Tools for the construction of correct programs : an overview

Eindhoven University of Technology
Department of Mathematics and Computing Science

Tools for the Construction of Correct Programs: an Overview

by

Michael Franssen

97/06

ISSN 0926-4515

editors: prof.dr. R.C. Backhouse
prof.dr. J.C.M. Baeten

Reports are available at:
http://www.win.tue.nl/win/cs

Computing Science Reports 97/06
Eindhoven, April 1997

Tools for the Construction of Correct Programs: an Overview

Michael Franssen

March 17, 1997

Abstract

In this document we describe the notion of correct programs, methods for the construction of correct programs and currently available tools for the construction of correct programs. Several aspects of the tools for the construction of correct programs are compared. The results of this overview are used to formulate requirements for a programming tool.

This research is the first part of the SOBU-project: 'A programming environment for the construction of correct programs'. The goal of this project is to develop an interactive programming environment for the creation of correct software in a Dijkstra/Hoare like fashion.

Contents

Introduction

This report presents the results of studies performed as a prelude to the development of an environment for deriving correct programs. Goal of this project is to create a programming environment which supports derivation of programs in a Dijkstra/Hoare- like fashion. In this approach to programming, a program is constructed simultaneously with its proof of correctness. The steps taken in this derivation are motivated by the requirements of the proof. In our opinion there are no tools yet that sufficiently support this style of programming.

In order to design a tool that does not suffer from the shortcomings of other tools, we started by studying existing tools. In this report we present an overview containing the results of our studies. We do not claim that the overview presented here is complete. Many of the listed systems are still under development and their features may change in the future. Also, we discuss only those features that we think are important to our project.

The report is constructed as follows:

First we introduce the notion of correctness of a program in chapter 1. The programming languages are divided into three categories: functional languages; imperative languages (including object oriented languages); and logical languages. For each language type it is shown how correctness of a program can be proved.

Next, we discuss three methods to obtain correct programs in chapter 2 . These methods are verification of programs; derivation of programs; and extraction of programs (described in section 2.1, section 2.2 and section 2.3 respectively). For verification and derivation of programs we also discuss a few existing tools in detail in order to give an impression of the current level of support for these methods of programming. For verification of programs we discuss an interactive tool (see 2.1.1) and a tool implemented within a general theorem prover (see 2.1.2). Derivation of programs is illustrated by the refinement calculus (see 2.2.1) and an automated program synthesizer (see 2.2.2). We also explain why extraction of programs is not suitable for our purposes.

Finally, in chapter 3, we give a brief overview of existing tools which may be used by programmers. These tools are divided into three categories: theorem provers; generic environments; and programming tools.

Chapter 1

Correct Programs

Correct programming, in contrast to conventional programming, requires a proof of correctness of the program. However, before we can speak of a correct program, we must define what is meant by correctness. Exactly this is done in this first chapter: we define the notion of a correct program. Since programs can be written in different kinds of languages we will also state in more detail how correctness can be proved for each kind of language. How to *obtain* a correct program and the corresponding proof is explained in the next chapter.

Every program that is written has to compute from some given input an entity that the programmer has in mind. The characteristics of the entity and the given input can be formally captured in logical formulas, called the program specification. The formula that captures the characteristics of the input is called the precondition and the formula capturing the characteristics of the required entity is called the postcondition.

A program is called *partially* correct if for any input satisfying the precondition, the program returns an entity satisfying the postcondition, *provided that it terminates.* A program is called *totally* correct if it is partially correct and it is *guaranteed to terminate* for any given input satisfying the precondition.

Unfortunately there are no formalisms yet that allow to specify the required efficiency of programs. Efficiency of a program is often determined after the program has been made, but not proved formally. This is a serious shortcoming of the formalisms: in 'real' programming the main concern of programmers is efficiency. For instance: to solve the traveling salesman problem one is willing to accept non-optimal solutions, if they can be computed efficiently and if they are not too bad approximations of the optimal solution.

Programs are written in a wide variety of languages. The number of programming languages and the features they offer is still increasing. However, all these languages are based on just three paradigms and therefore divided into three categories: functional languages, imperative languages and logical languages (see below for a note on object oriented languages). The intention of what correct programs are remains the same for all categories, but the way in which correctness is proved differs. In the following sections we will explain how correctness is proved for each language type.

Note on Object Oriented Languages

Some people may consider object oriented languages as a distinct category of languages. In this document however, we will consider these languages to be imperative languages with a special feature: classes as data types.

The class to which an object belongs can be seen as a special kind of record. This special record can contain functions and procedures[1], whereas a normal record can only hold variables. A child of such a class (i.e. a class that inherits from a previously defined class) can be seen as a record which has at least the same fields as its parent (or contains one dummy-variable of the parent-type).

[1] These functions and procedures are called methods in object oriented programming literature.

Even though objects are very handy for implementing many constructs and support a good modular approach in building software, it can be seen as a syntactic encoding of a construct that otherwise always had to be repeated by the programmer: A class can be replaced by a record type and a set of functions and procedures. The record type contains all data-fields of the original class, even if they are inherited. The methods of the class (also the inherited methods) are encoded by a set of functions and procedures that have an extra parameter with as type the record associated with the original class.

The advantage of the object oriented feature of the language is then mainly that the inherited methods of a class do not have to be re-implemented for the new class in order to maintain the original type of this new class.

What also is considered to be different in object oriented programming is communication between objects by message passing. However, in reality communication is not the right description for the passing of data between objects: data are passed to methods in the form of parameters, just like parameters are passed to functions and procedures in conventional imperative languages. The method called then first has to terminate before the calling object can proceed with its work, again like in a procedure- or function call. An exception to this rule is Looks^{tm}, the language of the Generalized Display Processor developed at the Eindhoven University of Technology for the DenK-project (see [Pee95]). This language does allow messages to be passed by calling a method of an object without waiting until termination of the method called. However, the mechanism used there could also be applied to languages without objects.

On the other hand, object oriented methods have led to different design strategies. A design made according to these strategies is hard to implement in languages that do not support object oriented programming. Also, the concept of inheritance requires different approaches to proofs of correctness of programs. In our project we aim at a tool for deriving correct programs in classical imperative languages. The concepts of object oriented programming are an interesting extension, but are beyond the scope of our initial goals.

1.1 Correctness of Functional Programs

In the functional programming paradigm a program is a function. The function takes the given input as parameters and returns the entity one wants to compute. Because this paradigm allows normal mathematical functions to be used as executable programs, correct programming is often easy. This is demonstrated by the following example:

Let F be a function that computes from a natural number i the i'th Fibonacci number. Then F is defined by:

$$\begin{array}{lcl} F.0 & = & 0 \\ F.1 & = & 1 \\ F.i & = & F.(i-1) + F.(i-2) \quad 2 \leq i \end{array}$$

A functional program f that computes the function F can now easily be written. For instance take the following program:

$$\begin{array}{l} \mathit{letrec}\ f \equiv \lambda i : \mathit{int}.\ \mathit{if}\ i = 0\ \mathit{then}\ 0 \\ \qquad\qquad\qquad\qquad\quad \mathit{else\ if}\ i = 1\ \mathit{then}\ 1 \\ \qquad\qquad\qquad\qquad\quad \mathit{else}\ f.(i-1) + f.(i-2) \end{array}$$

In this notation *letrec* binds a name to a (possibly) recursive function, λ is used to create a function, followed by the input parameter. After the first dot the actual function definition is placed.

The order in which $f.(i-1)$ and $f.(i-2)$ are computed is irrelevant. This also holds if more functions are used in more interacting ways. The mathematical notion of function is that a function applied to some arguments is just another encoding of its value. Computing this value is equal to replacing the applied function (one value) by a simpler notation of the same value. Since this notion is adopted by functional languages the order of evaluating functions is arbitrary.

However, implementations always use a fixed order, but this is transparent to the user of the language.

The correctness of the above function is evident, so we will use it as an example on how to actually prove the correctness. To prove correctness we use the specification, which we will construct first. The Fibonacci numbers are only defined for natural numbers, while the program takes an integer as parameter. Therefore our precondition will be $0 \leq i$. The result of the function must be the i'th Fibonacci number ($F.i$), so the postcondition is $f.i = F.i$. The program must compute the correct Fibonacci number for every input satisfying the precondition, hence the program specification can be denoted as:

$$\forall i : int.(0 \leq i \Rightarrow f.i = F.i)$$

Proving this proposition is then equivalent to proving partial correctness of the program f. Proving can easily be done by induction on i.

In order to prove the total correctness of f we can (in this case) suffice by noting that for a correct input value i computation of $f.i$ only requires computing a finite number of $f.j$ for j lower than i, but never below 0.

Although this program is correct, it is not efficient. For instance: to compute $f.5$ the program computes $f.4$ and $f.3$. To compute $f.4$ it again computes $f.3$, hence $f.3$ is computed twice.

In functional programming there are techniques to transform the program f into a more efficient one. These techniques use mathematical reasoning about equality of functions and often result in more complex, but equivalent functions. The advantage of these derived functions is that they are designed in such a way that they can be computed more efficiently. We will not discuss functional programming techniques here, but only use some results of their application.

For instance, applying these techniques to f could result in the auxiliary function g, that takes three integers as parameters. The specification of g is given by:

$$\forall i : int.(0 \leq i \Rightarrow g.0.1.i = F.i)$$

A program for g could be:

$$letrec\ g \equiv \lambda a, b, c : int.\ \ if\ c = 0\ then\ a \\ else\ g.b.(a + b).(c - 1)$$

g contains only one recursive call, which also is the last function call in each computation step. Such a function is called tail-recursive and can be implemented more efficiently than arbitrary recursive functions. That g is more efficient than f follows from the fact that every result of g is computed only once, while the 'depth' of the recursion is bounded by i, like was the case for f.

Once we have function g we can replace f by $f \equiv \lambda i : int.(g.0.1.i)$. However, to prove the correctness of g with respect to its specification is more difficult. A nice way to do this is first to prove that for g a stronger specification holds, which is obtained by applying functional programming techniques to f:

$$\forall i : int.\forall j : int.(0 \leq j \Rightarrow g.(F.i).(F.(i + 1)).j = F.(i + j))$$

The original specification is then the special case when $i = 0$.

The above example sketches how functional programs are developed and how their correctness is proved. In functional languages one often starts with a trivially correct (but inefficient) function and then transforms it step by step into a more efficient one, possibly using auxiliary functions. Proving partial correctness can be done by proving a single proposition: if the input satisfies the precondition then the result satisfies the postcondition. To prove total correctness it is shown that the recursion is well-founded.

1.2 Correctness of Imperative Programs

The imperative paradigm exploits the semantic concept of a state machine. The machine contains a set of variables that can hold values. A state is described by an assignment of values to variables and can change in the course of time.

A program is built out of statements. In contrast to functional programs, imperative programs have to be executed in a fixed order. Operationally one can think of a program counter that refers to the statement of the program that is to be executed next. Statements come in two flavors: statements to change the state and statements to describe the order in which the machine must execute the program.

Imperative languages are the most common languages. This is probably because they can be interpreted very operationally. Programmers with little or no knowledge about mathematics or logic (hobbyists) can easily imagine a computer putting values into variables and proceeding with the next action. Functional languages require much knowledge of mathematics. Logical languages can probably be read by hobbyists, but when writing them they will often encounter problems getting their programs to terminate.

When proving correctness of imperative programs one starts with annotating the program. This annotating is done by inserting logical propositions as comment between the statements. These formal comments are called assertions. The idea behind the assertions is that before the machine executes a statement the assertion before this statement, and after execution of the assignment the assertion behind this statement, must be true.

The program specification is now given by two assertions: one at the very beginning of the program (the precondition) and one at the end (the postcondition). The precondition states that certain variables contain the input values and the postcondition states that some variables contain the computed entity.

Consider an imperative version of the program g of the previous section:

$\{0 \leq i\}$
$a := 0;$
$b := 1;$
$j := 0;$
$\{(0 \leq j \leq i) \wedge (a = F.j) \wedge (b = F.(j+1))\}$
while $j \neq i$ **do**
 $r := a;$ $\{(0 \leq j < i) \wedge (a = F.j) \wedge (b = F.(j+1)) \wedge (r = F.j)\}$
 $a := b;$ $\{(0 \leq j < i) \wedge (a = F.(j+1)) \wedge (b = F.(j+1)) \wedge (r = F.j)\}$
 $b := a + r;$ $\{(0 \leq j < i) \wedge (a = F.(j+1)) \wedge (b = F.(j+2))\}$
 $j := j + 1$ $\{(0 \leq j \leq i) \wedge (a = F.j) \wedge (b = F.(j+1))\}$
od;
$\{(j = i) \wedge (0 \leq j \leq i) \wedge (a = F.j) \wedge (b = F.(j+1))\}$
$r := a$
$\{r = F.i\}$

The assertion at the beginning of the program claims that program-variable i contains a value at least zero[2]. The assertion at the end claims that the required entity ($F.i$) is stored in variable r. We will not show the proof of correctness of all the assertions, but concentrate on the most important ones.

The assertion before the **while**-statement and the last assertion within its body are exactly the same: $\texttt{inv} \equiv (0 \leq j \leq i) \wedge (a = F.j) \wedge (b = F.(j+1))$. However, the first occurrence of **inv** can be strengthened with $j = 0$, while for the latter one we are certain (by the statement $j := j + 1$) that $j \neq 0$. Because execution of the loop body obviously does not change the truth of this assertion, **inv** is called an invariant of the loop.

It is easy to see that indeed the invariant **inv** is true right before the **while**-statement: the values of a, b and j are $0, 1$ and 0 respectively. If we substitute these values in **inv** we have $(0 \leq 0 \leq i) \wedge (0 = F.0) \wedge (1 = F.1))$, which follows from the initial condition $0 \leq i$ and the definition of F.

Whenever the loop body is executed we know that the invariant is true, since it was true initially and its truth was not destroyed by any previous execution of the loop body. (This argument is

[2] We will assume for now that all variables are declared and have type integer.

similar to induction: if the loop body did not destroy the truth of the invariant during any previous execution, it will not destroy its truth during this execution.)

Also we know that in addition to the invariant, we have $j \neq i$ at the beginning of the execution of the loop body, since this condition was checked in order to enable the current execution. After some manipulation with r, a and b, the value of j is increased. After this increment operation the invariant must hold again. Now if the invariant must hold for the value j after it has been incremented by one, it must hold for $j+1$ right before this increment is performed. In general, we have for *any* assignment $x := e$ of a value e to a variable x that $P[x := e]$ must hold *before* the assignment in order for P to hold *after* the assignment[3], or, otherwise said, $\{P[x := e]\}x := e\{P\}$.

To prove correctness of the invariant we now prove as an example that

$$(0 \leq j \leq i) \wedge (a = F.j) \wedge (b = F.(j+1))$$

holds after the assignment $j := j + 1$, by proving that

$$(0 \leq j + 1 \leq i) \wedge (a = F.(j+1)) \wedge (b = F.(j+1+1))$$

held before this assignment[4]:

$$\begin{array}{lll} & (0 \leq j+1 \leq i) \wedge (a = F.(j+1)) \wedge (b = F.(j+1+1)) & (1) \\ \equiv & \{j+1+1 = j+2, (0 \leq j+1 \leq i) \equiv (-1 \leq j \leq i-1)\} & (A) \\ & (-1 \leq j \leq i-1) \wedge (a = F.(j+1) \wedge (b = F.(j+2)) & (2) \\ \Leftarrow & \{(0 \leq j < i) \Rightarrow (-1 \leq j \leq i-1)\} & (B) \\ & (0 \leq j < i) \wedge (a = F.(j+1)) \wedge (b = F.(j+2)) & (3) \end{array}$$

The last proposition is exactly equal to the assertion right before the assignment that increments j and therefore the invariant indeed holds after the increment of j.

When the loop terminates, the invariant still holds. In addition to the invariant the negation of the guard of the loop also holds, since otherwise the loop would not have terminated. These two facts are denoted in the assertion behind the **while**-statement. Finally we can prove the correctness of the postcondition, by proving that the assertion behind the **while**-statement implies $(r = F.i)[r := a]$. This last proof is trivial, since $a = F.j$ and $i = j$ are conjuncts of the assertion before the assignment $r := a$.

The total correctness of this program follows from the fact that when the loop body is executed the expression $i - j$ is greater than zero, but decreases as an effect of the body-execution. Since a positive expression of type integer cannot decrease infinitely, the loop will terminate. An expression like $i - j$ is an upper bound to the number of iterations of the loop and is therefore called a bound function.

1.3 Correctness of Logical Programs

In logical languages programs are sets of logical formulas. These formulas can contain free variables that are implicitly universally quantified. Moreover, all formulas have the same form: a conjunction of several positive literals that imply the truth of yet another literal. In a formula:

$$A_1 \wedge \ldots \wedge A_n \Rightarrow B$$

The A_i are called the *antecedents* of the formula, B is the *succeedent*. These kind of formulas are called Horn-clauses[5]. A logical program states a certain amount of 'knowledge'.

Executing a logical program is done by asking questions. A question is formulated by a literal. If this literal does not contain any free variables the answer to the question is a simple 'yes' or

[3] To some people this gives rise to some confusion, since the axiom $\{P\}x := e\{P[x := e]\}$ looks so natural. That this axiom is wrong is shown by the following simple example: $\{x = 0\}x := 1\{1 = 0\}$

[4] The format below means that (1) $\equiv$ (2) because of (A) and that (2) $\Leftarrow$ (3) by (B); hence: (3) $\Rightarrow$ (1)

[5] The Prolog denotation of $A_1 \wedge \ldots \wedge A_n \Rightarrow B$ is: $B : -A_1, \ldots, A_n$.

'no'. The answer should be 'yes' if the truth of the literal can be derived from the given knowledge and 'no' otherwise. If the literal contains free variables the answer consists of a set of formulas representing 'solutions' to the question. These solutions are calculated by executing all possible substitutions of values for the variables in such a way, that the closed proposition obtained by this substitution is derivable from the program.

Since the answer to a question is derived from the knowledge according to formal logic, every logical program is partially correct. Hence logical programming provides correctness by definition. But does the language also provide total correctness automatically?

To provide total correctness automatically, a logical program always has to terminate. If deriving an answer to the question asked were decidable, this would be the case: a logical program would then *always* provide a correct answer for any given problem. In particular, the language would then be able to decide the halting problem of Turing machines, for which it is known that it is undecidable. Therefore, no useful logical language will ever provide total correctness automatically, now or in the future. The programmer still personally has the task to provide for total correctness.

In our example, the definition of F itself can be denoted as a program. However, we have to code numerical computations as literals: instead of a function F we get a literal $F(i, X)$, that holds whenever X is the ith Fibonacci number. The program in pseudo-Prolog is denoted as follows:

$$F(0,0).$$
$$F(1,1).$$
$$F(i+2, X) \Leftarrow F(i+1, A), F(i, B), X = A + B.$$

The question to compute the nth Fibonacci number is then $F(n, X)$ for given $n \geq 0$. The computed answer is $X = N$, where N is the correct Fibonacci number.

Since the program *is* the specification, partial correctness is evident. This example program is also totally correct: for any (sub-)question no more than one rule can be applied and in only one way. From the fact that the conclusion of the formulas always has a first argument that is greater than the first arguments of the premises, we can conclude that repetitive application of the rules is well-founded. This means that for every question $F(n, X)$ for given n there is only one possible derivation. This derivation leads also to the correct substitution for X. Hence the computation will always terminate with the correct answer.

When searching for an answer to the question asked, the logical language tries to use the program rules in a fixed order. If a derivation set up in this order fails, the language cancels the last step taken in the derivation and tries to apply the program rule that comes next. This process is called backtracking or exhaustive search.

Applying a formula from the program is done in a backward style. To derive a question using a program formula, a substitution is computed that makes the question equal to the succedent of that program formula. This substitution is then also applied to the antecedents of the formula used. In order to obtain a derivation for the original question we now have to find derivations for the premises. Therefore, the premises can be regarded as sub-questions.

If there are finitely many possible derivations the program will always terminate, but if there are infinitely many derivations computation may not terminate. The programmer then has to state the program rules in such an order that if an answer to a question exists for some instance of the free variables, the corresponding derivation is found in finite time. As a consequence, the programmer has to know exactly in what order the program is used to construct a derivation for the question.

To actually prove termination of logical programs is very hard. By the nature of the execution of logical programs every termination proof relies on the order of execution of the separate formulas. Some proving-techniques are given in [Apt97]. These techniques are based on well-foundedness of the constructed derivation trees.

From the above we see that a logical program is easier read than written. When reading a logical program only the logical contents of the formulas matters. From this contents we can conclude what the program should do. If the program is totally correct, the execution will always

provide an answer, which by definition is correct. If we write a program we cannot concentrate solely on the logical contents of the rules, but we also have to be concerned with the order in which the rules are used when an answer is computed.

Chapter 2

Correct Programming

There are three approaches that can be used to obtain a program that is correct with respect to its specification. Theories have been developed for all approaches. Also, several tools have been built that support programming by implementing these theories. In the following sections we elaborate on each of the methods and their supporting tools and summarize their advantages and disadvantages.

However, before we elaborate on the different methods we first introduce them and give an impression of how they are used. The methods are:

1. First write a program and then prove its correctness.
2. Use the specification to derive a program and its proof of correctness simultaneously.
3. First derive a constructive proof of the existence of the entity we want to compute and then extract a program from this proof.

The first method is used the most. We first have to write a program, using not much more than our intuition. Then we write the program specification and finally we prove the correctness of the program. An elaboration of this method can be found in section 2.1. We will refer to it as 'verification of programs'.

If we take the second approach we start with a formal specification of the entire program. Next, we rewrite this specification in such a way that parts of it can easily be met (or established) by a simple program. This leaves us with the problem of meeting (or establishing) the rest of the specification, which we attack the same way as we attacked the entire programming problem. We will call this method 'derivation of programs' and give a more detailed explanation in section 2.2.

The third method is the most abstract one. To obtain a program that computes an entity with certain specification, we construct a constructive proof of the existence of such entity. Since the proof is constructive, a proof of existence of an entity contains an algorithm that actually computes this entity. A program implementing this algorithm is then extracted from this proof automatically. We describe this method in section 2.3 and refer to it by 'extraction of programs'.

2.1 Verification of Programs

If we use verification of programs as our programming strategy, we first write a program according to a specification we only have in mind. Next we want to verify if the written program is correct with respect to our intuitive specification. For this we have to formalize the intuitive specification. During the formalization process, we keep one eye at the program and try to write the specification in such a way, that proving correctness of the program is easy in the end. Finally we can prove correctness of the program using the theories related to the programming language. (This last step is omitted far too often.)

We have already demonstrated the verification of programs in the previous chapter. The programs to compute the i'th Fibonacci number were given before they were formally specified or proved. When we proved the correctness of programs it was done "by hand", which means that even a program which we proved to be correct can be wrong due to errors in our proof. After all we can make errors in a proof just like we can make errors in a program.

We can prevent errors in our proofs, by using mechanical tools to verify our proofs. Such tools exist in many forms and are based on a variety of formal systems. In this section we will discuss two tools, that not only verify our proofs, but also assist in constructing them. Besides, both tools support a programming language. For this supported language they can automatically compute the proof-obligations that guarantee correctness of a program written in this language. These proof-obligations are called verification conditions and therefore we call these tools 'verification conditions generators' (VCG's).

2.1.1 An Interactive Tool for Verification of Programs

Our first example VCG is described in chapter 10 of [RT89]. Although the authors only use it to demonstrate a generator for interactive programming environments, it is a good example. It demonstrates all elements required for the verification of programs.

First the authors define a simple imperative programming language, consisting of the most important statements: skip, assignment, if-then-else and while-do. Variables do not need to be declared and are always of type integer. All expressions of the language are valid, since there are no operations like *div* that require the second argument to be nonzero. The tool will support the programmer in verifying programs written in this language.

Next the authors of [RT89] define an assertion language, in which the programmer has to write the program specification. This is necessary, because in order to support verification the tool has to understand the specification and hence it must be written in a pre-defined assertion language. In this VCG the assertion language consists of terms of first order predicate logic. In this logic we have the usual connectives $\wedge$, $\vee$, $\Rightarrow$ as well as universal and existential quantifications over elements of a set. In fact, we only have quantifications over the set of integers, due to the simplicity of the programming language. As we described in the previous chapter, the specification, written in the assertion language, is added as formal comment to the program. This comment is called annotation.

The programming language and the assertion language are not mutually independent. There is only one grammar, which describes annotated programs. This grammar is constructed in such a way, that boolean expressions in the programming language are a sub-language of the assertion language. The assertion language is more powerful in the sense that it has quantifications. The advantage of this approach is that expressions that occur as a condition in the programming language can directly be used within the assertion language. We have seen this in the example of section 1.2, where the loop-condition $j \neq i$ occurred negated as $j = i$ in the assertion following the while-statement.

The programmer constructs annotated programs with a structure-editor that is generated automatically by the environment generator[1]. The annotation occurs at fixed places in the program code: within loops (while-statements) and at the beginning and the end of the program. Even though one is allowed to edit the assertions at any time of the editing session, one is not guided in the construction of the program by the form of these assertions. We will see later that this is a fundamental difference with derivation of programs, where the specification leads the way in constructing the program.

This tool automatically computes the verification conditions as soon as the programmer has a complete program. The algorithm to compute the verification conditions is explicitly encoded in the input for the environment generator. All theory about the correctness of these conditions is considered Meta-theory and has to be proved manually, outside the system.

[1] Note that the entire tool was used to demonstrate a generator for programming environments.

We will demonstrate this with the example program for Fibonacci numbers from chapter 1 on page 7. First, we introduce an abbreviation for the invariant, since the verification conditions are based upon it: $P \equiv (0 \leq j \leq i) \wedge (a = F.j) \wedge (b = F.(j+1))$. We then get three verification conditions:

1. $0 \leq i \Rightarrow (P[j := 0][b := 1][a := 0])$
2. $(P \wedge j \neq i) \Rightarrow (P[j := j+1][b := a+r][a := b][r := a])$
3. $(P \wedge \neg(j \neq i)) \Rightarrow ((r = F.i)[r := a])$

Verification condition 1 claims that the invariant is valid right before execution of the loop. Verification condition 2 claims that the invariant is indeed invariant under execution of the loop body and verification condition 3 claims that upon termination of the program the postcondition holds.

Instead of creating a verification condition for every assignment, the tool computes a single verification condition for a serie of statements, by performing a serie of substitutions on the assertions. (See $P[j := 0][b := 1][a := 0]$ in verification condition 1 above.) These series of statements may consist of skip-, assignment-, and selection-statements. For a loop this approach would result in a proposition, that is a fix-point of an equation of propositions. This equation of propositions stems from the equality of the following program parts:

$$\textbf{while } G \textbf{ do } S \textbf{ od}$$
$$\equiv$$
$$\textbf{if } G \textbf{ then } S;\ \textbf{while } G \textbf{ do } S \textbf{ od else skip fi}$$

Because a fix-point of an equality of propositions is not always computable, the fix-point should be explicitly present in the verification conditions. However, programmers do not want to have a verification condition that contains a fix-point. Therefore, the tool requires that the invariants of the loops are specified explicitly. It then uses these invariants to compute verification conditions that do not contain fix points. This eases the work of the programmer, since constructing proofs for propositions that are fix points is everything but easy.

After the verification conditions are computed, the programmer has to prove them manually. This kind of proving is supported by the system as follows: proofs are represented as derivations in a Gentzen-system. The authors have implemented this by creating syntactic terms for each rule of the Gentzen-system. Every syntactic term then corresponds to a legal derivation in the Gentzen system. The programmer can use the structure editor to edit these terms and consequently, to prove the verification conditions.

The system also provides the programmer with an automated theorem prover (ATP) to prove parts of the verification conditions. This ATP is specified in the input file for the environment generator. The ATP attempts to create a derivation for the condition the programmer has to prove when invoking the ATP. Because this ATP produces a syntactic term corresponding to a derivation in the Gentzen system, every constructed proof will be correct.

Since it is not decidable whether or not a first order predicate formula can be proved, the ATP will not always be successful. It may fail, even if the formula one wants to prove is true. Matters become worse if the assertion language is more powerful, like second, or even higher order logic. For these logics, good algorithms for automated theorem proving have yet to be found.

Now that we have discussed the entire tool, it is time to summarize its strong and weak points. We start with the strong ones:

- The editor is user-friendly. We have a single editor to manipulate programs, assertions and proofs.
- Proving is made easier by the built-in automated theorem prover. Using an ATP is feasible because all assertions are in first order predicate logic.
- We only have to learn a restricted kind of expressions, since boolean expressions in the programming language are immediately usable within the assertion language.

The weak points of the system are mainly due to consequences of its extreme simplicity:

- The amount of assertions within the program is fixed and small. Assertions only occur at the beginning and end of our program and at the beginning of a loop. When we read a program we want to have more comments, like assertions at arbitrary places in the program. Comments then immediately tell us what we (hope to) have achieved at this point of the program.
- The system requires from the programmer to annotate each loop with an invariant, but it does not help him or her to obtain one. For programmers this is problematic, since good invariants are hard to construct. Some aid in this process would be more than appreciated.
- Also, the programming language is too limited. For any real programming problem one needs a more powerful language. Since the programming environment is generated automatically by the synthesizer generator of [RT89], we can relatively easy extend the programming language. But beware: for every extension we make to the programming language, we will also need new inference rules to compute the corresponding verification conditions. Creating inference rules is a delicate job: in the past, inference rules have been proposed that turned out to be incorrect (see [GL80, GHL78]).

2.1.2 Verification of Programs within a General Theorem Prover

The second VCG we discuss is found in [HM96]. It is designed to verify mutually recursive procedures. It also demonstrates how a VCG can be embedded in a general purpose theorem prover. Homeier and Martin used HOL, the Higher Order Logic theorem prover.

In a general purpose theorem prover one language already exists: a language to formalize mathematical concepts and reason about them. Within this language one can introduce sets, elements of sets, functions on sets and even axioms. Together these elements form the theory one wants to reason about. For instance, one can introduce the class of boolean expressions (**bool**) and its syntactic constructors like **true** : **bool**, **false** : **bool**, **or** : **bool** $\rightarrow$ **bool** $\rightarrow$ **bool** and **and** : **bool** $\rightarrow$ **bool** $\rightarrow$ **bool**. Next, axioms defining the properties of **not** and **and** have to be declared. As soon as the theory is defined, it is possible to prove propositions like $(\forall x : \textbf{bool} \bullet (x \textbf{ and } x) = (x \textbf{ or } x))$. The theorem prover checks that the results are indeed derivable from the theory defined within the logic on which the theorem prover is based.

Homeier and Martin define their programming language exactly in this way. First they introduce sets that represent programs, boolean expressions, integers etc, along with the corresponding syntactic constructors like:

$$\begin{array}{lcl} \textbf{skip} & : & \textbf{program} \\ \leq & : & \textbf{int} \rightarrow \textbf{int} \rightarrow \textbf{bool} \\ \textbf{if} & : & \textbf{bool} \rightarrow \textbf{prog} \rightarrow \textbf{prog} \rightarrow \textbf{prog} \end{array}$$

The programming language also supports procedures with value and result parameters. Like in the first VCG, variables are not introduced and always have type integer. However, this time the language also has an increment operator $++$, that is treated like an expression. For any variable x the value of $++x$ is just the value of x, but as a side-effect the value of x is incremented by one. This special construct has its impact on the semantics of the language: The state can change as an effect of evaluating an expression, while usually the state can only change when executing a statement.

The assertion language and the programming language are introduced simultaneously. This is necessary, because while-statements require an invariant written in the assertion language (the while construction is defined as **while** : **assert** $\rightarrow$ **bool** $\rightarrow$ **prog** $\rightarrow$ **prog**). Also, procedures require a pre- and postcondition. Basically, the assertion language is first order predicate logic. The assertion language is independent from the programming language: we cannot directly use boolean expressions of the programming language within our assertions. Whenever we need an expression of the program in an assertion, we explicitly have to translate it. We can assure that

a translation is correct by proving soundness of the translation within the theorem prover, which is possible because all the theory needed for this proof is defined *within* the theorem prover.

Also, Homeier et al define the programming language semantics within the theorem prover. This is necessary, since the program correctness depends on these semantics. To compute verification conditions axiomatic semantics are needed . Axiomatic semantics specify the correctness of the annotation of a program directly, but it is a delicate job to define the correct axioms. In the past axiomatic semantics have been proposed for procedures, that later turned out to be wrong (see [GL80, GHL78]). Therefore an indirect approach is taken: first, Homeier et al use structural operational semantics to specify the language semantics. Structural operational semantics merely define the effect of executing a program on the machine state. These semantics are less error-prone, because they are more intuitive to a language designer than axiomatic semantics. Next, the axioms required by the VCG are defined as theorems and finally these are proved to be correct by using the theorem prover. This is possible because all theory required for this proof is also defined within the theorem prover. This indirect approach guarantees the correctness of the axioms, even for the complex construct of mutual recursive procedures.

The Verification Condition Generator itself is programmed as a proof-tactic in HOL. This tactic requires two parameters to work: the program one wants to verify and the postcondition it should establish. Except for the verification conditions, this VCG also produces the precondition of the main program.

The only assertions the programmer can put in his program are loop-invariants and the pre- and postconditions of procedures. These assertions are not optional, but they are needed to be able to compute the verification conditions. Without these assertions, computing the verification conditions would become difficult and also the verification conditions would contain fix points. The postcondition of the main program is given when calling the VCG and the precondition cannot be specified by the programmer at all.

Also, the programmer is not supported in writing the program. The program has to be entered into the system as one large HOL-term, just like any other syntactic term defined within HOL. This is also the case for assertions: the assertions within the program, like the postcondition, are syntactic terms within HOL and have to be entered as a whole.

After the programmer applied the VCG-tactic to his program and the corresponding postcondition, he can directly prove the validity of the verification conditions within HOL. This is possible because the verification conditions are objects of the HOL-object language. Automated theorem provers are not provided, because the verification conditions are terms in Higher Order Logic. For this logic no good automating algorithms are known.

We will summarize strong and weak points for this VCG as well. The strong points it has, stem mainly from its embedding in a general theorem prover.

- The VCG supports a powerful language with complex programming constructs like mutually recursive procedures and expressions that cause changes in the state.
- We can construct Meta-proofs over the programming language within this system as well. The entire language, its semantics and the assertion language are defined within a single formalism.
- Correctness is always and at any level guaranteed by the system. Because everything is formalized within one system, even the most complex inference rules are proved to be correct.

Using a general theorem prover also has some disadvantages. Theorem provers are not built to write programs. They are built to assist in proving theorems.

- The VCG does not support the programmer in writing his program in any way. The programmer has to manually write the entire program and its specification before he can enter it into the system. This also implies that, again, the programmer does not get any support in constructing invariants for loops.

- The language definition does not allow the programmer to insert assertions at arbitrary places in the program. In this respect this VCG is even worse than the first VCG we discussed: even the precondition cannot be specified explicitly. It is computed automatically by the VCG and hence, could turn out to be entirely different from what the programmer had in mind. Also the postcondition is not a part of the annotated program itself, but an extra argument of a tactic that computes the precondition and the verification conditions.
- Due to the embedding of the language, we have three levels of expressions: (1) Boolean expressions of the programming language. (2) Expressions of the assertion language. (3) The HOL-object language. Despite the fact that any expression in (1) can be formed in (2) and any expression in (2) can be formed in (3), the programmer has to translate his terms explicitly whenever he uses a lower level term in a higher level construct. (For instance, when he uses a guard within an assertion).
- The programmer cannot use automated theorem provers to prove the verification conditions for his program. We can prove the verification conditions directly within HOL, but these are elements in Higher Order Logic, for which no good automating algorithms are known.

2.2 Derivation of Programs

It is also possible to derive programs from formal specifications. When using this method, we first have to write a formal specification of the entity we want to compute. We then rewrite this specification in such a way that we can split off smaller sub-problems that we can easily solve. Because we did not yet demonstrate this way of program development, we will first give an example.

In the following example we show how we *derive* an imperative program to compute the i'th Fibonacci number from its specification. The precondition is $0 \leq i$ and the postcondition is $r = F.i$. In our context we consider i to be a constant and r a program variable. We want to derive a program S_0, such that $\{0 \leq i\}S_0\{r = F.i\}$ is a valid annotation of S_0.

First we rewrite the postcondition $r = F.i$ to $(r = F.j) \wedge (j = i)$. We have now created the possibility to establish the first conjunct by repetitive altering of r and j. The additional second conjunct we created is directly computable. Note that $r = F.j$ is directly computable if $j = 0$ or $j = 1$. In case $j = 0$, we also know that $j \leq i$, since $0 \leq i$ is stated in the precondition. This gives rise to implement S_0 by a loop that initializes j with 0 and increases j in its body until $j = i$, while $r = F.j$ is kept invariant. This loop has the required postcondition $(r = F.j) \wedge (j = i)$, because $r = F.j$ is invariant, and $j = i$ must be valid in order for the loop to terminate. We now have the following partial implementation:

$$
\begin{array}{l}
\{0 \leq i\} \\
S_1; \\
j := 0; \ \{r = F.j\} \\
\textbf{while } j \neq i \textbf{ do } \{(r = F.j) \wedge (j \neq i)\} \\
\quad S_2; \\
\quad j := j + 1 \\
\quad \{r = F.j\} \\
\textbf{od } \{(r = F.j) \wedge (j = i)\}\{r = F.i\}
\end{array}
$$

We now have to solve two programming problems: S_1 and S_2. To compute the specifications for these programs, we use the weakest preconditions calculus given in [Dij76]. The weakest preconditions are computed by a function wp. If S is a program and Q is a proposition, then $wp(S, Q)$ is the weakest precondition P such that $\{P\}S\{Q\}$ is a correctly annotated program. We start with S_2:

$$
\begin{array}{ll}
 & \{(r = F.j) \wedge (j \neq i)\}S_2\{wp(j := j + 1, r = F.j)\} \\
\equiv & \\
 & \{(r = F.j) \wedge (j \neq i)\}S_2\{r = F.(j + 1)\}
\end{array}
$$

Obviously, S_2 must assign $F.(j+1)$ to r. Unfortunately, there is no formula for $F.(j+1)$ and the precondition does not provide any information on it either. Therefore, we choose to strengthen the invariant by introducing a fresh variable h to maintain the value of $F.(j+1)$. The new invariant becomes $(r = F.j) \wedge (h = F.(j+1))$ and the partial program becomes:

$\{0 \leq i\}$
S_1;
$j := 0$; $\{(r = F.j) \wedge (h = F.(j+1))\}$
while $j \neq i$ **do** $\{(r = F.j) \wedge (h = F.(j+1)) \wedge (j \neq i)\}$
S_2;
$j := j + 1$
$\{(r = F.j) \wedge (h = F.(j+1))\}$
od $\{(r = F.j) \wedge (h = F.(j+1)) \wedge (j = i)\}\{r = F.i\}$

According to the weakest precondition calculus, the new specification for S_2 is:

$$\{(r = F.j) \wedge (h = F.(j+1)) \wedge (j \neq i)\} S_2 \{(r = F.(j+1)) \wedge (h = F.(j+2))\}$$

According to the precondition, the value S_2 has to assign to r is now given by h. The new value for h is $F.(j+2)$, which by definition is $F.j + F.(j+1)$ (We can easily show that $0 \leq j$ by strengthening the invariant with $0 \leq j$). The latter expression is directly computable from the values of r and h stated in the precondition. Hence, a correct implementation of S_2 is $r, h := h, r + h$.

The last step in deriving the program is the implementation of S_1. We derive:

$$\begin{array}{l} \{0 \leq i\} S_1 \{wp(j := 0; (r = F.j) \wedge (h = F.(j+1)))\} \\ \equiv \\ \{0 \leq i\} S_1 \{(r = F.0) \wedge (h = F.1)\} \end{array}$$

By using the definition of F, we rewrite the postcondition of S_1 to $(r = 0) \wedge (h = 1)$. We implement S_1 by $r, h := 0, 1$. The final program reads:

$\{0 \leq i\}$
$r, h := 0, 1$;
$j := 0$; $\{(r = F.j) \wedge (h = F.(j+1))\}$
while $j \neq i$ **do** $\{(r = F.j) \wedge (h = F.(j+1)) \wedge (j \neq i)\}$
$r, h := h, r + h$;
$j := j + 1$
$\{(r = F.j) \wedge (h = F.(j+1))\}$
od $\{(r = F.j) \wedge (h = F.(j+1)) \wedge (j = i)\}\{r = F.i\}$

Finally, we can prove termination by using $i - j$ as the bound function. The invariant is strengthened by adding the conjunct $(0 \leq j) \wedge (j \leq i)$. Proving the correctness of this invariant is left as an exercise to the reader. Termination is guaranteed by the correctness of the following statements:

1. $(r = F.j) \wedge (h = F.(j+1)) \wedge (0 \leq j) \wedge (j \leq i) \wedge (j \neq i) \Rightarrow (i - j) > 0$
2. $\{(r = F.j) \wedge (h = F.(j+1)) \wedge (0 \leq j) \wedge (j \leq i) \wedge (j \neq i) \wedge (i - j = X)\}$
   $r, h := h, r + h; j := j + 1$
   $\{i - j < X\}$

Note that by deriving the program, we have now obtained a program that is slightly simpler than the program given in chapter 1 (the latter one has an assignment right after **od**). Also it has the same efficiency as the functional program g in chapter 1.

The method of strengthening the invariant is a very powerful method. In general it is an alternative to computing values that are required within the loop directly. The actual computation

of the value is transferred to an other place in the program, by assuming that the value is stored in some variable. If it then turns out that the required value can be computed simultaneously with some other required values, we have obtained a more efficient algorithm than one that always computes each value at the place where it is needed. For instance, in our example we did not explicitly compute $F.(j+1)$ at the place we needed it, but computed it along with the required value of $F.j$.

To our knowledge there are no tools that support the actual derivation of programs. Therefore, we will discuss some tools that use a similar approach of program construction: program refinement. In fact program refinement is another way of deriving programs, but it does not really use the postcondition to guide the way in constructing the program. Instead, it allows the programmer to *refine* more abstract programs to more explicit ones. The refinement calculus then assures the programmer that the replacement of the abstracter program by the more explicit one is valid.

2.2.1 The Refinement Calculus

We will not formally explain the refinement calculus. Instead, we describe the refinement tool introduced in [vW94]. In this tool a simple programming language is embedded within the general theorem prover HOL. By using Grundy's window inference tool for refinement in HOL [Gru92], a programmer can construct programs using the refinement calculus. The tool can also be used to apply data-refinement.

The embedding used by von Wright, is based on the weakest precondition calculus of Dijkstra (see [Dij76]). Von Wright models the programming language by creating a few layers that model parts of the language theory:

- Machine states are modeled by a polymorphic type, which is usually instantiated by a tuple type. A state then is a tuple which contains one component for each program variable.
- Assertions are modeled by predicates, which in turn are modeled by functions from a state to a boolean. A predicate P is said to hold in state s, iff $Ps = True$. Implications, conjunctions and disjunctions on predicates are directly mapped to $\Rightarrow$, $\wedge$ and $\vee$ of the booleans. Using arbitrary functions from states to booleans as predicates allows the programmer to create very powerful specifications, since HOL allows the construction of higher order predicates.
- Substitutions on predicates are modeled by combinations of λ-abstractions and applications. In fact, every substitution is a function from predicates to predicates.
- Programs are modeled by functions from predicates to predicates in the fashion of Dijkstra's predicate transformer semantics (axiomatic semantics) [Dij76, DS90]. The function that models a program gives for every postcondition a precondition that must hold before executing the corresponding program, to assure that upon termination of the program the postconditions hold. In other words: to prove that a program establishes a certain postcondition, we have to prove correctness of the predicate that we obtain by applying the program to the postcondition. Hence, to prove $\{P\}S\{Q\}$, we have to prove $P \Rightarrow S\ Q$. This corresponds to the weakest precondition calculus of programs.

Although this formalism allows us to use *any* function from predicates to predicates as a program, we consider only those functions that correspond to program constructs in the language we work with.

Von Wright uses a simple guarded command language in which guards are predicates. The assignment statement is very general: an assignment is a function from states to states, lifted to the level of predicate transformers. In general, languages do not provide such a general assignment statement. It is only possible to change the value of some variables, usually not more than one at a time. Assertions like *assert* P, where P is a predicate, are considered to be normal statements. An assertion is much like a skip command: if the assertion predicate holds the assertion does nothing. If the predicate does not hold, the assertion statement does not terminate. It then establishes no postcondition at all, not even *true*.

Von Wright also introduces a *nondeterministic assignment* statement, called *nondass m*. m represents a relation between states. When executing *nondass m* , from a state u, it terminates in a state u', such that $m\ u\ u'$ holds. If there are several u' for which the relation m with u holds, *nondass* nondeterministically chooses one. If there exists no u' for which $m\ u\ u'$ holds, *nondass* establishes any postcondition, including *false*. Intuitively, *nondass* represents program parts that are not yet implemented and therefore it also has to be able to represent non-implementable parts of the program. Of course, we want to derive programs in which no *nondass* statements occur, but we need it during the derivation to fill gaps of the program for which we have not yet found a solution.

Refinement of programs can be done without knowledge of the postcondition of the program. We say that a program S *refines* a program T if for all predicates p the proposition $Tp \Rightarrow Sp$ holds, meaning that S can establish the same postcondition but requires only a weaker precondition. All constructs of the programming language are monotonic: if T occurs in some construct $C(T)$ and S refines T, then $C(S)$ refines $C(T)$. We can prove this monotonicity within the formalism for any construct, meaning that given a proof for S refines T, we derive a proof for $C(S)$ refines $C(T)$. Hence, we have to prove the correctness of:

$$\frac{\Gamma \vdash S \text{ refines } T}{\Gamma \vdash C(S) \text{ refines } C(T)}$$

Once the monotonicity has been proved for all constructs we refine programs using these proofs.

The window-inference tool described in [Gru92] supports this kind of refinement: it allows the programmer to focus on the refinement of subcomponents, rather than forcing him to refine the entire program at once. We can expand the window inference tool with new constructs, by simply providing proofs of monotonicity for these new constructs.

With this system a programmer can construct a program using the refinement calculus. The programmer first enters the program specification, given as a pre- and postcondition, say P and Q respectively. Since there is no program constructed yet that meets the specification, we use a single *nondass m* that implements the entire program. The relation m for the nondass depends on the specification. It is defined as $m \equiv \lambda u.\lambda u'.Pu \Rightarrow Qu'$. Hence, if we execute *nondass m* from a state u in which Pu holds, then it can only terminate in states u' in which Qu' holds, since otherwise $Pu \Rightarrow Qu'$ would be false. If Qu' is true for all states u' this does not work, since it implies that $Pu \Rightarrow Qu'$ is true for all states u and u'. However, in practice we will never want the postcondition Q to be true in all states.

The programmer then applies several refinement steps on the program *nondass m* in order to obtain a more specific program, that meets the same, or even a stronger specification (the specification can be stronger in the sense that it has a weaker precondition). Whenever the programmer applies one of the theorems that come with the system, he has to fill in all elements of the refining program. Usually he will use other nondeterministic assignments for program parts that he has not yet solved. Although it is not explicitly mentioned in [vW94], the actual goal of the programmer is to refine until he has obtained a program for which certain well-formedness conditions hold. These well-formedness conditions will certainly claim (among other conditions) that the final program should not contain any *nondass* statements and that guards should not contain any quantifications.

Data-refinement can also be performed with this refinement tool. Data-refinement allows us to first derive an abstract kind of program that works with sets, and then replace all variables that represent sets by variables of more explicit types that implement the sets. The corresponding operations performed on variables of the abstract type will then also be automatically transformed to operations on the more explicit type.

Constructing programs with the refinement calculus has some advantages:

- The construction of the programs is supported by the system. The programmer does not have to enter the entire program at once.
- Correctness of the program with respect to its specification is verified during the entire

process of constructing the program. It is not verified after the programmer has written the entire program.

- We can have variables of several types. In all the previous systems we considered all variables have type integer. Also the formalism of von Wright allows the use of local variables (we did not discuss this though).
- During program construction it is possible to use arbitrary predicates as guards of statements.
- Assertions can be inserted at any place of the program, since they are considered to be normal commands.

Drawbacks of the tool are mainly caused by the way the refinement calculus is embedded and supported by the system:

- This system still does not support the construction of invariants. The programmer has to fill in the various parameters when he applies a refinement step.
- Automating the proving process is still difficult. Since any function from states to booleans may be used as a predicate, higher order predicates can be formed.
- The user interface is still primitive. Even though Grundy's window inference allows the programmer to concentrate on parts of the program, all commands still have to be entered manually in a command line interface.
- The programming language is not strictly defined. It is not stated what types are actually supported by the language. Also the language's constructs are too primitive for any real-life application.

2.2.2 Automatic Synthesis of Imperative Programs

In [Chr93], Heine Christensen describes a tool that automatically generates imperative programs from logical specifications. The tool can not yet generate large or complex programs, but it demonstrates that deriving and computing programs is really possible. In this subsection we briefly describe the programming- and specification language supported by Christensen's tool and the algorithm that synthesizes programs.

The programming language supported by Christensen's tool is inspired by the language used in [Gri81] and Pascal. It supports variables of type boolean, integer, record types and arrays in any combination. Pointers are not allowed in order to avoid difficulties with aliasing in the specification language. The statements in the language are skip; multiple assignment; selection; iteration (multiple guards in selection and iteration are allowed); and procedure calls. Procedure calls are only used when the programmer directly implements in the programming language. They are not synthesized. In order to synthesize a program, it has to be formally specified by a pre- and a postcondition in the specification language.

The tool uses two assertion or specification languages: one for preconditions and one for postconditions. The assertion language for preconditions consists of the following constructs:

- Boolean and integer expressions, with all the usual operators, including conditional integer expressions, conditional and (cand) and conditional or (cor) on booleans; and the power-operator for integers.
- Bounded quantifications over explicit domains of integers. For instance, (A i in$[0..n].x[i] = i$) expresses that $x[i] = i$ holds for all i in $\{0..n\}$. Unbounded quantifications are not allowed. Hence, one cannot specify a 'Meta'-function, say *square* on integers, outside the programming language by $(\forall i : integer.square(i) = i * i)$. Allowed quantifiers are forall; exists; sum; number; maximum; and minimum.

The assertion language for postconditions is equal to the assertion language for preconditions with two extensions:

1. Behind the postcondition, one must give a list stating the changeable variables. The synthesized program will only change these variables and considers all the other variables as constants. This allows the user to use constants in the specification that are implicitly universally quantified. We will call this kind of constants *problem variables*, since they represent parameters of the specified problem.

2. One can refer to the initial value of a changeable variable by suffixing its name with a zero. The intitial value is the value of a variable at the moment that the preconditions holds. This is necessary, for example, when one wants to express that an array has been sorted without changing the contents of the array.

Once a program is specified, the user can start the synthesizer, which attempts to generate an implementation meeting the specifications. Synthesizing programs is done by recursively applying the following scheme of four steps:

1. If the precondition implies the postcondition, the synthesizer implements the program with a skip.

2. If a skip is not sufficient, the synthesizer attempts to generate a multiple assignment statement. The order in which values are assigned to the variables and the required expressions are deduced by analysis of the assertions and the dependencies of the variables. We will explain more about this analysis below.

3. Sometimes a condition C is found during the search for an assignment. If C is not implied by the precondition, Christensen's tool chooses to continue synthesizing conditional assignments until all cases are covered. All cases are covered when the precondition implies the disjunction of the conditions associated to the generated assignments. If this is successful, a selection statement is generated with the conditions as guards and the associated assignments as the guarded statements.

4. Finally, if the problem cannot be solved directly with the previous steps, the synthesizer attempts to create a loop. It obtains the required invariant, termination condition, and bound function by applying standard techniques for generating loop invariants. These standard techniques are discussed below. Once the invariant, the guard, and the bound function are known, the synthesizer computes the specification for the loop body. The loop body is then generated by recursive application of the synthesizer algorithm to the computed specification.

Analysis of Assertions for Generating Multiple Assignment Statements

Generating assignment statements is done by iterative guessing. Every guess is then verified for correctness against the specification of the program. To determine the order of assignment, the specification assertions are analyzed for dependencies of variables on other variables. For instance, in the assertion $r = (\text{Min } i \text{ in } [0..n].x[i])$ the variable r depends on the value of variable n. In this case, expressions for r are guessed after expressions for n are guessed. Christensen distinguishes between weak and strong dependencies. A variable x is weakly depending on variable set S, if given the values of the variables in S there exists *at most* one value for x, such that the postcondition holds. A variable x is strongly depending on variable set S, if given the values of the variables in S there exists *exactly one* value for x, such that the postcondition holds. In the latter case, the expression for the strongly dependent variable x can be computed, once the expressions for the variables in S have been chosen. If a variable is independent, the synthesizer guesses as expression one of $x+1$, $x-1$, 0, -1, $+1$, and $+$ or $-$ all computable subexpressions of the output assertion.

When expressions for all variables have been guessed, the synthesizer tests whether or not the assignment meets the specification and if so, what the additional conditions are (see step 3).

If no condition can be found under which the assignment meets the specification, another set of assignments is searched by backtracking. Generating an assignment fails, if even after backtracking and with additional conditions no legal assignment can be found.

Standard Techniques for Generating Loop Invariants

Another open problem in the four step scheme is the generation of loop invariants, loop termination conditions, and bound function. Currently the synthesizer uses two standard techniques to obtain them. These techniques have been introduced in [Gri81] and have been adapted to the formalism used by the synthesizer. They were both used in the example at the beginning of section 2.2.

The first technique removes a directly computable conjunct from the postcondition and uses its negation as the guard of the loop. It is then assured that upon termination of the loop the guard is false and hence, the conjunct is true. The remainder of the postcondition is used as the invariant of the loop, to ensure correctness of the entire postcondition upon termination. As bound function an inequality with a changeable variable in the invariant is used. Guessing techniques are used to obtain the appropriate expression. We refer to this technique by 'removing a conjunct'.

The second technique used by the synthesizer replaces a constant with a variable. First a constant, say N, is chosen in the postcondition. N is always a constant in the index set of a top-level quantifier or in a power construction. Next, the invariant P is generated to be the postcondition with the constant replaced by a fresh variable, say i. P is then strengthened by adding limits for i. These limits are N itself and another value z that either exactly empties the index set or makes the power construction 0. Initially, i is set to z. If $z < N$, the guard G of the generated loop is $i < N$ and the bound function is $N - i$. If $z > N$, the guard is $i > N$ and the bound function is $i - N$. Finally a loop body is synthesized that preserves the invariant. Correctness of the loop is guaranteed: $I \wedge \neg G \Rightarrow R$, since I contains the limits for i. Also the synthesizer checks whether or not the generated bound function is legal. The second technique is referred to as 'replacing a constant'.

Christensen claims that these two techniques, combined with the proposed guesses for assignments, are capable of solving many problems. For instance, insertion sort can be generated. Christensen proposes that the synthesizer should be included in a semantics-oriented editor. In such an editor, the user of the system must be able to give hints about the techniques the synthesizer must apply in order to efficiently derive correct programs. Positive characteristics of such an editor are:

- The programmer only has to outline how he or she wants to construct the program. The construction itself is done automatically and therefore correct.
- Simple program parts can be generated entirely automatically and free the programmer from tedious repetitive work.
- The administration of proof-obligations and their proofs is supported by the system.

The usage of a semantics oriented editor as proposed by Christensen also has some drawbacks:

- All programs have to be fully specified. Even simple modules have to be proved correct that are usually programmed without assertions.
- The specification language does not allow meta-specifications. We cannot use unbounded quantifications to range over infinite sets like the one of integers.
- Synthesizing program parts is slow. The examples in [Chr93] show that for a program counting the maximum row sum below the diagonal of an array half an hour of synthesizing time is needed. Manually this can be done much faster, including manual formal verification.

2.3 Extraction of programs

The last method of correct programming we discuss in this chapter is extraction of programs. It is described in more detail in [PM89]. In contrast to the previous methods, this method will not be described in detail here, since the proposed method is not appropriate for our project. Instead, we will describe the principles underlying the method and use these to explain why the method does not seem to be appropriate for our goals.

Extraction of programs is based on the observation that constructive proofs for existential quantifications contain an algorithm to compute a witness for the specification. In other words, a constructive proof of $\exists x.P(x)$ contains an algorithm that computes an x for which $P(x)$ holds.

To derive a program that computes an output entity $y : Y$ from an input entity $x : X$, such that $P(x, y)$ holds, the programmer first proves that for every $x : X$ such a $y : Y$ exists. Hence, he proves $\forall x : X.\exists y : Y.P(x, y)$. If the proof is constructive, he can extract a program from this proof. This program is a function $f : X \rightarrow Y$ such that $\forall x : X.P(x, fx)$ holds.

In Coq, extraction is done fully automatically, based on the extraction function described in [PM89]. To make this possible, a variant of the Calculus Of Construction is used, in which there is a distinction between informative and non-informative propositions. The informative propositions denote specifications of programs. The non-informative propositions represent logical contents. Elements of informative propositions contain programs meeting the specification represented by the informative proposition. Elements of non-informative propositions represent proofs of the non-informative proposition. During the extraction of a program from a proof, all proofs of non-informative propositions are discarded, since these do not contribute to the computation of the witness, but to the proof that the computed witness meets the specification.

To construct a program that computes from input $x : X$ the output $y : Y$, such that $P(x, y)$ holds, the programmer first constructs a proof t of $\forall x : X.\exists y : Y.P(x, y)$ within Coq. In this proposition, X and Y are informative and $P(x, y)$ is non-informative. Since the proof is entirely constructed within Coq it is fully constructive. Once t is constructed, the programmer invokes Coq's extraction function to extract a program from t. Intuitively, the extraction function traverses t and computes a function $f : X \rightarrow Y$, such that $P(x, f(x))$ is "*true*". The function f itself is a term in F_ω, the system designed by Girard.

Although program extraction provides a safe way to construct correct programs, it is not suitable for our purposes. Our objections to the method are the following ones:

- The program does not become visible until the entire proof has been constructed, since program extraction cannot be applied to a partial proof. Consequently, the programmer cannot see whether the program he is constructing contains parts that obviously could be implemented more efficiently.
- It is not possible to skip (parts of) a proof. It will often happen that propositions have to be proved in which the programmer is not interested at the time they are requested. For instance, the programmer may wish to postpone the proof of well-foundedness of a recursive function.
- Program extraction only works for the construction of functional programs.
- Programming has become a proof oriented matter. Programmers want to concentrate on the program, not its proof.
- Automating parts of the proving process is difficult, since program extraction requires a logic like F_ω or the Calculus of Constructions. For these logics good automating algorithms are not yet known.

However, extracting programs is useful for obtaining prototype programs that compute entities of which the existence has been proved. If a mathematician has proved that for a certain class of functions a least fix point exists, it is possible to immediately extract a program for the

computation of this fix point. This program may be too inefficient to be used in a program that requires many fix points of this kind, but for the mathematician it will be adequate.

Also, the extraction of programs is a beautiful theoretical result. It demonstrated that it is actually possible to immediately derive working programs from constructive proofs, which is a notable property of constructivism.

Chapter 3

Existing Tools: an Overview

In this chapter we present an overview of existing tools in the fields of theorem proving and computer programming. The tools for theorem proving support the verification, construction and/or automatic construction of formal proofs for correct conjectures. The tools for computer programming are divided in two groups: generic environments and programming tools. Generic environments assist in creating programming environments for languages that can be specified by the user of the generic environment. Programming tools assist in writing programs in a pre-defined language.

In the following sections we will discuss the results displayed in table 3.1. We do this by analyzing and comparing the systems in each category. Table 3.1 is compiled to the best of our knowledge. Note that most of the systems discussed in this chapter are still under development and hence, many data in the table may be outdated soon. Also, it may turn out during further research that despite our efforts, we misinterpreted some of the information we have about the systems. The list is not intended to be complete. Many other systems exist and may be added in the future.

For more detailed information about the systems, we refer to table 3.2. It contains for every system discussed in this chapter, either the URL of its World Wide Web home-page or a bibliographic reference.

3.1 Explanation of the Criteria

In this section we explain the criteria found at the top of the columns in table 3.1. Also, we give the meaning of the possible cell-values in each column. The scores of the systems on the criteria are discussed in the following two sections.

kind Kind indicates in what category we placed the system. The abbreviations are: Th.P. for Theorem Prover, Gen.E. for Generic Environments, and Pr.T. for Programming Tool.

generality The generality of a system can vary between performing a single, specific task, being useful for several tasks or being customizable for a large class of tasks. The scores are 0, +, and ++ respectively.

drive What has been the reason for the creators of the system to implement it? The desire for a system can arise from theory (T) or practice (P). We assign a T to a system if there first was a lot of theory that the creators wanted to support. Systems with a P in this column were designed because there was a need for the support they give. Most systems should be rated with a value in between T and P. Also, what is considered theory and what is considered practice depends on the point of view of the user. The values in the table indicate our point of view.

	Kind	Generality	drive	support type	automation	user interface	language	logic	extendability	custom notation
Coq	Th.P.	++	T	V,C	+	+	ML	++	0	+
Lego	Th.P.	++	P	V,C	−	+	ML	++	0	−
Isabelle	Th.P.	++		V,C	0	+	ML		+	++
HOL	Th.P.	+	P	V,C	−	+	ML	+	+	++
Nuprl	Th.P.	+	T	V,C	0	+	ML/Lisp	+	+	++
PVS	Th.P.	+	P	V	+	+	Lisp	+	+	++
KIV	Th.P.	+	P	C	+	++	PPL	+	+	0
Larch Prover	Th.P.	0	P	C	++	+		0	−	−
LeanTap	Th.P.	0	P	A	++	+	Prolog	0	−	−
Otter	Th.P.	0	P	A	++	0	C	0	−	−
The synthesizer gen.	Gen.E.	++	P	C	++	++		++	CS	
ASF+SDF	Gen.E.	++	P	C	++	++	Centaur	++	RW	
The Maintainers Ass.	Pr.T.	0	P	V,C	0	++	WSL	0	RW	
Homeier et al	Pr.T.	0	P	V	0	+	HOL	0	PL	
Refinement Calc.	Pr.T.	+	T	C	−	++	HOL	+	PL	
Christensen's	Pr.T.	0	T	A	++	+	Prolog	0	PL	
								supported language	supports	

Table 3.1: A brief overview of existing tools.

Name	Home-page URL or bibliographic reference
Coq	http://pauillac.inria.fr/coq/systeme_coq-eng.html
Lego	http://www.dcs.ed.ac.uk/home/lego
Isabelle	http://www.cl.cam.ac.uk/users/lcp/ml-aftp/
HOL	http://lal.cs.byu.edu/lal/hol-desc.html
Nuprl	http://www.cs.cornell.edu/Info/Projects/NuPrl
PVS	http://www.csl.sri.com/sri-csl-pvs.html
KIV	http://i11www.ira.uka.de/~kiv/KIV-KA.html
Larch Prover	http://larch.lcs.mit.edu:8001/larch/LP/overview.html
LeanTap	http://emmy.ira.uka.de/~posegga/leantap/leantap.html
Otter	http://www.mcs.anl.gov/home/mccune/ar/otter/index.html
The synthesizer gen.	[RT89]
ASF+SDF	http://www.cwi.nl/~gipe/asf+sdf.html
The Maintainers Ass.	http://www.dur.ac.uk/~dcs1ejy/Bylands/
Homeier et al	[HM96]
Refinement Calc.	http://cs.anu.edu.au/~Jim.Grundy/rcalc.html
Christensen's	[Chr93]

Table 3.2: List of the World Wide Web home-pages or bibliographic references of the systems.

support type A tool can provide several kinds of support. We distinguish three kinds: Verification (V), construction (C), and automatic construction (A). In verification the tool only checks whether the given input meets certain constraints or not. Sometimes the user has to give hints. Tools for construction interactively assist the user in reaching his goal. Automatic construction tools attempt to reach the user's goal all by themselves, without interaction of the user. In this case, the user only has to specify his goal.

automation The level of automation indicates how much the user has to do himself and how much the tool can do automatically. There are several theorem provers for constructing proofs, that are partially automated in order to relieve the user from proving trivial conjectures. We have four levels: no automation at all (-), naive automation (0), parametric or programmable automation (+), and sophisticated automation (++). Whether or not + is better than 0 depends on how well the user works with the offered degrees of freedom of the automation.

user interface Although the user-interface has no influence on the actual power of the tool, it determines a great deal of its practical application. Good user interfaces are better accepted by end-users. The possible values are: no interface (0, input comes from a file), command line interface (+), and graphical point-and-click interface (++).

language This is either the language in which the tool is programmed or the system on which it is based. The language (among other aspects) determines the portability and speed of the tool.

logic Proofs developed within a theorem prover are always based upon a formal logic. In strong logics more theorems can be formalized than in weaker logics. First order predicate logic (0) is the weakest logic that is used in the tools under consideration. Other tools use higher order logics (+). We also consider the calculus of constructions (++) as a higher order logic, but this one is so powerful that we indicate it differently.

extendability In this column we judge the extendability of the system. If extending the logic and the commands of the system is not possible, we rate it with −. If extending is possible, but the extensions have to be programmed in the underlying language or system, we rate it with 0. If extending is supported by the system, we rate it with +.

custom notation Custom notation indicates how well the syntax accepted by a theorem prover can be adjusted to the theory defined within it. For instance, if we reason about natural numbers, we want to use an infix notation for addition and multiplication. This kind of adjustment is not supported by all theorem provers. Therefore, we distinguish four levels of flexibility: no adjustments can be made (–), adjustments have to be programmed in the language underlying the theorem prover (0), the theorem prover provides limited or clumsy means for adjustment (+) and adjustment is well supported by the system (++).

supported language Supported language indicates the range of languages supported by a programming tool. Possible values are: supports only a fixed language (0), supports an extendable language (+), and the language can entirely be specified by the user (++).

supported aspects What aspects of the programming language are supported? We use the following values: Only supports the most elementary aspects of the language (CFS, context free syntax), checks the "spelling" of a program to such extent, that it can be translated when correct (CS, context dependent syntax), program parts can be replaced by equals or be re-written according to re-write rules supported by the system (RW, re-writing), and the tool supports reasoning about the correctness of programs in the given language (PL, programming logic).

3.2 Theorem Provers

Now that we know the meaning of the criteria and their values, we explain why the systems are rated the way they are. We will not do this by explaining every score assigned to each of the systems, but rather by comparing the systems and emphasizing the most important differences and features. All the general claims in the following discussion only apply to the systems in this overview. There will always be systems that do not fit into this scheme, now nor in the future.

To the end-users, the theorem provers present themselves in two flavors: proof assistants and automatic theorem provers. A proof assistant's main task is to take care of the administration of a proof. The user constantly enters commands (called tactics), telling the assistant what to do next. The automated theorem provers try to construct the entire proofs themselves. The user then merely enters a conjecture, which the theorem prover attempts to prove. To the user this distinction will be clear immediately, since the first systems are highly interactive, while the latter ones are more automated.

3.2.1 Proof Assistants

The main differences in proof assistants are their underlying logic and the customizability of the notation. For the average user the strength of the underlying logic (for proof assistants higher order logic (+) or the calculus of constructions (++)) of these systems is not immediately clear. This explains also why weaker systems like HOL (Higher Order Logic) and PVS (Prototype Verification System) are more used in practice than Coq[1] and Lego, which have more powerful logics. Apparently, the extendability of the logic and the flexibility of the user interface by means of flexible syntax are more important to a user of a system than the strength of its logic. Therefore, practically based systems are used more.

The most practically based proof assistants are HOL (see [GM93]), PVS and KIV (Karlsruhe Interactive Verifier). HOL is often applied in VLSI-design and has been extended to meet the requirements in this area. More recently, HOL has been used for proving mathematical theorems and reasoning about program correctness (see also Homeier's tool and the refinement calculator below). PVS is a specification verification system. It is specifically designed to deal with industrially scaled specifications. The KIV system is created to prove correctness of programs. However,

[1] Coq is pronounced as COC, Calculus Of Constructions

the way in which the programs are constructed is the way in which proofs of theorems are constructed in general theorem provers, which is not the way in which a programmer wants to work. Also, proving correctness of programs requires proving many trivial propositions.

The automation in the proof assistants is usually very naïve. For instance, consider Coq, Nuprl (see [CAB86]) and PVS. In Coq the user specifies a set of proved conjectures and declares a set of definitions that may be unfolded during the proof search. Coq's auto-tactic then naïvely searches through all proofs that can be constructed with the specified conjectures for a proof of the current proof goal. In Nuprl there also exist a few tactics to search proofs for conjectures about the set of integer numbers. These tactics rewrite the proposition to a normal form and then apply heuristics to decide on the correctness of the proposition. This works only for a limited class of propositions. Automatically proving propositions about integer numbers cannot be done in Coq: unlike Nuprl, Coq does not contain the set of integer numbers as a primitive and it does not support extensions of the automatic proof facility. Automatic proof search in PVS is based on model checking. In model checking a proposition is proved by considering all possible models and checking for each model whether or not the proposition holds.

3.2.2 Automated Theorem Provers

The automated theorem provers are always based on first order logics (score 0 on logic). This is because for these logics good automation is possible. The most popular automatic proof construction algorithms are tableaux methods (used in LeanTap, see also [Oph92, dK95]) and resolution methods (used in Otter, see also [dN95]). For both of these methods completeness has been proved: If a proposition is correct, then there exist a tableau- and resolution proof for it. However, finding such a proof is not decidable: any algorithm could search for a proof forever without finding one, since the search space is infinite. Automatic proving is a one-way street: If a proof is found, then the conjecture is correct. If a proof is not found, it may still be correct, but the program is not able to decide on the correctness. Therefore, automatic theorem provers like Otter and the Larch prover are extended with algorithms that find counter examples. If the prover does not find a proof within a certain amount of time, Otter and the Larch prover attempt to find a model in which the proposition does not hold. Similarly but opposite to proof search: if no such model can be found, the conjecture can still be wrong, but this cannot be proved by the system.

The Larch prover uses the possibility of automatic proof search for the construction of correct conjectures. Unlike other automatic theorem provers it does not assume that the initially given conjecture is correct. Instead, it tries to find flaws in the conjecture and reports them to the user. The user can then correct these flaws and let the corrected conjecture be processed by the Larch prover again.

The automated systems are typically not extendable, because the level of automation limits the logic. Extendability of the logic would cause great difficulties in the automatic proof search. Also the syntax of the systems is inflexible. Since the logic is simple and not extendable, no complex theories that need adjustment of the prover's syntax can be formalized within it.

3.3 Computer Programming

The differences in tools for computer programming are such that we divided them in two kinds: Generic environments and programming tools. The generic environments produce tools for a programming language that the user can specify. Programming tools support constructing or verifying programs in a specific, built-in language. Hence, generic environments produce programming tools. Also, tools that generate other tools are more general, since they can produce programming tools for a wide range of languages. Programming tools for a specific language are usually more powerful, since they can exploit properties that are specific to the language they support.

3.3.1 Generic Environments

The synthesizer generator (see [RT89]) and the ASF+SDF (Algebraic Specification Formalism + Syntax Definition Formalism, see [vDHK96]) environment both generate syntax-directed editors for given language specifications. The ASF+SDF environment takes not only a syntax of a language, but also a set of re-write rules that will be supported by the generated tool. The re-write rules can contain abstract variables, which is why the ASF+SDF environment is also called a META-environment. By specifying re-write rules that transform correct programs into other correct programs, the tools generated by the ASF+SDF environment support correctness preserving transformations. The synthesizer generator generates tools that only support syntax directed editing. On the other hand, the synthesizer generator has also been used to create a programming tool that can also verify the correctness of programs (see chapter 2). To do this, however, the user (also the authors in this case) had to specify the specification logic himselve. This way, proving program correctness became a purely syntactical matter supported by the generated tool.

3.3.2 Programming Tools

The programming tools are specifically designed to support programming in a specific language. Therefore these tools can give more sophisticated support, like verification of programs and refinement. Since we consider only four programming tools, we will give for each system a short description.

The Maintainers Assistant is a tool that supports correctness preserving transformations on the Wide Spectrum Language (WSL). The transformations are supported by a set of built in re-write rules. WSL is designed to encode high level specifications as well as low level implementations.

The remaining tools (of Homeier et al, the Refinement Calculator and Christensen's tool) all support the programming logic, but all in a different way, namely for verification, construction and automatic construction respectively.

Homeier et al programmed a verification condition generator (VCG) for mutually recursive procedures in HOL. The correctness of the programming logic is guaranteed, by proving the soundness of the axioms of the language with respect to its structural operational semantics within HOL. This construction makes it hard to extend the language, since all axioms about new constructs would have to be derived. The actual VCG is programmed in ML, which is the implementation language and meta language of the HOL system.

Von Wright's refinement calculator uses the programming logic to refine programs. Refinement of programs is similar to re-writing of programs, but takes advantage of the context in which the re-writing takes place. For instance, we can refine $\{x = a\}x := x + 1\{x > a\}$ by $\{x = a\}x := x + 2\{x > a\}$, because both programs are correct with respect to the specification. We cannot perform this mutation by re-writing though, since $x := x + 1$ cannot be replaced by $x := x + 2$ in an arbitrary context. The programming language supported by this tool can easily be extended. Within the tool, programs are formalized as functions from predicates to predicates (predicate-transformer semantics) and hence, we can extend the language by defining more constructors for this kind of functions.

Christensen's tool uses Gries's programming methodology, combined with a programming logic for a simple While language. It automatically derives programs from specifications in first order predicate logic. The main importance of this tool is that it demonstrates that *computing* programs is really possible. In practice the time needed to derive simple programs is such that it is more efficient to write and verify them manually. Also, no complex programs have yet been computed with this tool.

3.4 Conclusions

Although a few tools exist that support the construction of correct programs, none of these tools uses the Dijkstra/Hoare style of programming. The refinement calculator is based on the same

basic principles as the Dijkstra/Hoare style, but does not support the programming methods described in [Dij76, Gri81, Kal90]. Christensen's tool uses the Dijkstra/Hoare style of programming, but does not provide interactive support to a programmer and thereby limits the complexity of the programs. Instead it attempts to derive programs fully automatically. Moreover, the truth of the assertions accepted by Christensen's tool are directly computable, which indicates their limited expressional power.

Without exceptions the programming tools are large and do not run on average sized computers. If programmers are to use a tool on a larger scale, this tool must run on their own computers which usually are average sized. We believe that often the size could be limited by directly implementing the tool in an executable language. The existing tools are built in layers:

1. Tools like The Refinement Calculator, Homeier's tool, and KIV are implemented in a general theorem prover, like HOL.

2. General theorem provers are mostly implemented in ML or another functional language.

3. The functional language is interpreted or compiled by an imperative program.

This layered design of the systems requires not only large, but also very fast computers in order to run the systems. If a system is directly implemented in an imperative language it becomes faster and smaller at the same time.

Also, tools for correct programming have user-interfaces that are based on constructing proofs, not programs. Tools with user-interfaces for the construction of programs do not support checking correctness on the logical level. An exception is the tool generated with The Synthesizer Generator in [RT89]. Unfortunately, this tool only supports verification of programs and not the derivation of programs. The programming and specification language supported by this tool are rather weak.

The requirements of the tool we want to construct in our project are as follows: the tool should support the derivation of programs in the Dijkstra/Hoare fashion. To avoid a rejection of the tool by the programmers at forehand, it must also be possible to use the tool to write programs without proving correctness. The programmer can then decide for himself when he wants to have formal support to construct program parts. The tool also must be able to run on average sized computers. This can be achieved by directly implementing the tool in an imperative language instead of using the usual layered approach. Finally the tools should have a program-oriented user interface, opposed to the proof-oriented user interfaces provided by the existing tools.

Bibliography

[Apt97] Krzysztof R. Apt. *From Logic Programming to Prolog.* Prentice Hall international series in Computer Science. Prentice Hall, 1997.

[CAB86] R.L. Constable, S.F. Allen, and H.M. Bromley. *Implementing mathematics with the Nuprl proof development system.* Prentice-Hall, 1986.

[Chr93] Heine Christensen. Synthesis of programs from logic specifications using programming methodology. *Structured Programming*, 14:173–186, 1993.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall International, 1976.

[dK95] Eric de Kogel. *Equational Proofs in Tableaux and Logic Programming.* PhD thesis, Tilburg University, 1995.

[dN95] Hans de Nivelle. *Ordering Refinements of Resolution.* PhD thesis, Delft University of Technology, 1995.

[DS90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics.* Springer, 1990.

[GHL78] J. Guttag, J. Horning, and R. London. A proof rule for euclid procedures. *Formal Description of Programming Language Concepts*, pages 211–220, 1978.

[GL80] David Gries and G. Levin. Assignment and procedure call proof rules. *ACM TOPLAS*, 2:564–579, 1980.

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, Cambridge, 1993.

[Gri81] David Gries. *The Science of Programming.* Springer, 1981.

[Gru92] Jim Grundy. A window inference tool for refinement. In Cliff B. Jones, Roger C. Shaw, and Tim Denvir, editors, *5th Refinement Workshop*, pages 230–254. Springer, 1992.

[HM96] Peter V. Homeier and David F. Martin. Mechanical verification of mutually recursive procedures. In M.A. McRobbie and J.K. Slaney, editors, *Automated Deduction CADE-13*, Lecture Notes in Artificial Intelligence. Springer, July/August 1996.

[Kal90] Anne Kaldewaij. *Programming: the derivation of algorithms.* Prentice-Hall international series in Computer Science. Prentice Hall, 1990.

[Oph92] W.J. Ophelders. *Automated Theorem Proving Based Upon a Tableau-Method With Unification Under Restrictions: Theory, Implementation and Empiracal Results.* PhD thesis, Tilburg University, 1992.

[Pau94] Lawrence C. Paulson. *Isabelle: a generic theorem prover.* Springer, Berlin, 1994.

[Pee95] Eric A.J. Peeters. *Design of an Object-Oriented Interactive Animation System.* PhD thesis, Eindhoven University of Technology, 1995.

[PM89] Christine Paulin-Mohring. Extracting F_ω's programs from proofs in the calculus of constructions. In *Sixteenth Annual ACM symposium on Principles of Programming Languages*, pages 89–104, Austin, Texas, Januari 1989. ACM, ACM press.

[RT89] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator, A System for Constructing Language-Based Editors.* Texts and Monographs in Computer Science. Springer-Verlag, 1989.

[vdBvDK96] Mark van den Brand, Arie van Deursen, and Paul Klint. Industrial applications of asf+sdf. Technical Report CS-R; 9622, Centrum voor Wiskunde en Informatica, Amsterdam, 1996.

[vDHK96] Arie van Deursen, Jan Heering, and Paul Klint. *Language Prototyping, an Algebraic Specification Approach*, volume 5 of *AMAST: Series in Computing.* World Scientific, 1996.

[vW94] Joakim von Wright. Program refinement by theorem prover. In David Till, editor, *6th Refinement Workshop*, pages 121–150. Springer, 1994.

Computing Science Reports

Department of Mathematics and Computing Science
Eindhoven University of Technology

In this series appeared:

93/01	R. van Geldrop	Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.
93/02	T. Verhoeff	A continuous version of the Prisoner's Dilemma, p. 17
93/03	T. Verhoeff	Quicksort for linked lists, p. 8.
93/04	E.H.L. Aarts J.H.M. Korst P.J. Zwietering	Deterministic and randomized local search, p. 78.
93/05	J.C.M. Baeten C. Verhoef	A congruence theorem for structured operational semantics with predicates, p. 18.
93/06	J.P. Veltkamp	On the unavoidability of metastable behaviour, p. 29
93/07	P.D. Moerland	Exercises in Multiprogramming, p. 97
93/08	J. Verhoosel	A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.
93/09	K.M. van Hee	Systems Engineering: a Formal Approach Part I: System Concepts, p. 72.
93/10	K.M. van Hee	Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.
93/11	K.M. van Hee	Systems Engineering: a Formal Approach Part III: Modeling Methods, p. 101.
93/12	K.M. van Hee	Systems Engineering: a Formal Approach Part IV: Analysis Methods, p. 63.
93/13	K.M. van Hee	Systems Engineering: a Formal Approach Part V: Specification Language, p. 89.
93/14	J.C.M. Baeten J.A. Bergstra	On Sequential Composition, Action Prefixes and Process Prefix, p. 21.
93/15	J.C.M. Baeten J.A. Bergstra R.N. Bol	A Real-Time Process Logic, p. 31.
93/16	H. Schepers J. Hooman	A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27
93/17	D. Alstein P. van der Stok	Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.
93/18	C. Verhoef	A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.
93/19	G-J. Houben	The Design of an Online Help Facility for ExSpect, p.21.
93/20	F.S. de Boer	A Process Algebra of Concurrent Constraint Programming, p. 15.
93/21	M. Codish D. Dams G. Filé M. Bruynooghe	Freeness Analysis for Logic Programs - And Correctness, p. 24
93/22	E. Poll	A Typechecker for Bijective Pure Type Systems, p. 28.
93/23	E. de Kogel	Relational Algebra and Equational Proofs, p. 23.
93/24	E. Poll and Paula Severi	Pure Type Systems with Definitions, p. 38.
93/25	H. Schepers and R. Gerth	A Compositional Proof Theory for Fault Tolerant Real-Time Distributed Systems, p. 31.
93/26	W.M.P. van der Aalst	Multi-dimensional Petri nets, p. 25.
93/27	T. Kloks and D. Kratsch	Finding all minimal separators of a graph, p. 11.
93/28	F. Kamareddine and R. Nederpelt	A Semantics for a fine λ-calculus with de Bruijn indices, p. 49.
93/29	R. Post and P. De Bra	GOLD, a Graph Oriented Language for Databases, p. 42.
93/30	J. Deogun T. Kloks D. Kratsch H. Müller	On Vertex Ranking for Permutation and Other Graphs, p. 11.

93/31	W. Körver	Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.
93/32	H. ten Eikelder and H. van Geldrop	On the Correctness of some Algorithms to generate Finite Automata for Regular Expressions, p. 17.
93/33	L. Loyens and J. Moonen	ILIAS, a sequential language for parallel matrix computations, p. 20.
93/34	J.C.M. Baeten and J.A. Bergstra	Real Time Process Algebra with Infinitesimals, p.39.
93/35	W. Ferrer and P. Severi	Abstract Reduction and Topology, p. 28.
93/36	J.C.M. Baeten and J.A. Bergstra	Non Interleaving Process Algebra, p. 17.
93/37	J. Brunekreef J-P. Katoen R. Koymans S. Mauw	Design and Analysis of Dynamic Leader Election Protocols in Broadcast Networks, p. 73.
93/38	C. Verhoef	A general conservative extension theorem in process algebra, p. 17.
93/39	W.P.M. Nuijten E.H.L. Aarts D.A.A. van Erp Taalman Kip K.M. van Hee	Job Shop Scheduling by Constraint Satisfaction, p. 22.
93/40	P.D.V. van der Stok M.M.M.P.J. Claessen D. Alstein	A Hierarchical Membership Protocol for Synchronous Distributed Systems, p. 43.
93/41	A. Bijlsma	Temporal operators viewed as predicate transformers, p. 11.
93/42	P.M.P. Rambags	Automatic Verification of Regular Protocols in P/T Nets, p. 23.
93/43	B.W. Watson	A taxomomy of finite automata construction algorithms, p. 87.
93/44	B.W. Watson	A taxonomy of finite automata minimization algorithms, p. 23.
93/45	E.J. Luit J.M.M. Martin	A precise clock synchronization protocol,p.
93/46	T. Kloks D. Kratsch J. Spinrad	Treewidth and Patwidth of Cocomparability graphs of Bounded Dimension, p. 14.
93/47	W. v.d. Aalst P. De Bra G.J. Houben Y. Kornatzky	Browsing Semantics in the "Tower" Model, p. 19.
93/48	R. Gerth	Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.
94/01	P. America M. van der Kammen R.P. Nederpelt O.S. van Roosmalen H.C.M. de Swart	The object-oriented paradigm, p. 28.
94/02	F. Kamareddine R.P. Nederpelt	Canonical typing and II-conversion, p. 51.
94/03	L.B. Hartman K.M. van Hee	Application of Marcov Decision Processe to Search Problems, p. 21.
94/04	J.C.M. Baeten J.A. Bergstra	Graph Isomorphism Models for Non Interleaving Process Algebra, p. 18.
94/05	P. Zhou J. Hooman	Formal Specification and Compositional Verification of an Atomic Broadcast Protocol, p. 22.
94/06	T. Basten T. Kunz J. Black M. Coffin D. Taylor	Time and the Order of Abstract Events in Distributed Computations, p. 29.
94/07	K.R. Apt R. Bol	Logic Programming and Negation: A Survey, p. 62.
94/08	O.S. van Roosmalen	A Hierarchical Diagrammatic Representation of Class Structure, p. 22.
94/09	J.C.M. Baeten J.A. Bergstra	Process Algebra with Partial Choice, p. 16.

94/10	T. verhoeff	The testing Paradigm Applied to Network Structure. p. 31.
94/11	J. Peleska C. Huizing C. Petersohn	A Comparison of Ward & Mellor's Transformation Schema with State- & Activitycharts, p. 30.
94/12	T. Kloks D. Kratsch H. Müller	Dominoes, p. 14.
94/13	R. Seljée	A New Method for Integrity Constraint checking in Deductive Databases, p. 34.
94/14	W. Peremans	Ups and Downs of Type Theory, p. 9.
94/15	R.J.M. Vaessens E.H.L. Aarts J.K. Lenstra	Job Shop Scheduling by Local Search, p. 21.
94/16	R.C. Backhouse H. Doornbos	Mathematical Induction Made Calculational, p. 36.
94/17	S. Mauw M.A. Reniers	An Algebraic Semantics of Basic Message Sequence Charts, p. 9.
94/18	F. Kamareddine R. Nederpelt	Refining Reduction in the Lambda Calculus, p. 15.
94/19	B.W. Watson	The performance of single-keyword and multiple-keyword pattern matching algorithms, p. 46.
94/20	R. Bloo F. Kamareddine R. Nederpelt	Beyond β-Reduction in Church's $\lambda\rightarrow$, p. 22.
94/21	B.W. Watson	An introduction to the Fire engine: A C++ toolkit for Finite automata and Regular Expressions.
94/22	B.W. Watson	The design and implementation of the FIRE engine: A C++ toolkit for Finite automata and regular Expressions.
94/23	S. Mauw and M.A. Reniers	An algebraic semantics of Message Sequence Charts, p. 43.
94/24	D. Dams O. Grumberg R. Gerth	Abstract Interpretation of Reactive Systems: Abstractions Preserving $\forall$CTL*, $\exists$CTL* and CTL*, p. 28.
94/25	T. Kloks	$K_{1,3}$-free and W_4-free graphs, p. 10.
94/26	R.R. Hoogerwoord	On the foundations of functional programming: a programmer's point of view, p. 54.
94/27	S. Mauw and H. Mulder	Regularity of BPA-Systems is Decidable, p. 14.
94/28	C.W.A.M. van Overveld M. Verhoeven	Stars or Stripes: a comparative study of finite and transfinite techniques for surface modelling, p. 20.
94/29	J. Hooman	Correctness of Real Time Systems by Construction, p. 22.
94/30	J.C.M. Baeten J.A. Bergstra Gh. Ştefanescu	Process Algebra with Feedback, p. 22.
94/31	B.W. Watson R.E. Watson	A Boyer-Moore type algorithm for regular expression pattern matching, p. 22.
94/32	J.J. Vereijken	Fischer's Protocol in Timed Process Algebra, p. 38.
94/33	T. Laan	A formalization of the Ramified Type Theory, p.40.
94/34	R. Bloo F. Kamareddine R. Nederpelt	The Barendregt Cube with Definitions and Generalised Reduction, p. 37.
94/35	J.C.M. Baeten S. Mauw	Delayed choice: an operator for joining Message Sequence Charts, p. 15.
94/36	F. Kamareddine R. Nederpelt	Canonical typing and Π-conversion in the Barendregt Cube, p. 19.
94/37	T. Basten R. Bol M. Voorhoeve	Simulating and Analyzing Railway Interlockings in ExSpect, p. 30.
94/38	A. Bijlsma C.S. Scholten	Point-free substitution, p. 10.
94/39	A. Blokhuis T. Kloks	On the equivalence covering number of splitgraphs, p. 4.

94/40	D. Alstein	Distributed Consensus and Hard Real-Time Systems, p. 34.
94/41	T. Kloks D. Kratsch	Computing a perfect edge without vertex elimination ordering of a chordal bipartite graph, p. 6.
94/42	J. Engelfriet J.J. Vereijken	Concatenation of Graphs, p. 7.
94/43	R.C. Backhouse M. Bijsterveld	Category Theory as Coherently Constructive Lattice Theory: An Illustration, p. 35.
94/44	E. Brinksma J. Davies R. Gerth S. Graf W. Janssen B. Jonsson S. Katz G. Lowe M. Poel A. Pnueli C. Rump J. Zwiers	Verifying Sequentially Consistent Memory, p. 160
94/45	G.J. Houben	Tutorial voor de ExSpect-bibliotheek voor "Administratieve Logistiek", p. 43.
94/46	R. Bloo F. Kamareddine R. Nederpelt	The λ-cube with classes of terms modulo conversion, p. 16.
94/47	R. Bloo F. Kamareddine R. Nederpelt	On Π-conversion in Type Theory, p. 12.
94/48	Mathematics of Program Construction Group	Fixed-Point Calculus, p. 11.
94/49	J.C.M. Baeten J.A. Bergstra	Process Algebra with Propositional Signals, p. 25.
94/50	H. Geuvers	A short and flexible proof of Strong Normalazation for the Calculus of Constructions, p. 27.
94/51	T. Kloks D. Kratsch H. Müller	Listing simplicial vertices and recognizing diamond-free graphs, p. 4.
94/52	W. Penczek R. Kuiper	Traces and Logic, p. 81
94/53	R. Gerth R. Kuiper D. Peled W. Penczek	A Partial Order Approach to Branching Time Logic Model Checking, p. 20.
95/01	J.J. Lukkien	The Construction of a small CommunicationLibrary, p.16.
95/02	M. Bezem R. Bol J.F. Groote	Formalizing Process Algebraic Verifications in the Calculus of Constructions, p.49.
95/03	J.C.M. Baeten C. Verhoef	Concrete process algebra, p. 134.
95/04	I. Hidders	An Isotopic Invariant for Planar Drawings of Connected Planar Graphs, p. 9.
95/05	P. Severi	A Type Inference Algorithm for Pure Type Systems, p.20.
95/06	T.W.M. Vossen M.G.A. Verhoeven H.M.M. ten Eikelder E.H.L. Aarts	A Quantitative Analysis of Iterated Local Search, p.23.
95/07	G.A.M. de Bruyn O.S. van Roosmalen	Drawing Execution Graphs by Parsing, p. 10.
95/08	R. Bloo	Preservation of Strong Normalisation for Explicit Substitution, p. 12.
95/09	J.C.M. Baeten J.A. Bergstra	Discrete Time Process Algebra, p. 20
95/10	R.C. Backhouse R. Verhoeven O. Weber	Math$\int$pad: A System for On-Line Prepararation of Mathematical Documents, p. 15
95/11	R. Seljée	Deductive Database Systems and integrity constraint checking, p. 36.
95/12	S. Mauw and M. Reniers	Empty Interworkings and Refinement

		Semantics of Interworkings Revised, p. 19.
95/13	B.W. Watson and G. Zwaan	A taxonomy of sublinear multiple keyword pattern matching algorithms, p. 26.
95/14	A. Ponse, C. Verhoef, S.F.M. Vlijmen (eds.)	De proceedings: ACP'95, p.
95/15	P. Niebert and W. Penczek	On the Connection of Partial Order Logics and Partial Order Reduction Methods, p. 12.
95/16	D. Dams, O. Grumberg, R. Gerth	Abstract Interpretation of Reactive Systems: Preservation of CTL*, p. 27.
95/17	S. Mauw and E.A. van der Meulen	Specification of tools for Message Sequence Charts, p. 36.
95/18	F. Kamareddine and T. Laan	A Reflection on Russell's Ramified Types and Kripke's Hierarchy of Truths, p. 14.
95/19	J.C.M. Baeten and J.A. Bergstra	Discrete Time Process Algebra with Abstraction, p. 15.
95/20	F. van Raamsdonk and P. Severi	On Normalisation, p. 33.
95/21	A. van Deursen	Axiomatizing Early and Late Input by Variable Elimination, p. 44.
95/22	B. Arnold, A. v. Deursen, M. Res	An Algebraic Specification of a Language for Describing Financial Products, p. 11.
95/23	W.M.P. van der Aalst	Petri net based scheduling, p. 20.
95/24	F.P.M. Dignum, W.P.M. Nuijten, L.M.A. Janssen	Solving a Time Tabling Problem by Constraint Satisfaction, p. 14.
95/25	L. Feijs	Synchronous Sequence Charts In Action, p. 36.
95/26	W.M.P. van der Aalst	A Class of Petri nets for modeling and analyzing business processes, p. 24.
95/27	P.D.V. van der Stok, J. van der Wal	Proceedings of the Real-Time Database Workshop, p. 106.
95/28	W. Fokkink, C. Verhoef	A Conservative Look at term Deduction Systems with Variable Binding, p. 29.
95/29	H. Jurjus	On Nesting of a Nonmonotonic Conditional, p. 14
95/30	J. Hidders, C. Hoskens, J. Paredaens	The Formal Model of a Pattern Browsing Technique, p.24.
95/31	P. Kelb, D. Dams and R. Gerth	Practical Symbolic Model Checking of the full μ-calculus using Compositional Abstractions, p. 17.
95/32	W.M.P. van der Aalst	Handboek simulatie, p. 51.
95/33	J. Engelfriet and JJ. Vereijken	Context-Free Graph Grammars and Concatenation of Graphs, p. 35.
95/34	J. Zwanenburg	Record concatenation with intersection types, p. 46.
95/35	T. Basten and M. Voorhoeve	An algebraic semantics for hierarchical P/T Nets, p. 32.
96/01	M. Voorhoeve and T. Basten	Process Algebra with Autonomous Actions, p. 12.
96/02	P. de Bra and A. Aerts	Multi-User Publishing in the Web: DreSS, A Document Repository Service Station, p. 12
96/03	W.M.P. van der Aalst	Parallel Computation of Reachable Dead States in a Free-choice Petri Net, p. 26.
96/04	S. Mauw	Example specifications in phi-SDL.
96/05	T. Basten and W.M.P. v.d. Aalst	A Process-Algebraic Approach to Life-Cycle Inheritance Inheritance = Encapsulation + Abstraction, p. 15.
96/06	W.M.P. van der Aalst and T. Basten	Life-Cycle Inheritance A Petri-Net-Based Approach, p. 18.
96/07	M. Voorhoeve	Structural Petri Net Equivalence, p. 16.
96/08	A.T.M. Aerts, P.M.E. De Bra, J.T. de Munk	OODB Support for WWW Applications: Disclosing the internal structure of Hyperdocuments, p. 14.
96/09	F. Dignum, H. Weigand, E. Verharen	A Formal Specification of Deadlines using Dynamic Deontic Logic, p. 18.
96/10	R. Bloo, H. Geuvers	Explicit Substitution: on the Edge of Strong Normalisation, p. 13.
96/11	T. Laan	AUTOMATH and Pure Type Systems, p. 30.
96/12	F. Kamareddine and T. Laan	A Correspondence between Nuprl and the Ramified Theory of Types, p. 12.
96/13	T. Borghuis	Priorean Tense Logics in Modal Pure Type Systems, p. 61
96/14	S.H.J. Bos and M.A. Reniers	The I^2C-bus in Discrete-Time Process Algebra, p. 25.
96/15	M.A. Reniers and J.J. Vereijken	Completeness in Discrete-Time Process Algebra, p. 139.
96/17	E. Boiten and P. Hoogendijk	Nested collections and polytypism, p. 11.

96/18	P.D.V. van der Stok	Real-Time Distributed Concurrency Control Algorithms with mixed time constraints, p. 71.
96/19	M.A. Reniers	Static Semantics of Message Sequence Charts, p. 71
96/20	L. Feijs	Algebraic Specification and Simulation of Lazy Functional Programs in a concurrent Environment, p. 27.
96/21	L. Bijlsma and R. Nederpelt	Predicate calculus: concepts and misconceptions, p. 26.
96/22	M.C.A. van de Graaf and G.J. Houben	Designing Effective Workflow Management Processes, p. 22.
96/23	W.M.P. van der Aalst	Structural Characterizations of sound workflow nets, p. 22.
96/24	M. Voorhoeve and W. van der Aalst	Conservative Adaption of Workflow, p.22
96/25	M. Vaccari and R.C. Backhouse	Deriving a systolic regular language recognizer, p. 28
97/01	B. Knaack and R. Gerth	A Discretisation Method for Asynchronous Timed Systems.
97/02	J. Hooman and O. v. Roosmalen	A Programming-Language Extension for Distributed Real-Time Systems, p. 50.
97/03	J. Blanco and A. v. Deursen	Basic Conditional Process Algebra, p. 20.
97/04	J.C.M. Baeten and J.A. Bergstra	Discrete Time Process Algebra: Absolute Time, Relative Time and Parametric Time, p. 26.
97/05	J.C.M. Baeten and J.J. Vereijken	Discrete-Time Process Algebra with Empty Process, p. 51.