representations. SPARK programs consist of standardised statement types, like assignment statements, control constructs and so on. It will simplify the graph generation if the SUBPROG-SPIDER output is also standardised.

Preferably, these three information types should be kept separate from each other rather than in one monolithic term, since it is easier to deal with a number of smaller chunks of data. A more detailed description of the Internal code representation is given in appendices A and B.

4.3 The Kernel

The Kernel uses the Internal code representation and the Original SPARK code to derive meta-data and logical properties from the SPARK program.

The Kernel consists of three processes. The Graph Builder is first applied to the Internal code representation. As its name suggests, the Graph Builder creates a Graph, similar to those shown in figure 4.2. The Graph is then passed to the Graph analyser. This is the most important part of AUTOGAP. The Graph Analyser looks for algorithmic patterns in the graph representation of SPARK code. The result of the graph analysis is Meta-data and Logical properties. The Meta-data is written directly to a file for use at a later stage by SPADE-PP. The Logical properties, if any are found, are merged with the Original SPARK code to produce a new source file containing new annotations. We also append the Logical properties and the Meta-data to the new source file.

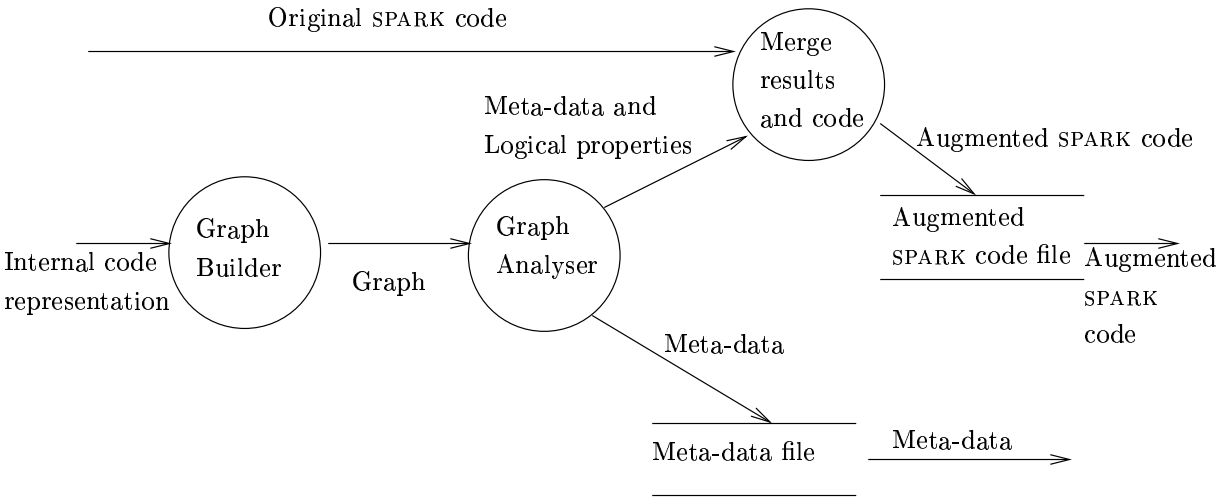

Figure 4.11: **DFD-1-2.** Level 2 Data Flow Diagram. Breaks the Kernel process from DFD-1 up into its constituent parts. A graph is built and analysed. The output is SPARK code with added annotations and a fact file.

The Graph Builder makes up the last stage of translation, taking the Internal code representation as input and turning it into a Graph. The Graph Builder's design is shown in figure 4.12.

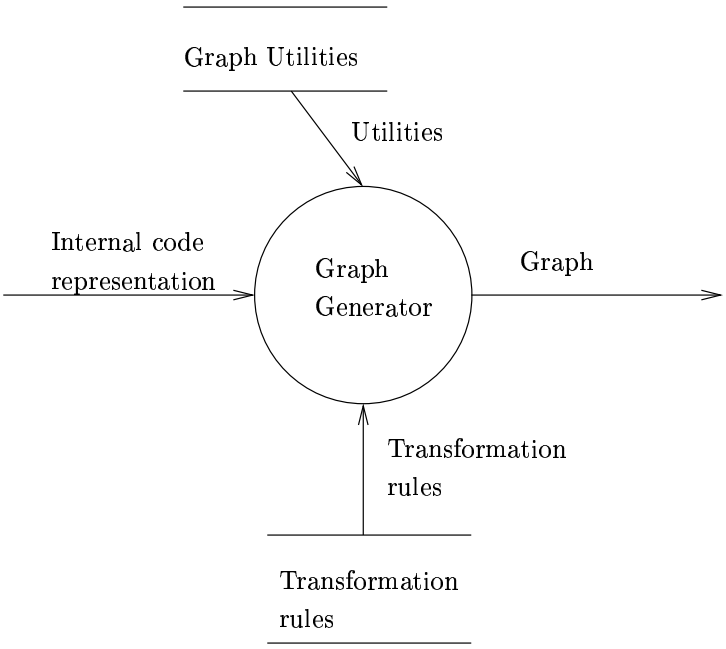

Figure 4.12: **DFD-1-2-1**. Level 3 Data Flow Diagram. The Graph Builder has a Graph Generator as its core, and uses a set of Graph building utilities and Transformation rules to make a graph of the Internal code representation.

The Graph Generator will process the Internal code representation and produce the Graph. In order to achieve this, some Transformation rules are needed, one for each possible construct that can appear in the Internal code representation. Also, some Utilities for graph building are used, in the form of some standard methods that create vertices, edges and so on.

The Graph Analyser is shown in greater detail in the level 3 DFD of figure 4.13. Its design is quite simple. Given the Graph and a library of algorithmic patterns, it will traverse the Graph looking for instances of the patterns. Although its design is simple, the operation of the Graph Analyser is complicated. In section 4.3.2 we elaborate on this. The Graph analyser will process the Graph in several "passes", each time adding to the knowledge of the previous pass. Each algorithmic pattern corresponds to a separate type of pass.

The Graph Analyser, the core of AUTOGAP, is generic. The Graph Analyser has a library of algorithmic patterns that are applied in order to generate the required meta-data. Existing patterns can be changed and new patterns added to the library, enabling the Graph Analyser to look for and generate new types of meta-data.

For any program only one graph will be created. But because the Graph Analyser has an interchangeable library of algorithmic patterns, it will be able to find many different patterns for the same program despite traversing a single graph.

Merging the results and code is the last process in AUTOGAP. This module will create a new SPARK file, containing the original code augmented with annotations. The module is made up of two distinct processes, the Add Annotations and the Append Meta-data & Logical properties. An overview is given in figure 4.14.

The Augmented SPARK code will consist of the Original SPARK code with annotations (derived from the Logical properties) in the appropriate places

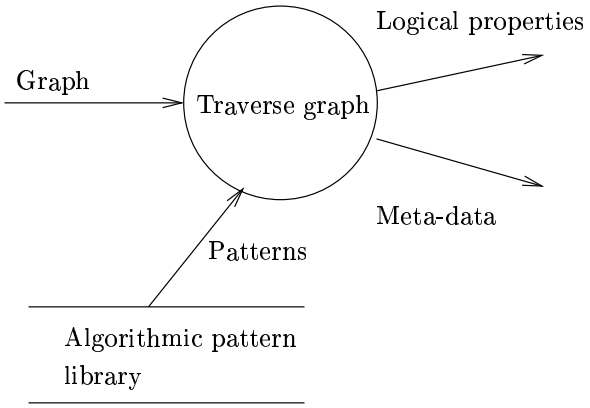

Figure 4.13: **DFD-1-2-2.** Level 3 Data Flow Diagram. Expands the Graph Analyser process from DFD-1-2. Meta-data and Logical properties are generated by traversing the graph looking for instances of algorithmic patterns.

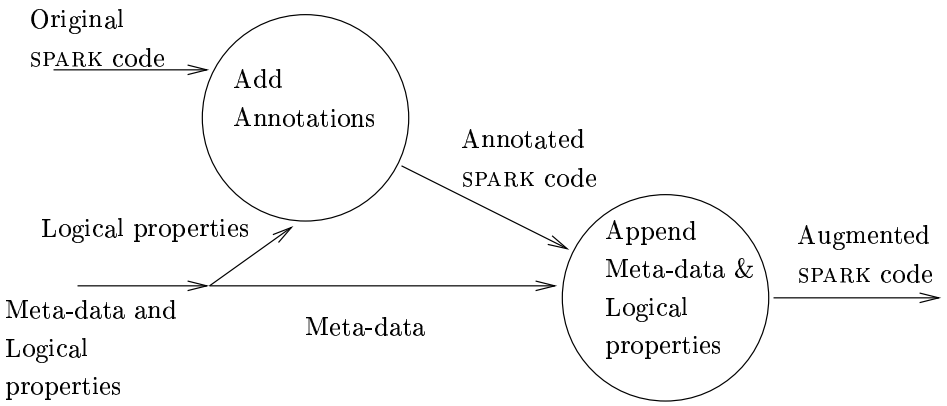

Figure 4.14: **DFD-1-2-3**. Level 3 Data Flow Diagram. Creating a new SPARK file involves merging Logical properties and Meta-data with the Original SPARK code.

and the Meta-data appended onto the end. Adding annotations is further decomposed into two processes, Create Annotations and Merge, as displayed in figure 4.15.

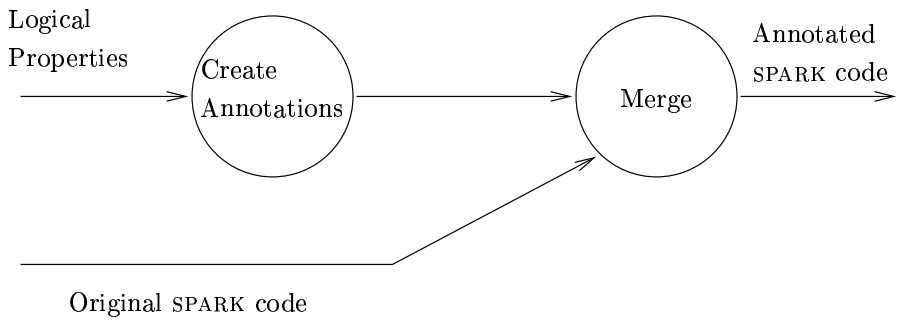

Figure 4.15: **DFD-1-2-3-1**. Level 4 Data Flow Diagram. The Annotated SPARK code is arrived at by first creating annotations from the Logical properties and then merging these annotations with the Original SPARK code.

Let us now take a closer look at the Graph

4.3.1 The Graph

While the Graph Analyser is the single most important module in AUTOGAP, the Graph is the most important data structure. If the Graph is implemented well, it will ease the task of the Graph Analyser. An important goal is that the Graph should be a flexible structure that allows prototyping of heuristics. By keeping the Graph as general as possible, the heuristics can be altered quickly to look for different algorithmic patterns.

As for the other two SPARK representations (the Parsed structure and the Internal code representation), the Graph must preserve the exact meaning of the program to be analysed. Moreover, it should also be easy to search, by containing as much supporting information as possible. Supporting information is "administrative" data on the code such as supplementary comments attached to edges and vertices. Figure 4.16 displays an example

graph for a program.

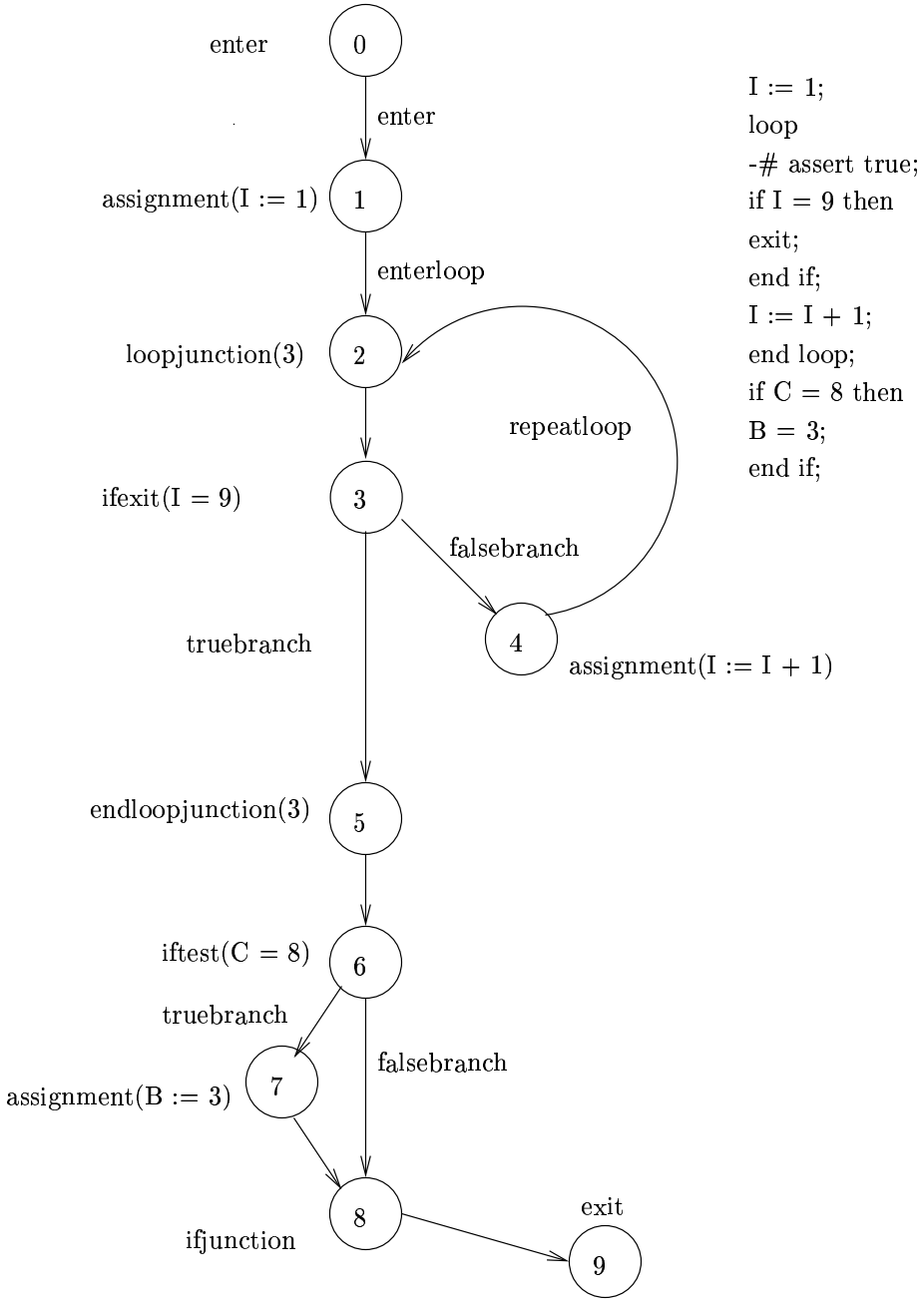

Figure 4.16: An example graph showing some standard graph constructs. Most importantly, loops and test statements are standardised.

The Graph is made up of several standardised components, built from edges and vertices. The number of components is kept as low as possible, and using these subassemblies we are able to represent any SPARK program in a uniform way.

Overall structure

The Graph is a collection of edges and vertices. There should be one entry and one exit vertex for every program, so that the start and end can be easily identified. The vertices will encode the program statements.

The vertices should be numbered. The Internal code representation keeps the sequential ordering of program statements, so it is natural to create the Graph by processing this representation from start to end, and so that the order of program statements can be encoded explicitly in the Graph. In addition, the vertices should have an associated type and program statement. Examples of vertex types in figure 4.16 are `loopjunction`, `ifexit`, `enter`, `iftest`, `endloopjunction`, `assignment`, `ifjunction` and `exit`.

Edges are somewhat simpler, and have three attributes. Firstly, each edge must have a start and end vertex. Secondly, edges can have comments attached to them. An example of a comment is the edge from vertex 3 to vertex 5, which is tagged "truebranch". Thirdly, the edges must encode direction, since we need to know which direction we are traversing the graph in.

Standard components - loops and tests

As mentioned, we would like the Graph to be built from a small number of basic subassemblies. Figure 4.16 shows that *loops* are translated into a standard format. Firstly, there is a unique start and end point for a loop. Vertex 2, labelled `loopjunction(3)` and vertex 5 (`endloopjunction(3)`)

are the start point and end point for the loop respectively. Because we can have an arbitrary number of loops inside loops, we need an easy way of discriminating between loops. We choose to identify a loop by the line number of its `--# assert true` statement in the code, as shown for vertices 2 and 5. All exit statements in a loop are translated into simple tests. If the test is true then the loop is left, otherwise the next statement in the loop body is executed. At the end of the loop body, an edge connects the vertex representing the last loop statement to the start of the loop (the edge labelled `repeatloop`).

The *tests* that make up exit statements in a loop are the same as those representing other tests such as if-then-else. For every condition there is a vertex, and out of that vertex comes a false and a true edge. Every test has a junction node which acts as a collection point for the branches of the test. After the branches have been brought together, the rest of the code that does not belong to the test statement follows.

4.3.2 The Graph Analyser and the Graph

The Graph Analyser carries out a bottom-up analysis of the SPARK program by traversing the Graph. The Graph Analyser performs several passes over the Graph, each time adding to its knowledge of the SPARK program. Here we give the pseudocode for the algorithms that encode the heuristics. Note that some of the heuristics depend upon the results of others, so the order in which the heuristics are called is important. The order in which we list the pseudocode here is the actual order in the implementation. Also note that since we are using *heuristics*, it would be possible to encode them in different ways depending upon the meaning that different persons assign to the concepts. Through prototyping we have arrived at definitions that work well in practice. Now we define some variables that are common to all the heuristics.

```
Start_Loop - the first graph node of any loop

End_Loop - the last graph node of the loop that Start_Loop starts

Loop_Id - the loop identifier for the loop bounded by Start_Loop
          and End_Loop, that is, the line in the SPARK program where
          the default loop invariant (--# assert true) appears
```

The first and simplest heuristic is `init_val`, which finds the initial values of variables.

```
for all variables V used in the program unit
    Node_1 = the first node where V is assigned;

    if V is used in a test or an assignment before Node_1 then
        exit;
    else
        Value = the value that V is assigned at Node_1;
        Save(init_value(V, Value));
```

Variables that have their values defined outside the program, for example parameters to procedures, are not taken into account by `init_val`.

`counter` finds loop counter variables. A loop counter variable is a variable that is incremented or decremented within a loop and appears in that loop's exit condition.

```
for all loop exit condition nodes Exit
    E_Loop = the final node of the loop that Exit belongs to;

    S_Loop = the first node of the loop that Exit belongs to;

    L_Id = the loop identifier for the loop bounded by
              S_Loop and E_Loop;

    for all variables V mentioned in Exit's test

    if V is incremented or decremented within the loop bounded
       by S_Loop and E_Loop then

        Save(counter(V, L_Id));
```

Next is `mono`, which identifies variables that are monotonically decreasing or increasing within a loop. Formal verification of loop code is difficult, and therefore it will be useful to identify variables that exhibit structured changes in value during execution of the loop.

```
for all variables V used in the program unit
    Node_1 = the first node where V is incremented by a positive value;

    find Start_Loop and End_Loop such that V appears within that loop;

    if V is not assigned any other values than a simple increment
       by a positive value within the loop found above then
          Save(value(V, this_iteration) is larger than
               value(V, previous_iteration)
               in loop Loop_ID);
    else
          exit;
```

The code for monotonically decreasing variables is similar, but considers variables that are decremented instead. It would be possible for a variable to be assigned via division or multiplication and still be monotonically increasing, but we have found that such code almost never occurs.

`index_var` finds index variables. An index variable is any variable that occurs as an index into an array, either in a test or in an assignment statement. The heuristic for finding array partitions processes loop code and uses `index_var`, which is why there are two versions of `index_var`. The one described below finds index variables within loops. The other one is not restricted to loops, but is so similar to the heuristic below that we omit its pseudocode.

```
for all nodes N used in the program unit that involves a test or
    an assignment

    if N is within the loop bounded by Start_Loop and End_Loop then

        Statement = the code statement associated with N
```

```
A_List = list of all arrays Array_Name in Statement;
I_List = list of all indices I for those arrays in Statement;
Result_List = Merge A_List and I_List;


for all elements on Result_List
    Save(index_var(Array_Name, I));
```

The `bounds` heuristic finds the maximum or minimum value for monotonic variables involved in simple loop exit conditions.

```
for all variables V in the program unit
    if V is monotonically increasing or decreasing within loop
         Loop_Identifier and
       V is compared to a constant value in that loop's exit condition
         using the '=' operator

         C_Val = the constant value V is compared to in the exit
                 condition;

         if V is monotonically increasing then
             Save(bound(V, C_Val, upper_bound);
         else
             Save(bound(V, C_Val, lower_bound);
```

Array partitions are found within loops. They arise when a monotonically increasing or decreasing variable is used to index an array. For monotonic index variables the array partition grows from the initial value of the index variable towards the final value of the index variable. The initial value of the index variable is used to denote one end of the partition while the variable itself denotes the other end. For a partition to exist, the index variable must also be a counter variable. The name of this heuristic is `partition`.

```
Find Start_Loop, End_Loop and Loop_Id;
Array_List = use the index_var heuristic to find all arrays Array_Name
             and their associated index variables for the loop Loop_Id;
```

```
for all index variables I on Array_List
    if I has an initial value and
       I is a counter variable for the loop Loop_Id and
       I is monotonically increasing or decreasing within that loop then

           Expand Array_List to include the initial value of I and its
           monotonic nature;
    else
           remove I from Array_List;

for all elements on Array_List
    if the index variable for the array is monotonically increasing
    then
        Save(partition(Array_Name, Lower_Bound = initial value of I,
                       Upper_Bound = I, Loop_Id))
    else
        Save(partition(Array_Name, Lower_Bound = I,
                       Upper_Bound = initial value of I, Loop_Id))
```

`invariant` automatically generates loop invariants. This is a very hard problem, so we concentrate upon finding simple invariants of the form `variable <operator> variable`. We focus on the loop exit condition, since this often gives a clue to how program variables relate to each other. It would be easy to find a large number of invariants by listing variables that do not change within the loop, but we do not consider such trivial cases. Also, we start off by looking for monotonically increasing or decreasing variables. Such variables often appear within loop invariants.

```
for all variables M that are either monotonically increasing or decreasing

    L_Id = the loop identifier for the loop within which Mono_Variable
           is monotonic;

    for all Exit nodes for the loop L_Id

        Statement = the exit condition for Exit;

        if M appears within Statement then

            Other_Var = the other variable within Statement;

            Condition =  the relationship between Mono_Variable and
```

```
                    Other_Var within Statement;

    InvarCondition = the invariant relationship between
                     Mono_Variable and Other_Var within the loop
                     L_Id ;

    Save(Invariant(InvarCondition, Loop_Id));
```

This concludes the AUTOGAP design. We have decomposed the rough picture in figure 4.1 into modules simple enough for implementation. Three main subsystems have been found, the LW-SPARK-PARSER, SUBPROG-SPIDER and the Kernel. We have described how each module works and given an outline of how each heuristic works.

Chapter 5

Implementation

We now describe how the design was implemented. Since AUTOGAP is intended for integration with NUSPADE at a later date, we write this chapter not only with description but also maintenance in mind. First we give an overview of the implementation and then we consider the LW-SPARK-PARSER, SUBPROG-SPIDER and the Kernel in turn. The final part of this chapter examines how AUTOGAP is implemented in terms of a library system which makes it extendible and how extensions can be made.

5.1 Overview

AUTOGAP is implemented in seven different languages - C, Prolog, Awk, Lex, Yacc, Sed and Unix shell scripts. In total it consists of roughly 12500 lines of code, of which Prolog makes up almost 8000. AUTOGAP is written on the Linux platform.

5.1.1 Choice of languages

LW-SPARK-PARSER is written in Lex and Yacc with C. The reason we chose this technology was firstly that it provides a mature and well tested platform for building non-trivial parsers and secondly that we already had some

experience with this combination. The Kernel is written entirely in Prolog, which was chosen because it is a logic programming language and hence ideally suited to the kind of logic analysis that AUTOGAP performs. Also, Prolog has the ability to manipulate and reason with records just as easily as with strings and numbers. This makes Prolog ideally suited for AUTOGAP. We believe it would be extremely hard to engineer AUTOGAP using imperative languages like C or Java. The scripting languages Awk, Sed and the basic Unix shell script were selected because they constitute a very powerful trio of tools for overall management of Unix software. The scripts for controlling the different subsystems of AUTOGAP are written in these languages.

5.1.2 Top level control

The three subsystems of AUTOGAP make up three separate programs and each is executed via its own command on the command line. In order to control these programs and to manage the output a shell script called `autogap` has been written. Thus the user types `autogap <SPARK source file>` in order to apply AUTOGAP to a SPARK program unit. The `autogap` script also calls further scripts written in Sed and Awk in order to create formatted files for the results. Two files are created to hold output - one called `autogresults`, which lists all logical properties and meta-data for use by NUSPADE and one with the suffix `.autogap` that is a report containing the original code with any discovered invariants in place and other results appended. Appendix C contains the main script and shows an example of running AUTOGAP.

5.2 LW-SPARK-PARSER

LW-SPARK-PARSER is implemented in Lex and Yacc and was very difficult to create because of its size and complexity, which is why we spent so much time in second term on it. The Lexer was created using Lex while the Parser consists of a Yacc grammar with actions specified in C. The grammar appears in Appendix G. The Parser builds the Parse tree, which is processed by the Formatter and turned into the Parsed structure. Praxis Critical Systems Ltd. kindly provided the entire SPARK grammar in a form that was readily translated into Yacc. The Lexer and Parser consist of mostly repetitive code, so we give attention to the Parse tree instead. The Parse tree is a C data structure that consists of nodes and links. These are the attributes of each node.

- *node_type.* This is the name of the grammar rule that the node corresponds to.
- *token_string.* If the corresponding grammar rule has an associated identifier (as is true for leaf nodes), this field contains that identifier. Otherwise it contains the empty string. The Lexer stores all identifiers in a buffer that is made available to the Parser.
- *field_1, field_2, ..., field_6.* These are pointers to the children of the current node. Any unused pointers are set to `null`.
- *need_comma.* This is a boolean flag that is set to true for nodes whose grammar rules do not originally have an associated identifier but are required to hold an identifier anyway. This flag is used by the Formatter.
- *visit_count.* This is a simple count of how many times the Formatter has looked at a node. When the value of this field equals the value of

the field that contains the children count, all children of a node have been processed and the Formatter may proceed to the next node.

- *already_visited.* This is another formatting flag, used by the Formatter when processing the nodes to indicate that no further processing of a node is necessary.

- *no_of_children.* A count of how many children a node has, depending on how many of the `field_` pointers are non-null.

- *already_parenthesized.* This boolean flag is set to true if a node that requires extra parentheses has had those parentheses printed to the output.

The Parse tree is an exact grammatical representation of the source code. The Formatter transforms the tree into a Prolog term (the Parsed structure), which is better suited to treatment by SUBPROG-SPIDER. The Formatter consists of two C routines, named `traverse` and `visit_node`, which were both notoriously hard to write correctly.

5.3 SUBPROG-SPIDER

All of SUBPROG-SPIDER was written by the eminent Bill J. Ellis, research associate in the Dependable Systems Group. He is the main implementor of NUSPADE and took on the job of creating SUBPROG-SPIDER because we realised that mutual co-operation would benefit both NUSPADE and AUTOGAP. SUBPROG-SPIDER is coded in Prolog and consists of a set of rewrite rules and an engine for applying them. The size of SUBPROG-SPIDER is just under 1700 lines, and is in addition to the 12500 we mentioned in section 5.1 (since SUBPROG-SPIDER was not implemented by us).

The result of rewriting the Parsed structure is the Internal code representation, which is a kind of pseudocode for the SPARK program unit. The

Internal code representation consists of a number of terms, the most important of which are called `sparkType` and `sparkCode`. `sparkType` gives definitions of all user defined data types while the actual pseudocode is contained within `sparkCode`. The BNF for the pseudocode language appears in Appendix B.

5.4 The Kernel

The Kernel consists of routines for creating the Graph and extracting algorithmic properties from it by heuristic analysis. The Kernel is written entirely in Prolog.

5.4.1 The Graph

The Graph is implemented as a loose collection of edges and vertices. This means that vertices and edges are created separately and "tied" together to form the Graph. The Prolog database is used to store all the information about the graph. Traversing the Graph is therefore done by searching the database looking for vertices and edges that connect them.

The two main predicates used to create the Graph are `addSequenceCode` and `addLoopCode`. As their names suggest, `addSequenceCode` creates the part of the Graph that consists of ordinary sequential code (such as assignment statements and if-then-else tests) while `addLoopCode` processes loops.

5.4.2 The heuristics

Each heuristic is kept in a separate file, apart from the loop invariants heuristic, which spans five files. This keeps the implementation modular, to allow for easy extension in the future.

The implementation closely adheres to the pseudocode given in the previous chapter, although it is obviously much more complex. The code for

the heuristics utilises the Prolog backtracking mechanism and the Prolog `findall` predicate for finding all solutions of a predicate. This allows the code to be kept as simple as possible, but at the same time sacrifices efficiency. However, since it only takes a couple of seconds to execute AUTOGAP on the test programs, this does not significantly decrease performance.

Note that the heuristics are executed in the order given in chapter 4. They all store their results in the Prolog database so that the succeeding heuristics can make use of them.

5.5 The library organisation

To adhere to the requirement that AUTOGAP should be extendible with new heuristics we have organised the implementation of the heuristics as a library of files. The library definition is given in the file `config.pl`. Extending the AUTOGAP system is done by replacing this file by a new configuration file.

There are two parts to the configuration file. The first part is a list of Prolog `include` statements, which lists all the heuristics files that are needed for compilation. The second part of the configuration file is a predicate called `installConfiguration`. The first part of this predicate is a list of lists which gives the order of the heuristics, their names and the number of arguments. There is one list element for each heuristic, which is itself a list. This is the format used to describe the heuristics:

```
[PredicateName, PredicateArity, AssertionName, AssertionArity]
```

where `PredicateName` and `PredicateArity` is the name and number of arguments for the heuristic, and `AssertionName` and `AssertionArity` is the name and number of arguments for the Prolog statement that is saved to the database as a fact when the heuristic has finished. The second part of the `installConfiguration` predicate saves the names of the assertions

and predicates to the Prolog database. The information given in `config.pl` is used to execute all the heuristics in the given order and collect their results. The code which does this is contained within the file `autogapmain.pl`. Given `config.pl` as an example configuration file it should be easy for an experienced Prolog programmer to modify AUTOGAP with new heuristics.

Chapter 6

Evaluation and results

We now show the results of the tests and then evaluate AUTOGAP against the requirements.

6.1 Results

The LW-SPARK-PARSER and SUBPROG-SPIDER was tested on a corpus of SPARK programs during implementation . AUTOGAP was then evaluated as a whole on 19 different SPARK programs. The programs we have focused on are searching and sorting routines, because such programs are often hard to prove and contain a lot of information for extraction by AUTOGAP. The following tables lists the programs and a summary of which heuristics gave results. The full report files with the source code and including results appear in Appendix F. First, we define what the shorthand notation for program names means.

Shorthand	Program
P1	Binary search
P2	Bubble sort
P3	Dutch flag problem [1]
P4	Gaussian elimination [21]
P5	Exchange sort [23]
P6	Find [12]
P7	Insertion sort
P8	Sum of integer series [23]
P9	Linear search
P10	Majority voting algorithm [1]
P11	Finding maximum value element in array
P12	Merge routine
P13	Finding minimum value element in array
P14	Pivot routine from quicksort [8]
P15	Polish flag problem, version 1 [7], [14]
P16	Polish flag problem, version 2 [2]
P17	Prime number generator [23]
P18	Selection sort
P19	Square root [23]

Table 6.1: Shorthand notation for algorithms used in testing.

We now give the results of the application of AUTOGAP to these algorithms. Note that the Roman literals denote a particular heuristic, as follows:

I Initial values

II Counter variables

III Monotonically increasing variables

IV Monotonically decreasing variables

V Bounds

VI Index variables

VII Array partitions

VIII Loop invariants

[1] See http://www.csse.monash.edu/~lloyd/tildeAlgDS/Sort/Flag/ and [1].

[2] Two solutions to the Polish flag problem have been used for testing.

Program	Heuristic								Logical property	Meta-data
	I	II	III	IV	V	VI	VII	VIII		
P1	*	*								M1 - M8
P2	*	*	*			*	*			M9 - M17
P3	*	*	*	*		*	*	*	L1	M18 - M31
P4	*	*	*	*	*	*		*	L2 - L3	M32 - M53
P5	*	*	*	*	*	*	*	*	L4 - L5	M54 - M69
P6	*	*	*	*		*				M70 - M84
P7	*	*	*	*	*	*	*	*	L6 - L7	M85 - M95
P8	*	*	*							M95 - M100
P9	*	*	*		*	*	*	*	L8	M101 - M106
P10	*	*	*		*	*	*	*	L9	M107 - M115
P11	*	*	*			*	*			M116 - M121
P12	*	*	*	*		*				M122 - M142
P13	*	*	*			*	*			M143 - M148
P14	*	*	*			*	*			M149 - M158
P15	*	*	*	*		*	*	*	L10	M159 - M169
P16	*	*	*	*		*	*	*	L11	M170 - M180
P17	*	*	*		*	*		*	L12	M181 - M191
P18	*	*	*			*	*			M192 - M204
P19	*	*	*					*	L13	M205 - M214

Table 6.2: Test results showing which heuristics triggered for each test program and the properties that were found.

So, for example, a '*' character in column V means that the Bounds heuristic gave a result for the corresponding program. The logical properties and the meta-data found for each program are listed in tables 6.3 - 6.10. In Appendix E there is an explanation of the syntax used for the meta-data and logical properties.

We note that since all the test algorithms contain loops, counter variables are found for all of them. Also, initial values are found for all the programs, and monotonically increasing variables are discovered in all but one program.

The Bounds heuristic only gives results for variables whose bounds are fixed to constant values. Giving constant bounds for variables is quite uncommon, so this heuristic does not give results very often.

Time did not permit us to develop Loop invariant heuristics as advanced as we hoped, so invariants are only discovered for ten of the programs. These are the invariants that AUTOGAP discovered:

Logical properties	
L1	`--# assert R<=M;`
L2	`--# assert I<=100;`
L3	`--# assert J<=100;`
L4	`--# assert Idx>=2;`
L5	`--# assert I<=100;`
L6	`--# assert I<=100;`
L7	`--# assert J>=1;`
L8	`--# assert Ans<=100;`
L9	`--# assert K<=100;`
L10	`--# assert I<=J;`
L11	`--# assert M<=W;`
L12	`--# assert Candidate<=1999;`
L13	`--# assert I<=N_Max;`

Table 6.3: The loop invariants (logical properties) discovered by AUTOGAP.

The meta-data facts that were discovered now follow in tables 6.4 - 6.10.

Meta-data	
M1	`initial_val('Left', integer(1))`
M2	`initial_val('Right', integer(100))`
M3	`initial_val('Placement', -(integer(1)))`
M4	`initial_val('Middle',(variable('Left') +variable('Right'))/integer(2))`
M5	`initial_val('Found', variable('False'))`
M6	`count_var('Right', 20)`
M7	`count_var('Left', 20)`
M8	`count_var('Found', 20)`
M9	`initial_val('I', integer(1))`
M10	`initial_val('Temp',element(variable('Sort_Me'), variable('J')-integer(1)))`
M11	`initial_val('J', integer(1))`
M12	`count_var('I', 13)`
M13	`count_var('J', 20)`
M14	`mono_inc(value('I',now), greaterthan, value('I',previous), line(13))`
M15	`mono_inc(value('J',now), greaterthan, value('J',previous), line(20))`
M16	`index_var('J', 'Sort_Me')`
M17	`array_partition('Sort_Me', integer(1), 'J', 20)`
M18	`initial_val('R', integer(1))`
M19	`initial_val('M', integer(100))`
M20	`initial_val('W', integer(100))`
M21	`initial_val('Temp', element(variable('Flag'), variable('M')-integer(1)))`
M22	`count_var('R', 21)`
M23	`count_var('M', 21)`
M24	`mono_inc(value('R',now), greaterthan, value('R',previous), line(21))`
M25	`mono_dec(value('M',now), lessthan, value('M',previous), line(21))`
M26	`mono_dec(value('W',now), lessthan, value('W',previous), line(21))`
M27	`index_var('M', 'Flag')`
M28	`index_var('R', 'Flag')`

Table 6.4: Meta-data derived by AUTOGAP.

Meta-data	
M29	index_var('W', 'Flag')
M30	array_partition('Flag', 'M', integer(100), 21)
M31	array_partition('Flag', integer(1), 'R', 21)
M32	initial_val('I', integer(1))
M33	initial_val('Size_1', integer(101))
M34	initial_val('Max', variable('I'))
M35	initial_val('Temp', element(variable('Equation_Set'), variable('I')))
M36	initial_val('Start_1', variable('I')+integer(1))
M37	initial_val('J', variable('Start_1'))
M38	initial_val('K', variable('I'))
M39	count_var('I', 20)
M40	count_var('J', 29)
M41	count_var('K', 43)
M42	count_var('J', 55)
M43	count_var('K', 60)
M44	mono_inc(value('I',now), greaterthan, value('I',previous), line(20))
M45	mono_inc(value('J',now), greaterthan, value('J',previous), line(29))
M46	mono_inc(value('J',now), greaterthan, value('J',previous), line(55))
M47	mono_inc(value('K',now), greaterthan, value('K',previous), line(43))
M48	mono_dec(value('K',now), lessthan, value('K',previous), line(60))
M49	bound('I', 100, upper)
M50	bound('J', 100, upper)
M51	index_var('J', 'Equation_Set')
M52	index_var('Max', 'Equation_Set')
M53	index_var('I', 'Equation_Set')
M54	initial_val('I', integer(1))
M55	initial_val('Idx', integer(101))
M56	initial_val('Temp', element(variable('Arr'), variable('Position')))
M57	initial_val('Max', element(variable('Arr'),integer(1)))
M58	initial_val('Position', integer(1))

Table 6.5: Meta-data derived by AUTOGAP.

Meta-data	
M59	count_var('I', 32)
M60	count_var('Idx', 22)
M61	mono_inc(value('I',now), greaterthan, value('I',previous), line(32))
M62	mono_dec(value('Idx',now), lessthan, value('Idx',previous), line(22))
M63	bound('I', 100, upper)
M64	bound('Idx', 2, lower)
M65	index_var('I', 'Arr')
M66	index_var('Position', 'Arr')
M67	index_var('Idx', 'Arr')
M68	array_partition('Arr', integer(1), 'I', 32)
M69	array_partition('Arr', 'Idx', integer(101), 22)
M70	initial_val('M', integer(1))
M71	initial_val('N', integer(100))
M72	initial_val('R', element(variable('Arr'),variable('F')))
M73	initial_val('I', variable('M'))
M74	initial_val('J', variable('N'))
M75	initial_val('Temp', element(variable('Arr'), variable('I')))
M76	count_var('M', 41)
M77	count_var('N', 41)
M78	count_var('I', 55)
M79	count_var('J', 55)
M80	mono_inc(value('I',now), greaterthan, value('I',previous), line(63))
M81	mono_inc(value('I',now), greaterthan, value('I',previous), line(72))
M82	mono_dec(value('J',now), lessthan, value('J',previous), line(72))
M83	index_var('I', 'Arr')
M84	index_var('J', 'Arr')
M85	initial_val('I', integer(1))
M86	initial_val('Idx', element(variable('To_Sort'), variable('I')))
M87	initial_val('J', variable('I'))
M88	count_var('I', 18)
M89	count_var('J', 25)
M90	mono_inc(value('I',now), greaterthan, value('I',previous), line(18))

Table 6.6: Meta-data derived by AUTOGAP.

Meta-data	
M91	mono_dec(value('J',now), lessthan, value('J',previous), line(25))
M92	bound('I', 100, upper)
M93	bound('J', 1, lower)
M94	index_var('J', 'To_Sort')
M95	array_partition('To_Sort', 'J', variable('I'), 25)
M96	initial_val('No_Terms', variable('Terms'))
M97	initial_val('Sum', integer(0))
M98	initial_val('Terms', integer(1))
M99	count_var('Sum', 21)
M100	mono_inc(value('Terms',now), greaterthan, value('Terms',previous), line(21))
M101	initial_val('Ans', integer(1))
M102	count_var('Ans', 18)
M103	mono_inc(value('Ans',now), greaterthan, value('Ans',previous), line(18))
M104	bound('Ans', 100, upper)
M105	index_var('Ans', 'A')
M106	array_partition('A', integer(1), 'Ans', 18)
M107	initial_val('K', integer(1))
M108	initial_val('E', integer(1))
M109	initial_val('Winner', element(variable('All_Votes'), tick('Index','First')))
M110	count_var('K', 31)
M111	mono_inc(value('K',now), greaterthan, value('K',previous), line(31))
M112	mono_inc(value('E',now), greaterthan, value('E',previous), line(31))
M113	bound('K', 100, upper)
M114	index_var('K', 'All_Votes')
M115	array_partition('All_Votes', integer(1), 'K', 31)
M116	initial_val('I', integer(1))
M117	initial_val('Max_Value', element(variable('Arr'), tick('Index','First')))
M118	count_var('I', 19)
M119	mono_inc(value('I',now), greaterthan, value('I',previous), line(19))
M120	index_var('I', 'Arr')

Table 6.7: Meta-data derived by AUTOGAP.

Meta-data	
M121	`array_partition('Arr', integer(1), 'I', 19)`
M122	`initial_val('Left_End', variable('Middle')-integer(1))`
M123	`initial_val('No_Elements', variable('Right')-` `variable('Left')+integer(1))`
M124	`initial_val('Tmp_Pos', variable('Left'))`
M125	`initial_val('I', integer(1))`
M126	`count_var('Left', 41)`
M127	`count_var('Middle', 41)`
M128	`count_var('Left', 65)`
M129	`count_var('Middle', 79)`
M130	`count_var('I', 94)`
M131	`mono_inc(value('Left',now), greaterthan,` `value('Left',previous), line(41))`
M132	`mono_inc(value('Left',now), greaterthan,` `value('Left',previous), line(65))`
M133	`mono_inc(value('Middle',now), greaterthan,` `value('Middle',previous), line(41))`
M134	`mono_inc(value('Middle',now), greaterthan,` `value('Middle',previous), line(79))`
M135	`mono_inc(value('Tmp_Pos',now), greaterthan,` `value('Tmp_Pos',previous), line(41))`
M136	`mono_inc(value('Tmp_Pos',now), greaterthan,` `value('Tmp_Pos',previous), line(65))`
M137	`mono_inc(value('Tmp_Pos',now), greaterthan,` `value('Tmp_Pos',previous), line(79))`
M138	`mono_inc(value('I',now), greaterthan,` `value('I',previous), line(94))`
M139	`mono_dec(value('Right',now), lessthan,` `value('Right',previous), line(94))`
M140	`index_var('Left', 'To_Sort')`
M141	`index_var('Middle', 'To_Sort')`
M142	`index_var('Tmp_Pos', 'Temp')`
M143	`index_var('Right', 'To_Sort')`
M142	`index_var('Right', 'Temp')`
M143	`initial_val('I', integer(1))`
M144	`initial_val('Min_Value', element(variable('Arr'),` `tick('Index','First')))`
M145	`count_var('I', 18)`
M146	`mono_inc(value('I',now), greaterthan,` `value('I',previous), line(18))`
M147	`index_var('I', 'Arr')`

Table 6.8: Meta-data derived by AUTOGAP.

Meta-data	
M148	array_partition('Arr', integer(1), 'I', 18)
M149	initial_val('Left', integer(1))
M150	initial_val('Right', integer(100))
M151	initial_val('Pivot_El', integer(0))
M152	initial_val('First_Key', element(variable('Arr'), variable('Left')))
M153	initial_val('Start_Left', variable('Left')+integer(1))
M154	initial_val('Up', variable('Start_Left'))
M155	count_var('Up', 28)
M156	mono_inc(value('Up',now), greaterthan, value('Up',previous), line(28))
M157	index_var('Up', 'Arr')
M158	array_partition('Arr', variable('Start_Left'), 'Up', 28)
M159	initial_val('I', integer(1))
M160	initial_val('J', integer(101))
M161	initial_val('T', element(variable('Flag'), variable('I')))
M162	count_var('I', 25)
M163	count_var('J', 25)
M164	mono_inc(value('I',now), greaterthan, value('I',previous), line(25))
M165	mono_dec(value('J',now), lessthan, value('J',previous), line(25))
M166	index_var('I', 'Flag')
M167	index_var('J', 'Flag')
M168	array_partition('Flag', integer(1), 'I', 25)
M169	array_partition('Flag', 'J', integer(101), 25)
M170	initial_val('M', integer(1))
M171	initial_val('W', integer(100))
M172	initial_val('Temp', element(variable('Flag'), variable('M')))
M173	count_var('M', 32)
M174	count_var('W', 32)
M175	mono_inc(value('M',now), greaterthan, value('M',previous), line(32))
M176	mono_dec(value('W',now), lessthan, value('W',previous), line(32))
M177	index_var('M', 'Flag')
M178	index_var('W', 'Flag')
M179	array_partition('Flag', integer(1), 'M', 32)
M180	array_partition('Flag', 'W', integer(100), 32)

Table 6.9: Meta-data derived by AUTOGAP.

Meta-data	
M181	initial_val('Size', integer(3))
M182	initial_val('Divisor', integer(1))
M183	initial_val('Candidate', integer(3))
M184	initial_val('Prime', variable('True'))
M185	count_var('Divisor', 35)
M186	count_var('Candidate', 25)
M187	mono_inc(value('Size',now), greaterthan, value('Size',previous), line(25))
M188	mono_inc(value('Divisor',now), greaterthan, value('Divisor',previous), line(35))
M189	mono_inc(value('Candidate',now), greaterthan, value('Candidate',previous), line(25))
M190	bound('Candidate', 1999, upper)
M191	index_var('Size', 'Arr')
M192	initial_val('I', integer(1))
M193	initial_val('J', integer(2))
M194	initial_val('Min', variable('I'))
M195	initial_val('Temp', element(variable('Sort_Array'), variable('I')))
M196	count_var('I', 22)
M197	count_var('J', 32)
M198	mono_inc(value('I',now), greaterthan, value('I',previous), line(22))
M199	mono_inc(value('J',now), greaterthan, value('J',previous), line(32))
M200	index_var('J', 'Sort_Array')
M201	index_var('Min', 'Sort_Array')
M202	index_var('I', 'Sort_Array')
M203	array_partition('Sort_Array', integer(2), 'J', 32)
M204	array_partition('Sort_Array', integer(1), 'I', 22)
M205	initial_val('Sqrt', variable('X0'))
M206	initial_val('Eps', integer(1.0E-06))
M207	initial_val('N_Max', integer(9))
M208	initial_val('X0', variable('N')/integer(2.0))
M209	initial_val('X_Old', variable('X0'))
M210	initial_val('I', integer(1))
M211	count_var('I', 36)
M212	count_var('X0', 36)
M213	count_var('X_Old', 36)
M214	mono_inc(value('I',now), greaterthan, value('I',previous), line(36))

Table 6.10: Meta-data derived by AUTOGAP.

6.2 Evaluation against requirements

The AUTOGAP system has been implemented and works according to the requirements we set out. The system analyses a useful subset of SPARK source code in a bottom-up fashion and generates algorithmic properties. AUTOGAP has been implemented in co-operation with the implementors of NUSPADE in order to ensure that it can be included within NUSPADE in the future. The results from AUTOGAP are made available to NUSPADE as a single file of algorithmic properties, using a syntax that has been agreed with the NUSPADE developers. Also, AUTOGAP creates a report file that displays the time of execution and the original source code with any discovered loop invariants in place, together with the algorithmic properties appended onto the end. AUTOGAP is extendible, so that new heuristics can be added in the future. Let us list the contributions AUTOGAP makes to the NUSPADE project.

- *Provides meta-data for constraining proof search.* The meta-data that AUTOGAP produces can be used by the NUSPADE proof planner (SPADE-PP) to constrain the search for proofs. This will reduce the time taken to arrive at a proof. Also, the additional information that the meta-data gives can provide additional clues for how proof can be carried out, as compared to NUSPADE using VCs only. The role of such meta-data in the discovery of loop invariants was illustrated in section 2.2.

- *Automatically generates loop invariants from unannotated code.* The report file generated by AUTOGAP has all discovered invariants embedded within the code. Also, all other results are appended onto the code as comments. The user is therefore given instant feedback on his code.

- *Enhances the* SPARK *VC-generation.* By embedding loop invariants within the code, improved VCs can be produced for use by NUSPADE.

This increases the chance of a successful proof without user intervention.

- *Helps* NUSPADE *achieve a higher degree of automation.* The above three properties aid NUSPADE so that less user interaction is needed to carry out proofs.

- *Represents a novel approach to analysing programs in support of proof.* The heuristic approach that AUTOGAP takes is uncommon. As a prototype for analysing code in addition to conventional VC-generation, the system provides some answers for how this should be carried out. This is further investigated in the discussion in the next chapter.

- *Provides a vehicle for further prototyping of heuristics.* AUTOGAP is a prototype system and does not represent a "final" solution to the problem of extracting information from source code. It has been engineered by means of a library system, which allows the NUSPADE developers to add and remove heuristics easily.

Chapter 7

Conclusion

Let us summarise the achievements and discuss what could be done to improve AUTOGAP.

7.1 Summary of contribution

- AUTOGAP- a static analyser for SPARK code has been designed, prototyped and evaluated. It consists of approximately 12500 lines of code.
- A number of heuristics have been developed and tested using AUTOGAP.
- As a framework, AUTOGAP is flexible and extendible. Now that the framework is complete more heuristics (algorithmic patterns) and their associated properties can be investigated.
- AUTOGAP has been judged ready for testing with the NUSPADE system in the near future (once the NUSPADE proof planner is ready).

7.2 Limitations and suggestions for future work

While the requirements of AUTOGAP have been met, AUTOGAP is not a perfect system. AUTOGAP is a prototype, a first try at generating logical properties and meta-data using algorithmic patterns.

There are many more heuristics that could be useful for proof. For example, we have not considered the values of individual array elements in any of the heuristics we have implemented. In [17] Katz and Manna emphasise such analysis because it can yield advanced loop invariants. Due to the complexity and time it would take to implement such heuristics, they have not been considered in AUTOGAP. We would recommend that such heuristics are given due attention in any future development of meta-data and logical properties generation.

Also, the loop invariant heuristics of AUTOGAP generate relatively basic invariants, and do not discover invariants for all the test programs. For example, consider the bubble sort test program, which appears as the second program in Appendix F. The exit conditions for the loops are $I > Size$ and $J > I$. Because of the way our heuristics are formulated, no invariants are discovered. If given more time, we would develop more heuristics for invariant generation that look more closely at the relationship between variables, for example by comparing different exit conditions in nested loops and deciding which counter variables relate. Now that the framework is in place, extending AUTOGAP with new heuristics requires significantly less effort than we have put into it so far.

We have not attempted to use any advanced mathematical reasoning, but rather applied common sense and intuition in the AUTOGAP heuristics. Invariant generation is an extraordinary hard problem - "it is very difficult to look at an existing program and guess what ... assertions should be" [1].

[1] C. A. R. Hoare in "Assertions: a personal perspective" (unpublished, but available at http://www.research.microsoft.com/~thoare/6Jun_assertions_personal_perspective.htm

While [17], our main inspiration, gives some techniques that we have not used, others have suggested different approaches. A lot of research has been conducted on the automatic discovery of invariants, and [15], [22] and [9] give techniques alternative to those we have used. Loop invariant discovery is important since invariants are crucial for proof of non-trivial programs, and so any future work on heuristics for code analysis should take this into account, preferably in combination with the type of array heuristics mentioned above.

The use of annotations in code is not commonly accepted among programmers, except within the high end of safety-critical systems development. Indeed, [18] regards program annotations as a major bottleneck in development. Most current static analysers expect the programmer to annotate the code, and then the analysers carry out a top-down analysis. An advantage of the methods used in AUTOGAP is that annotations for loop invariants can be generated automatically, thus liberating the programmer of some of the annotation burden. However, it is probably not feasible to generate all necessary annotations automatically, even if a much more powerful static analyser is developed - "the ultimate goal of automatic assertion generation is almost certainly unattainable" [17]. The best way forward is to have users interact with the machine. For example, a more intelligent analyser could ask the user detailed questions when it got stuck. Also, if programmers give part of the assertions this would shorten the process.

To fully exploit the potential of bottom-up analysis, the analyser should be tightly integrated with the theorem prover or proof planner. Ideally, the verification package should invoke the analyser with requests for specific information when proofs are failing due to lack of data on the program. This would focus the analyser's effort and give clues to the sort of questions that the analyser might ask the user. Obviously, this means that a common set of heuristics must be used within the analyser and the rest of the

verification package. Also, the full power of bottom-up code analysis can only be exploited if combined with top-down methods for VC-generation. Proof is based upon translating programs into VCs, so we cannot do without top-down analysis. But by introducing bottom-up methods we can gain additional leverage and make more proofs go through automatically.

With respect to the AUTOGAP implementation, it would be useful if we used a more user-friendly format for the appended results in the augmented code file. At the moment, these results are displayed in the same format as they are in the file for use by NUSPADE. While the current form is certainly understandable for experienced programmers, it could be improved. Also, we left out some important constructs from the subset of SPARK. It would be good if the parser handled more of SPARK, especially while- and for-loops.

7.3 Summary and conclusion

AUTOGAP has been successfully implemented as a prototype system for bottom-up analysis of code. AUTOGAP provides a crucial component in testing the hypothesis that *Knowledge of the algorithmic patterns that occur within a program can significantly increase automation of program proof.* AUTOGAP will now be used in conjunction with NUSPADE. We have shown that it is possible to use bottom-up heuristic methods to derive properties of source code. Two kinds of properties were investigated. Firstly, conventional program properties that are essential to proof construction. Secondly, meta-data on programs that provides useful constraints for a theorem prover when searching for a proof. AUTOGAP is extendible and provides possibilities for further experiments with heuristics. Used in context with conventional top-down VC-generation methods, the bottom-up approach helps increase the automation of program proof.

7.4 Postscript

On a more personal basis, it is safe to say that the development of AUTOGAP has been a huge challenge. In creating AUTOGAP we confronted a very interesting combination of theoretical and practical problems. It took a long time to understand the underlying principles, but once the theory was established, implementation proceeded at great speed.

As mentioned, AUTOGAP is implemented in seven different languages. This shows how it ties in with much of the work done in the degree course. The most useful courses seen from this perspective has been the work on compilers in third year, scripting languages and Prolog in third year and all the programming courses.

The author is much indebted to the supervisor and the rest of the Dependable Systems Group for the chance to get an insight into some of the hardest research problems in computer science.

Bibliography

[1] Roland C. Backhouse. *Program Construction and Verification.* Prentice-Hall International Series in Computer Science. Prentice-Hall, 1986.

[2] J. G. P. Barnes. *High Integrity Ada The SPARK Approach.* Addison-Wesley, 1997.

[3] Jean-Francois Bergeretti and Bernard A. Carré. Information-Flow and Data-Flow Analysis of while-Programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37–61, January 1985.

[4] Alan Bundy. Proof Planning. In B. Drabble, editor, *Proceedings of the 3rd International Conference on AI Planning Systems*, pages 261–267, 1996.

[5] Alan Bundy. A Survey of Automated Deduction. In Michael J. Wooldridge and Manuela Veloso, editors, *Artificial Intelligence Today. Recent Trends and Developments*, volume 1600 of *Lecture Notes in Computer Science*, pages 153–174. Springer, 1999.

[6] Edmund M. Clarke and Jeanette M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28, 1996.

[7] Edsger W. Djikstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[8] Michael B. Feldman. *Data Structures With Ada.* Prentice-Hall, 1985.

[9] David Gries. *The Science of Programming.* Springer-Verlag, 1981.

[10] David Harel. *Algorithmics The Spirit of Computing.* Addison-Wesley, 2nd edition, 1992.

[11] Michael G. Hinchey and Jonathan P. Bowen, editors. *Applications of Formal Methods.* Prentice Hall, 1995.

[12] C. A. R. Hoare. Proof of a Program: FIND. *Communications of the ACM*, 14(1):39–45, January 1971.

[13] Andrew Ireland. The Use of Planning Critics in Mechanizing Inductive Proofs. In *Logic Programming and Automated Reasoning*, pages 178–189, 1992.

[14] Andrew Ireland. The Polish Flag Problem. In *Clam-Spark (CS) Notes*. 2002. http://www.cee.hw.ac.uk/~air/clamspark/project-pages/cs-notes/.

[15] J. X. Su, D. L. Dill, and C. W. Barett. Automatic generation of invariants in processor verification. In M. Srivas and A. Camilleri, editors, *First international conference on formal methods in computer-aided design*, volume 1166, pages 377–388, Palo Alto, CA, USA, 1996. Springer Verlag.

[16] Jonathan Jacky. *The Way of Z Practical Programming with Formal Methods*. Cambridge University Press, March 1997.

[17] Shmuel M. Katz and Zohar Manna. A Heuristic Approach to Program Verification. In Nils J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pages 500–512, Stanford, CA, August 1973. William Kaufmann.

[18] K. Rustan M. Leino. Extended Static Checking: A Ten-Year Perspective. *Lecture Notes in Computer Science*, 2000:157–175, January 2001.

[19] The UK Ministry of Defence. *The procurement of safety critical software in defence equipment (Requirements), (Interim Defence Standard 00-55*. apr 1991.

[20] Michael J. Pont. *Software Engineering with C++ and CASE Tools*. Addison-Wesley, 1996.

[21] Robert Sedgewick. *Algorithms in C++*. Addison-Wesley, 1992.

[22] Jamie Stark and Andrew Ireland. Invariant Discovery via Failed Proof Attempts. *Lecture Notes in Computer Science*, 1559:271–288, 1999.

[23] Richard Wiener and Richard Sincovec. *Programming in ADA*. John Wiley & Sons, 1983.

Appendix A

Internal code representation

The Parsed structure is very complicated and mundane to read, so instead we give an example of the Internal code representation. This is the output from SUBPROG-SPIDER for the minimum value source code (minvalue.adb).

```
sparkVariable(variable('Arr'), 'Int_Array', parameter(in)).

sparkVariable(variable('Min_Value'), 'Integer', parameter(out)).

sparkVariable(variable('I'), 'Integer', local(uninitilised)).


sparkType('Index', fromto(integer(1),integer(100))).


programName('Min').

sparkCode('Min',
          useVariable([variable('Arr'),
                       variable('Min_Value'),
                       variable('I')]),
          codeSequence([becomes(variable('Min_Value'),
                                element(variable('Arr'),
                                tick('Index','First'))),
                       becomes(variable('I'),tick('Index','First')),
                       loop(codeSequence([
                              invariant(18),
                              ifexit(variable('I')>tick('Index','Last'),
                                 codeSequence([])),
                              if(<=(element(variable('Arr'),variable('I')),
                                  variable('Min_Value')),
                                codeSequence(
                                    [becomes(variable('Min_Value'),
                                      element(variable('Arr'),variable('I')))]),
                                 codeSequence([])),becomes(variable('I'),
                                          variable('I')+integer(1))])])]).
```

Appendix B

Internal code representation BNF

The Internal code representation language is surprisingly small despite the significant subset of SPARK that we are working with. Note that `Then_codeSequence` and `Else_codeSequence` are only present to increase readability.

```
codeSequence ::= codeStatement | codeStatement codeSequence | EMPTY

loopCodeSequence ::= Invariant codeSequence

codeStatement ::= AssignmentStatement | ReturnStatement | IfStatement |
                  LoopStatement | InvariantStatement | IfexitStatement

AssignmentStatement ::= becomes(variable, codeExpression)

ReturnStatement ::= return(variable, codeExpression)


IfStatement ::= if(booleanCondition,
                   Then_codeSequence,
                   Else_codeSequence)

Then_codeSequence ::= codeSequence
Else_codeSequence ::= codeSequence

Loop_statement ::= loop(Loop_codeSequence)

Invariant ::= invariant(integer)

Ifexit_Statement ::= ifexit(booleanCondition, Then_codeSequence)

booleanCondition ::= Identifier comparisonOperator Identifier

codeExpression ::= codeExpression + Term | codeExpression - Term
```

```
Term ::= Term * Factor | Term/Factor | Factor

Factor ::= (codeExpression) | Identifier

Identifier ::= variable | Number

Number ::= integer | floatingpointnumber

comparisonOperator ::= > | = | < | >= | <=
```

Appendix C

Running AUTOGAP

In order to use AUTOGAP, just type `autogap <sparkfilename>` at the command line:

```
bash-2.05b$ autogap binarysearch.adb
------------------------------------------------
Spider----------------------------------------
------------------------------------------------
Input file:standard.parsed
Output file:parsed.spider
------------------------------------------------
Installing predicates...done
Finished graph construction
Running heuristics... Found all solutions for initVal(_30581,_30574)
Found all solutions for countVar(_30782,_30775)
Found all solutions for monoInc(_30977,_30970)
Found all solutions for monoDec(_31168,_31161)
Found all solutions for bounds(_31359,_31352)
Found all solutions for indexVar
Found all solutions for partition(_31716,_31709,_31702,_31695)
Found all solutions for invariant(_31917,_31910)
done
Saving meta-data to file...done
bash-2.05b$
```

Figure C.1: Running AUTOGAP on "binarysearch.adb".

This is the main AUTOGAP script, which is invoked by the `autogap` command:

```
#!/bin/bash
# File "autogap" - the main executable for autogap.
# Author: Tommy Ingulfsen, May 2003.

# Assign the name of the input file to IN.
IN=$1

# Cut off the part before .adb and assign it to
# PREDOT, then use this to create the output file to
# be named after the input file.
PREDOT=`echo $IN | sed -e 's/^\(.*\)\..*$/\1/g'`

# Parse the input.
parser2 $1 > standard.parsed

# Apply the Spider to the parsed result.
spider.exe standard.parsed parsed.spider

# Run Autogap (results stored in autogapswapped and in
# autogresults).
autogap.exe

# Create header for the output file.
echo "    AUTOGAP OUTPUT          " > autogap.output
date >> autogap.output
cat autogap.output | sed s/^/--\ /g > $PREDOT.autogap
echo " " >> $PREDOT.autogap
echo " " >> $PREDOT.autogap

# autogap.invariant contains the invariant that is created
# (in legal SPARK), so substitute that invariant into the original
# code.
callsubstitute autogap.invariant $1

# If Autogap has found any invariants the result of the above call
# for substitution is contained within file autogapswapped. So
# write that new source code to file and append all meta-data and
# program properties as comments. Otherwise just do the append
# (no invariants found).
if [ -f autogapswapped ]
then
cat autogapswapped >> $PREDOT.autogap
echo " " >> $PREDOT.autogap
echo " " >> $PREDOT.autogap
echo "--  ALGORITHMIC PROPERTIES               " >> $PREDOT.autogap
echo " " >> $PREDOT.autogap
cat autogresults | sed s/^/--\ /g >> $PREDOT.autogap
```

```
# Remove the temporary file for inserting invariants, otherwise this
# will be used again when no invariants are found and insert the
# previous invariant in a totally wrong place.
rm autogapswapped
else
cat $1 >> $PREDOT.autogap
echo " " >> $PREDOT.autogap
echo " " >> $PREDOT.autogap
echo "--  ALGORITHMIC PROPERTIES               " >> $PREDOT.autogap
echo " " >> $PREDOT.autogap
cat autogresults | sed s/^/--\ /g >> $PREDOT.autogap
fi

# Remove temporary file
rm autogap.output
```

Appendix D

Requirements

This appendix contains a formal listing of each AUTOGAP requirement.

D.1 Explanation

Requirement #:

Each requirement is numbered to enable easy cross-referencing.

Requirement type:

Functional or non-functional.

Description:

A succinct description of the requirement.

Rationale:

A justification of the requirement.

Fit criterion:

An objective measure of the requirement, so that it is possible to test whether the solution successfully implements the requirement.

Dependencies:

A list of other requirements that depend upon this one.

D.2 Requirements listing

Requirement #1
Requirement type: Non-functional.
Description: The system should be compatible with the software of the NUSPADE project.
Rationale: The system should aid and complement the NUSPADE software.
Fit criterion: Since the NUSPADE software is not yet finished, and will not be finished before this dissertation project is over, it is not possible to evaluate this requirement fully. However, by co-operating with the developers of the NUSPADE software it should be possible to achieve such compatibility.
Dependencies: None.

Requirement #2
Requirement type: Functional
Description: The system should be able to parse a subset of SPARK programs.
Rationale: In order to extract information from the SPARK code, we need an internal representation of the code. To achieve this we must parse it. AUTOGAP should concentrate on parsing a useful subset of the SPARK language.
Fit criterion: After applying the Examiner, the system written in this dissertation should be able to parse SPARK programs without errors.
Dependencies: None.

Requirement #3
Requirement type: Functional
Description: The system should analyse a single program unit.
Rationale: While SPARK software may consist of more than one source code file, it is not necessary to analyse more than one program in order to demonstrate the feasibility of AUTOGAP. This requirement thus simplifies the implementation of AUTOGAP.
Fit criterion: A single SPARK program should be analysed.
Dependencies: None.

Requirement: #4:
Requirement type: Functional
Description: The system should be designed so as to work after the user has applied the SPARK Examiner to his or her code.
Rationale: When the SPARK Examiner has been successfully run, we know for sure that the code is syntactically correct. This simplifies the parsing.
Fit criterion: None.
Dependencies: 2

Requirement #5
Requirement type: Functional
Description: The system should build an internal representation of the code in the form of a flow graph structure.
Rationale: To analyse the code, we need an internal representation. A graph is the most common and natural data structure for this type of application.
Fit criterion: None.
Dependencies: 7

Requirement #6
Requirement type: Functional
Description: The system should analyse SPARK programs to generate meta data from them.
Rationale: This states the goal of the project.
Fit criterion: The final testing will reveal if this has been met.
Dependencies: None.

Requirement #7
Requirement type: Functional
Description: The system should analyse source code by analysing the graph structure that is made.
Rationale: If the graph structure and the interface between it and the analysis is good, then each of these parts can be implemented separately, creating a modular system that is easy to change.
Fit criterion: The final code should reflect this requirement. No analysis should take place on structures other than the graph.
Dependencies: None.

Requirement #8
Requirement type: Functional
Description: The analysis should be carried out in a bottom-up fashion.
Rationale: Bottom-up means that no extra information about the code (existing annotations) are taken into consideration. AUTOGAP should analyse the code alone.
Fit criterion: The final code should reflect this requirement.
Dependencies: None.

Requirement #9
Requirement type: Functional
Description: Where possible, new annotations should be added to the original source file, derived from the algorithmic patterns.
Rationale: If we can create new facts about the SPARK program, then the SPADE theorem prover might be able to perform a complete proof automatically or with less interaction with the user.
Fit criterion: Any new source files should contain new annotations.
Dependencies: 10.

Requirement #10
Requirement type: Functional
Description: If new annotations can be derived from a source file, then the system should incorporate these at a suitable place and write back a new source file containing everything from the old source in addition to the new annotations.
Rationale: By keeping the new and old source files separate, it will become easier to test the system since the test file itself is not destroyed.
Fit criterion: No old files should be destroyed or changed.
Dependencies: None.

Requirement #11
Requirement type: Functional
Description: The result of the algorithmic patterns generation should be a set of text files (in addition to any new annotations incorporated in the source file).
Rationale: This conforms to the SPARK approach, which entails that each subsystem creates its own output files, not overwriting any others. By using text files, communication with the NUSPADE system is eased.
Fit criterion: A single or more text files containing program properties and meta-data should be produced for every execution of the system.
Dependencies: 1.

Requirement #12
Requirement type: Functional
Description: The SPARK source code that is to be analysed should contain the default assertion, `assert true;` on the line immediately succeeding the start of all loops.
Rationale: In order to identify the point in the source code where a loop is started we need an identifier. When parsing the SPARK code it is easy to recognise the assertion and record the line number where it occurred. This will also help in inserting any new annotations at the correct place. Demanding this conforms to the existing SPARK approach, since the original SPARK static analyser automatically adds the default annotation at the start of each loop where none exists.
Fit criterion: All test files should have the default annotation at the start of each loop.
Dependencies: 10, 11.

Requirement #13
Requirement type: Functional
Description: AUTOGAP should be an extendible framework so that new heuristics can be added in the future.
Rationale: AUTOGAP should be regarded as a prototype system for generating information on source code. Its utility will be greatly increased if it is possible to add new heuristics after it is finished.
Fit criterion: An experienced programmer should be able to create new heuristics by understanding how the graph and some of the existing heuristics work. Some sort of library system should be implemented, so that heuristics can be removed and added easily.
Dependencies: None.

Appendix E

Definitions of algorithmic patterns

This is a more formal listing of the algorithmic patterns that AUTOGAP looks for in SPARK code. Note that **Syntax** describes the output of each heuristic as it appears in the file that is generated for use by NUSPADE. This is also the syntax for the results that are appended onto each reports file.

Name: `mono`

Type: Monotonically increasing/decreasing variables.

Syntax:

```
mono_inc(value(Var, now),
         greaterThan,
         value(Var, previous),
         line(LoopStart))

mono_dec(value(Var, now),
         lessThan,
         value(Var, previous),
         line(LoopStart))
```

Description: `Var` is a monotonically increasing or monotonically decreasing program variable within the loop has its annotation at line `LoopStart`. Such variables either consistently increase or decrease but do not oscillate in relative value within that loop.

Precondition: A variable that is incremented but never decremented is monotonically increasing. A variable that is decremented but never incremented is monotonically decreasing (simplification). The increment or decrement must be of the form `Var := Var + integer`.

Name: `init`
Type: Initial values of variables.
Syntax: `initial_val(Var, Value)`
Description: The first value that the program variable `Var` takes is `Value`.
Precondition: `Var` is explicitly assigned a value `Value` before being used.

Name: `counter`
Type: Counter variables.
Syntax: `count_var(Var, Line)`
Description: `Var` is a counter variable for the loop that has its annotation at line `Line`. Such variables are used for loop control.
Precondition: `Var` appears in a loop control condition and is incremented or decremented within the loop.

Name: `index`
Type: Index variables.
Syntax: `index_var(Var, Array)`
Description: `Var` is associated with array `Array` because it indexes `Array`.
Precondition: `Var` appears as an index into `Array` at least once.

Name: `partition`
Type: Array partitions.
Syntax: `partition(Array, Start, End, Line)`
Description: Throughout execution, there is a partition from `Start` to `End` in array `Array` within the loop that has its annotation at line `Line`. `Start` and `End` can be program variables or constants.
Precondition: Any variables must be counter variables and be monotonically increasing or decreasing within the loop referred to by `Line`. Any variables must also be index variables for the array `Array`, and their initial value must be known.

Name: `invariant`
Type: Loop invariants.
Syntax: `invar(Line, Expression)`
Description: `Expression` is a loop invariant for the loop that has its annotation at line `Line`. Typically `Expression` contains a simple comparison between program variables or a variable and a constant, e.g. `I<=J`.
Precondition: `Expression` is derived from the loop exit condition for the loop and the monotonic nature of the variable(s) that occur within the exit condition.

Name: `bounds`
Type: Variable bounds.

Syntax:
`bound(Var, Value, upper)`
`bound(Var, Value, lower)`

Description: The upper or lower bound on variable `Var` is the constant `Value`.
Precondition: Variables with upper bounds must be monotonically increasing. Variables with lower bounds must be monotonically decreasing.

Appendix F

Report files generated by

AUTOGAP

Here are the reports generated by AUTOGAP

```
--     AUTOGAP OUTPUT
-- Sat May 31 20:36:58 BST 2003


procedure Binary_Search(Search_Array : in IntArray;
                        Item_To_Find : in Integer;
                        Placement : out Integer)
is
   subtype Index is Integer range 1..100;

   Left : Index;
   Right : Index;
   Middle : Index;
   Found : Boolean;

begin

   Left := Index'First;
   Right := Index'Last;
   Found := False;
   Placement := -1;

   loop
      --# assert true;
```

```
      if Right < Left then

         exit;
      end if;

      if Found then
         exit;
      end if;

      Middle := (Left + Right)/2;

      if Item_To_Find = Search_Array(Middle) then
            Placement := Middle;
            Found := True;
      end if;

      if Item_To_Find < Search_Array(Middle) then
            Right := Middle - 1;
      else
         Left := Middle + 1;
      end if;


   end loop;

end Binary_Search;


--  ALGORITHMIC PROPERTIES

-- initial_val('Left', integer(1)).
-- initial_val('Right', integer(100)).
-- initial_val('Placement', -(integer(1))).
-- initial_val('Middle', (variable('Left')+variable('Right'))/integer(2)).
-- initial_val('Found', variable('False')).
-- count_var('Right', 20).
-- count_var('Left', 20).
-- count_var('Found', 20).
```

```
--     AUTOGAP OUTPUT
-- Sun Jun  1 15:17:49 BST 2003

procedure Bubble_Sort(Sort_Me : in out IntArray)
is
   subtype Index is Integer range 1..100;

   Temp : Index;
   I : Index;
   J : Index;

begin

   I := Index'First;
   loop
      --# assert true;

      exit when I > Size;

      J := 1;
      loop
         --# assert true;
         exit when J > I;

         if Sort_Me(J-1) > Sort_Me(J) then
            Temp := Sort_Me(J-1);
            Sort_Me(J-1) := Sort_Me(J);
            Sort_Me(J) := Temp;
         end if;

         J := J + 1;

      end loop;

      I := I + 1;
   end loop;
end Bubble_Sort;

--  ALGORITHMIC PROPERTIES

-- initial_val('I', integer(1)).
-- initial_val('Temp', element(variable('Sort_Me'),variable('J')-integer(1))).
-- initial_val('J', integer(1)).
-- count_var('I', 13).
-- count_var('J', 20).
-- mono_inc(value('I',now), greaterthan, value('I',previous), line(13)).
-- mono_inc(value('J',now), greaterthan, value('J',previous), line(20)).
-- index_var('J', 'Sort_Me').
-- array_partition('Sort_Me', integer(1), 'J', 20).
```

```
--     AUTOGAP OUTPUT
-- Sun Jun  1 16:23:00 BST 2003


procedure Dutch_Flag(Flag : in out ArrayOfColours)
is

   subtype Index is Integer range 1..100;

   R : Index;
   M : Index;
   W : Index;
   Temp : Colour;

begin

   R := Index'First;
   M := Index'Last;
   W := Index'Last;




   loop
      --# assert R<=M;
      exit when R = M;

      -- if element is white then no problem
      if Flag(M-1) = White then
         M := M - 1;

      end if;

      -- if element is red we must swap with the
      -- Rth element
      if Flag(M-1) = Red then

         Temp := Flag(M-1);
         Flag(M-1) := Flag(R);
         Flag(R) := Temp;

         R := R+1;
      end if;

      -- if the element is blue we must first make
      -- room in the sector that is to become blue
      -- and then swap the elements
      if Flag(M-1) = Blue then
```

```
         M := M - 1;
         W := W - 1;

         Temp := Flag(M);
         Flag(M) := Flag(W);
         Flag(W) := Temp;

      end if;

   end loop;
end Dutch_Flag;


--  ALGORITHMIC PROPERTIES

-- initial_val('R', integer(1)).
-- initial_val('M', integer(100)).
-- initial_val('W', integer(100)).
-- initial_val('Temp', element(variable('Flag'),variable('M')-integer(1))).
-- count_var('R', 21).
-- count_var('M', 21).
-- mono_inc(value('R',now), greaterthan, value('R',previous), line(21)).
-- mono_dec(value('M',now), lessthan, value('M',previous), line(21)).
-- mono_dec(value('W',now), lessthan, value('W',previous), line(21)).
-- index_var('M', 'Flag').
-- index_var('R', 'Flag').
-- index_var('W', 'Flag').
-- array_partition('Flag', 'M', integer(100), 21).
-- array_partition('Flag', integer(1), 'R', 21).
-- invar(21, variable('R')<=variable('M')).
```

```
--     AUTOGAP OUTPUT
-- Sun Jun  1 15:27:57 BST 2003


procedure Eliminate(Equation_Set : in out Matrix)
is
   subtype Index is Integer range 1..100;

   Max : Index;
   Temp : Index;
   Start_1 : Index;
   Size_1 : Index;
   I : Index;
   J : Index;
   K : Index;
begin

   Size_1 := Index'Last + 1;
   I := Index'First;
   loop
      --# assert I<=100;
      exit when I = Index'Last;

      Max := I;
      Start_1 := I+1;
      J := Start_1;

         loop
            --# assert J<=100;
            exit when J = Index'Last;

            if Equation_Set(J, I) > Equation_Set(Max, I) then
               Max := J;
            end if;

            J := J + 1;
         end loop;

         K := I;

         loop
            --# assert true;
            exit when K > Size_1;

            Temp := Equation_Set(I, K);
            Equation_Set(I, K) := Equation_Set(Max, K);
            Equation_Set(Max, K) := Temp;

            K := K + 1;
```

```
         end loop;

         J := Start_1;
         loop
            --# assert true;
            exit when J > Index'Last;

            K := Size_1;
            loop
               --# assert true;
               exit when K < I;
               Equation_Set(J, K) := Equation_Set(J, K) -
                      Equation_Set(I, K)*Equation_Set(J, I)/Equation_Set(I, I);
               K := K - 1;
            end loop;

            J := J + 1;
         end loop;

         I := I + 1;
   end loop;
end Eliminate;

--  ALGORITHMIC PROPERTIES

-- initial_val('I', integer(1)).
-- initial_val('Size_1', integer(101)).
-- initial_val('Max', variable('I')).
-- initial_val('Temp', element(variable('Equation_Set'),variable('I'))).
-- initial_val('Start_1', variable('I')+integer(1)).
-- initial_val('J', variable('Start_1')).
-- initial_val('K', variable('I')).
-- count_var('I', 20).
-- count_var('J', 29).
-- count_var('K', 43).
-- count_var('J', 55).
-- count_var('K', 60).
-- mono_inc(value('I',now), greaterthan, value('I',previous), line(20)).
-- mono_inc(value('J',now), greaterthan, value('J',previous), line(29)).
-- mono_inc(value('J',now), greaterthan, value('J',previous), line(55)).
-- mono_inc(value('K',now), greaterthan, value('K',previous), line(43)).
-- mono_dec(value('K',now), lessthan, value('K',previous), line(60)).
-- bound('I', 100, upper).
-- bound('J', 100, upper).
-- index_var('J', 'Equation_Set').
-- index_var('Max', 'Equation_Set').
-- index_var('I', 'Equation_Set').
-- invar(20, variable('I')<=integer(100)).
-- invar(29, variable('J')<=integer(100)).
```

```
--     AUTOGAP OUTPUT
-- Sun Jun  1 15:29:38 BST 2003


-- File "exchange.adb". Implementation file for the exchange sort
-- algorithm. Taken from "Programming in Ada" by Richard Wiener
-- and Richard Sincovec.

-- Author: Tommy Ingulfsen, May 2003.

procedure Exchange_Sort(Arr : in out Int_Array)
is
   subtype Index is Integer range 1..100;

   Temp : Index;
   Idx : Integer;
   Max : Index;
   Position : Index;
   I : Index;

begin

   Idx := Index'Last +1;

   loop
      --# assert Idx>=2;

      Idx := Idx - 1;

      -- find max value in array from 1 to Idx
      Max := Arr(1);
      Position := 1;

      I := Index'First;
      loop
         --# assert I<=100;

         if I = Index'Last then
            exit;
         end if;


         if Arr(I) >= Max then
            Max := Arr(I);
            Position := I;
         end if;

         I := I + 1;
```

```
      end loop;

      -- exchange Arr(Position) with Arr(Idx)
      Temp := Arr(Position);
      Arr(Position) := Arr(Idx);
      Arr(Idx) := Temp;

      exit when Idx = 2;
   end loop;

end Exchange_Sort;



--  ALGORITHMIC PROPERTIES

-- initial_val('I', integer(1)).
-- initial_val('Idx', integer(101)).
-- initial_val('Temp', element(variable('Arr'),variable('Position'))).
-- initial_val('Max', element(variable('Arr'),integer(1))).
-- initial_val('Position', integer(1)).
-- count_var('I', 32).
-- count_var('Idx', 22).
-- mono_inc(value('I',now), greaterthan, value('I',previous), line(32)).
-- mono_dec(value('Idx',now), lessthan, value('Idx',previous), line(22)).
-- bound('I', 100, upper).
-- bound('Idx', 2, lower).
-- index_var('I', 'Arr').
-- index_var('Position', 'Arr').
-- index_var('Idx', 'Arr').
-- array_partition('Arr', integer(1), 'I', 32).
-- array_partition('Arr', 'Idx', integer(101), 22).
-- invar(32, variable('I')<=integer(100)).
-- invar(22, variable('Idx')>=integer(2)).
```

```
--     AUTOGAP OUTPUT
-- Sun Jun  1 15:31:26 BST 2003


-- File "find.ads", specification file for the Find algorithm.
-- Taken from Tony Hoare's paper in the Communications of the
-- ACM, Volume 14, Number 1, January 1971.

-- Problem: Find that element of an array Arr(1:Size) whose
--          value is Fth in order of magnitude, and to
--          rearrange the array in such a way that this element
--          is placed in Arr(F), and furthermore, all elements
--          with subscripts greater than F have greater values.
--          Thus on completion of the program the following
--          relationship will hold:

-- Arr(1), Arr(2), ..., Arr(F-1) <= Arr(F) <= Arr(F+1), .., Arr(Size)



-- Author: Tommy Ingulfsen, May 2003
procedure H_Find(Arr : in out Int_Array;
                 F : in Integer)
is
   subtype Index is Integer range 1..100;

   M : Index;
   R : Index;
   I : Index;
   J : Index;
   Temp : Index;
   N : Index;


begin

   -- M <= F & For all P, Q ( 1 <= P < M <= Q <= Size => Arr(P) <= Arr(Q) )
   -- F <= N & For all P, Q ( 1 <= P <= N < Q <= Size => Arr(P) <= Arr(Q) )

   M := Index'First;
   N := Index'Last;

   -- reduce middle part
   loop
      --# assert true;

      if M >= N then
         exit;
      end if;
```

```
-- M <= I & For all P : ( 1 <= P <= I  => Arr(P) <= R)
-- J <= Size & For all Q : ( J <= Q <= Size => R <= Arr(Q) )

R := Arr(F);
I := M;
J := N;

loop
   --# assert true;

   if I > J then
      exit;
   end if;

   -- increase I and decrease J
   loop
      --# assert true;

      if Arr(I) >= R then
         exit;
      end if;

      I := I + 1;

      loop
         --# assert true;

         if R >= Arr(J) then
            exit;
         end if;


         J := J - 1;

         -- Arr(J) <= R <= Arr(I)
         if I <= J then
            Temp := Arr(I);
            Arr(I) := Arr(J);
            Arr(J) := Temp;

            -- Arr(I) <= R <= Arr(J)
            I := I + 1;
            J := J - 1;
         end if;
      end loop;
   end loop;
```

```
            if F <= J then
               N := J;
            elsif I <= F then
               M := I;
            end if;

            if F > J then
               exit;
            end if;

            if I > F then
               exit;
            end if;
      end loop;
   end loop;
end H_Find;



--  ALGORITHMIC PROPERTIES

-- initial_val('M', integer(1)).
-- initial_val('N', integer(100)).
-- initial_val('R', element(variable('Arr'),variable('F'))).
-- initial_val('I', variable('M')).
-- initial_val('J', variable('N')).
-- initial_val('Temp', element(variable('Arr'),variable('I'))).
-- count_var('M', 41).
-- count_var('N', 41).
-- count_var('I', 55).
-- count_var('J', 55).
-- mono_inc(value('I',now), greaterthan, value('I',previous), line(63)).
-- mono_inc(value('I',now), greaterthan, value('I',previous), line(72)).
-- mono_dec(value('J',now), lessthan, value('J',previous), line(72)).
-- index_var('I', 'Arr').
-- index_var('J', 'Arr').
```

```
--     AUTOGAP OUTPUT
-- Sun Jun  1 15:41:21 BST 2003


-- File "insert.adb" - implements the insertion sort
-- algorithm.

-- Author: Tommy Ingulfsen, May 2003.

procedure Insertion_Sort(To_Sort : in out IntArray)
is
   subtype Index is Integer range 1..100;
   Idx : Index;
   J : Index;
   I : Index;

begin

   I := Index'First;

   loop
      --# assert I<=100;
      exit when I = Index'Last;

      Idx := To_Sort(I);
      J := I;

      loop
         --# assert J>=1;

         if J = 1 then
            exit;
         end if;

         if To_Sort(J-1) <= Idx then
            exit;
         end if;

         To_Sort(J) := To_Sort(J-1);
         J := J - 1;
      end loop;

      To_Sort(J) := Idx;
      I := I + 1;
   end loop;

end Insertion_Sort;
```

```
--  ALGORITHMIC PROPERTIES

-- initial_val('I', integer(1)).
-- initial_val('Idx', element(variable('To_Sort'),variable('I'))).
-- initial_val('J', variable('I')).
-- count_var('I', 18).
-- count_var('J', 25).
-- mono_inc(value('I',now), greaterthan, value('I',previous), line(18)).
-- mono_dec(value('J',now), lessthan, value('J',previous), line(25)).
-- bound('I', 100, upper).
-- bound('J', 1, lower).
-- index_var('J', 'To_Sort').
-- array_partition('To_Sort', 'J', variable('I'), 25).
-- invar(18, variable('I')<=integer(100)).
-- invar(25, variable('J')>=integer(1)).
```

```
--     AUTOGAP OUTPUT
-- Sun Jun  1 15:43:17 BST 2003


-- File "intseries.ads", specification file for the algorithm
-- that sums an integer series. Taken from "Programming in Ada"
-- by Richard Wiener and Richard Sincovec.Terms**2;
-- The problem is to determine how many terms are needed in the
-- series 1*1 + 2*2 + 3*3 + ... + N*N so that the sum just
-- exceeds 10000.

-- Author: Tommy Ingulfsen, May 2003.

procedure Sum_Exceeds_10K(No_Terms: out Integer)
is
   Sum : Integer;   -- keeps track of running sum
   Terms : Integer; -- keeps track of number of terms

begin

   Sum := 0;
   Terms := 1;

   loop
      --# assert true;

      Sum := Sum + Terms*Terms;
      exit when Sum >= 10000;

      Terms := Terms + 1;
   end loop;

   No_Terms := Terms;
end Sum_Exceeds_10K;


--  ALGORITHMIC PROPERTIES

-- initial_val('No_Terms', variable('Terms')).
-- initial_val('Sum', integer(0)).
-- initial_val('Terms', integer(1)).
-- count_var('Sum', 21).
-- mono_inc(value('Terms',now), greaterthan, value('Terms',previous), line(21)).
```

```
--     AUTOGAP OUTPUT
-- Sun Jun  1 15:45:42 BST 2003


-- File "linearsearch.adb", implements a linear search of an array
-- of integers.

-- Author: Tommy Ingulfsen, May 2003.

procedure Search(A : in IntArray;
                 Find : in Integer;
                 Ans : out Integer)
is

   subtype Index is Integer range 1..100;

begin

   Ans := Index'First;

   loop
      --# assert Ans<=100;



      if A(Ans) = Find then
         exit;
      end if;

      if Ans = Index'Last then
         exit;
      end if;

      Ans := Ans + 1;
   end loop;
end Search;


--  ALGORITHMIC PROPERTIES

-- initial_val('Ans', integer(1)).
-- count_var('Ans', 18).
-- mono_inc(value('Ans',now), greaterthan, value('Ans',previous), line(18)).
-- bound('Ans', 100, upper).
-- index_var('Ans', 'A').
-- array_partition('A', integer(1), 'Ans', 18).
-- invar(18, variable('Ans')<=integer(100)).
```

```
--     AUTOGAP OUTPUT
-- Sun Jun  1 15:47:09 BST 2003


-- File "majority.adb". Implements the majority voting
-- algorithm.

-- Problem: Given n voters and n candidates, determine whether a
--          candidate "Winner" in the ballot receives more than half
--          the votes.

-- The problem is modelled using an array of Size items, say
-- b b c b d d c c c c c c d b a e ...and so on,
-- using the enumerated type Vote_List.
-- type Vote_List is array (Index) of Character;

-- Author: Tommy Ingulfsen, May 2003. The algorithm is taken from
--         Roland C. Backhouse's "Program Construction and Verification".

package body Majority is
procedure Majority_Vote( All_Votes : in Vote_List;
                         Winner : out Character)
is
   subtype Index is Integer range 1..100;

   K : Index;
   E : Index;

begin

   K := Index'First;
   Winner := All_Votes(Index'First);
   E := Index'First;
   loop
      --# assert K<=100;

      if K = Index'Last then
         exit;
      end if;


      if All_Votes(K) = Winner then
         E := E + 1;
      elsif 2*E = K then
         Winner := All_Votes(K);
         E := E + 1;
      end if;
```

```
      K := K + 1;
   end loop;

end Majority_Vote;
end Majority;


--  ALGORITHMIC PROPERTIES

-- initial_val('K', integer(1)).
-- initial_val('E', integer(1)).
-- initial_val('Winner', element(variable('All_Votes'),tick('Index','First'))).
-- count_var('K', 31).
-- mono_inc(value('K',now), greaterthan, value('K',previous), line(31)).
-- mono_inc(value('E',now), greaterthan, value('E',previous), line(31)).
-- bound('K', 100, upper).
-- index_var('K', 'All_Votes').
-- array_partition('All_Votes', integer(1), 'K', 31).
-- invar(31, variable('K')<=integer(100)).
```

```
--     AUTOGAP OUTPUT
-- Sun Jun  1 15:50:31 BST 2003


-- File "maxvalue.adb", implementation file for the algorithm
-- that finds the largest value in an array of integers.

-- Author: Tommy Ingulfsen, May 2003.

procedure Max(Arr : in Int_Array;
              Max_Value : out Integer)
is
   subtype Index is Integer range 1..100;

   I: Integer;

begin

   Max_Value := Arr(Index'First); -- initialisation

   I := Index'First;
   loop
      --# assert true;

      if I > Index'Last then
         exit;
      end if;


      if Arr(I) >= Max_Value then
         Max_Value := Arr(I);
      end if;

      I := I + 1;
   end loop;
end Max;


--  ALGORITHMIC PROPERTIES

-- initial_val('I', integer(1)).
-- initial_val('Max_Value', element(variable('Arr'),tick('Index','First'))).
-- count_var('I', 19).
-- mono_inc(value('I',now), greaterthan, value('I',previous), line(19)).
-- index_var('I', 'Arr').
-- array_partition('Arr', integer(1), 'I', 19).
```

```
--     AUTOGAP OUTPUT
-- Sun Jun  1 15:54:02 BST 2003


-- File "merge.adb", the implementation file for the merge algorithm.
-- Taken from http://linux.wku.edu/~lamonml/algor/sort/.

-- The merge sort splits the list to be sorted into two equal
-- halves, and places them in separate arrays. Each array is recursively
-- sorted, and then merged back together to form the final sorted list.

-- The merge sort is by nature recursive, and so cannot be implemented
-- in SPARK. The non-recursive implementations all make use of subroutines,
-- which is not included in the subset of SPARK that we are working with.
-- However, the core algorithm in the merge sort is the actual merging
-- routine, and this is presented here.

-- Elementary implementations of the merge sort make use of three arrays,
-- one for each half of the data set and one to store the sorted
-- list in. The below algorithm merges the arrays in-place, so only
-- two arrays are required.

-- Author: Tommy Ingulfsen, May 2003.

procedure Merging(To_Sort : in out IntArray;
                  Temp : in out IntArray;
                  Left : in out Index;
                  Middle: in out Index;
                  Right : in out Index)
is
   subtype Index is Integer range 1..100;

   Left_End : Index;
   No_Elements : Index;
   Tmp_Pos : Index;
   I : Index;

begin

   Left_End := Middle - 1;
   Tmp_Pos := Left;
   No_Elements := Right - Left + 1;

   loop
      --# assert true;

      if Left > Left_End then
         exit;
      end if;
```

```
   if Middle > Right then
      exit;
   end if;

   if To_Sort(Left) <= To_Sort(Middle) then
      Temp(Tmp_Pos) := To_Sort(Left);
      Tmp_Pos := Tmp_Pos + 1;
         Left := Left + 1;
   else
      Temp(Tmp_Pos) := To_Sort(Middle);
      Tmp_Pos := Tmp_Pos + 1;
      Middle := Middle + 1;

   end if;
end loop;


loop
   --# assert true;

   if Left > Left_End then
      exit;
   end if;

   Temp(Tmp_Pos) := To_Sort(Left);
   Left := Left + 1;
   Tmp_Pos := Tmp_Pos + 1;

end loop;


loop
   --# assert true;

   if Middle > Right then
      exit;
   end if;

   Temp(Tmp_Pos) := To_Sort(Middle);
   Middle := Middle + 1;
   Tmp_Pos := Tmp_Pos + 1;

end loop;

I := 1;

loop
   --# assert true;
```

```
      if I > No_Elements then
         exit;
      end if;

      To_Sort(Right) := Temp(Right);
      Right := Right - 1;

      I := I + 1;
   end loop;
end Merging;


--  ALGORITHMIC PROPERTIES

-- initial_val('Left_End', variable('Middle')-integer(1)).
-- initial_val('No_Elements', variable('Right')-variable('Left')+integer(1)).
-- initial_val('Tmp_Pos', variable('Left')).
-- initial_val('I', integer(1)).
-- count_var('Left', 41).
-- count_var('Middle', 41).
-- count_var('Left', 65).
-- count_var('Middle', 79).
-- count_var('I', 94).
-- mono_inc(value('Left',now), greaterthan, value('Left',previous), line(41)).
-- mono_inc(value('Left',now), greaterthan, value('Left',previous), line(65)).
-- mono_inc(value('Middle',now), greaterthan, value('Middle',previous), line(41)).
-- mono_inc(value('Middle',now), greaterthan, value('Middle',previous), line(79)).
-- mono_inc(value('Tmp_Pos',now), greaterthan, value('Tmp_Pos',previous), line(41)).
-- mono_inc(value('Tmp_Pos',now), greaterthan, value('Tmp_Pos',previous), line(65)).
-- mono_inc(value('Tmp_Pos',now), greaterthan, value('Tmp_Pos',previous), line(79)).
-- mono_inc(value('I',now), greaterthan, value('I',previous), line(94)).
-- mono_dec(value('Right',now), lessthan, value('Right',previous), line(94)).
-- index_var('Left', 'To_Sort').
-- index_var('Middle', 'To_Sort').
-- index_var('Tmp_Pos', 'Temp').
-- index_var('Right', 'To_Sort').
-- index_var('Right', 'Temp').
```

```
--     AUTOGAP OUTPUT
-- Sun Jun  1 16:35:16 BST 2003


-- File "minvalue.adb", implementation file for the algorithm
-- that finds the smallest value in an array of integers.

-- Author: Tommy Ingulfsen, May 2003.

procedure Min(Arr : in Int_Array;
              Min_Value : out Integer)
is
   subtype Index is Integer range 1..100;

   I : Integer;
begin

   Min_Value := Arr(Index'First); -- initialisation

   I := Index'First;
   loop
      --# assert true;
      if I > Index'Last then
         exit;
      end if;

      if Arr(I) <= Min_Value then
         Min_Value := Arr(I);
      end if;

      I := I + 1;
   end loop;
end Min;


--  ALGORITHMIC PROPERTIES

-- initial_val('I', integer(1)).
-- initial_val('Min_Value', element(variable('Arr'),tick('Index','First'))).
-- count_var('I', 18).
-- mono_inc(value('I',now), greaterthan, value('I',previous), line(18)).
-- index_var('I', 'Arr').
-- array_partition('Arr', integer(1), 'I', 18).
```

```
--     AUTOGAP OUTPUT
-- Sun Jun  1 16:37:05 BST 2003


-- File "pivot.ads", the implementation of the algorithm
-- that quicksort uses to find the pivot element. Taken from
-- Michael B. Feldman's "Data Structures with Ada".

-- Author: Tommy Ingulfsen, May 2003.

procedure Find_Pivot(Arr : in Int_Array;
                     Pivot_El : out Integer)
is
   subtype Index is Integer range 1..100;

   Left : Index;
   Right : Index;
   First_Key : Integer;
   Start_Left : Index;
   Up : Index;

begin

   Left := Index'First; -- Array starts at 1
   Right := Index'Last;
   First_Key := Arr(Left);
   Pivot_El := 0;  -- Initialise to error value
   Start_Left := Left+1;

   Up := Start_Left;
   loop
      --# assert true;

      if Up > Right then
         exit;
      end if;


      if Arr(Up) > First_Key then
         Pivot_El := Up;
      end if;

      if Arr(Up) < First_Key then
         Pivot_El := Left;
         exit;
      end if;

      Up := Up + 1;
   end loop;
```

```
end Find_Pivot;


--  ALGORITHMIC PROPERTIES

-- initial_val('Left', integer(1)).
-- initial_val('Right', integer(100)).
-- initial_val('Pivot_El', integer(0)).
-- initial_val('First_Key', element(variable('Arr'),variable('Left'))).
-- initial_val('Start_Left', variable('Left')+integer(1)).
-- initial_val('Up', variable('Start_Left')).
-- count_var('Up', 28).
-- mono_inc(value('Up',now), greaterthan, value('Up',previous), line(28)).
-- index_var('Up', 'Arr').
-- array_partition('Arr', variable('Start_Left'), 'Up', 28).
```

```
--     AUTOGAP OUTPUT
-- Sun Jun  1 16:39:04 BST 2003


-- File "polish1.adb". This is an implementation of the
-- Polish Flag problem algorithm.

-- Problem: Given an array of colours (red or white), sort
--          it so that all the red elements precede all the
--          white ones. This is a simpler version of the
--          Dutch flag problem.

-- Author: Bill James Ellis (the actual code),
--         Tommy Ingulfsen (the packaging)

-- Partition the colours of flag.
procedure Partition_Section(Flag: in out ArrayOfColours)
is
   subtype Index is Integer range 1..100;

   I: Index;
   J: Integer;
   T: Colour;
begin
   I:= Index'First;
   J:= Index'Last + 1;

   loop
      --# assert I<=J;

      exit when I=J;


      if Flag(I)=Red then
         I:=I+1;

      else
         J:=J-1;
         T:=Flag(I);
         Flag(I):=Flag(J);
         Flag(J):=T;
      end if;
   end loop;

end Partition_Section;
```

```
--  ALGORITHMIC PROPERTIES

-- initial_val('I', integer(1)).
-- initial_val('J', integer(101)).
-- initial_val('T', element(variable('Flag'),variable('I'))).
-- count_var('I', 25).
-- count_var('J', 25).
-- mono_inc(value('I',now), greaterthan, value('I',previous), line(25)).
-- mono_dec(value('J',now), lessthan, value('J',previous), line(25)).
-- index_var('I', 'Flag').
-- index_var('J', 'Flag').
-- array_partition('Flag', integer(1), 'I', 25).
-- array_partition('Flag', 'J', integer(101), 25).
-- invar(25, variable('I')<=variable('J')).
```

```
--     AUTOGAP OUTPUT
-- Sun Jun  1 16:40:50 BST 2003


-- File "polish2.adb". This is the specification file
-- for the Polish Flag problem algorithm.

-- Problem: Given an array of colours (red or white), sort
--          it so that all the red elements precede all the
--          white ones. This is a simpler version of the
--          Dutch flag problem.
--          The algorithm is taken from Roland C. Backhouse's
--          "Program Construction and Verification".

-- The loop invariant given in the book is

-- 0 <= M <= W <= Size and
-- For all elements I, 0 <= I <= M: Flag(I) = Black
-- For all elements I, M <= I <= W: Flag(I) = White

-- Author:  Tommy Ingulfsen, May 2003.

procedure Partition_Section_2(Flag: in out ArrayOfColours)
is
   subtype Index is Integer range 1..100;

   M : Index;
   W : Index;
   Temp : Colour;

begin
   M := Index'First;
   W := Index'Last;

   loop
      --# assert M<=W;

      if M = W then
         exit;
      end if;

      -- if the current element is red, no change
      if Flag(M) = Red then
         M := M + 1;

         -- if it is white, it should be red, so swap
      else
         W := W - 1;
         Temp := Flag(M);
```

```
         Flag(M) := Flag(W);
         Flag(W) := Temp;
      end if;

   end loop;

end Partition_Section_2;




--  ALGORITHMIC PROPERTIES

-- initial_val('M', integer(1)).
-- initial_val('W', integer(100)).
-- initial_val('Temp', element(variable('Flag'),variable('M'))).
-- count_var('M', 32).
-- count_var('W', 32).
-- mono_inc(value('M',now), greaterthan, value('M',previous), line(32)).
-- mono_dec(value('W',now), lessthan, value('W',previous), line(32)).
-- index_var('M', 'Flag').
-- index_var('W', 'Flag').
-- array_partition('Flag', integer(1), 'M', 32).
-- array_partition('Flag', 'W', integer(100), 32).
-- invar(32, variable('M')<=variable('W')).
```

```
--     AUTOGAP OUTPUT
-- Sun Jun  1 17:01:53 BST 2003


-- File "primes.ads", specification file for the prime number
-- series generator. The algorithm generates the prime
-- integers up to 1999. It is taken from Richard Wiener and
-- Richard Sincovec's "Programming in Ada".

-- Author: Tommy Ingulfsen, May 2003.

procedure Make_Primes(Arr : in out Int_Array;
                      Size : out Index)  -- size of array
is
   subtype Index is Integer range 1..2000;

   Divisor : Integer;
   Candidate : Integer;  -- a possible prime
   Prime : Boolean;

begin

   Candidate := 3;
   Arr(1) := 2;    -- take care of 2,3 separately
   Arr(2) := 3;
   Size := 3;

   loop
      --# assert Candidate<=1999;

      Divisor := 1;
      Candidate := Candidate + 2;  -- look at next candidate
      Prime := True;

      -- assume Candidate is innocent until proved
      -- guilty (only need use odd Divisors up to square root
      -- of Candidate)
      loop
         --# assert true;

         if Divisor * Divisor > Candidate then
            exit;
         end if;

         -- no need to keep testing
         Divisor := Divisor + 2;
         if Candidate mod Divisor = 0 then
            Prime := False;
            exit;
```

```
         end if;

      end loop;

      -- if no numbers where found for which
      -- Candidate mod Divisor = 0, then Candidate is
      -- prime, so save it in array
      if Prime = True then
         Arr(Size) := Candidate;
         Size := Size + 1;
      end if;

      exit when Candidate = 1999;
   end loop;

end Make_Primes;



--  ALGORITHMIC PROPERTIES

-- initial_val('Size', integer(3)).
-- initial_val('Divisor', integer(1)).
-- initial_val('Candidate', integer(3)).
-- initial_val('Prime', variable('True')).
-- count_var('Divisor', 35).
-- count_var('Candidate', 25).
-- mono_inc(value('Size',now), greaterthan, value('Size',previous), line(25)).
-- mono_inc(value('Divisor',now), greaterthan, value('Divisor',previous), line(35)).
-- mono_inc(value('Candidate',now), greaterthan, value('Candidate',previous), line(25)).
-- bound('Candidate', 1999, upper).
-- index_var('Size', 'Arr').
-- invar(25, variable('Candidate')<=integer(1999)).
```

```
--     AUTOGAP OUTPUT
-- Sun Jun  1 17:05:48 BST 2003


-- File "selection.adb" - contains code to implement the selection
-- sort. Algorithm taken from http://linux.wku.edu/~lamonml/algor/sort/.

-- Author : Tommy Ingulfsen, May 2003.

procedure Selection_Sort(Sort_Array : in out IntArray)
is

   subtype Index is Integer range 1..100;


   Min : Integer;
   Temp : Integer;
   I : Index;
   J : Index;

begin

   I := Index'First;

   loop
      --# assert true;

      if I > Index'Last - 1 then
         exit;
      end if;

      Min := I;

      J := Index'First+1;
      loop
         --# assert true;

         if J > Index'Last then
            exit;
         end if;

         if Sort_Array(J) < Sort_Array(Min) then
            Min := J;
         end if;

         J := J + 1;
         end loop;

         Temp := Sort_Array(I);
```

```
         Sort_Array(I) := Sort_Array(Min);
         Sort_Array(Min) := Temp;

         I := I + 1;
   end loop;
end Selection_Sort;


--  ALGORITHMIC PROPERTIES

-- initial_val('I', integer(1)).
-- initial_val('J', integer(2)).
-- initial_val('Min', variable('I')).
-- initial_val('Temp', element(variable('Sort_Array'),variable('I'))).
-- count_var('I', 22).
-- count_var('J', 32).
-- mono_inc(value('I',now), greaterthan, value('I',previous), line(22)).
-- mono_inc(value('J',now), greaterthan, value('J',previous), line(32)).
-- index_var('J', 'Sort_Array').
-- index_var('Min', 'Sort_Array').
-- index_var('I', 'Sort_Array').
-- array_partition('Sort_Array', integer(2), 'J', 32).
-- array_partition('Sort_Array', integer(1), 'I', 22).
```

```
--     AUTOGAP OUTPUT
-- Sun Jun  1 15:01:13 BST 2003


-- File "sqrt.ads", the implementation file for the algorithm
-- that finds the square root of a number. Taken from
-- "Programming in Ada" by Richard Wiener and Richard Sincovec.
-- It is an adaptation of Newtons algorithm.
-- Note that because we don't support subprogram calls, the
-- calls to abs() in the book's implementation are removed.


-- Author: Tommy Ingulfsen, May 2003.

-- Finds the square root of N
procedure Square_Root(N : in Float;
                      Sqrt: out Float)
is
   Eps : Float; -- a very small number
                -- used as sentinel
   N_Max : Integer; -- max no of iterations
   X0 : Float;
   X_Old : Float;
   I : Integer;

begin
   Eps := 0.000001;
   N_Max := 9;

   if N >= 0.0 then

      X0 := N/2.0; -- initial guess for square root
      X_Old := X0; -- save old value

      -- update X0 until it is close to the exact
      -- value of the square root of N

      I := 1;
      loop
         --# assert I<=N_Max;

         if I = N_Max then
            exit;
         end if;

         exit when (X0*X0 - N) <= Eps*N;

         X0 := (X0*X0 + N) / (2.0*X0);
```

```
         exit when (X0 - X_Old) <= Eps*X0;

         X_Old := X0;

         I := I + 1;

      end loop;

      Sqrt := X0;
   end if;

end Square_Root;


--  ALGORITHMIC PROPERTIES

-- initial_val('Sqrt', variable('X0')).
-- initial_val('Eps', integer(1.0E-06)).
-- initial_val('N_Max', integer(9)).
-- initial_val('X0', variable('N')/integer(2.0)).
-- initial_val('X_Old', variable('X0')).
-- initial_val('I', integer(1)).
-- count_var('I', 36).
-- count_var('X0', 36).
-- count_var('X_Old', 36).
-- mono_inc(value('I',now), greaterthan, value('I',previous), line(36)).
-- invar(36, variable('I')<=variable('N_Max')).
```

Appendix G

SPARK subset

This appendix contains the grammar for the subset of SPARK that AUTOGAP uses. As a short summary, the constructs of full SPARK that have been left out are:

- *All of the library system.* Full SPARK contains an elaborate system that allows programmers to organise their code in a modular fashion. Since AUTOGAP is only required to analyse a single program unit there is never any need for more than one source file.
- *Multi-dimensional arrays.* We feel that although arrays of more than one dimension are not unusual in SPARK, AUTOGAP will still be able to analyse a useful subset of all SPARK programs if it only considers arrays of one dimension.
- *Subprogram calls.* It would significantly complicate the analysis of programs if we allowed calls to functions or procedures. We choose to focus on self-contained programs instead.
- *Advanced loop exit conditions.* We assume that the loop exit conditions will contain only a single comparison. That is, no more than one operator is involved.

- *While-loops and for-loops.* The parsing is complicated by the inclusion of all possible loops. Since the SPARK Examiner translates all loops into the basic `loop ... endloop` we chose to use only the basic loop.

G.1 Yacc grammar without actions

Items in capital letters denote lexemes, while a lower case item denotes a rule. To save space we have omitted the C code that builds the parse tree, as that code is very repetitive.

```
proper_body : subprogram_body

numeric_literal : decimal_literal
                | based_literal

decimal_literal : INTEGER_NUMBER
                | REAL_NUMBER


based_literal : BASED_INTEGER
              | BASED_REAL

basic_declaration : object_declaration
                  | full_type_declaration
                  | subtype_declaration



object_declaration : constant_declaration
                   | variable_declaration

constant_declaration :  identifier_list COLON RW_CONSTANT
```

```
                        type_mark BECOMES  expression SEMICOLON

                      | identifier_list COLON RW_CONSTANT
                        BECOMES expression SEMICOLON


variable_declaration : identifier_list COLON type_mark SEMICOLON

                     | identifier_list COLON type_mark BECOMES expression SEMICOLON

                     | identifier_list COLON RW_ALIASED type_mark SEMICOLON

                     | identifier_list COLON RW_ALIASED type_mark
                       BECOMES expression SEMICOLON

identifier_list : identifier_list COMMA IDENTIFIER

                | IDENTIFIER


full_type_declaration :  RW_TYPE IDENTIFIER RW_IS type_definition SEMICOLON


type_definition : enumeration_type_definition

                | integer_type_definition

                | real_type_definition

                | array_type_definition

subtype_declaration :  RW_SUBTYPE IDENTIFIER RW_IS subtype_indication SEMICOLON


subtype_indication :   type_mark constraint

                   | type_mark


type_mark : dotted_simple_name

constraint : range_constraint

           | floating_point_constraint

           | fixed_point_constraint
```

```
          | index_or_discriminant_constraint

range_constraint : RW_RANGE arange

arange : attribute

       | simple_expression DOUBLE_DOT simple_expression

enumeration_type_definition : LEFT_PAREN
                              enumeration_literal_specification COMMA
                              enumeration_type_definition_rep RIGHT_PAREN

                            | LEFT_PAREN
                              enumeration_literal_specification
                              RIGHT_PAREN

integer_type_definition : range_constraint

real_type_definition :  floating_point_constraint

                     | fixed_point_constraint

floating_point_constraint :  floating_accuracy_definition

                          | floating_accuracy_definition range_constraint


floating_accuracy_definition : RW_DIGITS simple_expression



fixed_point_constraint : fixed_accuracy_definition

                       | fixed_accuracy_definition range_constraint


fixed_accuracy_definition : RW_DELTA simple_expression
```

```
array_type_definition : constrained_array_definition

constrained_array_definition : RW_ARRAY index_constraint RW_OF type_mark

index_subtype_definition : type_mark RW_RANGE BOX

index_constraint : LEFT_PAREN index_constraint_rep RIGHT_PAREN

index_constraint_rep : index_constraint_rep COMMA type_mark
                     | type_mark

declarative_part : renaming_declaration_rep
                   initial_declarative_item_rep later_declarative_item_rep
                 | initial_declarative_item_rep later_declarative_item_rep
                 | renaming_declaration_rep initial_declarative_item_rep
                 | initial_declarative_item_rep
                 | renaming_declaration_rep later_declarative_item_rep
                 | later_declarative_item_rep
                 | renaming_declaration_rep

initial_declarative_item_rep :
      initial_declarative_item_rep basic_declarative_item
      | basic_declarative_item

basic_declarative_item :  basic_declaration
                       | representation_clause

basic_proof_declaration : type_assertion
```

```
later_declarative_item_rep :
      later_declarative_item_rep later_declarative_item

    | later_declarative_item_rep apragma

    | later_declarative_item


name : simple_name

     | name LEFT_PAREN name_argument_list RIGHT_PAREN


simple_name : IDENTIFIER


dotted_simple_name : dotted_simple_name POINT IDENTIFIER

                   | IDENTIFIER

prefix :  name

name_argument_list : positional_argument_association

positional_argument_association : expression


attribute : prefix attribute_designator

attribute_designator : attribute_designator ATTRIBUTE_IDENT attribute_designator_opt

                     | ATTRIBUTE_IDENT attribute_designator_opt

attribute_designator_opt : LEFT_PAREN expression RIGHT_PAREN

                         | LEFT_PAREN expression COMMA expression RIGHT_PAREN

                         |

expression : relation
```

```
            | relation RW_AND expression_rep1
            | relation RW_ANDTHEN expression_rep2
            | relation RW_OR expression_rep3
            | relation RW_ORELSE expression_rep4
            | relation RW_XOR expression_rep5

expression_rep1 : expression_rep1 RW_AND relation
                | relation
expression_rep2 : expression_rep2 RW_ANDTHEN relation
                | relation

expression_rep3 : expression_rep3 RW_OR relation
                | relation

expression_rep4 : expression_rep4 RW_ORELSE relation
                | relation

expression_rep5 : expression_rep5 RW_XOR relation
                | relation



relation : simple_expression
         | simple_expression relational_operator simple_expression
         | simple_expression inside arange
         | simple_expression outside arange
         | simple_expression inside name
         | simple_expression outside name
```

```
inside : RW_IN

outside : RW_NOTIN

simple_expression : simple_expression binary_adding_operator term
                  | simple_expression_opt


simple_expression_opt : term
                      | unary_adding_operator term


term : term multiplying_operator factor
     | factor



factor : primary
       | primary DOUBLE_STAR primary
       | RW_ABS primary
       | RW_NOT primary
primary : numeric_literal
        | CHARACTER_LITERAL
        | STRING_LITERAL
        | name
        | qualified_expression
        | LEFT_PAREN expression RIGHT_PAREN
       | attribute
```

```
relational_operator : EQUALS
                    | NOT_EQUAL
                    | LESS_THAN
                    | LESS_OR_EQUAL
                    | GREATER_THAN
                    | GREATER_OR_EQUAL


binary_adding_operator : PLUS
                       | MINUS
                       | AMPERSAND


unary_adding_operator : PLUS
                      | MINUS

multiplying_operator : MULTIPLY
                     | DIVIDE
                     | RW_MOD
                     | RW_REM

qualified_expression : name APOSTROPHE aggregate

sequence_of_statements : sequence_of_statements statement
                       | statement
```

```
statement : simple_statement

          | compound_statement


simple_statement : null_statement

                 | assignment_statement

                 | procedure_call_statement
                 | exit_statement

                 | return_statement




compound_statement : if_statement

                   | loop_statement


null_statement : RW_NULL SEMICOLON


assignment_statement :  name BECOMES expression SEMICOLON



if_statement : RW_IF condition RW_THEN sequence_of_statements
              elsif_part else_part RW_END RW_IF SEMICOLON


elsif_part: elsif_part RW_ELSIF condition RW_THEN sequence_of_statements

            |


else_part: RW_ELSE sequence_of_statements

           |
```

```
condition : expression


loop_statement : loop_statement_opt RW_LOOP loop_invariant
                 sequence_of_statements end_of_loop RW_END
                 RW_LOOP SEMICOLON

               | simple_name COLON loop_statement_opt RW_LOOP sequence_of_statements
                 end_of_loop RW_END RW_LOOP simple_name SEMICOLON



loop_statement_opt : iteration_scheme

                   | iteration_scheme loop_invariant

                   |



loop_parameter_specification : IDENTIFIER forward type_mark

                             | IDENTIFIER forward type_mark RW_RANGE arange

                             | IDENTIFIER backward type_mark

                             | IDENTIFIER backward type_mark RW_RANGE arange

forward : RW_IN

backward :RW_IN RW_REVERSE

loop_invariant : assert_statement


exit_statement : RW_EXIT SEMICOLON

               | RW_EXIT RW_WHEN condition SEMICOLON

               | RW_EXIT simple_name SEMICOLON

               | RW_EXIT simple_name RW_WHEN condition SEMICOLON
```

```
return_statement : RW_RETURN expression SEMICOLON


assert_statement : ANNOTATION_START RW_ASSERT RW_TRUE SEMICOLON


subprogram_specification : procedure_specification
                         | function_specification


procedure_specification : RW_PROCEDURE IDENTIFIER
                        | RW_PROCEDURE IDENTIFIER formal_part


function_specification : RW_FUNCTION designator formal_part RW_RETURN type_mark
                       | RW_FUNCTION designator RW_RETURN type_mark


designator : IDENTIFIER



formal_part : LEFT_PAREN formal_part_rep RIGHT_PAREN

formal_part_rep : formal_part_rep SEMICOLON parameter_specification
                | parameter_specification


parameter_specification :  identifier_list COLON mode type_mark

mode : in_mode
```

```
      | inout_mode

      | outmode

      |


in_mode : RW_IN


inout_mode : RW_IN RW_OUT


outmode : RW_OUT

subprogram_body : procedure_specification RW_IS subprogram_implementation

                | function_specification RW_IS subprogram_implementation



subprogram_implementation : pragma_rep declarative_part
                            RW_BEGIN sequence_of_statements
                            RW_END designator SEMICOLON


                          | pragma_rep RW_BEGIN
                            sequence_of_statements RW_END
                            designator SEMICOLON

                          | pragma_rep RW_BEGIN
                            code_insertion RW_END
                            designator SEMICOLON



hidden_part : HIDE_DIRECTIVE
```