Program Verification with the Mathematical Software System Theorema

Diplomarbeit

zur Erlangung des akademischen Grades
Diplom−Ingenieur (Fachhochschule)

Eingereicht von

Martin Kirchner

Juli 1999

Betreuer: Univ.−Prof. DDr. Bruno Buchberger, RISC Hagenberg
Begutachter: Dipl.−Ing. Dr. Franz Lichtenberger, FHS Hagenberg

Abstract

With *Theorema,* the Research Institute for Symbolic Computation in Hagenberg attempts to develop an integrated and comprehensive mathematical environment, with emphasis on a "natural" mathematical language with clear semantics and the support of mathematical proving. Within the scope of the thesis we have extended *Theorema* by a verification system for simple sequential programs.

The verification system is designed for educational purposes. It provides an environment for Software Engineering students to learn the basics of formal methods in software development: the formal specification of programs, the annotation of programs and the formal proof of correctness.

The most important part of the verification system is a tool, which takes a specification and an annotated program and generates so-called verification conditions. These are mathematical statements which are to be proved; either by "hand", using an existing automated *Theorema* prover, or with an interactive interface to a *Theorema* prover. In the thesis we discuss the implementation of such a verification condition generator based on a "predicate transformer" function, and describe the verification rules in detail.

The user interfaces for specifications and programs follow the conventions of *Theorema* and are also presented elaborately. The programs to verify are written in *Theorema Procedural Language*, a small language designed in the scope of this thesis to enable procedural programming within the functional environment of *Theorema*. The embedding of this language into the *Theorema* system allows to *specify, implement, verify* and *execute* programs within one uniform environment. We demonstrate the use of the system in numerous examples and provide a tutorial for future users.

Kurzfassung

Mit *Theorema* wird am RISC Institut in Hagenberg ein integriertes und umfassendes mathematisches Softwaresystem entwickelt, dessen Schwerpunkte auf einer "natürlichen"mathematischen Sprache mit klarer Semantik sowie der Unterstützung von mathematischen Beweisen liegen. Im Rahmen dieser Diplomarbeit haben wir *Theorema* um ein Verifikationssystem für einfache sequentielle Programme erweitert.

Das Verifikationssystem ist für didaktische Zwecke ausgelegt. Studenten in Software Engineering sollen mit dem System die Grundlagen formaler Methoden bei der Softwareentwicklung lernen: Programme formal zu spezifizieren, zu annotieren und die Korrektheit zu beweisen.

Der wichtigste Teil des Verifikationssystems ist ein Werkzeug, das aus der Spezifikation und dem annotierten Programm sogenannte "Verifikationsbedingungen" generiert. Dies sind mathematische Aussagen, die es zu beweisen gilt; entweder mit der "Hand", mit existierenden automatischen *Theorema*-Beweisern oder mit einer interaktiven Beweiser-Schnittstelle. Wir diskutieren die Implementierung dieses Werkzeugs auf Basis einer "Predicate Transformer"−Funktion und beschreiben die verwendeten Verifikationsregeln im Detail.

Die Benutzerschnittstellen zur Eingabe von Spezifikationen und Programmen sind an die Koventionen von *Theorema* angepaßt und werden ebenfalls ausführlich dargestellt. Die zu verifizierenden Programme werden in *Theorema Procedural Language* geschrieben, einer im Rahmen dieser Diplomarbeit entwickelten kleinen Sprache, die prozedurale Programmierung in der funktionalen Umgebung von *Theorema* ermöglicht. Die Einbettung der Sprache in *Theorema* erlaubt die *Spezifikation*, *Implementierung*, *Verifikation* und *Exekution* von Programmen innerhalb einer einheitlichen Umgebung. Die Benutzung des Systems wird anschaulich in einem "Tutorial" und anhand zahlreicher Beispiele demonstriert.

Acknowledgements

First of all, I want to thank my supervisor, Franz Lichtenberger. He is currently reorganizing mathematics education in Software Engineering and was the actual initiator of this thesis. He always had time when I needed it and gave me best support.

I am also very grateful to Wolfgang Windsteiger; without his help I would not have been able to implement the verification system within the *Theorema* system. His ideas in the design of the system and his advice in several critical issues were very valuable.

It was an interesting experience to participate in the *Theorema* Working Group. Many members were very helpful; I want to mention in particular Daniela Vasaru–Dupre, Franz Kriftner, Koji Nakagawa and Tudor Jebelean (in alphabetical order). Thanks to Daniela for swapping the work place with me.

Furthermore, I am indebted to Felix Kossak. He has organized that I could write my thesis at the RISC institute, and he gave me a "roof over my head" when I came to Hagenberg in March.

I am very grateful to Gregory Curtis for proofreading the thesis and, moreover, for being a perfect English teacher over the years.

I want to express my deep respect for Bruno Buchberger. He has given me a "taste" of what is possible and he has taught me things which go beyond mathematics. Without his efforts, I would not have been in Hagenberg; more than that: nothing would have happened in this little town.

Of course I am very grateful to my parents for the financial support during my study in Hagenberg. It would not have been possible without them.

Finally, I want to thank my "fair lady" Petra that she came to Hagenberg with me for the time of the thesis.

M. K.

Note

Due to a problem with *Mathematica* the postscript version of the thesis does not contain figures. The (three) figures are not essential for the general understanding. For a "full" version please refer to the printed document.

1. Introduction

1.1 Motivation

What should students in Software Engineering learn in their mathematics lessons?

We believe that it would be most important that the students receive basic training for the use of *formal methods in software development*: to learn how to express themselves in a formally exact way, and to learn how to prove, for example the correctness of programs [Lichtenberger98].

However, in mathematical education, these issues are traditionally either just taken for granted or only touched on the surface.

The Polytechnical University Hagenberg, in contrast, intends to put great emphasis on exactly those areas in the first year of the Software Engineering curriculum. This will soon be practically supported by several software tools:

- *Mathematica,* a commercial computer algebra system
- *Theorema*, a comprehensive mathematical environment built "on top of" *Mathematica*
- An interactive proving interface within *Theorema*
- A program verification system within *Theorema*, the topic of this thesis

As opposed to doing maths with pencil and paper, all these tools accept only *syntactically correct* input, and it is anticipated that this will naturally lead the students to express themselves in a formally exact way.

Nowadays, software increasingly penetrates nearly all areas of life; the systems get bigger and more complex. On the other hand, the quality of software is still a major problem; software products are in general highly unreliable (compared to products of other technical areas) mostly due to errors in the code. In this respect, software engineers are often not aware of the high responsibility placed upon them.

Considering this, we think that it is a natural approach to raise the consciousness of the young software engineers by means of program verification, a scientific method to really *ensure* that a program is correct for *all* possible input (with respect to its specification).

With the verification system developed in the scope of this thesis, we hope to contribute to a modern education for more professional software engineers. Together with the interactive proving interface, which is developed in another thesis [Kossak99], we provide a didactical framework for formal program verification in one uniform environment.

1.2 The Environment: *Theorema*

Theorema is a mathematical software system which is being developed at the RISC-Insitute (Research Institute for Symbolic Computation) in Hagenberg, Austria by the *Theorema* Working Group under the leadership of Professor Bruno Buchberger.

Theorema is designed as a comprehensive working environment for mathematicians and mathematically-oriented scientists. It aims to close the gap between computing and proving by adding proving support to an existing computer algebra system (*Mathematica*).

Theorema supports all key aspects of numerical and symbolic computation as well as the presentation of the work performed within this environment.

The current distinctive features of *Theorema* are [Buchberger98]:

- "Usual" mathematical syntax (close to natural style)
- Integration of an existing rewrite-rule language (*Mathematica*) into the logic
- Imitation of human proof style
- Natural language formulation of proofs
- Combination of functor-style programming with proving
- Integration of proving into textbook generation and interactive teaching
- Emphasis on generating the routine parts of proofs

Theorema is written "on top of" *Mathematica* and takes advantage of its computational power, its convenient typesetting and its extensible syntax facility. *Theorema* also uses *Mathematica* concepts like nested cells and hyperlinks in the notebooks.

Up-to-date information on *Theorema* can be found on the *Theorema* homepage www.theorema.org.

1.3 Overview

In the scope of this thesis we have built a simple and easy-to-use system where one can *specify* and *implement* sequential programs, *verify* them and *execute* them in one uniform environment, *Theorema*.

- We have defined a source language within *Theorema* with language constructs to represent all major concepts of procedural programming (assignment, sequence, condition, loop, block, procedure call), including assertions and loop invariants.
- We have established environments for writing specifications and programs consistent with the general input style of *Theorema*.

- A major part of our work was to build a verification condition generator which, based on certain rules, translates formally specified programs into mathematical formulae which then can be proved.
- We have shown how to prove verification conditions with existing *Theorema* provers.
- We have made the programs executable within *Theorema*; all the constraints (pre-, postconditions and invariants) can in general be verified during execution.
- We have tested the system with numerous examples and have written a very practical tutorial for the future users.

In the next section we provide the reader with a concise "Theoretical Background", where the basic notions are explained and several practical aspects of program verification are discussed.

After giving a "History and State of the Art" of verification systems, we describe the top-level structure of our system in the next section, "The Verification System". Then we go into more detail presenting the *Theorema Procedural Language*, its concepts, a detailed description of the grammar as well as the semantics of the various language constructs.

In the subsequent section, "Implementation of the Verification Condition Generator", we analyze different approaches to an implementation and present a natural solution. After that we explain the "Verification Rules" which the VCG implements and give an example for every rule. In the next sections, we provide future users with a tutorial and a number of examples which show the capabilities of the system.

Concluding, we give a critical "Summary" on what has been achieved and what would be the next steps in the development of the system.

2. Theoretical Background

2.1 Basic Notions

2.1.1 The Terms Statement, Condition, and Program

In the scope of the thesis *statement* will mostly denote single commands of programming languages (like assignments, conditionals, etc) as this meaning is very common in Computer Science.

A *program* is built up of a sequence of (program-) statements and usually represents a complete algorithm. A program can be either a procedure (with output parameters but without a return value) or a function procedure (with a return value but without output parameters).

Also, in the context of programming, we decided to call a formula which denotes a truth value *condition* (like pre- or postcondition or the condition in an *IF* statement).

2.1.2 Formal Specification

The main question in designing a program is whether it is *correct*. If that is not the case then it does not make much sense to start discussing other issues like efficiency, stabilility, structuredness, etc. But what does *correctness* really mean? It sounds "absolute", but if one analyzes the word, one can see that this is a binary predicate: Something can only be correct relative to some requirements. In our case an a program can only be called correct if there is a *specification* which formalizes the requirements and if the program *meets* the specification.

Many software engineers use natural language to define a problem or the requirements for an program. However, mostly due to its ambiguity, natural language is insufficient in order to specify a program *exactly* [Klaeren83]. Moreover, if one wants to really be sure that a program is correct (relative to a specification), one has to *prove* it formally. Therefore the specification needs to be on a formal level, too. For a discussion of the different levels of formality see [Gries81].

A specification basically determines *what* (statically) the problem actually is, the program describes *how (*dynamically) a certain input state is transformed into a desired output state. The specification determines the "effect" of the program and makes a static connection between what is "given" and what is to be "found". To specify programs, it suffices to define the valid initial states of the variables *before* the program is executed (in a *precondition* or *input assertion*) and the desired states of the variables *after* the program has been executed (in a *postcondition* or *output assertion*).

For example, a program which computes the factorial *f* of a natural number *n* could have the following specification:

Precondition: $n \geq 0$

Postcondition: $f = n!$

We have already defined in the term "program" in the previous section. A program meeting this specification could, for example, be:

```
f := 1;
i := n;
WHILE[i > 1,
  f := f * i;
  i := i - 1]
```

2.1.3 Correctness Statement

The following is called the "Meta Problem" of program verification (x,y are vectors of variables):

Given: a Problem P (specified by a precondition Q(x) and a postcondition R(x,y))
and an algorithm A.

Quesition: Does algorithm A solve the problem P?

This question is called *correctness statement* and should be read like this:

"For all possible values of the input variables x:
If the precondition Q(x) is fulfilled
the algorithm A yields such values of the output variables y
such that the postcondition R(x,y) is fulfilled."

If this statement is true we say that the algorithm A meets the specification.

The correctness statement is usually written as a so-called "*Hoare Triple*":

{Q}A{R}.

Such a triple is the foundation of *Hoare Logic*, which has its name from its creator, Tony Hoare. In the late sixties, early seventies, he developed a verification theory for procedural languages based on predicate logic [Hoare69]. As mentioned in [Gries81], this was the starting point of a whole school of research on the axiomatic definition of programming languages. In many publications the semantics of programming languages have been defined with Hoare Logic (e.g. Pascal in [Hoare/Wirth73] or [Hohlfeld92]).

2.1.4 Partial and Total Correctness

Sometimes an algorithm (or a program, resp.) doesn't stop but gets caught in an "infinite loop". The correctness statement above is true only if the algorithm terminates. It is called "statement of *total correctness*".

However, usually it is easier to build the (weaker) statement of *partial correctness* (or *conditional correctness*):

"...If the precondition Q(x) is fulfilled *and* the Algorithm A terminates

the Algorithm A yields ..."

Total correctness is logically equivalent to partial correctness with an additional *statement of termination*:

"...if Q(X) is fulfilled, the program stops."

So if one wants to prove the total correctness of a program relative to a specification one has to prove the partial correctness *and* the termination. We will show later how that can be done.

But how do we prove that a program is correct (relative to a specification)? How can we reason with regard to programs?

2.1.5 Proving Partial Correctness

The usual method in program verification is to *translate* a formally specified program into proveable predicate logic formulae called *verification conditions* (sometimes also called *proof obligations*) such that the truth of these verification conditions implies the correctness of the program.

Verification conditions do not contain any references to programming language phrases, they are formulae describing relationships among variables of the underlying data types of the programming language.

Hoare Logic provides a set of rules for this task of "translation" into verification conditions.

For example the rule for an assignment is as follows:

The verification condition for

$\{Q\}\ V := E\ \{R\}$

is $Q \Rightarrow R'$,

where R' is obtained by substituting every occurence of V in R by E.

Example:

Given: A specification for a program Square which should take (an integer number) x as input parameter and return an output parameter $y = x^2$.

Find: (1) A program which solves this problem.

(2) A proof of correctness.

Solution:

(1) $\{ \text{True} \}\ y := x * x\ \{ y = x^2 \}$

(2) With the rule for assignments one obtains as verification condition

$True \Rightarrow x * x = x^2$,

which is obviously true.

Likewise there are rules for other statements of procedural languages in Hoare Logic.

However, there are some statements (e.g. WHILE) which need additional information, so-called *annotations*, in order to be provable.

2.1.6 Annotation of Programs

A program is *annotated* by inserting statements (often called *assertions* or *invariants*) expressing conditions at certain intermediate points. These describe relationships between variables which are meant to "hold" during exectution. That means, whenever control passes through that point, the assertion should evaluate to *True*.

Assertions could be inserted at any point in the control structure, but in fact they are only necessary in loops as so-called *loop invariants* (*inductive assertions).* During execution, loop invariants have to evaluate to true *before and after* each iteration (invariant means constant or unchanging).

A loop has many invariants. For example, the predicate $x*0=0$ is an invariant of every loop since it is always true. An appropriate loop invariant for the example "Factorial" (shown in the section 2.1.2, "Formal Specification") would e.g. be $(i \geq 0) \wedge (f = n! / i!)$.

Annotating a program is often non-trivial and needs a good understanding of how the algorithm works. The idea of the invariants is mostly identical to the basic design idea. That is a main reason why [Dijkstra76] recommends that the development of the program should go hand in hand with finding the right annotations and the program verification. [Gries81] shows that programming is more effective if one thinks about the invariant before coding a loop and also gives heuristics for developing invariants.

Logically there is no difference between an assertion at the beginning of a program (the precondition), one at the end of a program (the postcondition) or somewhere in the middle: they all express conditions that are meant to hold at these points during execution. Nevertheless we want to view the pre- and postcondition as "special annotations" for a program and combine them in a "specification".

2.1.7 Proving Total Correctness

To verify *total correctness*, it is necessary to prove that every program statement terminates, in particular iterative statements like, for instance, WHILE statements.

Therefore we need an additional technique in order to *guarantee* termination. The common method is to construct a term *t* which has the following properties:

- *t* is based on program variables which change with every iteration of the loop,
- the value of *t* is always a natural number (no matter how often the loop is iterated), and
- the value of *t* is decreased with *every* single iteration.

If one can find such a term, then obviously the loop *must* be finite. An appropriate termination term for the example "Factorial" (shown in the section 2.1.2, "Formal Specification") would e.g. simply be *i*.

This property of a loop is often called *variant* of the loop, as opposed to the *invariant* discussed above.

There is another situation where nontermination could occur, namely where one program calls another program (a procedure or a function) which then does not terminate. However, the termination of those "subprograms" is has to be be proved separately.

Of course also recursive or mutually recursive calls of "subprograms" could lead to nontermination. This issue is discussed in big detail in [Homeier95].

2.1.8 Weakest Preconditions and Predicate Transformation

Above we have shown a verification method using inference rules of Hoare Logic. However, there exists another strategy for program verification using the so-called "weakest precondtion predicate transformer" (*wp*) developed by E. W. Dijkstra [Dijkstra76]. The two approaches are related, although different.

Let's assume that we want to verify a program where we know the postcondition but *not* the precondition:

$\{?\}\ S\ \{R\}$

In general there could be arbitrarily many preconditions Q which are valid for the program S and postcondition R. However, there is one precondition which describes the *maximal* set of possible initial states such that the execution of S would lead to a state satisfying R. It is called the *weakest precondition*.

But what does "weak" really mean? A condition b is *weaker* than c if and only if $c \Rightarrow b$. Conversly, c would in this case be called *stronger* than b.

Again, in other words: If a condition is weaker than another one, it describes more possible states.

Simple Example:

Given: A program $S : y := x * x$

A postcondition $R : y \geq 4$

Find: The weakest precondition Q.

Solution:

The precondition $x \geq 2$ would guarantee that R is valid after execution.

Stronger preconditions like $x \geq 3$, $x = 3$, etc. would also be valid preconditions.

However, the weakest precondition is:

$Q : (x \leq -2) \vee (x \geq 2)$

The weakest precondition of a program S and a postcondition R is denoted by

$wp\ (S, R)$

and is a predicate which describes the set of all initial states that will *guarantee termination*

of S in a state satisfying R. That can also be expressed as a Hoare Triple:

$$\{wp(S,R)\}\ S\ \{R\}$$

So if one wants to verify $\{Q\}\ S\ \{R\}$ using *wp* one can first determine $wp(S, R)$ and then prove

$$Q \Rightarrow wp(S, R).$$

Like in Hoare Logic, there are inference rules for *wp* which translate program statements into mathematical formulae; however, the rule for iterative statements like WHILE is not applicable in practice [Futschek89].

The function *wp* for one certain statement can be viewed as a function which only takes a predicate (the postcondition) and returns another predicate (the weakest precondition). Therefore *wp* is also called a *predicate transformer*. More on predicate transformation can be found in [Dijkstra76].

With the function *wp* it is also possible to define the semantics of programming languages [Dijkstra76].

2.2 Discussion of Program Verification

2.2.1 Motivation for Program Verification

Since writing software is a creative task done by humans, it is prone to errors. Virtually every major software system on the market contains errors. Software usually gives correct results "most of the time", and occasional collapses of the system are viewed as "normal". Such a degree of unreliability would not be acceptable in most other technical fields. However, it is tolerated in software because there is no apparent alternative.

In recent years a lot of improvements in the software development process have been made, the software organizations nowadays are in general much more "professional" than they used to be. Still, if one looks at the quality of software nowadays, it is obvious that the current techniques applied are not sufficient.

We believe that it would be a major improvement if formal methods and, in particular, program verification became widely used. The reliability of software would increase immensly; entire categories of (e.g. run-time) errors could be eliminated forever. And we dare say that software would become even cheaper.

Nevertheless, program verification is already performed successfully in safety-critical systems (in fields like air and railway traffic control, aircraft control, banking, etc.), where software errors can easily lead to catastrophes.

2.2.2 Program Verification versus Testing

In principle there are two methods to check whether a program does what is intended: testing and verification.

- In the process of *testing,* the program is executed with a selection of input values and it is checked whether the program produces the desired effect. However, usually programs can *not* be tested for *all* possible combinations of input data. Thus testing can not really *ensure* the correctness of a program.
- In contrast, *verification* is a formally exact method to *prove* the consistency between specification and program for *every* possible input. By means of verification, one can immensly increase the confidence in the correctness of a program and really *be sure* that a program meets its specification.

Typically, a software system goes through numerous tests and still fails under some unfortunate combination of circumstances. As expressed by Dijkstra, the problem is: "Program testing can be used to show the presence of errors, but never to show their absence!" [Dijkstra72] .

In contrast to testing, formal verification can trace *every* path through a system and consider *every* possible combination of circumstances; one can be certain that nothing has been left out. It relies on mathematical methods of proof to ensure the *completeness* and *correctness* of every step.

In some cases testing is extremely time-consuming, especially when programs would need an unknown amount of time (e.g. a few months) until they terminate. Then program verification is a very reasonable alternative.

Nevertheless, we do not want to claim that a verified program should not be tested, too.

Testing is performed on the object code. If the compiler introduces errors it is only possible to find them with testing, because formal verification is performed on the source code.

If one has proved that a program meets its specification, it still can be the case that the specification does not exactly capture the requirements (in practice, clients of a software company often do not know exactly what they want and their requirements are likely to change while the software is being developed). In such cases, testing can show very soon if there is a problem.

2.2.3 Benefits of Program Verification

Clearly, program verification usually causes more work than testing. However, it has some additional benefits which are often overlooked:

- With program verification, one can gain deep insight into the program and really understand the idea behind it and its limitations. One often gets a clue for a better program [Buchberger/Lichtenberger81].
- Program verification supports goal-oriented program development [Gries91].
- The annotations for program verification are a good documentation for the program, too [Gries91].

A very high percentage of the costs of a software project is due to the detection of errors, especially in later phases of the software life cycle. The sooner an error is detected, the

cheaper it is. So clearly, program verification can reduce the costs for error detection and maintenance.

Even if a proof of correctness fails, it usually indicates what exactly has to be corrected in the program. In addition, it helps to detect errors in the specification or gives information about how the loop invariants have to be changed or extended. Last but not least, an unsuccessful proof often gives more insight into the problem and its solution (or contributes to better solutions).

2.2.4 Computer-Aided Program Verification

If one performs program verification with pencil and paper, one soon encounters the following problems. First, the generation of the verification conditions can often be quite tedious and it is a task which is highly prone to errors. Furthermore, the proofs are often long and involve a lot of fiddly details, even for small programs.

This situation naturally calls for computer support in the verification process which immensly reduces the probability of making errors in the generation of verification conditions or in the proof (of course, this requires correct tools!).

A *verification condition generator* implements the inference rules for translating a formally specified (and annotated) program into predicate logic formulae, which are to be proved. It can be compared to a compiler: It receives a program as input, analyzes it and produces some output depending on the syntactic structure of the program. In fact, it *is* a compiler which produces code for an abstract machine (the theorem prover) with predicate logic as the machine language.

A *verification condition generator generator* is a tool similar to a "compiler compiler". It takes a description of the syntax and semantics of a programming language as input and generates a verification condition generator [Baumgartner88].

Of course, also computer *support for the proof* of the verification conditions is desireable. There are several possible levels, from a tool which just records and maintains the proof as it is constructed step by step, via a semi-automatic proof (where the user has to interact in crucial steps of the proof) to a fully-automatic proof.

Many verification conditions are "trivial" (like for instance $x>0 \Rightarrow x>0$) such that a simplifier suffices for the proof step. On the other hand, some verification conditions can not be proved at all. Nevertheless, we presume that within *Theorema* it will soon be possible to have an automatic prover for *typical* verification conditions.

Summarizing, with the aid of strong tool support for routine tasks, the human can concentrate on the *creative steps* in program verification; finding appropriate loop invariants and proving of simplified verification conditions which cannot be proved automatically.

2.2.5 Limitations of Program Verification

We do not want to claim that program verification is the "ultimate solution" to the problems of programming. There are some severe limitations which have to be kept in mind.

Most of the limitations of program verification are due to its relative complexity.

- Only programs in a simple programming language with a very *limited set of language constructs* can be verified (e.g. a lot of problems arise with pointers or more sophisticated language concepts like classes). Therefore verification is usually performed only on a small-scale algorithmic level.
- In most cases not complete software systems are verified but only single "safety-critical" algorithms. But how can one make sure that *everything* which is critical has been considered?
- Annotating a program can be very difficult if it is not done parallel to programming.
- It is usually not possible to prove fully automatically and proofs can get long and complicated. In fact, verification conditions don't even have to be provable.

A formal specification is the prerequisite for program verification (E.G. Dijkstra: "A program is just half of a conjecture"). The only thing which can be said about a verified program is that it *meets the given formal specification*.

Writing a specification is somehow similar to writing a program – though at a higher level of abstraction. Trivial problems are quite easy to specify. But it is much more likely to make mistakes with "real" problems. And, for example, specifying what it means for a user interface to be correct needs additional, sophisticated techniques.

It is important to keep in mind that even if a perfectly specified program is formally verified we cannot absolutely trust its execution.

- The operating system or even the hardware could have errors.
- Programs can still be used incorrectly or in a way which was not originally intended.
- The program verification itself might contain an error. There may be errors in the verification rules used, in the implemetation of the verification condition generator (if such a tool is used) or in the proof (e.g. if it was performed by hand with pencil and paper).

What can be concluded from the previous sections is that verification should always be used "hand in hand" with other techniques (testing, hazard analysis, etc.).

2.2.6 Why Is Program Verification not Popular?

In spite of the relative age and maturity of program verification, it is rarely used in practice. It is almost a strictly academic domain; the average software engineer has only a very vague idea of program verification. What could be the reasons for that?

Program verification can only be performed by someone who has appropriate knowledge and experience with formal methods (e.g. one who is able to reason at an abstract mathematical level). Clearly, the training in this area is usually far from being sufficient.

Program verification is easiest if done parallel to program development. Often a programmer is quite impatient and wants to see the result of his work very soon, instead of "doing some boring maths" in advance. This is another educational problem; more emphasis should be placed on training students in a professional personal software development process.

Another reason why program verification is not widely used is that the existing software tools are just too complex and usually only prototypes. These tools require high mathematical sophistication for working within the complex proof systems.

Concluding, we want to list a few things which should, from our point of view, be done in order to make program verification more popular:

- Students should learn the basics of program verification early in their education so that they become familiar with them, and clearly see the benefits.
- Tools should be developed which are easy to use without in-depth mathematical knowlegde (more requirements for verification tools are listed in [Clarke/Wing96]).
- *Typical* verification conditions should be automatically proveable.

We believe that the verification system built in the scope of this thesis is a step in the right direction, towards a more professional and scientific education in Software Engineering.

3. History and State of the Art

3.1 History

This overview of the history of program verification was partially taken from [Hohlfeld92] and [Homeier95].

The origins of systems for formal program verification date back to the sixties, early seveties. One of the first verification systems was presented by J.C. King [King70], and was based on the verification theory of Floyd [Floyd67]. It was written in Assembler for the verification of programs in a simple ALGOL-dialect. It did not contain an automatic prover but a powerful simplifier.

The first implementations of verification condition generators appeared in the early 1970's, of which the verifier for partial correctness of Igarashi, London, and Luckham [Igarashi/London/Luckham75] is a characteristic example. It was based on an axiomatic semantics for a substantial subset of Pascal (including procedures). The only mechanized part was the verification condition generator itself.

Ragland also developed a verification condition generator and even verified its soundness [Ragland73]. It was written in Nucleus, a language Ragland had invented to have the expressiveness to write a verification condition generator, and also to be verifiable itself.

Probably the best-known verification system of that time is the Stanford Pascal Verifier [SPV79]. It consisted of a verification condition generator (the first based on the Hoare calculus), a simplifier and a "Rule Handler" (where self-defined rules could be given by the user as axioms for the proof). This verification system was successfully used to prove the correctness of "real" applications, as for example of a complier.

Another system, which was developed in the early 80's, was called GYPSY (University of Texas, [GOO85]). It was written in LISP for the verification of a specially designed language similar to Pascal, but containing specific statements for program verification. GYPSY was a comprehensive collection of tools consisting of a syntax-driven editor, a vecrification condition generator, an algebraic simplifier and an automatical prover.

Later in the 80's, Gray presented a verification condition generator for teaching axiomatic semantics to undergraduate students [Gray87]. The verification conditions produced were to be proven by hand by the students. The language considered resembled a subset of Pascal, and contained input and output statements, as well as procedure calls with both value and variable parameters.

Apart of King's Verifier all the systems mentioned were designed for the verification of PASCAL-like languages. However, Boyer and Moore developed verification systems for LISP [Boyer/Moore75] and FORTRAN [Boyer/Moore80]. The powerful verification condition generators covered much of these "real" programming language, with the advantage that the verification conditions generated could then be proven with the aid of the Boyer-Moore theorem prover.

Moriconi wrote the first verification condition generator generator. His Meta-VCG [Moriconi80] produced a verification condition generator from a formal definition of the semantics of a language.

3.2 State of the Art

In contrast to the stand-alone tools for program verification in the past, there have been several shallow embeddings of programming languages in theorem proving environments in recent years, including the creation of verification condition generators. New concepts have been developed and big advances have been made in the verification of concurrent programs.

Nowadays, program verification is only one branch in the large field of formal methods. The research in that area nowadays focuses on specification languages and a verification method called *Model Checking,* where all possible states and paths in a program are *tested*. Furthermore, verification systems these days are more widely used in *hardware* verification.

For a good overview of the current state of formal methods, see [Clarke/Wing96]. An excellent index and a collection of information with links is located in the Formal Methods Virtual Library [H1]. A comprehensive survey on tools supporting the use of formal methods can be found in the FME Tools Database [H2].

In the following we present three selected state-of-the-art verification systems, PVS, Sunrise and KIV.

3.2.1 PVS

The Program Verification System PVS is the most popular verification system nowadays. The following information is taken from the homepage of PVS [H3].

PVS consists of a specification language integrated with support tools and a theorem prover. It is intended to capture the state-of-the-art in mechanized formal methods that it can be used for significant applications.

PVS uses Gnu or X Emacs to provide an integrated interface to its specification language and prover. Extensive help, status−reporting and browsing tools are available, as well as the ability to generate typeset specifications (in user−defined notation) using LaTeX. Proof trees and theory hierarchies can be displayed graphically using Tcl/Tk.

The theorem prover of PVS provides a collection of ∃inference procedures that are applied interactively under user guidance within a sequent calculus framework.

However, PVS is still a research prototype, constantly extended by new capabilities. PVS is a large and complex system and it takes several month to learn to use it effectively. Nevertheless, PVS has been installed at hundreds of sites in North America, Europe, and Asia. Applications include a microprocessor for aircraft flight-control, diagnosis and scheduling algorithms for fault-tolerant architectures, and requirements specification for portions of the Space Shuttle flight-control system.

3.2.2 Sunrise

Sunrise is a verification system within the HOL theorem prover [H4]. It contains a deep embedding of a small Pascal-like programming language ("Sunrise") within the HOL theorem prover, and provides a verification condition generator for the semi-automatic creation of proofs of total correctness for programs within HOL.

The verification condition generator is itself verified as sound, and that soundness proof is contained within and checked by the HOL system. This permits the significant simplification of proofs of total correctness of Sunrise programs while retaining complete security. The system also contains tools to ease using mutually recursive datatypes and conditional rewriting.

3.2.3 KIV

The Karlsruhe Interactive Verifier (KIV) is a comprehensive environment for the development of correct software, including formal specification and verification. The development of the KIV system started in 1986 at the University of Karlsruhe and is now pursued together with several other german universities (mainly Ulm [H5]).

KIV relies on a first−order algebraic specification language to describe hierarchically structured software systems. Specification components are implemented by stepwise refinement using (functional) program modules. KIV offers a powerful interactive verification component for module correctness based on proof tactics.

KIV supports incremental verification and visualizes the impacts of modifications on the correctness of other components. A correctness management keeps track of the development graph (visualization of the development process), proof obligations and proofs. Specifications, proofs, and verified components can easily be reused.

The system has been tested in a large number of case studies and is at the threshhold to industrial applications.

4. The Verification System

In this section we describe the top-level structure of the verification system and discuss the representation of programs and specifications. We describe the behaviour of the verification condition generator and its integration into *Theorema*.

4.1 The Structure

Building the verification system within *Theorema* has several advantages:

- A separate lexical analyzer or syntax checker is not necessary; it is more or less built-in in *Mathematica.*
- An intermediate language is not necessary. By applying the convenient "pattern matching" in *Mathematica* verification rules can be implemented which operate *directly on the code.*
- A seperate simplifier for the verification conditions is not necessary – simplification can be done by existing *Theorema* simplifiers.

Hence, the structure of our verification system is much simpler than of those discussed in the survey.

Programs are written in the *Theorema Procedural Language* which is described in the section 5.2, "The Source Language". The environments for writing programs and specifications are presented in the following section.

4.2 Programs and Specifications

One major goal of the whole system is that it be a training environment for young software engineers. We believe that for didactical reasons ("think before you hack") it is important to make "Specification" a separate entity – to really define in advance *what* has to be done before working on *how* it should be done (see "Theoretical Background").

A program and its specification are not necessarily in a "1:1" relation. A single program could meet more than one specification and, conversly, there could be a lot of different implementations for one specification (for instance different algorithms for a sorting problem). Thus between program and specification there is an "m:n" relation and they should strictly be decoupled.

So what is clearly needed are two separate data structures:

- "Specification" which contains a pre- and a postcondition and an interface definition
- "Program" which encapsulates program code and also an interface definition

An interface definition consists of the program name (interface name) and the parameters, attributed as input-, transient- or output parameters (We do not yet support data types, as

discussed in the "Concepts" of *Theorema Procedural Language*). A function procedure has an additional return value in the interface definition.

As mentioned in the introduction one major goal of this thesis is to meet the conventions for the general *Theorema* input style. *Theorema* provides environments for giving definitions, theorems, lemmas, etc. which are not part of its Higher Order Logic language, but build a "meta-language" called *Theorema FormalText Language*.

"Specification" and "Program" are environments similar to the mathematical environments mentioned above. This becomes clear when considering the follwing three different situations (according to Bruno Buchberger in a seminar talk):

- A specification is written but there is no implementation. Mathematically this would correspond to an *axiom*.
- A program is written but not annotated. This can mathematically be viewed as a *definition*.
- A program is written and annotated to meet a certain specification. Mathematically this can be viewed as a *theorem* (actually a correctness theorem).

Thus, writing a program could also be viewed as "writing a mathematical definition in a procedural style". Therefore it is natural that a *Theorema* program does not only look similar to a definition but also follows the existing scheme for the internal representation in *Theorema*. The implementation is presented in the section 5.1, "The Environments Program and Specification".

4.3 Verification Condition Generation

The verification condition generator takes an annotated program with pre- and postcondition (mostly taken from the specification) and produces as output a *Theorema*-`Lemma` with a collection of formulae, i.e. the verification conditions.

The verification condition generator is a translator based on a list of inference rules. It is recursive on the structure of the code and works back-to-front statement by statement. Internally it repeatedly modifies the postcondition using a predicate transformer such that at the end the result is the verification condition

$$\textit{precondition} \Rightarrow \textit{transformed postcondition}$$

together with a list of verification conditions from the loop invariants.

Currently the VCG is (more or less) a stand-alone program within *Theorema*; it has to be called separatly before calling a prover for the resulting lemma. However, it is planned that it should become part of a "Program Prover" such that one can call the prover for the program (i.e. the correctness theorem) directly where the VCG is just one (automatied) step.

For details on the implementation, see the section 5.3, "The Verification Condition Generator".

4.4 Proving with *Theorema*

Proving the verification conditions with *Theorema* provers was not actually part of this thesis. However, we give a general introduction on how to prove verification conditions within *Theorema*.

Whether a verification condition can be proved by *Theorema* depends mostly on the functions (and operations) occuring in the program. The knowledge about the used functions can be either built-in in the prover (in so-called "special provers") or supplied by the user in a knowledge base for the `Prove` function. The knowledge base usually contains definitions or propositions about functions in *Theorema.* If calls of function procedures occur in a program one has to "manually" transform the specification into a *Theorema* proposition, as it is explained in the section 5.4.10, "Calls of Function Procedures". However, in future versions of the system it will be allowed to put specifications into knowledge bases, and the appropriate proposition will then be generated automatically from the specification.

The current version of *Theorema* already includes a few very sophisticated automatic provers, like for instance an induction prover for natural numbers, an induction prover for tuples, a predicate logic prover and a prover based on simplification (and several combinations of these).

However, the requirements for automatically proving typical verification conditions are high. Verification conditions are usually universally quantified implications containing inequalities. If arrays are used in the program, verification conditions also contain tuples and operations on tuples.

Currently, there is no prover in *Theorema* with built-in knowledge about inequalities. However, if one supplies additional knowledge on the properties of inequalities, then it is possible that simple verification conditions can be proved automatically.

Currently the user needs some know-how about how to handle these provers and which provers to employ. In future, however, a specific "Program Verification Prover" should combine all the capabilities for proving typical verification conditions automatically.

4.5 Execution of Programs

Similar to what has been discussed in "Programs and Specifications", the execution of a program is logically equivalent to the execution of a function definition in *Mathematica*, i.e. calling a mathematical function with certain parameters.

In *Theorema*, one function is responsible for all execution: `Compute`. To execute a program, one has to give the interface name and the actual parameters (similar to a procedure call). In addition, the evaluator used must be `ProgramEvaluator`. In a call to `Compute` one can specify the type of evaluator with the option `by→`.

When programs are executed, there are two different situations:

- Executing a function procedure returns a value.

  For instance, calling `Compute` of a function procedure "Factorial" with an input parameter 5 returns the value 120. In a *Theorema* session this would look as follows:

  ```
  Compute[Fact[5], by → ProgramEvaluator]

  120
  ```

- Executing a "normal" procedure modifies (sets) the output variables to certain values and usually does not return a value. `Compute`, however, instead of setting the output variables of a procedure in a global scope returns a list (a tuple) of rules of the form

  output-variable → *value*.

  For instance, calling `Compute` of a procedure "Factorial" with an input parameter 5 and an output parameter *f* returns ⟨*f*→5⟩. Again, in a *Theorema* session this would look as follows:

  ```
  Compute[Fact[5, f], by → ProgramEvaluator]

  ⟨f → 120⟩
  ```

An additional feature of the execution is that (for specified programs) `Compute` can (by supplying the option `VerifyConditions→True`) verify the validity of pre- and postcondition. Further checks can also be performed at the level of single statements (e.g. the invariants have to hold), which is described in detail in the section 5.2.3, "Semantics". The checks are at least possible where the conditions do not contain quantifiers with infinite ranges. What can be evaluated depends, in fact, on the implementation of the sematics of quantifiers in *Theorema*.

5. Implementation

5.1 The Environments Program and Specification

As discussed in "Programs and Specifications" the entities "Specification" and "Program" are introduced to *Theorema* as special *FormalText* elements. They have the following basic structure:

- **`Specification`**[*label*, *interface*, *precondition*, *postcondition*]
- **`Program`**[*label*, *interface*, *code*]

The *label* is an arbitrary string to identify the specification and the program, respectively.

The *interface* consists of an interface name and parameters attributed with arrows; for example the interface definition `SomeProc[↓u,↕v,↑w]` has the interface name `SomeProc`, an input parameter `u`, a transient parameter `v`, and `w` as an output parameter. The parameter concept with arrows is very easy and clear, input parameters can be viewed as "call by value", the transient and output parameters as "call by reference".

The *precondition* and the *postcondition* are marked by `Pre→` and `Post→` followed by arbitrary *Theorema* formulae.

The *code* is a sequence of statements (the actual program) in the *Theorema Procedural Language*.

As it was mentioned in the previous section, we want to verify if pre- and postcondition are valid *during execution*. Therefore we have to "overload" the "Program" structure by the following possiblities:

- **`Program`**[*label*, *interface*, *code, specification*]
- **`Program`**[*label*, *interface*, *precondition*, *code, postcondition*]

The *specification* is an option marked by `Specification→` and is a means to "link" the program with an existing specification.

In the second case a program can be annotated with a *precondition* and a *postcondition* without the need for writing a separate "Specification".

All these additional parameters are optional; however, one cannot "overspecify" a program by giving a *specification* and a *pre-* and *postcondition.*

Example:

```
Specification ["Maximum", Max[↓a, ↓b, ↑c],
 Pre → True,
 Post → ((c = a) ∧ (a ≥ b)) ∨ (c = b)
]

Program ["Maximum", Max[↓a, ↓b, ↑c],
 IF [a ≥ b,
  c = a,
  c = b
 ],
 Specification → Specification ["Maximum"]
]
```

This example shows the "natural" syntax of the two environments. In the following, we will discuss the *functionality* "behind" the environments. In principle there are different situations where a program (and a specification, resp.) is used: for verification, for execution and as a "sub-program" called from another program.

5.1.1 Using a Program for Verification

For *verification* the static data is needed. We want to call the verification condition generator e.g. in the following way:

```
VCG[ Program["Maximum"], Specification["Maximum"] ]
```

Therefore it is required that `Program["Maximum"]` and `Specification["Maximum"]` refers to internal data structures, where all the information (label, interface, pre- and postcondition, code) is encapsulated.

In *Theorema,* such data structures are *Mathematica* expressions which do *not* evaluate and follow certain conventions. For instance, the data structure for a Lemma in *Theorema* is (simplified)

•`lma`[*label*, *formula*].

The bullet "•" in front of `lma` denotes an internal *Theorema* data structure. When the user writes a new `Lemma`[*label*, *formula*], a new *Mathematica* inference rule is established such that `Lemma`[*label*] always evaluates to the internal data structure shown above.

Since a major requirement for the verification system is that it be consistent with *Theorema* style, the environments "Program" and "Specification" produce similar inference rules:

`Specification`[*label*] = •`spec`[*label*, *interface*, *precondition*, *postcondition*]

`Program`[*label*] = •`prog`[*label*, *interface*, *precondition*, *code, postcondition*]

The verification condition generator now operates on the internal data structures •`prog` and •`spec`.

5.1.2 Using a Program for Execution

As explained in the section 4.5, "Execution of Programs", the function `Compute` is responsible for *executing* a program. It has to be called like in the following example:

```
Compute[Max[5,7,out], by→ProgramEvaluator]
```

Hence, another inference rule is needed where a program call, consisting of the interface name (here `Max`) and *arbitrary actual parameters* (here `5,7,` and `out`), refers to a data structure which encapsulates the "functionality" of the program.

Again, *Theorema* already provides a general concept for such a situation. For instance a universally quantified formula

$$\forall_{x \in N} \; x \geq 0$$

is represented internally as `ForAll[x∈N, x≥0]`. Since in *Theorema* there are evaluation rules for the function `ForAll` it would be evaluated instantly. In order to avoid this it is renamed to `™ForAll` beforehand such that its evaluation is under control of *Theorema*. Similar to the bullet, the "™" is a special prefix, denoting a "functional data structure" in *Theorema* which "holds" all the data of the function but is never directly evaluated.

Similarly, we use a "™-data-structure" called `™Program` for our purpose. The inference rule constructed for the example above would be:

```
Max[a_,b_,c_] := ™Program[ Max[↓a,↓b,↑c], True,
                                ((c=a)∧ (a≥ b)) ∨(c=b),
                                IF [a≥b,c=a,c=b]
                              ]
```

Note that this is actually a *Mathematica* function definition. The interface name (`Max`) can now be called with arbitrary actual parameters which then substitute the formal parameters in `™Program`, in the interface definition and in the program code.

Interface names of programs are unique in the system. If a program is written with the same interface name as an already existing program, the inference rule for the old program will be overwritten. Hence, if one wants to have two different versions of a program, they must have different interface names, too.

5.1.3 Using a Program in a Procedure Call

For *calling* a program from another program we want to write, for instance, the following:

```
Max[a+b, 5, c]
```

The requirements for this situation are exactly the same as for the preceding one (program execution), so it does not need to be handled differently when a new program is written.

5.2 The Source Language

Theorema, as a mathematical software system, presents a strictly functional interface to the user. Embedding a verification system in *Theorema*, however, has required us to extend it with a procedural programming language.

5.2.1 Concepts

The *Theorema Procedural Language* adds a procedural aspect to *Theorema*. In fact, this language combines *procedural* and *functional* style.

All basic concepts of a procedural language are supported:

- Assignment
- Sequence
- Condition
- Loop
- Block
- Procedure and Function Call

Single statements like `IF, WHILE, FOR` have functional style. For instance the statement IF (similar to the *Mathematica* function *If*) takes the condition and the statements for the two branches "then" and "else" as *arguments*, e.g. `IF[i<j, j:=i, i:=j]`.

In addition to the concepts of usual programming language the *Theorema Procedural Language* includes a few other features:

- The possibility of annotating a program is part of the language, e.g. as an optional parameter to `WHILE`.
- Mathematical expressions can be written in the *Theorema* Language, e.g. quantifiers can be used in a condition for a `WHILE` loop or on the right side of an assignment. This powerful feature is shown in "HasElement" in the section "Examples" (7.10.).
- Interfaces are defined in a very intuitive and concise style using arrows.

In order to guarantee that a program is processed correctly it must occur within a "Program" environment (described in the previous section).

What we have left totally aside in the scope of the thesis is the issue of data types. In the current version, the program execution relies totally on the user to give correct values as parameters. In a later version of the language, data types will most probably be added by using predicates, like for instance `IsInteger[i]` in an interface definition.

5.2.2 Syntax

The following is a grammar (a description of the syntax) of the *Theorema Procedural Language* in Backus-Naur-Form (as for example in [Aho/Ullman77]). Terminal symbols are printed in bold face.

Code →

statement-sequence

statement-sequence→

statement

| statement-sequence **;** statement

statement →

elementary-statement

| if-statement

| while-statement

| for-statement

elementary-statement →

variable **:=** term

| variable **:=** formula

| procedure-call

| **Module [{** identifier-list **} ,** statement **]**

identifier-list →

IDENTIFIER

| identifier-list , IDENTIFIER

if-statement →

IF [formula **,** statement-sequence **]**

| IF [formula **,** statement-sequence **,** statement-sequence **]**

while-statement →

WHILE [formula **,** statement-sequence **]**

| **WHILE [** formula **,** statement-sequence **,** invariant **]**

| **WHILE [** formula **,** statement-sequence **,** invariant **,** terminationTerm **]**

for-statement →

FOR [variable**,** term **,** term **,** statement-sequence **]**

| **FOR [** variable**,** term **,** term **,** term **,** statement-sequence **]**

| FOR [variable**,** term **,** term **,** statement-sequence **,** invariant **]**

| **FOR [** variable**,** term **,** term **,** term **,** statement-sequence **,** invariant **]**

invariant →

Invariant → formula

terminationTerm →

TerminationTerm → term

procedure-call →

IDENTIFIER **[** argument-sequence **]**

variable →

IDENTIFIER

| $\text{IDENTIFIER}_{\text{term}}$

argument-sequence →

term

| formula

| argument-sequence **,** term

| argument-sequence **,** formula

term →

variable

| FUNCTION−IDENTIFIER **[** argument-sequence **]**

formula →

True

| **False**

| variable

| PREDICATE−IDENTIFIER **[** argument-sequence **]**

As one can see, square brackets are used for the arguments of **IF**, **WHILE**, **FOR** or **Module**, and for function or procedure calls. Note that the arrows **→** in the rules for **Invariant** and **TerminationTerm** are terminal symbols. The variable can be either a normal program variable denoted by an IDENTIFIER or an array variable, denoted by an IDENTIFIER with a term in the subscript.

The follwing information makes the grammar complete:

- Comments are, like in *Mathematica,* embraced by **(*** and ***)**. Comments can be nested.
- IDENTIFIERs are all identifiers possible in *Mathematica.*
- A FUNCTION−IDENTIFIER denotes a function returning a number. This can either be a function procedure defined in a Program environment or a *Mathematica* function without side effect.
- A PREDICATE−IDENTIFIER is similar to a FUNCTION−IDENTIFIER and denotes a predicate returning a **True** or **False**.

5.2.3 Semantics

In this section we just give a short verbal description of the semantics of each statement. The verification rules presented in a later section can also be read as a more formal definition of the semantics of *Theorema Procedural Language*.

Assignments

$V := E$

The program state is changed by assigning the value of the term E to the variable V.

Array Assignments

$V_{E_1} := E_2$

The program state is changed by assigning the value of the term E_2 to the E_1 -th element of the array variable V.

Sequences of Statements

$S_1 \; ; \; \ldots \; ; S_n$

The statements $S_1 \; ; \; \ldots \; ; S_n$ are executed in that order.

Blocks

Module [{ $V_1 \, , \, \ldots \, , V_n$ } , S]

S is executed with $V_1 \, , \, \ldots \, , V_n$ as local variables. The initial values of $V_1 \, , \, \ldots \, , V_n$ inside the block are unspecified. In the current version of the language variables with the same names as the local variables should never occur outside of the block.

One-armed Conditionals

IF [C , S]

If the condition C can be evaluated to true, then S is executed. Otherwise nothing is done.

Two-armed Conditionals

IF [C , S_1 , S_2]

If the condition C can be evaluated to true, then S_1 is executed, otherwise S_2 is executed.

WHILE–Loops

WHILE [C , S , I , T]

If the condition *C* can be evaluated to true, then *S* is executed and the `WHILE` statement is then repeated. If *C* is false or cannot be evaluated, then nothing is done. Thus, *S* is repeatetly executed until the value of *C* becomes false. If this is never the case, the execution of the statement never terminates.

If the invariant or the termination term are violated (if they do not "hold"), the execution is stopped and an appropriate error message is printed. Again, as explained in the section 4.5, "Execution of Programs", this feature depends on value of the option `VerifyConditions→` supplied to the `Compute` function.

FOR−Loops

`FOR` [*V* , E_1 , E_2 , *S* , *I*]

If $E_1 \leq E_2$ can be evaluated to true, then *S* is executed $(E_2 - E_1)+1$ times with the variable *V* taking on the sequence of values E_1, E_1+1, ... , E_2 in succession. The expressions E_1 and E_2 are evaluated only once at the beginning of the execution of this statement; if that is not possible, then the statement has no effect.

If the invariant is violated (if it does not "hold"), the execution is stopped and an appropriate error message is printed (again, depending on the value of the option `VerifyConditions→`).

Procedure Calls

p[V_1, V_2, …, V_n]

Procedures are abstractions: the single statements of a procedure are normally not of interest to the calling program; only the interface and the effect specified in the pre- and postcondition are relevant.

Procedures in our system don’t have side effects (since it is not intended that global variables are used).

When a procedure call is executed, it is checked whether the procedure is actually implemented and the formal parameters in the program are replaced by the actual parameters. When the procedure is then executed, it is checked whether pre- and postcondition are valid (again, depending on the value of the option `VerifyConditions→`).

5.2.4 Conventions

In principle, we trust the user and *Mathematica* that the input is syntactically sound. However, there are a few things which one has to keep in mind when writing programs in this language:

- The most important convention is that input variables may not be changed within a program. If this is ignored, it is very likely that the verification condition generator produces erroneous results.
- As described in the "Motivation", the verification system will mainly be used for didactical purposes for Software Engineering students. As global variables should be avoided in professional software engineering we simply ignored the issue of global variables in the scope of the thesis.
- The arguments for program calls must be reasonable (e.g. no expressions as output variables).

5.3 The Verification Condition Generator

In this section we want to explain the principles on which the verification condition generator (VCG) relies. Firstly we present an approach based on Hoare Logic and explain why it is not sufficient for our requirements. The second approach relies on a "weakest precondition predicate transformer" and also has some serious limitations. Finally, as a consequence of the two approaches, we describe our solution based on a function which we call "Extended Predicate Transformer".

5.3.1 VCG with Rules for Hoare Triples

The verification rules of Hoare Logic for "standard" statements are well-documented, e.g. in [Gordon98]. So a VCG based on those inference rules can quite easily be implemented in *Mathematica* by applying the concept of pattern matching.

In this approach, the VCG is a function which takes an annotated program with pre- and postcondition and returns a list of verification conditions. It processes a sequence of statements back-to-front (in order to avoid the problematic issue of forward substitution) by recursively applying the appropriate rule for each statement.

Although this approach works fine, there is one problem: those rules "work" with Hoare Triples {Q} S {R} and require a pre- and a postcondition for *each* program statement (the rule for assignments is an exception). This is illustrated by the following example:

```
Program["Simple", Simple[↓a, ↑b],
  Pre → True,
   b := 10;
   IF[ a > b,
    b := 2 * a
   ],
   Post → ( b = 10 ) ∨ ( b = 2 * a )
 ]
```

As mentioned above, the VCG works back-to-front and would thus in the first step call a rule for the `IF` statement. The postcondition is clear – but what should it take as a precondition?

One common solution is that the user has to give annotations, e.g. in front of every `IF` statement. However, with this method the user has to provide more than the minmal set of necessay annotations, the loop invariants (as discussed in "Theoretical Background"). Hence, this approach is not satisfying.

So the obvious idea is to *generate* the intermediate assertions required by these rules. For the simple example above those assertions are intuitively clear: We just need one assertion in front of the `IF` statement, namely `(b=10)`.

But how can this be done *automically*? What if there were a complicated WHILE-loop instead of the simple assignment `b:=10`?

5.3.2 VCG with Rules for Weakest Preconditions

In the next approach, we attempted to use Dijkstras weakest precondition predicate transformer (*wp*). It is a recursive function which works back-to-front like the VCG in the first approach. However, it has one advantage: It does not need intermediate assertions between a sequence of statements. The function *wp* only takes a statement (or a sequence of statements respectively) and the postcondition as it is shown in the algebraic notation [Klaeren83] :

$$wp: \langle\ stat\ ,\ post\ \rangle \mapsto pre$$

It returns the weakest precondition which is the postcondition *transformed by certain inference rules* for the different statements. This and the connection to Hoare Logic are explained in the section on "Weakest Preconditions" (2.1.8.).

However, one can see quite soon (as described in [Futschek89]) that iterative statements are a problem with *wp*. It can be extremely complicated to construct the weakest precondition of a WHILE statement. So implementing a "pure" *wp* function which automatically computes the weakest precondition of all possible statements is not feasable.

A possiblity to circumvent this problem is to use invariants like in Hoare Logic. As discussed in "Theoretical Background", the loop invariants have to be valid before and after each iteration, and particularly *before the first* iteration. Hence, an invariant can also be viewed as a precondition to the loop (although it is usually not the weakest).

This solution has two consequences. First, the function does not necessarily return the weakest precondition for all statements and should consequently not be called *wp*. Second, it has to have the "side effect" of producing verification conditions for the invariants (like in Hoare Logic); and therefore it does not meet the above algebraic specification anymore.

In a natural way that leads to a function which we call *EPT* (Extended Predicate Transformer).

5.3.3 VCG using an "Extended Predicate Transformer"

The Extended Predicate Transformer *EPT* is actually more than a predicate transformer. Like the function *wp,* it takes a statement and a postcondition, but it returns two "data structures": the precondition (the transformed postcondition) and a list of verification conditions.

$$EPT: \langle\ stat, post\ \rangle \mapsto \langle\ pre, vcList\ \rangle$$

Example:

For the following program

```
Program["Factorial Version 1",
     Fact[↓n, ↑f],
     Pre → n ≥ 0,
       f := 1;
       i := n;
       WHILE[ i > 1,
         f := f * i;
           i := i - 1,
         Invariant → ( ( i ≥ 0 ) ∧ ( f = n! / i! ) )
        ],
     Post → ( f = n! )
]
```

EPT would return the new precondition

$$n \geq 0 \bigwedge (1 = \frac{n!}{n!})$$

and a list of verification conditions from the `WHILE`-invariant containing the two formulae:

$$i \geq 0 \bigwedge (f = \frac{n!}{i!}) \bigwedge (i > 1) \Rightarrow (i - 1 \geq 0) \bigwedge (f * i = \frac{n!}{(i-1)!})$$

$$i \geq 0 \bigwedge (f = \frac{n!}{i!}) \bigwedge (\neg (i > 1)) \Rightarrow (f = n!)$$

As discussed in "Theoretical Background", it is easy to close the gap between Hoare Logic and *wp*, and similarly we embed this extended predicate transformer into the VCG of our first approach such that we can just call `VCG[Program[x]]` and get verification conditions.

In the example above, the constructed verification condition is obtained by building an implication with the given precondition and the "new" precondition (the transformed postcondition).

$$n \geq 0 \Rightarrow n \geq 0 \bigwedge (1 = \frac{n!}{n!})$$

It is then joined to the list of verification conditions and transformed in a *Theorema*-`Lemma` and returned as output.

`Lemma ( Factorial Version 1 ) :`

`for any :` n, f

$$(i \geq 0 \bigwedge (f = \frac{n!}{i!})) \bigwedge i > 1 \Rightarrow i - 1 \geq 0 \bigwedge (f * i = \frac{n!}{(i-1)!}) \quad \text{(WHILE . Inv)}$$

$$(i \geq 0 \bigwedge (f = \frac{n!}{i!})) \bigwedge (i \not> 1) \Rightarrow (f = n!) \quad \text{(WHILE . Final)}$$

$$n \geq 0 \Rightarrow n \geq 0 \bigwedge (1 = \frac{n!}{n!}) \quad \text{(Init)}$$

With the pattern matching concept of *Mathematica* [Wolfram96], it is easy to devide the data of the fully annotated program into interface definition, precondition, postcondition and code. The EPT function takes the code and the postcondition and (recursively) applies a transformation rule for every single statement back-to-front such that the output is first a list of the accumulated verification conditions and second the transformed postcondition (the new precondition before the first statement). Then the "last" verification condition

precondition ⇒ *transformed postcondition*

is joined to the list.

The list of verification conditions is transformed into a *Theorema* formula list. Parameter variables (from the interface definition) and local variables (from `Modules`) occur as free

variables in these formulae. In the last step of the processing the formula list is turned into a *Theorema*-`Lemma` where these variables are universally quantified.

With EPT it is possible to minimize the number of necessary annotations, such that the user has to only give invariants for loops, and nothing more. In the following section we present the translation rules which the EPT function implements.

5.4 Verification Rules

In the previous section we have given an informal description of the EPT function. Now we want to go into more detail and present all the single verification rules which EPT implements.

As mentioned in the preface, a major goal of this thesis is to keep it useable for the mathematically oriented readers as well as for software engineers. Therefore we give two equivalent definitions for every rule: one in a functional *Mathematica*-like notation, the other in natural language.

For example the rule for an empty statement "`;`" is as follows:

- In functional notation:

  `EPT[ ; , R ]` $\rightarrow$ `{ R , {} }`

- In natural language:

  An empty statement

  "`;`"

  with a postcondition *R*

  has *R'* as precondition

  where *R*' is equal to *R*

  and does not produce any verification conditions.

The description in natural language makes the logical structure of the rules clear by extensive use of indentation. On the outer (leftmost) level it always shows the input and output of the function EPT, namely the statement and a postcondition as input, and a (new) precondition and verification conditions as output.

The application of every single rule is illustrated by simple examples (which are given in Hoare notation for conciseness).

In the following, variables are denoted by *V*, expressions by *E*, statements by *S*, conditions (in `IF` and `WHILE`) by *C*, postconditions by *R* and *Q* and invariants by *I*.

5.4.1 Assignments

`EPT[`V`:=`E`,`R`]` $\rightarrow$ `{` R_E^V `, {} }`

R_E^V is a notion for the substitution of all occurences of V in R by E. This notion, used by [Futschek89], is equivalent to other common notions like $R[V|E]$ or $R_V[E]$.

An assignment

$V := E$

with a postcondition R

produces the (weakest) precondition R'

where R' results from R by substituting every occurence of the variable V by the expression E

and no verification conditions.

Example:

The program (in Hoare-Notation)

```
{x=0} x:=x+1 {x=1}
```

produces the following verification condition:

```
x=0 ⇒ (x+1=1)
```

(which is clearly true).

Explanation of the Example Relating to Fig. 3:

The VCG is called basically with the precondition `x=0`, the program code `x:=x+1`, and the postcondition `x=1` (in this explanation we don't go into details of the implementation, e.g. concerning the interface).

Calling `EPT[x:=x+1, x=1]` yields `{(x+1=1), {}}`. The final verification condition is obtained by firstly forming the implication of the precondition `x=0` with the transformed postcondition `(x+1=1)` and secondly joining it with the (empty) list of verification conditions `{}`.

In the following, we will not present the examples in such detail and assume that the reader can build the intermediate steps according to Fig. 3.

5.4.2 Array Assignments

In [Gordon98] it is shown in a good example why an array assignment $A_{E_1} := E_2$ has to be treated as an ordinary assignment

$$A := A_{E_1 \leftarrow\!\!\!\!\leftarrow E_2}$$

where the term $A := A_{E_1 \leftarrow\!\!\!\!\leftarrow E_2}$ denotes an array identical to A, except that the E_1 - th element is changed to have the value E_2.

$$EPT[A_{E_1} := E_2, R] \rightarrow \{ A := A_{E_1 \leftarrow\!\!\!\!\leftarrow E_2}, \{\} \}$$

An array assignment

$A_{E_1} := E_2$

with a postcondition R

produces the (weakest) precondition R'

which results from R by substituting all occurences of the array variable A by A'

and A' results from A *by* replacing the E_1 - th element with E_2

and no verification condition.

Example:

The program (in Hoare-Notation)

$\{v=E\}$

$a_1 := v + 1$

$\{a_1 = E+1\}$

produces the following verification condition:

$(v = E) \Rightarrow ((a_{1 \leftarrow (v+1)})_1 = E + 1)$

The term $(a_{1 \leftarrow (v+1)})_1$ evaluates to $v+1$ and, hence, the implication is clearly true.

5.4.3 Sequences

$$\texttt{EPT[}S_1\texttt{;}\ldots\texttt{;}S_{n-1}\texttt{;}S_n\texttt{,}R\texttt{]} \rightarrow \{\ \texttt{EPT[}S_1\texttt{;}\ldots\texttt{;}S_{n-1}\texttt{,(EPT[}S_n\texttt{,}R\texttt{])}_1\texttt{],}\ \{\texttt{(EPT[}S_1\texttt{;}\ldots\texttt{;}S_{n-1}\texttt{,}R\texttt{])}_2 \cup \texttt{(EPT[}S_n\texttt{,}R\texttt{])}_2\}\ \}$$

As discussed above, EPT returns a list with two elements: a transformed postcondition and a list of "accumulated" verification conditions. $\texttt{(EPT[}S_n\texttt{,}R\texttt{])}_1$ denotes the transformed postcondition, $\texttt{(EPT[}S_n\texttt{,}R\texttt{])}_2$ denotes the list of verification conditions.

A sequence of statements

$S_1 \ ; \ \ldots \ ; S_{n-1} \ ; S_n$

with a postcondition R

produces the precondition R'

which is obtained by computing (recursively) the precondition of

$S_1 \ , \ \ldots \ , S_{n-1}$ with respect to a postcondition Q

where Q is obtained by computing the precondition of the

single statement S_n with respect to the postcondition R.

and also the verification conditions from the invariants of $S_1 \ ; \ \ldots \ ; S_{n-1} \ ; S_n$.

Example:

The program (in Hoare-Notation)

```
{x=X ∧ y=Y}
   tmp:=x; x:=y; y:=tmp
{x=Y ∧ y=X}
```

produces the following verification condition:

```
(x=X) ∧ (y=Y) ⇒ (y=Y) ∧ (x=X)
```

(which is clearly true).

The intermediate steps are as follows:

```
(1)  EPT[ tmp:=x; x:=y; y:=tmp , x=Y ∧ y=X ]
(2)  EPT[ tmp:=x; x:=y , x=Y ∧ tmp=X ]
(3)  EPT[ tmp:=x , y=Y ∧ tmp=X ]
(4)  {(y=Y) ∧ (x=X), {} }
```

The result of calling `EPT` (4) consists, equal to the examples above, of a transformed postcondition and an (empty) list of verification conditions. The transformed postcondition can be found on the right side of the implication of the final verification condition `(x=X) ∧ (y=Y) ⇒ (y=Y) ∧ (x=X)`.

5.4.4 Blocks

EPT [`Module` [{ V_1 , … , V_n } , S] , R] → EPT [S , R]

A block

`Module` [{ V_1 , … , V_n } , S]

with a postcondition R

produces the precondition R'

which is obtained by computing the precondition of

S with respect to the postcondition R

and no verification condition

(with the syntactic condition that none of V_1 , … , V_n do occur in R.

That would lead to undefined results).

Example:

The program (in Hoare-Notation)

```
{x=X ∧ y=Y}
   Module[{tmp},tmp:=x;x:=y;y:=tmp]
{x=Y ∧ y=X}
```

produces the same verification condition as

```
{x=X ∧ y=Y}
   tmp:=x; x:=y; y:=tmp
{x=Y ∧ y=X}
```

since `tmp` does not occur in the postcondition `{x=Y ∧ y=X}`.

The variable `tmp` is only local and does not have a value after execution. Why `Modules` should be used is explained in the section 6.7.2, "Where to Use Blocks".

5.4.5 Conditionals

Conditional statements are either one-armed (only the "then" branch) or two-armed (with the "else" branch). After presenting the rules for both we give an example for the verification of a two-armed conditional statement.

`EPT[IF[`C`,`S`],`R`]` → { (C⇒(`EPT[`S`,`R`]`)$_1$) ∧ (¬C⇒R), { C ⇒ (`EPT[`S`,`R`]`)$_2$ } }

If S contains loops, then (`EPT[`S`,`R`]`)$_2$ returns the accumulated verification conditions of the invariants. Of course this is only interesting in case C is true and there is a loop in S. Therefore `EPT[IF[`C`,`S`],`R`]` returns C ⇒ (`EPT[`S`,`R`]`)$_2$ as a verification condition.

A one-armed conditional

IF [C , S]

with a postcondition R

produces the precondition R' which is $(C \Rightarrow Q) \wedge (\neg C \Rightarrow R)$

where Q is obtained by computing the precondition of

S with respect to the postcondition R

and also the verification conditions from the invariants of S.

$$\texttt{EPT[IF[}C\texttt{,}S_1\texttt{,}S_2\texttt{],}R\texttt{]} \rightarrow \{ (C \Rightarrow (\texttt{EPT[}S_1\texttt{,}R\texttt{]})_1) \vee (\neg C \Rightarrow (\texttt{EPT[}S_2\texttt{,}R\texttt{]})_1), \{ C \Rightarrow (\texttt{EPT[}S_1\texttt{,}R\texttt{]})_2, \neg C \Rightarrow (\texttt{EPT[}S_2\texttt{,}R\texttt{]})_2 \} \}$$

A two-armed conditional

IF [C , S_1 , S_2]

with a postcondition R

produces the precondition R' which is $(C \Rightarrow Q_1) \wedge (\neg C \Rightarrow Q_2)$

where Q_1 is obtained by computing the precondition of

S_1 with respect to the postcondition R

and Q_2 is obtained by computing the precondition of

S_2 with respect to the postcondition R

and also the verification conditions from the invariants of S_1 and S_2.

Example:

The program (in Hoare-Notation)

```
{True}
   IF[x≥y, max:=x, max:=y]
{max=Maximum[x,y]}
```

(where `Maximum` is a function defined within *Theorema*)

produces the following verification condition:

```
True ⇒ ( (x≥y) ⇒ (x=Maximum[x,y]) ) ∧
          ( (¬(x≥y)) ⇒ (y=Maximum[x,y]) )
```

where the right part of the implication is the new precondition (transformed postcondition) obtained by calling `EPT` with the above rule.

5.4.6 WHILE–Loops without Verification of Termination

EPT[WHILE[*C*,*S*,*I*],*R*] → { *I*, { (*I*∧¬*C*) ⇒ *R*,(*I*∧*C*) ⇒ (EPT[*S*,*I*])$_1$ } }

A loop

WHILE [*C* , *S* , *I*]

with a postcondition *R*

has the invariant *I* as a precondition

and produces the following verification conditions:

$(I \wedge \neg C) \Rightarrow R$

$(I \wedge C) \Rightarrow I'$

where *I'* is obtained by computing the precondition of *S* with respect to the postcondition *I*.

Example:

The program (in Hoare-Notation)

```
{r=x ∧ q=0}
      WHILE[ y≤r,
            r:=r−y; q:=q+1,
            Invariant→(x=r+y*q) ]
{x=r+y*q ∧ r<y}
```

produces the following verification conditions:

```
(x=r+y*q) ∧ ¬(y≤r) ⇒ (x=r+y*q) ∧ r<y
(x=r+y*q) ∧ y≤r ⇒ (x=(r−y)+y*(q+1))
(r=x) ∧ (q=0) ⇒ (x=r+y*q)
```

where the first two are generated by the WHILE rule as shown above and the third verification condition is the implication of the precondition and the invariant.

5.4.7 WHILE–Loops with Verification of Termination

EPT[WHILE[*C*,*S*,*I*,*T*],*R*] →

{ *I*, { (*I* ∧ ¬*C*) ⇒ *R*,

(*I* ∧ *C* ∧ (*T*=n)) ⇒ (EPT[*S*,*I* ∧ (*T* < n)])$_1$,

(*I* ∧ *C*) ⇒ (*T*≥*0*) } }

In the second verification condition of the list, the value of *T* (which always has to be a natural number) is "stored" in the constant n. After the execution of the loop (verification of the loop body by recursively calling EPT) the value of *T* is compared with n and must have decreased in order to prove termination.

A loop

WHILE [C , S , I , T]

with a postcondition R

has the invariant I as a precondition

and produces the following verification conditions:

$(I \wedge \neg C) \Rightarrow R$

$(I \wedge C \wedge (T=n)) \Rightarrow I'$

where I' is obtained by computing the precondition of

S with respect to the postcondition $I \wedge (T \prec n)$.

$(I \wedge C) \Rightarrow (T \geq 0)$

Example:

The program from the last example with a termination term added

```
{r=x ∧ q=0}
      WHILE[ y≤r,
        r:=r−y; q:=q+1,
        Invariant→(x=r+y*q),
        TerminationTerm→r−y]
{x=r+y*q ∧ r<y}
```

produces the following verification conditions:

```
(x=r+y*q) ∧ ¬(y≤r) ⇒ (x=r+y*q) ∧ r<y
(x=r+y*q) ∧ y≤r ∧ (r=T1) ⇒ (x=(r−y)+y*(q+1)) ∧ (r−y)<T1
(x=r+y*q) ∧ y≤r ⇒ r≥0
(r=x) ∧ (q=0) ⇒ (x=r+y*q)
```

where the first three are generated by the WHILE rule as shown above and the third verification condition is the implication of the precondition and the invariant.

By supplying the termination term and proving the generated verification conditions the total correctness of the program is proved.

5.4.8 FOR–Loops

$\texttt{EPT[FOR[}V\texttt{,}E_1\texttt{,}E_2\texttt{,}S\texttt{,}I\texttt{],}R\texttt{]} \rightarrow$

$\{ I_V^{E_1} \texttt{ , } \{ \texttt{ (}I \wedge \texttt{ (}E_1 \leq V\texttt{) } \wedge \texttt{ (}V \leq E_2\texttt{)) } \Rightarrow \texttt{ (EPT[}S\texttt{,}I_V^{V+1}\texttt{])}_1\texttt{ , } I_V^{E_2+1} \Rightarrow R \} \}$

A loop

FOR [V , E_1 , E_2 , S , I]

where a variable V ranges from the lower bound E_1 to the upper bound E_2 with a postcondition R

has the invariant I' as a precondition

where I' results from I by replacing every occurence of the variable V with E_1 and produces the following verification conditions:

$(I \wedge (E_1 \leq V) \wedge (V \leq E_2)) \Rightarrow I'$

where I' is obtained by computing the precondition of

S with respect to the postcondition Q

and Q results from I by substituting every occurence of the variable V by V+1.

$I'' \Rightarrow R$

where I'' results from I by replacing every occurence of the variable V with E_2+1

(with the syntactic condition that neither V, nor a variable occuring in E_1 or E_2 are changed inside S).

Example:

The program (in Hoare-Notation)

```
{m≥1 ∧ x=0}
FOR[ i, 1, m, x:=x+i, Invariant→(x=(i*(i-1)/2)) ]
{x=(m*(m+1)/2)}
```

produces the following verification conditions:

$(x = \frac{(i-1)*i}{2}) \bigwedge 1 \leq i \bigwedge i \leq m \Rightarrow (x + i = \frac{((1+i)-1)*(1+i)}{2})$

$(x = \frac{((m+1)-1)*(m+1)}{2}) \Rightarrow (x = m * \frac{m+1}{2})$

$m \geq 1 \wedge (x = 0) \Rightarrow (x = \frac{(1-1)*1}{2})$

The first two verification conditions are generated by the FOR rule as shown above and the third verification condition is the implication of the precondition and the invariant (the new precondition returned by EPT).

5.4.9 Procedure Calls

As already mentioned, procedures are programs with a procedural interface definition (without a functional return value). All programs written (or at least specified) in a *Theorema* session can be called from another program.

For the verification of a procedure call, it suffices that the pre- and the postcondition of this procedure are known. This fact is very helpful for developing programs top-down with the method of stepwise refinement. The procedure does not have to be implemented in order to enable verification of a program which calls this procedure, it only has to be specified.

Of course, for the verification of a program all (sub-)procedures have to be correct, hence the correctness of the called procedure has to be shown seperatly.

Since the rule for procedure calls is a bit complicated and we do not want to introduce new notations (in order to keep it simple), we only present the description in natural language.

A Procedure Call

$p[\ \overline{E}\ ,\ \overline{U}\ ,\ \overline{V}\ ]$

where $\overline{E}$ is a sequence of expressions as actual input parameters

and $\overline{U}$ is a sequence of variables as actual transient parameters

and $\overline{V}$ is a sequence of variables as actual output parameters

where the specification of the procedure p has

a precondition Q_p with the formal input and transient parameters as free variables

and a postcondition R_p with all formal parameters as free variables

and where Q_{pr} and R_{pr} result from Q_p and R_p by substituting the formal parameters with the actual parameters

with a postcondition R

produces the precondition Q which is $Q_{pr} \wedge \underset{\overline{u}', \overline{v}'}{\forall} (R_{pr}' \Rightarrow R')$

where $\overline{u}'$ and $\overline{v}'$ are sequences of new variables for all actual transient and output parameters

and R_{pr}' and R' result from R_{pr} and R by substituting those actual parameters with the new variables.

and no verification condition.

A more formal and detailed explanation of the rule for procedure calls can be found in [Futschek89].

Example:

To demonstrate this rule we present a program which just calls the "Factorial" procedure as it was shown in the section 5.3,"VCG using an Extended Predicate Transformer". It takes an input parameter `n` and returns the factorial in the output parameter `f`. In the specification it has a precondition `n≥0` and a postcondition `f=n!`. In the following example, however, we chose a different pre- and postcondition.

```
{in > 0}
  Factorial[in, out]
```

$$\left\{\text{out} = \prod_{j=1}^{\text{inj}} \right\}$$

The following verification condition is the result:

$$in > 0 \Rightarrow in \geq 0 \bigwedge \left(\underset{out1}{\forall} \left((out1 = in!) \Rightarrow \left(out1 = \prod_{j=1}^{in} \text{j} \right) \right) \right)$$

The variables `f` and `n` in the pre- and postcondition of the procedure are replaced by the actual parameters `in` and `out`. In the next step, the output variable `out`, also occuring in the postcondition of the above program, is substituted by a new variable, `out1`. Then the new precondition of the procedure is constructed according to the rule above. And finally, as always, an implication of the given precondition and the new precondition is built.

In the scope of our work we did not implement the verification of recursive procedures. For detailed information on recursion and mutually recursive procedures see [Homeier95].

5.4.10 Calls of Function Procedures

Expressions containing function calls are not handled differently than other expressions in a program; they are simply "inserted" into verification conditions. Hence, all the functions called within a program code "accumulate" in the verification conditions.

This will be demonstrated in the following simple example:

```
Specification ["Maximum", c = Max[↓a, ↓b],
 Pre → True,
 Post → ((c = a) ∧ (a ≥ b)) ∨ (c = b)
]

Program ["SquareOfMaximum", SMax[↓x, ↓y, ↑z],
 Pre → True,
  h := Max[x, y];
  z := h * h,
 Post → ((z = x * x) ∧ (x ≥ y)) ∨ (z = y * y)
]
```

Here we have the specification of a simple "Maximum" function procedure, which is then called from another program, "SquareOfMaximum". That is a procedure which takes two numbers `x` and `y`, computes the maximum of the two and returns the square of the maximum in the output variable `z`.

The only verification condition generated by "SquareOfMaximum" is:

$$True \Rightarrow ((Max[x, y] * Max[x, y] = x * x) \wedge x \geq y) \vee (Max[x, y] * Max[x, y] = y * y)$$

To handle the call of the function procedure in the proof of the verification conditions *additional knowledge* from the specification is needed, namely

$$True \Rightarrow ((Max[x,y]=x) \wedge (x \geq y)) \vee (Max[x,y]=y).$$

This is obtained from the postcondition by replacing the return variable `c` by the interface name `Max` with the *actual* input variables `x` and `y`.

The method shown in this example can be applied for arbitrary function procedures. It is not a rule for the verification condition generator, it is a technique for the *proof* of such programs. A more formal description of the method can be found in [Buchberger/Lichtenberger81].

Of course the correctness of such function procedures (like "normal" procedures) has to be proved separately.

6. User Tutorial

This tutorial is intended for students (or other users) who want to use the verification system. Understanding this tutorial requires basic knowledge in *Mathematica* and *Theorema*.

6.1 How to Write a Specification

```
Specification["Gaussian Sum", GaussSum[↓n, ↑s],
 Pre → (IsInteger[n] ∧ (n > 0)),
 Post → (s = Σ_{i=1,…,n} i)
]
```

This is a specification with the label "`Gaussian Sum`". It specifies a program (procedure) with the interface `GaussSum[↓n,↑s]`, where `n` is input parameter and `s` is output parameter. In the current version there is no necessitiy for the typization of variables.

Precondition and postcondition can be arbitrary formulas or predicates. Both, precondition and postcondition always have to be defined in a specification. If there is actually no precondition, one still has to give **`Pre→True`**.

An interface of a function procedure would have the following syntax: `s=GaussSum[↓n]`. For function procedures it is common that only input parameters are used, so one could write as well: `s=GaussSum[n]`.

After writing the above specification, it is stored internally under `Specification["Gaussian Sum"]`.

```
Specification["Gaussian Sum"]
```

•spec[GaussSum[ParIn[*n*], ParOut[*s*]], IsIntegerNumber[*n*] ∧ *n* > 0, *s* = Σ_{*i*=1,…,*n*} *i*]

6.2 How to Write a Program

```
Program ["Gaussian Sum Version 1", GaussSum[↓n, ↑s],
 s := 0;
 i := 1;
 WHILE[i ≤ n,
   s := s + i;
   i := i + 1
 ]
]
```

This is a program with the label "`Gaussian Sum Version 1`". Like programs (or procedures) in other languages, it has an interface definition which contains the name and

the parameters. `GaussSum[↓n,↑s]` is the procedural interface definition where `n` is input parameter and `s` is output parameter. Again, functional style would be possible, too: `s=GaussSum[↓n]`.

In order to guarantee that the verification condition generator works correctly the interface definition of the program must match exactly the one of the specification (e.g, the names of the parameters must be equal). The actual code consists of a sequence of statements (as described in "The Source Language") separated by semicolons.

At the end there is no need for a semicolon. Two semicolons without anything (or only blanks) in between are interpreted as an "empty statement".

The **`:=`** is an assignment like in Pascal. Multiple assignments like `x:=y:=z` are not allowed.

The `WHILE` statement has a functional form in this language. It consists of the loop condition and the loop body, which again is code (a sequence of statements).

The difference to the *Mathematica* `While` is that the loop condition is in *Theorema* Syntax (e.g. with = as equality instead of == or containing quanifiers).

Mind the convention that an input variable may not be changed within the program.

In the *Theorema Procedural Language* there is no BEGIN or END. However, there is also a block concept in the language, which again has a functional form.

```
Program["Gaussian Sum", GaussSum[↓n, ↑s],
 Module[{i},
  s := 0;
  FOR[i, 1, n,
    s := s + i
  ]
 ]
]
```

A `Module` "opens" a local scope and has two parameters: A list with local variables and a sequence of statements. It is the same `Module` as in *Mathematica*. Variables with the same names as the local variables should *not* occur outside of the block (at least in this version of the language). For a discussion where a Module is appropriate see the section 6.7.2. in the "Additional Topics".

Another thing has changed in the program: The `WHILE` statement is replaced by a `FOR`. The `FOR` is a loop which is easier to use than `WHILE` in case the programmer knows in advance *how often* the body should be executed. The semantics of the above is: `i` runs from `1` to `n`; `i` is always increased by one and `s:=s+1` is the actual loop body.

Another version of the `FOR` statement is possible where the user can define a positive or negative stepwidth. So the program above would produce the same result if one wrote `FOR[i,1,n,1,s:=s+1]` or even `FOR[i,n,1,-1,s:=s+1]`.

After writing the above program, it is stored internally under `Program["Gaussian Sum"]`.

```
Program["Gaussian Sum"]
```

```
•prog[Gaussian Sum, GaussSum[ParIn[n], ParOut[s]], True,
 True, Module[{i}, s := 0; ™For[i, 1, n, s := s + i]]]
```

6.3 How to Annotate a Program

With this verification system only a minimal set of annotations must be supplied, that is, invariants for all loops in a program. In *Theorema Procedural Language* the possibility of giving invariants is part of the language. The statements `FOR` and `WHILE` take an optional argument `Invariant→` followed by the invariant formula, which can be an arbitrary *Theorema* formula. However, everything which is used in invariants will finally appear in the verification condition, and it might not be possible to prove it with the existing automatic provers (this issue is discussed in more detail in the section 4.4, "Proving with *Theorema*").

The invariant for the example above is `s=((i-1)*i)/2`. In the following we show a possibility to specify the pre- and the postcondition directly in the program instead of having a separate specification.

```
Program["Gaussian Sum Version2", GaussSum[↓n, ↑s],
 Pre → (IsInteger[n] ∧ (n > 0)),
  s := 0;
  i := 1;
  WHILE[i ≤ n,
    s := s + i;
    i := i + 1,
    Invariant → (s = ((i - 1) * i) / 2)
  ],
 Post → (s = (n + 1) * n / 2)
]
```

However, we recommend to write the specification separately, before writing the program. Equal to giving the pre- and the postcondition as (optional) parameters to the program one could also "make the link" to the specification explicit by e.g. giving

```
Specification→Specification["Gaussian Sum"]
```

6.4 How to Generate Verification Conditions

The function for producing the verification conditions is called VCG. There are two different possibilities for calling it.

```
VCG[Program["Gaussian Sum Version2"]]
```

In this case the program has to be fully annotated with a pre- and postcondition (or with a link to a specification) as shown in "How to annotate a program". Otherwise VCG will not produce correct results.

```
VCG[Program["Gaussian Sum Version2"],
 Specification["Gaussian Sum"]]
```

Here the program will be verified using the pre- and postcondition from the specification (the ones of the program are "overruled"). Of course it has to have an interface definition matching the one of the program.

The output of the VCG function is a Theorema lemma (with a formula list containing the verification conditions) in so-called "Traditional Form":

Lemma (Gaussian Sum Version2) :
for any : n, s
(WHILE . Inv)
$$\left(s = \frac{(i-1)*i}{2}\right) \bigwedge i \leq n \Rightarrow \left(s + i = \frac{((i+1)-1)*(i+1)}{2}\right)$$
(WHILE . Final) $$\left(s = \frac{(i-1)*i}{2}\right) \bigwedge (i \not\leq n) \Rightarrow \left(s = \sum_{i=1,\ldots,n} i\right)$$
(Init) IsIntegerNumber[n] $\wedge$ $n > 0 \Rightarrow \left(0 = \frac{(1-1)*1}{2}\right)$

For proving one can refer to the Lemma with `Lemma["Gaussian Sum Version2"]`.

6.5 How to Verify Termination

The only statement in the *Theorema Procedural Language* which could cause nontermination is `WHILE`. Therefore the `WHILE` statement has an optional argument where the user can give a *termination term* which is built of the program variables in the loop and denotes a natural number which decreases with *every* iteration of the loop.

If you add a termination term to *every* `WHILE` statement in a program and generate verification conditions, then a proof of these implies total correctness.

The termination term for the above example is `n-i`.

```
Program["Gaussian Sum Version1", GaussSum[↓n, ↑s],
 s := 0;
 i := 1;
 WHILE[i ≤ n,
   s := s + i;
   i := i + 1,
   Invariant → (s = ((n - 1) * n) / 2),
   TerminationTerm → n - i
 ]
]
```

6.6 How to Use Arrays

Arrays are tuples, in fact *Theorema*-tuples. The index of the first element is always 1.

There are three operations on arrays needed in our context:

- The length of an array `A` is `|A|`.
- An element at position i of an array can be accessed by subscripting A_i.

- An element at position i of an array can be replaced by a value v either (in the code) by `A_i :=v` or (in conditions) by $A_{i \leftarrow v}$.

For arrays `a` and `c`, '`a:=c`' will be interpreted as copying the array denoted by '`c`' to a new memory space denoted by '`a`' so that there are now two different arrays with initially the same contents in the memory. Since the *Theorema Procedural Language* is currently a typeless language this assignment is not any different from an assignment of "normal" variables.

The use of arrays is demonstrated in "SimpleArray" in the section "Examples" (7.3).

6.7 Aditional Topics

6.7.1 Transient Parameters

If a program is intended to perform some modification of a parameter (like for instance a program *Sort* which sorts an array in place), it is often undesirable to have a seperate input and output parameter. In such a case it is much more natural to have a transient parameter.

In interface definitions in our verification system transient parameters are marked with an arrow ↕.

For example:

```
Sort[↕a]
```

Usually transient parameters lead to the following problem: The postcondition has to relate to the initial values of the (modified) transient parameters.

The solution is to "copy" the initial values of the transient parameters into some new constants (so-called ghost variables in [Backhouse86]) which are not modified in the program code. This is usually done in the precondition, as shown in the following example:

```
Specification["Sort", Sort[↕a],
 Pre → (a = A),
 Post → (IsSorted[a] ∧ HasSameElements[a, A])]
```

The array `a` is "stored" in `A` in the precondition, such that the postcondition (`IsSorted[]` and `HasSameElements[]` are predicates defined within *Theorema*) can relate to the initial value of `a`.

The same "trick" can be seen in the example "Swap" in the section "Examples" (7.2.).

6.7.2 Where to Use Blocks

In the *Theorema Procedural Language* a sequence of statements (separated by semicolons) does not have to be grouped by a block `Module`. However, if one wants to have local variables one *has* to use a `Module`. Local variables have the advantage that they do not have a value within the global scope of *Mathematica* after program execution with `Compute`.

6.7.3 Debug Output

In case the VCG produces doubtful results, or if one is interested in the application of the single rules, one can activate "debug output". Then the VCG prints out the name of every applied rule and some details of the intermediate data. This can be done by simply setting the debug level, which is a global variable in *Theorema*.

```
SetGlobals[DebugLevel -> 10]
```

7. Examples

In the following we present example programs that show the capabilities of the verification system. Each example basically consists of three parts: the specification of the program, the program itself (written in the *Theorema Procedural Language*), and finally the call and the output of the verification condition generator. In some examples we also demonstrate the execution of the programs by the `Compute` function.

7.1 Simple Example

This example is only a first "test". The program consits only of two assignments; so one can see how the rule for assignments is applied, performing substitutions "back-to-front".

```
Specification["Simple", Simple[↓b, ↑a],
 Pre → True,
 Post → (a = 2 * b + 3)]

Program["Simple", Simple[↓b, ↑a],
   t := 2 b + 5;
   a := t - 2,
 Specification → Specification["Simple"]]

VCG[Program["Simple"]]

Lemma ( Simple ) :
 for any : b, a
(Init)  True ⇒ ((2 * b + 5) - 2 = 2 * b + 3)

Compute[Simple[2, x], by → ProgramEvaluator]

⟨x → 7⟩
```

7.2 Swap

In this example the values of two variables are "swapped". It shows an interface with transient parameters and demonstrates the use of so-called "ghost variables", as discussed in the section 6.7.1, "Transient Parameters", in the tutorial.

```
Specification["Swap", SwapSpec[↕x, ↕y],
 Pre → ((x = X) ∧ (y = Y)),
 Post → ((x = Y) ∧ (y = X))]

Program["Swap", Swap[↕x, ↕y],
 Module[{tmp},
  tmp := x;
  x := y;
  y := tmp
 ]]
```

Here we use a `Module` to have `tmp` as a local variable. This is in contrast to the first example above, where the variable `t` would be set in a global scope.

```
VCG[Program["Swap"], Specification["Swap"]]
```

```
Lemma ( Swap ) :
 for any : x, y, tmp
(Init)  (x = X) ∧ (y = Y) ⇒ (y = Y) ∧ (x = X)
```

This very simple verification condition could for example be shown by the *Theorema* "Predicate Logic Prover" by calling

```
Prove[Lemma["Swap"], by → PredicateProver]
```

Transient parameters should have a value before execution. In the following, two variables a and b "swap' their values.

```
a = 0; b = 1;

Compute[Swap[a, b], by → ProgramEvaluator]

⟨a → 1, b → 0⟩
```

7.3 SimpleArray

In this example we demonstrate the use of arrays in the *Theorema Procedural Language*. The program takes an array `a` and a value `b` and returns an array `c` with the value of b as the first element.

```
Specification["SimpleArray", SimpleArraySpec[↓a, ↓b, ↑c],
 Pre → True, Post → (c = a_{1⇤b})]

Program["SimpleArray", SimpleArray[↓a, ↓b, ↑c],
 c := a;
 c_1 := b
 ]

VCG[Program["SimpleArray"], Specification["SimpleArray"]]

Lemma ( SimpleArray ) :
 for any : a, b, c
(Init)  True ⇒ (a⟦1 ⇤ b⟧ = a⟦1 ⇤ b⟧)
```

In this verification condition a⟦1⇤b⟧ denotes the same as $a_{1⇤b}$, it is just the current output format. The following call of `Compute` shows that arrays are in fact tuples.

```
Compute[SimpleArray[⟨2, 4, 1⟩, 0, out], by → ProgramEvaluator]

⟨out → ⟨0, 4, 1⟩⟩
```

7.4 Maximum

This example shows the application of the verification rule for an `IF` statement. The program uses the operator ">" to compute the maximum of two values. In the specification we also use a function `Max`, e.g. from *Mathematica*.

```
Specification["Maximum", Maximum[↓x, ↓y, ↑result],
  Pre → True,
  Post → (result = Max[x, y])]


Program["Maximum", Maximum[↓x, ↓y, ↑result],
  IF[x > y,
   result := x,
   result := y]]


VCG[Program["Maximum"], Specification["Maximum"]]
```

```
Lemma ( Maximum ) :
 for any : x, y, result
(Init)
  True ⇒ (x > y ⇒ (x = Max[x, y])) ∧ ((¬ (x > y)) ⇒ (y = Max[x, y]))
```

7.5 Remainder

This example is the first one with a loop and contains an invariant. The program computes the remainder `r` of two numbers `x` and `y`.

```
Specification["Remainder", Remainder[↓x, ↓y, ↑r],
 Pre → (True),
 Post → (∃ (x = r + y * q) ⋀ r < y)]

Program["Remainder", Remainder[↓x, ↓y, ↑r],
    r := x;
    q := 0;
    WHILE[ y ≤ r,
     r := r - y;
     q := q + 1,
     Invariant → (x = r + y * q)
   ]
]

VCG[Program["Remainder"], Specification["Remainder"]]
```

```
Lemma ( Remainder ) :
 for any : x, y, r
(WHILE . Inv)   (x = r + y * q) ∧ y ≤ r ⇒ (x = (r - y) + y * (q + 1))
(WHILE . Final)   (x = r + y * q) ∧ (y ≰ r) ⇒ ∃ (x = r + y * q) ⋀ r < y
                                              q
(Init)   True ⇒ (x = x + y * 0)
```

The above verification conditions do not contain a "statement of termination". Therefore, to verify total correctness, we can also supply a termination term with the loop (e.g. just `r`).

```
Program["Remainder", Remainder[↓x, ↓y, ↑r],
    r := x;
    q := 0;
    WHILE[ y ≤ r,
     r := r - y;
     q := q + 1,
     Invariant → (x = r + y * q),
     TerminationTerm → r
   ]
]

VCG[Program["Remainder"], Specification["Remainder"]]
```

Lemma (Remainder) :
for any : x, y, r

(WHILE . Inv + Term) $(x = r + y * q) \wedge y \leq r \wedge (r = \mathrm{T1}) \Rightarrow$
$(x = (r - y) + y * (q + 1)) \wedge r - y < \mathrm{T1}$

(WHILE . Final) $(x = r + y * q) \wedge (y \nleq r) \Rightarrow \exists_{q} (x = r + y * q) \bigwedge r < y$

(WHILE . Term) $(x = r + y * q) \wedge y \leq r \Rightarrow r \geq 0$

(Init) $\mathrm{True} \Rightarrow (x = x + y * 0)$

The constant `T1` is a new identifier (generated by the system) used to "store" the value of the termination term at the beginning of the loop and for comparison at the end of the loop.

7.6 Factorial

This example shows two different implementations of the well-known factorial function. Both take a natural number `n` as input and returns `n!` as the value of `f` (as defined in the specification). The first version of the program, based on a `FOR` loop, starts with `f=1` and successively multiplies `f` with the values `2` to `n`. The second version uses a `WHILE` loop works differently; it multiplies `f` with the values `n` down to `2`.

```
Specification ["Factorial", FactSpec[↓n, ↑f],
 Pre → (n ≥ 0),
 Post → (f = n!)]

Program ["Factorial Version 1", Fact1[↓n, ↑f],
  f := 1;
  i := n;
  FOR[i, 2, n,
   f := f * i,
   Invariant → ((f = (i - 1)!))
  ]
 ]

Program ["Factorial Version 2", Fact2[↓n, ↑f],
  f := 1;
  i := n;
  WHILE[i > 1,
   f := f * i;
   i := i - 1,
   Invariant → ((i ≥ 0) ∧ (f = n! / i!)),
   TerminationTerm → i
  ]
 ]
```

```
VCG[Program["Factorial Version 1"],
 Specification["Factorial"]]
```

Lemma (Factorial Version 1) :
 for any : n, f

(FOR . Inv1) $(f = (i - 1)!) \wedge 2 \leq i \wedge i \leq n \Rightarrow (f * i = ((1 + i) - 1)!)$

(FOR . Inv2) $(f = ((n + 1) - 1)!) \Rightarrow (f = n!)$

(Init) $n \geq 0 \Rightarrow (1 = (2 - 1)!)$

```
VCG[Program["Factorial Version 2"],
 Specification["Factorial"]]
```

Lemma (Factorial Version 2) :
 for any : n, f

(WHILE . Inv + Term) $\left(i \geq 0 \bigwedge \left(f = \frac{n!}{i!}\right)\right) \bigwedge i > 1 \bigwedge (i = \mathrm{T2}) \Rightarrow$

$$\left(i - 1 \geq 0 \bigwedge \left(f * i = \frac{n!}{(i-1)!}\right)\right) \bigwedge i - 1 < \mathrm{T2}$$

(WHILE . Final) $\left(i \geq 0 \bigwedge \left(f = \frac{n!}{i!}\right)\right) \bigwedge (i \not> 1) \Rightarrow (f = n!)$

(WHILE . Term) $\left(i \geq 0 \bigwedge \left(f = \frac{n!}{i!}\right)\right) \bigwedge i > 1 \Rightarrow i \geq 0$

(Init) $n \geq 0 \Rightarrow n \geq 0 \bigwedge \left(1 = \frac{n!}{n!}\right)$

Note the difference of the verification conditions of the two versions. However, all these verification conditions can easily be proved using elementary arithmetic knowledge (including a definition of the factorial function).

7.7 Fibonacci Numbers

The program in this example takes a natural number `n` as input, computes the nth Fibonacci numberand returns the value in `f`. The annotations rely on a function `Fibonacci` which has to be defined in *Theorema* for proving (actually there is such a funtion in *Mathematica*, see [Wolfram96]).

```
Specification["Fibonacci", FibonacciProcSpec[↓n, ↑f],
 Pre → (n > 0),
 Post → (f = Fibonacci[n])]

Program["Fibonacci", FibonacciProc[↓n, ↑f],
 Module[{h, i},
  i := n;
  f := 1;
  h := 1;
  WHILE[i > 1,
   h := h + f;
   f := h - f;
   i := i - 1,
   Invariant →
    (f = Fibonacci[n - i + 1]) ∧ (h = Fibonacci[n - i + 2]) ∧ (i > 0),
   TerminationTerm → i
  ]
 ]
]
```

```
VCG[Program["Fibonacci"], Specification["Fibonacci"]]
```

```
Lemma ( Fibonacci ) :
 for any : n, f, h, i
(WHILE . Inv + Term)   ((f = Fibonacci[(n - i) + 1]) ∧
        (h = Fibonacci[(n - i) + 2]) ∧ i > 0) ∧ i > 1 ∧ (i = T3) ⇒
    (((h + f) - f = Fibonacci[(n - (i - 1)) + 1]) ∧
        (h + f = Fibonacci[(n - (i - 1)) + 2]) ∧ i - 1 > 0) ∧ i - 1 < T3
(WHILE . Final)   ((f = Fibonacci[(n - i) + 1]) ∧
        (h = Fibonacci[(n - i) + 2]) ∧ i > 0) ∧ (i ≯ 1) ⇒
    (f = Fibonacci[n])
(WHILE . Term)   ((f = Fibonacci[(n - i) + 1]) ∧
        (h = Fibonacci[(n - i) + 2]) ∧ i > 0) ∧ i > 1 ⇒ i ≥ 0
(Init)   n > 0 ⇒
    (1 = Fibonacci[(n - n) + 1]) ∧ (1 = Fibonacci[(n - n) + 2]) ∧ n > 0
```

7.8 Egyptian Multiplication

The Egyptian Multiplication algorithm multiplies two integer numbers `x` and `y` and returns the result in `s`. This example is more complicated than the previous, because it has an IF statement in the loop.

Note that, despite the algorithm is not trivial, the invariant is quite simple. The function `IsEven` simply test if the input parameter is an even number; `IntegerPart` cuts away the fractional part.

```
Specification["Egyptian Multiplication", EgyptMult[↓x, ↓y, ↑s],
 Pre → True,
 Post → (s = x * y)]

Program["Egyptian Multiplication", EgyptMult[↓x, ↓y, ↑s],
 Module[{a, b},
  a := x;
  b := y;
  s := 0;
  WHILE[¬ (b = 0),
   IF[¬ IsEven[b], s := s + a];
   a := 2 * a;
   b := IntegerPart[b / 2],
   Invariant → (x * y = a * b + s)
  ]
 ]
]

VCG[Program["Egyptian Multiplication"],
 Specification["Egyptian Multiplication"]]
```

```
Lemma ( Egyptian Multiplication ) :
 for any : x, y, s, a, b
(WHILE . Inv)   (x * y = a * b + s) ∧ (b ≠ 0) ⇒ ((¬ (IsEven[b])) ⇒
        (x * y = (2 * a) * IntegerPart[b/2] + (s + a))) ⋀
     ((¬ (¬ (IsEven[b]))) ⇒ (x * y = (2 * a) * IntegerPart[b/2] + s))
(WHILE . Final)   (x * y = a * b + s) ∧ (¬ (b ≠ 0)) ⇒ (s = x * y)
(Init)   True ⇒ (x * y = x * y + 0)
```

7.9 Greatest Common Divisor

In this example we present two different implementations of the "Greatest Common Divisor" problem. In order to keep it simple, the specification and the invariants are based on a GCD function, e.g. defined in *Theorema*.

```
Specification["Greatest Common Divisor", GrCoDiv[↓m, ↓n, ↑z],
 Pre → (m ≥ 0) ∧ (n ≥ 0),
 Post → (z = GCD[m, n])]

Program["GCD Version 1", GrCoDiv[↓m, ↓n, ↑z],
 Module[{r},
  z := m;
  r := n;
  WHILE[¬ (r = z),
   IF[z > r,
    z := z - r,
    r := r - z],
   Invariant → (GCD[z, r] = GCD[m, n])
  ]
 ],
 Specification → Specification["Greatest Common Divisor"]
]

VCG[Program["GCD Version 1"],
 Specification["Greatest Common Divisor"]]
```

```
Lemma ( GCD Version 1 ) :
 for any : m, n, z, r
(WHILE . Inv)  (GCD[z, r] = GCD[m, n]) ∧ (r ≠ z) ⇒
     (z > r ⇒ (GCD[z - r, r] = GCD[m, n])) ∧
      ((¬ (z > r)) ⇒ (GCD[z, r - z] = GCD[m, n]))
(WHILE . Final)
   (GCD[z, r] = GCD[m, n]) ∧ (¬ (r ≠ z)) ⇒ (z = GCD[m, n])
(Init)  m ≥ 0 ∧ n ≥ 0 ⇒ (GCD[m, n] = GCD[m, n])
```

```
Program["Euclid", GrCoDiv[↓m, ↓n, ↑z],
 Module[{r, h},
  z := m;
  r := n;
  WHILE[¬ (r = 0),
   h := Mod[z, r];
   z := r;
   r := h,
   Invariant → (GCD[z, r] = GCD[m, n]),
   Specification → Specification["Greatest Common Divisor"]
  ]
 ]
]
```

```
VCG[Program["Euclid"],
 Specification["Greatest Common Divisor"]]
```

```
Lemma ( Euclid ) :
 for any : m, n, z, r, h
(WHILE . Inv)  (GCD[z, r] = GCD[m, n]) ∧ (r ≠ 0) ⇒
    (GCD[r, Mod[z, r]] = GCD[m, n])
(WHILE . Final)
   (GCD[z, r] = GCD[m, n]) ∧ (¬ (r ≠ 0)) ⇒ (z = GCD[m, n])
(Init)  m ≥ 0 ∧ n ≥ 0 ⇒ (GCD[m, n] = GCD[m, n])
```

For proving the verification conditions one needs knowlegde about the function `Mod`. It can be either a *Mathematica* (or *Theorema*) function or a function procedure which at least has to be specified and then transformed into methematical knowledge as shown in "Calls of Function Procedures".

For example, `Mod` as a function procedure could have the following specification (similar to the example "Remainder"):

```
Specification["Modulo", r = Mod[↓x, ↓y],
 Pre → True,
 Post → ((x = r + y * q) ∧ r < y)]
```

One could transform this specification into knowledge for using in the proof (according to what was described in the section 5.4.10, "Calls of Function Procedures"):

```
True ⇒ ( x = Mod[x, y] + y * q ) ∧ Mod[x, y] < y)
```

7.10 HasElement

This simple example demonstrates the use of an existential quantifier in the condition of an `IF` statement. The program takes an array `a` and a value `n` and tests if `n` occurs in `a` and sets the output variable `r` to the appropriate truth value. This program works for all kinds of "data types" for which the operator "=" is defined.

```
Specification["HasElement", HasElement[↓a, ↓n, ↑r],
 Pre → (True),
 Post → ((r = True) ⋀ (∃_{j=1,…,|a|} (n = a_j))) ⋁ (r = False)]

Program["HasElement", HasElement[↓a, ↓n, ↑r],
  IF[∃_{i=1,…,|a|} (n = a_i),
   r := True,
   r := False
    ]
]
```

```
VCG[Program["HasElement"], Specification["HasElement"]]
```

```
Lemma ( HasElement ) :
 for any : a, n, r
```

$$\text{(Init)}\quad \text{True} \Rightarrow \left(\underset{i=1,\ldots,|a|}{\exists} (n = a_i) \Rightarrow \right.$$

$$\left. (\text{True} = \text{True}) \bigwedge \underset{j=1,\ldots,|a|}{\exists} (n = a_j) \bigvee (\text{True} = \text{False}) \right) \bigwedge$$

$$\left(\left(\neg \left(\underset{i=1,\ldots,|a|}{\exists} (n = a_i) \right) \right) \Rightarrow \right.$$

$$\left. (\text{False} = \text{True}) \bigwedge \underset{j=1,\ldots,|a|}{\exists} (n = a_j) \bigvee (\text{False} = \text{False}) \right)$$

This verification condition is always true. Note that, in *Theorema,* the operator "∧" has precedence before "∨".

7.11 Multiple GCD

Here we show a more complicated example, where the greatest common divisor of all elements (natural numbers) in an array is computed. We used an (infix) operator "|" which returns *True* if one number devides the other and false otherwise.

$$\text{Specification}\left[\text{"MultipleGCD"}, \text{MultiGCD}[\downarrow a, \uparrow z],\right.$$

$$\text{Pre} \rightarrow \left((|a| \geq 2) \bigwedge \left(\underset{i=1,\ldots,|a|}{\forall} (a_i \geq 1) \right) \right),$$

$$\left. \text{Post} \rightarrow \left(\left(\underset{i=1,\ldots,|a|}{\forall} (z \mid a_i) \right) \bigwedge \left(\underset{y}{\forall} \left(\left(\underset{i=1,\ldots,|a|}{\forall} (y \mid a_i) \right) \Rightarrow (y \leq z) \right) \right) \right) \right]$$

The implementation is based on a procedure which computes the greatest common divisor for just two numbers.

$$\text{Program}\left[\text{"MultipleGCD"}, \text{MultiGCD}[\downarrow a, \uparrow z],\right.$$

$$\text{Module}\left[\{y\},\right.$$

$$z := a_1;$$

$$\text{FOR}\left[j, 2, |a|,\right.$$

$$\text{GrCoDiv}[a_j, z, y];$$

$$z := y,$$

$$\text{Invariant} \rightarrow \left(\left(\underset{i=1,\ldots,(j-1)}{\forall} (z \mid a_i) \right) \right.$$

$$\bigwedge \left(\underset{y}{\forall} \left(\left(\underset{i=1,\ldots,(j-1)}{\forall} (y \mid a_i) \right) \Rightarrow (y \leq z) \right) \right)$$

$$\left. \left. \bigwedge \left(\underset{i=1,\ldots,|a|}{\forall} (a_i \geq 1) \right) \right) \right]$$

$$\left. \right]$$

$$\left. \right]$$

In stepwise refinement one designes algorithms "top-down". For example, the procedure `GrCoDiv` only has to be specified (and does not yet have to be implemented) in order to enable verification (the specification can be found in the section 7.9.). Likewise, one could

design the procedure GrCoDiv based on a procedure Remainder which equally is only specified in the first place (this can be found in the section 7.5.).

VCG[Program["MultipleGCD"], Specification["MultipleGCD"]]

Lemma (MultipleGCD) :
for any : a, z, y

(FOR . Inv1) $\left(\forall_{i=1,\ldots,j-1} (z \mid a_i) \bigwedge \forall_{y} \left(\forall_{i=1,\ldots,j-1} (y \mid a_i) \Rightarrow y \le z\right) \bigwedge \forall_{i=1,\ldots,|a|} (a_i \ge 1)\right) \bigwedge 2 \le j \bigwedge j \le |a| \Rightarrow$
$(a_j \ge 0 \wedge z \ge 0) \bigwedge \forall_{y1} \left((y1 = \mathrm{GCD}[a_j, z]) \Rightarrow \forall_{i=1,\ldots,(1+j)-1} (y1 \mid a_i) \bigwedge \left(\forall_{i=1,\ldots,(1+j)-1} (y1 \mid a_i) \Rightarrow y1 \le y1\right) \bigwedge \forall_{i=1,\ldots,|a|} (a_i \ge 1)\right)$

(FOR . Inv2) $\forall_{i=1,\ldots,(|a|+1)-1} (z \mid a_i) \bigwedge$
$\forall_{y} \left(\forall_{i=1,\ldots,(|a|+1)-1} (y \mid a_i) \Rightarrow y \le z\right) \bigwedge \forall_{i=1,\ldots,|a|} (a_i \ge 1) \Rightarrow$
$\forall_{i=1,\ldots,|a|} (z \mid a_i) \bigwedge \forall_{y} \left(\forall_{i=1,\ldots,|a|} (y \mid a_i) \Rightarrow y \le z\right)$

(Init) $|a| \ge 2 \bigwedge \forall_{i=1,\ldots,|a|} (a_i \ge 1) \Rightarrow \forall_{i=1,\ldots,2-1} (a_1 \mid a_i) \bigwedge$
$\forall_{y} \left(\forall_{i=1,\ldots,2-1} (y \mid a_i) \Rightarrow y \le a_1\right) \bigwedge \forall_{i=1,\ldots,|a|} (a_i \ge 1)$

In (FOR.Inv1) the verification condition generator uses the rule for a procedure call for GrCoDiv (y1 is a new identifier generated).

7.12 Nested WHILE's

This example demonstrates that even in programs with nested WHILE statements one only has to supply the loop invariants (the "minimal set of annotations") for the verification condition generation.

Specification["Nested WHILE's", MyPower[↓x, ↓y, ↑z],
Pre → (y ≥ 0),
Post → (z = x^y)]

```
Program["Nested WHILE's", MyPower[↓x, ↓y, ↑z],
 Module[{a, b},
  IF[y = 0,
   z := 1,
   a := x;
   b := y;
   z := 1;
   WHILE[b > 0,
    b := b - 1;
    z := z * a;
    WHILE[EvenQ[b] ∧ (b > 0),
     b := b / 2;
     a := a * a,
     Invariant → (z * a^b = x^y)
    ](*WHILE*),
    Invariant → ((z * a^b = x^y) ⋀ (b > 0))
   ](*WHILE*)
  ](*IF*)
 ], (*Module*)
 Specification → Specification["Nested WHILE's"]
]

VCG[Program["Nested WHILE's"],
 Specification["Nested WHILE's"]]
```

Lemma (Nested WHILE' s) :
for any : x, y, z, 3, 0

(WHILE . Inv) $(z * 3^0 = x^y) \bigwedge (\text{True} \wedge 0 > 0) \Rightarrow \left(z * (3 * 3)^{\frac{0}{2}} = x^y\right)$

(WHILE . Final) $(z * 3^0 = x^y) \bigwedge (\neg (\text{True} \wedge 0 > 0)) \Rightarrow (z * 3^0 = x^y) \bigwedge 0 > 0$

(WHILE . Inv) $((z * 3^0 = x^y) \bigwedge 0 > 0) \bigwedge 0 > 0 \Rightarrow ((z * 3) * 3^{0-1} = x^y)$

(WHILE . Final) $((z * 3^0 = x^y) \bigwedge 0 > 0) \bigwedge (0 \not> 0) \Rightarrow (z = x^y)$

(Init)
$y \geq 0 \Rightarrow ((y = 0) \Rightarrow (1 = x^y)) \wedge ((\neg (y = 0)) \Rightarrow (1 * x^y = x^y) \wedge y > y)$

8. Summary

8.1 Goals Achieved

We have built a verification system which enables the user to *specify*, *implement*, *verify* and *execute* programs within one uniform environment – *Theorema*. We have designed the system for educational purposes and built a simple user interface, consistent with the look-and-feel of *Theorema*.

A major objective of our work was to build a verification condition generator. Based on an "Extended Predicate Transformer" function, it makes verification possible with only a minimal set of necessary annotations supplied by the user.

With the *Theorema Procedural Language*, we have designed a small language with ideal properties for program verification within the functional environment of *Theorema*. We have tested the system with numerous examples and provided a tutorial for future users of the system.

However, the system has some limitations. The most serious restriction is that, with current *Theorema* provers, typical verifications are difficult to prove automatically (but this thesis was actually not concerned with the proving). Another limitation of the verification system stems from its nature as a educational environment; the current version is not suitable for the verification of "real-life" programs.

Nevertheless, the goal of this thesis, which was to extend *Theorema* with a verification system for educational purposes, has clearly been achieved.

8.2 Next Steps

The system will first be used in Spring 2000 for mathematics education in the Software Engineering course at the Polytechnical University in Hagenberg. Together with the Interface for Interactive Proving in *Theorema,* it will form a comprehensive training environment for program verification.

Possible additions to the current system include:

- Adding type information to the *Theorema Procedural Language*. This could be done by using predicates, like for instance `IsInteger[i]`. These predicates are already part of the *Theorema* system, but verifying data types needs additional verification techniques.
- Writing a "Program Verification Prover" within *Theorema* which is specialized in the typical requirements of verification conditions. The verification condition generator could be embedded in the "Program Verification Prover" such that fully automated proving of (simple) programs is possible.

- Writing preprocessors which translate programs written in other programming languages (e.g. Pascal) to the *Theorema Procedural Language*.
- Adding the verification of recursive and mutually recursive procedures to the system.
- Writing a compiler for the *Theorema Procedural Language* for fast execution of verified programs. This will possibly be done when a compiler for arbitrary mathematical definitions in *Theorema* is developed.

It is expected that the verification system, including the *Theorema Procedural Language*, will be part of the β-version of *Theorema*.

References

[Aho/Ullman77] Alfred V. Aho, Jeffrey D. Ullman. *Principles of Compiler Design.* Addison-Wesley, 1977.

[Backhouse86] Roland C. Backhouse. *Program Construction and Verification.* Prentice-Hall, 1986.

[Baumgartner88] Gerald Baumgartner. *Generating Verification Systems from Language Definitions*, Master Thesis, University Linz, Department for Informatics, Austria 1988.

[Boyer/Moore75] R. S. Boyer and J. S. Moore. *Proving Theorems about LISP Functions.* Journal of ACM, Vol. 22, 1975.

[Boyer/Moore80] R. S. Boyer and J. S. Moore. *The FORTRAN Verification System.* In: VERkshop Workshop on Formal Verification, ACM SIGSOFT, Software Engineering Notes, Vol. 5, 1980.

[Buchberger98] Bruno Buchberger. *Theorema: The Current Status.* In: Second International Theorema Workshop. Technical Report, RISC Research Institute for Symbolic Computation, Hagenberg, Austria, 1998.

[Buchberger/Lichtenberger81] Bruno Buchberger and Franz Lichtenberger. *Mathematik für Informatiker I: Die Methode der Mathematik.* Springer–Verlag, 1981.

[Clarke/Wing96] Edmund M. Clarke and Jeannette M. Wing. *Formal Methods: State of the Art and Future Directions.* Carnegie Mellon University, Computer Science Department, 1991.

[Dijkstra72] Edsger W. Dijkstra. *Notes on structured programming.* In: O.–J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, Structured Programming. Academic Press, 1972.

[Dijkstra76] E.W Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[Floyd67] R. W. Folyd. *Assigning Meaning to Programs.* Mathematical Aspects of Computer Science, American Mathematical Society, 1967.

[Futschek89] Gerald Futschek. *Programmentwicklung und Verifikation.* Springer Verlag Wien New York, 1989.

[Good85] D. I. Good. *Revised Report on GYPSY 2.1 (Draft).* Technical Report, University of Texas at Austin, 1985.

[Gordon98] Mike Gordon. *Lecture Notes on Specification and Verification I.* www.cl.cam.ac.uk/users/mjcg/Teaching/SpecVer1/SpecVer1.html (accessed on June 28, 1999), Unsiversity of Cambridge Computer Laboratory, 1998.

[Gray87] D. Gray. *A Pedagogical Verification Condition Generator.* In: The Computer Journal, 30, June 1987.

[Gries81] D. Gries. *The Science of Programming*. Springer Verlag, 1981.

[Hoare69] C.A.R. Hoare. *An Axiomatic Basis for Computer Programming.* Comm ACM 12, 1969.

[Hohlfeld92] Bernhard Hohlfeld. *Einfuehrung in die Programmverifikation: Theorie und Anwendung der Hoareschen Methode am Beispiel der Programmiersprache PASCAL.* BI Wissenschaftsverlag, 1992.

[Hoare/Wirth73] C.A.R. Hoare and N. Wirth. *An Axiomatic Definition for the Programming Language Pascal.* Acta Informatica 2, 1973.

[Homeier95] Peter Vincent Homeier. *Trustworthy Tools for Trustworthy Programs: A Mechanically Verified Verification Condition Generator for the Total Correctness of Procedures*. Dissertation for the degree Doctor of Philosophy in Computer Science, Universitiy of California, Los Angeles, 1995.

[Igarashi/London/Luckham75] S. Igarashi, R. L. London and D. C. Luckham. *Automatic Program Verification I, A Logical Basis and its Implementation.* Acta Informatatica, Vol. 4, 1975.

[Klaeren83] Herbert A. Klaeren. *Algebraische Spezifikation.* Springer Verlag Wien New York, 1983.

[King70] J. C. King. *A Program Verifier*. Dissertation, Carnegie Mellon University, 1970.

[Kossak99] Felix Kossak. *An Interface for Interactive Proving with the Mathematical Software System Theorema*. Master thesis, Polytechnical University of Upper Austria, Department for Software Engineering, Hagenberg, Austria, 1999.

[Lichtenberger98] Franz Lichtenberger. *Mathematik für Software−Ingenieure: Gibt's das überhaupt?* In: Telematik, Zeitschrift des Telematik−Ingenieur−Verbandes, 2 / 1998.

[Moriconi80] M. S. Moriconi. *Automatic Generation of Verification Condition Generators*. SRI CSL report, 1980.

[Ragland73] Larry Calvin Ragland. *A Verified Program Verifier*. Technical Report 18, Department of Computer Sciences, The University of Texas at Austin, Austin, 1973.

[SPV79] *Stanford Pascal Verifier − User Manual*. Computer Science Department, Stanford University, 1979.

[Wolfram96] Stephen Wolfram. *The Mathematica Book*, 3rd ed. Wolfram Media / Cambridge University Press, 1996.

Vistited Homepages

[H1] http://www.comlab.ox.ac.uk/archive/formal−methods.html

Formal Methods Virtual Library. Accessed on June 28, 1999.

[H2] http://www.csr.ncl.ac.uk/projects/FME/InfRes/tools/name.html
Formal Tools Database. Accessed on June 28, 1999.

[H3] http://pvs.csl.sri.com/
PVS Homepage. Accessed on June 10, 1999.

[H4] http://www.cis.upenn.edu/~hol/sunrise/
A Verified Verification Condition Generator. Accessed on June 10, 1999.

[H5] http://www.informatik.uni−ulm.de/pm/kiv/kiv.html
KIV Homepage in Ulm. Accessed on June 10, 1999.

Table of Contents

Table of Contents

1.Introduction.....6

1.1. Motivation.....6
1.2. The Environment: Theorema.....6
1.3. Overview.....6

2. Theoretical Background..... 7

2.1. Basic Notions.....7
2.1.1. The Terms Statement..... 7
2.1.2. Formal Specification..... 7
2.1.3. Correctness Statement..... 7
2.1.4. Partial and Total Correctness..... 7
2.1.5. Proving Partial Correctness.....7
2.1.6. Annotation of Programs..... 7
2.1.7. Proving Total Correctness..... 7
2.1.8. Weakest Preconditions and Predicate Transformation..... 7
2.2. Discussion of Program Verification.....7
2.2.1. Motivation for Program Verification..... 7
2.2.2. Computer Aided Program Verification..... 7
2.2.3. Program Verification versus Testing.....7
2.2.4. Benefits of Program Verification.....7
2.2.5. Limitations of Program Verification..... 7
2.2.6. Why is Program Verification not popular?..... 7

3. History and State of the Art..... 8

3.1. History.....8
3.2. State of the Art..... 8

4. The Verification System.....9

4.1. The Structure..... 9
4.2. Programs and Specifications.....9
4.3. Verification Condition Generation.....10
4.4. Proving with Theorema..... 11
4.5. Execution of Programs.....11

5. Implementation..... 13

5.1. The Environments Program and Specification..... 13
5.2. The Source Language.....16
5.2.1. Concepts..... 16
5.2.2. Syntax..... 17

5.2.3. Semantics..... 19
5.2.4. Conventions.....21
5.3. The Verification Condition Generator.....22
5.3.1. VCG with Rules for Hoare Triples..... 22
5.3.2. VCG with Rules for Weakest Preconditions.....23
5.3.3. VCG using anExtended Predicate Transformer.....23
5.4. Verification Rules..... 25
5.4.1. Assignments.....25
5.4.2. Array Assignments.....26
5.4.3. Sequences..... 28
5.4.4. Blocks..... 29
5.4.5. Conditionals..... 29
5.4.6. WHILE−Loops without Verification of Termination.....31
5.4.7. WHILE−Loops with Verification of Termination.....31
5.4.8. FOR−Loops.....33
5.4.9. Procedure Calls.....34
5.4.10. Calls of Function Procedures.....35

6. User Tutorial..... 37

6.1. How to Write a Specification.....37
6.2. How to Write a Program.....37
6.3. How to Annotate a Program.....39
6.4. How to Generate Verification Conditions.....39
6.5. How to Verify Termination.....40
6.6. How to Use Arrays..... 40
6.7. Aditional Topics..... 41
6.7.1. Transient Parameters..... 41
6.7.2. Where to Use Blocks.....41
6.7.3. Debug Output..... 42

7. Examples..... 43

7.1. Simple Example..... 43
7.2. Swap..... 43
7.3. SimpleArray..... 44
7.4. Maximum..... 44
7.5. Remainder.....45
7.6. Factorial.....XXX
7.7. Fibonacci Numbers.....47
7.8. Egyptian Multiplication.....48
7.9. Greatest Common Divisor.....49
7.10. HasElement.....50
7.11. Multiple GCD..... 51
7.12. Nested WHILE's.....52

8. Summary.....54

8.1. Goals Achieved.....54

8.2. Next Steps.....54

References..... 56