Extending a formal verifiable language

Savelyev, Y.

Award date:
2009

Extending A Formal Verifiable Language

Master's Thesis

Yuriy Savelyev, 550671
August 2009
Eindhoven University of Technology
Computer Science and Engineering
Supervisor: Michael Franssen

Contents

Chapter 1

Introduction

The systematic and reoccurring problem of low quality software systems and its consequences has been identified already several decades ago. Fundamental complexity of software as a product of an engineering process creates different opportunities for programs to fail in meeting their specification. General programming or design errors, wrong interfacing of the modules, inadequate testing, incomplete requirements and management problems are just some examples of possible, sometimes very different ways which lead to undesired results. As an achievement of the ongoing effort to make "programming" less an art and more an engineering task, software engineering now consists of many disciplines which address different stages of software creation in an organized and systematic way.

Arguably the most fundamental aspect critical for "correctness" of a program is its algorithmic part. It is this computational part which is the most useful piece of any software system. When reasoning about a program which should be implementing some algorithm, a natural question is "Does this program really do what it is supposed to do?". So on one side we have a program which performs a computation and on the other side we have a description of the requirements. Checking whether the program meets this description is usually done by means of testing. This widely used and popular procedure involves, in general, feeding the program with some systematically chosen input and comparing the actual output to the expected one. Although testing can be very structured and thorough it is a lengthy and costly procedure and does not guarantee completely that a program is error free. As E.W. Dijkstra said, testing can only show the presence of errors, not their absence.

Another approach to ensure that a program meets its description is to let the latter guide the construction of the former in a more structured way than is the case during traditional programming, when the programmer is responsible for choosing data structures and composing program statements while following his or her subjective understanding of the requirements. This approach is based on the idea that it is possible to formally derive a program from a description of the problem. In this case the description itself must have sufficient structure, so a formal specification language is used for this purpose. The underlying theory for program derivation was developed by C.A.R. Hoare [10] and further studied and refined by E.W. Dijkstra and others. The kernel of this method is Hoare logic — a formal system which combines program fragments and their specification. The programming method advocated by these authors is often called a step-wise program refinement method. The set of possible program fragments is a set of operators and statements of a basic algorithmic language which is very small yet powerful enough for most algorithmic purposes. Starting from a "main"

specification of the problem a sequence of refinement steps produces increasingly fine-grained portions of the program from the specification until no more refinement is possible. These steps involve calculations with the chosen specification language (for example first-order logic) and application of the Hoare logic rules. Because each refinement step is constrained by these formal rules, the program is guaranteed to be *correct by construction*. That is, the program meets its specification, because a proof of its correctness has been constructed at the same time.

The step-wise refinement method based on Hoare logic was taught during the Bachelor course at Eindhoven Technical University. A special supporting tool called Cocktail [8] has been developed to be used both as a teaching tool and a proof of concept. Cocktail offers functionality to define specifications in first-order logic, gradually derive programs from their specifications and construct first-order proofs through a built-in combination of automated and interactive theorem provers. While serving well as a teaching tool, we must admit that a system of this kind is of little use for real-world applications. A programmer can not be bothered by fine-grained derivation steps, rules of the Hoare logic and often even first-order logic, which is usually the paramount of his/her knowledge of formal systems. Moreover, the step-wise refinement method is not scalable, even if it has tool support.

A way to satisfy the need for formally correct programs under the limitations imposed by practice is to use Hoare logic not to derive programs but to verify their correctness afterwards, that is after a program has been constructed by the programmer. This approach requires the same main ingredient — a specification for a given program in its entirety. Additionally, the program must contain some auxiliary annotations which are sub-specifications for some of its parts. In general lines the approach works as follows:

1. The programmer inputs a complete program written in the language supported by the verification system.

2. The programmer annotates the program with its main and auxiliary specifications, according to (informal) specification of the original program.

3. The system generates a set of verification conditions from the annotated program. These are theorems in some chosen logic and capture the correspondence of the program to its specification. Accordingly, verification conditions are generated by analyzing the program and its annotations, based on the rules of Hoare logic.

4. Verification conditions are checked and conclusions about the correctness of the program are made.

As we can see, such a verification system has three major aspects:

1. A formal language which is a programming language with built-in specification facilities.

2. Verification conditions and their generation.

3. A means to check verification conditions or to pass verification conditions to be checked by some external system.

Several systems and tools based on similar ideas around verifying complete programs exist, most notably ESC/Java, Spec# and Perfect Developer.

The topic of this thesis is extension and implementation of the formal verifiable language for a verification system based on the principles described above. This thesis serves not only as my Master's thesis, but also as a manual for potential users of the tool and as a reference guide. The rest of the thesis is organized as follows: Chapter 2 describes the tool, its motivation and main ideas. The formal background of the tool is presented in the second part of this chapter. The language supported by the tool is completely defined in Chapter 3. It is this part which is meant as a manual and contains also examples of programs. Verification conditions generated by the system are explained in Chapter 4. Some implementation details are found in Chapter 5. Finally, conclusions and directions of future work end the thesis in Chapter 6.

Chapter 2

Preliminaries

This chapter contains a description of the system, its purpose and motivation for some of its main features. Formal foundations on which the system builds are reviewed and explained in a separate section which follows after that.

2.1 Tool Description

2.1.1 General Considerations

Proponents of formal methods in software engineering from the academic world and industry have created several tools which are meant to be used in practice. These include the Eiffel programming language as an early example, ESC/Java [2], Spec# [5], Perfect Developer [4] and others. When conceiving such tools two dimensions of choice play a central role: what kind of semantics and which method(s) to combine in a tool. As for semantics, denotational, operational or axiomatic are the options. Step-wise refinement, incremental development, program extraction for functional programming and others belong to the methods. A formal specification mechanism is the most precise and reliable way to express what programs are supposed to do. In the context of formal approaches for software engineering, one can let a specification somehow guide the process of program construction in addition to purely documentational and communicational role. The exact mechanics of this depends on the chosen method and properties of the specification language itself.

Let us consider a possible combination of one of semantics, a programming method and a specification language. Hoare logic ([10] and Subsection 2.2.1) is a well-known programming logic which represents axiomatic semantics. Combined with first-order logic as a specification language and step-wise refinement method, it had been taught for many years at Eindhoven Technical University. This combination has the following characteristics:

- programs are specified by formulas of first-order logic;
- Hoare-triples combine program fragments and specifications;
- programs are derived step by step from their specifications;
- derived programs are correct by construction.

Many of the formal approaches share similar shortcomings and problems either of fundamental nature, like bad algorithmic complexity in case of program extraction, or of practical nature,

like tedious and complex calculations "on paper". Most of these problems unavoidably create a need of supporting tools and are solved by such automation with varying degree of success. The step-wise refinement method as based on Hoare logic also has serious drawbacks which severely limit its applicability in industrial situations:

- derivation of programs is too much work due to calculations within predicate logic;
- it is too difficult for programmers: they have to know and understand Hoare logic;
- Hoare logic is available only for simple programming languages;
- programmers do not want to derive programs because they usually "know" the solution in advance; and finally
- even with the right tools like Cocktail (see [8]), this method is not scalable to large programs or systems.

2.1.2 Goals and Features of the System

Having identified these problems, we can formulate a wish list for a tool meant to support formal methods while at the same time being realistic enough for industrial use:

- the tool is acceptable for a programmer: easy to use and understand;
- at the same time, the tool retains enough formal machinery as foundation;
- the tool must grow and evolve to be increasingly applicable in the real world.

The system described in this master thesis was conceived with these "requirements" in mind. Below follows a list of its features which are meant to meet them and remedy most of the mentioned problems.

- The system verifies completed programs with annotation instead of supporting their derivation by the step-wise refinement method. With this, a programmer can input a complete solution which is known in advance and avoid calculations.
- The specification language is first-order predicate logic. First of all, this logic is semi-decidable and therefore useful for automating proofs. Secondly, first-order predicate logic is certainly within the programmer's bounds of reach.
- The system is based on axiomatic semantics provided by the Hoare logic.
- The language of the system evolves in a bottom-up manner starting from a simple but already useful version as described in this thesis. The ultimate goal of the system is to support a realistic, compilable programming language. The tool could then be used as an actual programming environment (a so-called Verifying IDE), instead of just as a verification tool.
- An effort is made to avoid some problems and flaws of the existing systems. For example, verification conditions (see further) generated by ESC/Java are very big formulas which do not help in finding incorrect parts of a program in case a proof attempt fails. Our system generates separate verification conditions which are associated to program

fragments from which they originate.
Proving theorems with automated theorem provers may take a long time. While working with a tool which generates verification conditions, users correct errors found in the program and let the tool regenerate the conditions. These new formulas often turn out to be almost the same as before and repeatedly proving them in the situation of absence of any "proof" memory takes even more time. This has been reported by some users of Perfect Developer as a major inconvenience and a serious problem in the workflow. Coupling with a proof repository, as explained further, will add such memory to our system.
Another important feature is soundness of the system. ESC/Java, for example, is an extended static checker and is nor sound nor complete. Also, it is not really known which logic Perfect Developer is based on.

2.1.3 Workflow and Architecture

In general lines the operation of the tool can be divided in several major steps:

1. The programmer inputs a program and annotates it by providing the main and additional specifications, according to a (informal) description of the problem.

2. The system generates a set of verification conditions from the annotated program.

3. Verification conditions are checked and conclusions about the correctness of the program are made.

These three steps imply three major aspects of the system: a formal language consisting of program and specification parts (Chapter 3), verification conditions and their generation (Chapter 4) and a means to check verification conditions or to pass them to be checked by some external system (briefly explained in this subsection). The system's architecture accommodates this workflow and is depicted in Figure 2.1. The main components grouped

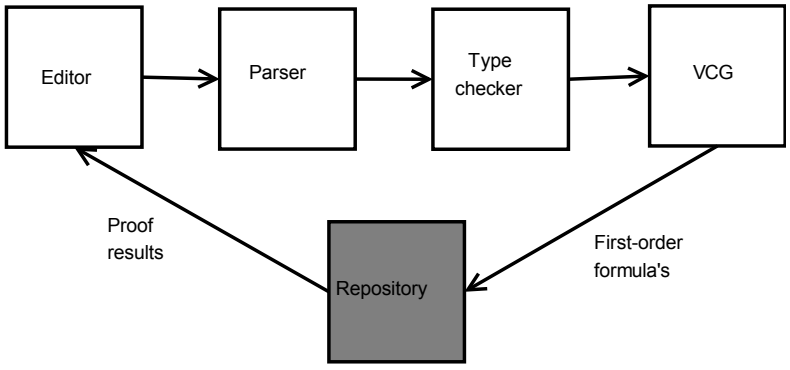

Figure 2.1: Global architecture of the system

by each step are: program editor, parser and type checker; verification condition generator (VCG); a proof repository.

The editor is meant for input of annotated programs which are then parsed (Section 5.2) and checked for usual well-formedness conditions, correct usage of scope and so on (Section 3.3). The VCG is the part of the system which analyzes the tree representation of programs and generates formulas which represent verification conditions. The main formal system of the tool – a set of rules of the Hoare logic – is implemented in the VCG. A proof repository is a work in progress (see [9]) on a system which can store theorems in some normal form, request proofs of new theorems from heterogeneous sources like automated theorem provers and proof assistants, store these proofs and retrieve/instantiate them when asked by a client. A repository can have a simple or a distributed structure and provides facilities to connect different sources to different clients acting as a translator of formats and representations of theories. However, from the tool's point of view, a repository is just a black box which can prove or reject theorems (verification conditions). The proof repository in the figure accepts the generated verification conditions and additional context generated by the tool and returns the results: if a set of generated conditions is consistent, then the program is correct with respect to its specification. Proof results are displayed for each generated condition separately.

2.2 Formal Background

2.2.1 Hoare Logic

Hoare logic is a deduction system for proving correctness of sequential programs. The central notion of this calculus is the Hoare-triple. A Hoare-triple combines a program fragment and its specification. Relating to the context of our system, we assume that specifications are first-order logic formulas. Let P,Q be specification formulas and S be a program fragment, then the following is a Hoare-triple:

$$\{P\}S\{Q\}.$$

Predicates P and Q describe the state before and after the execution of S. Of course, not every combination of these three building blocks makes sense. Such a Hoare-triple is said to be valid when the following is true:

> if the state before execution of S satisfies P, then the state after the execution of S satisfies Q.

For this reason P is a called a precondition of S and Q a postcondition respectively. However this meaning of a Hoare-triple is not precise enough because "after the execution of S" does not guarantee that S actually terminates, so a more fine-grain meaning is necessary. With respect to termination, total and partial correctness are distinguished. A Hoare-triple is valid under partial correctness when

> if the state before execution of S satisfies P and S terminates, then the state after the execution of S satisfies Q.

The case for total correctness has a stronger conclusion:

> if the state before execution of S satisfies P, then S terminates and the state after the execution of S satisfies Q.

The tool supports both partial and total correctness.

The actual derivation rules for Hoare logic are inductively defined on the possible forms of S — different kinds of program statements of some programming language. Usually it is a small programming language for which these rules can be defined easily. Guarded Command Language (GCL) has been introduced by Dijkstra [7] especially for investigation of program derivation and verification techniques. A program S is proven to be correct with respect to its specification by deriving the validity of the corresponding Hoare-triple. The validity of this Hoare-triple is derived from the validity of other Hoare-tripes which involve fragments of S, guided by the set of rules. The derivation rules for GCL are:

$$\{P\}\text{skip}\{P\} \qquad \text{(skip)}$$

$$\{Q(v := E)\}v := E\{Q\} \qquad \text{(assignment)}$$

$$\frac{\{P\}S_1\{R\} \qquad \{R\}S_1\{Q\}}{\{P\}S_1; S_2\{Q\}}, \quad \text{for some } R \qquad \text{(catenation)}$$

$$\frac{\{P \wedge B_1\}S_1\{Q\} \qquad \{P \wedge B_2\}S_2\{Q\}}{\{P\}\text{if } B_1 \to S_1 \ [] \ B_2 \to S_2 \text{ fi}\{Q\}} \quad P \Rightarrow B_1 \vee B_2 \qquad \text{(branching)}$$

$$\frac{\{P \wedge B\}S\{P\}}{\{P\}\text{do } B \to S \text{ od}\{P \wedge \neg B\}} \qquad \text{(repetition)}$$

$$\frac{P^* \Rightarrow P \qquad \{P\}S\{Q\} \qquad Q \Rightarrow Q^*}{\{P^*\}S\{Q^*\}} \qquad \text{(consequence)}$$

Repetition is the most complex statement of this simple language. It is the only statement which may make programs not terminate. Actually, the repetition rule above is for partial correctness only. The conditions of total correctness for this statement are explained in Subsection 4.2.5 and are not repeated here.

Because of the structure of these rules proofs are tree-like. Starting with a Hoare-triple for the whole program, a proof tree is built up; this process ends because program fragments inside Hoare-triples become only smaller and some rules are actually axioms. The rule of consequence is very important and must be frequently used to construct a proof tree because strengthening preconditions or weakening postconditions is sometimes the only way to proceed. However, it also makes the deduction system non-deterministic.

2.2.2 Weakest Preconditions

There is another way to construct proofs for programs. This method uses a notion of weakest precondition [7]. Proofs using weakest preconditions rely more on sequential structure of programs and are linear, in contrast to proof trees built from Hoare-triples. Our tool implements this proof technique with some modifications, as will be explained shortly.

Weakest precondition calculus is closely related to Hoare-triples. For a program fragment S and an assertion Q, the weakest precondition $wp(S, Q)$ is defined as the weakest assertion P_{wp}, such that $\{P_{wp}\}S\{Q\}$ is a valid Hoare-triple. Conversely, if $\{P\}S\{Q\}$ is a valid triple, then $P \Rightarrow wp(S, Q)$.

As with the rules of the Hoare logic, wp is defined inductively on the structure of programs, namely for every possible statement kind. In case of GCL, the repetition statement creates some trouble, because its weakest precondition in its pure form is a complex recursive expression based on the fixed-point semantics of repetition. Even worse, this weakest precondition may not be expressible in first-order logic and the connection to Hoare-triples is then lost, because the tool uses first-order logic only. A way to avoid this difficulty is to give an alternative definition of the weakest precondition for repetition and use some elements of the repetition rule of the Hoare logic to make it work. As a welcome side effect, this will lead to additional verification conditions being emitted by interesting and important parts of programs. These conditions make their sources directly recognizable. As a consequence of this approach, the system relies heavily on the notion of invariant of repetition, which captures its essence in a relatively easy to understand manner. Also, as the language contains procedure calls, wp is defined for this kind of statement as well.

Generally speaking, by relying on this alternative weakest precondition definition we avoid using Hoare logic rules at some difficult places and apply some of them to complement the weakest-precondition method. A small example with a repetition statement will illustrate this combination. Suppose we want to prove the validity of $\{P\}\text{do } G \to S \text{ od}\{Q\}$ and let the invariant of this repetition be I. Then, the following formulas capture the correctness of this small annotated program, without the need to construct a proof tree based on the Hoare's rules:

$$
\begin{aligned}
&P \Rightarrow wp(\text{do } G \to S \text{ od}, Q)\\
&I \wedge G \Rightarrow wp(S, I)\\
&I \wedge \neg G \Rightarrow Q.
\end{aligned}
$$

The connection between these formulas and the original Hoare triple is the rule of consequence. The fact that invariant I is provided by the user removes the problem of non-determinicity of this rule.

The inductive definition of wp based on the language of the system, additional verification conditions and the calculation procedure are presented in Chapter 4.

Another detail about how weakest preconditions are used in the system is worth mentioning here. If Q is a postcondition of S, then the state after execution of S should satisfy Q. However, S could be responsible only for a partial change of state. This means that S establishes only some "parts" of Q and other parts are established by other program fragments, which, obviously, precede S. For example, if the state space if defined by variables a, b and c and only a is possibly altered by S, then the validity of some sub-formulas of Q, which depend on b and c only, remains unaffected by execution of S. It is useful to have an additional calculation which splits Q in two sub-formulas based on their validity: the one altered (possibly violated) by S and the one unaltered by it. We will denote these "components" of the state satisfying Q by $Alters(S, Q)$ and $Keeps(S, Q)$ respectively. Why this calculation and splitting is useful becomes clear with the following simple example: without $Keeps$ and $Alters$ it would be necessary to prove that the part of a postcondition not established by a repetition is also not disturbed by it. This would mean that this part would have to be included in the invariant of this repetition making proof obligations (and therefore verification conditions) unnecessarily complex.

Finally, we show some properties of $Keeps$ and $Alters$ and their relation to the weakest precondition. First of all information about the state must be preserved by the split

completely: $Keeps(S,Q) \wedge Alters(S,Q) \Rightarrow Q$. But there should not be more information after the split than before: $Keeps(S,Q) \wedge Alters(S,Q) \Leftarrow Q$. Secondly, by definition $\{Keeps(S,Q)\}Q\{Keeps(S,Q)\}$ is a valid Hoare-triple. Lastly, using this and some simple properties of wp from [7] it is possible to derive that

$$\begin{aligned} & \{Keeps(S,Q) \wedge wp(S, Alters(S,Q))\}S\{Keeps(S,Q) \wedge Alters(S,Q)\} \\ \equiv & \\ & \{wp(S,Q)\}S\{Q\}. \end{aligned}$$

2.2.3 Strictly Positive Types

The language of the system allows to define and use inductive data types in programs. At the time when this feature were to be introduced in the language a natural question arose: which inductive definitions are well-behaving and should be allowed? The class of strictly positive types is well-known and used, for example, in Coq [1]. We start with a simple example of a list type. A list with elements of type A can be (somewhat loosely) defined as

$$List\ A = empty \mid A \dot{+} List\ A.$$

Here *empty* is an "immediate" element of $List\ A$ and $(\dot{+})$ is a constructor which creates lists from an element of type A and an existing list. $List\ A$ can be seen as an initial algebra with operator $empty \triangledown (\dot{+})$, where $\triangledown$ stands for "case". Then, the type of this operator is $1 + A \times List\ A \to List\ A$, where $+,\times$ and $\to$ are the usual type constructors. Because initial algebra is the smallest set, using the least fixed point operator μ we can reformulate the definition of list: $List\ A = \mu X.1 + A \times X$. This definition is strictly positive because it can be constructed from the following grammar of strictly positive types, where $\mathcal{C}$ is a constant type:

$$\mathcal{T} ::= 0 \mid 1 \mid \mathcal{T} + \mathcal{T} \mid \mathcal{T} \times \mathcal{T} \mid \mathcal{C} \to \mathcal{T} \mid X \mid \mu X.\mathcal{T}.$$

The class is called so because $\mathcal{T}$ may appear only strictly positively in function types, that is to the right of $\to$. The notion of positive occurrence is based on the type-theoretic relation between the function type and implication. For example, $A \to T$ is strictly positive with respect to T, $T \to A$ has a negative occurrence of T and $(T \to A) \to B$ has a positive occurrence of T, but not strictly positive.

The fact that strictly positive types can be described by a grammar means that strict positivity can be checked syntactically. Luckily, there is no need to implement this check in the system. Due to reuse (Section 5.1) of a particular term representation based on the Pure Type System $\lambda P-$ which requires function symbols to be applied to all arguments at once (see [8]) and because the system is first-order only, all inductive types defined within the syntax of the language are strictly positive.

Chapter 3

The Language

This chapter presents the complete language of the system. This part of the thesis is also intended as a user manual for the system, at least for its programming-related part.

The tool provides a GCL-based, non-deterministic, statically typed, non-recursive procedural language with built-in basic types, user-defined inductive types and copy semantics. The language is under development, with considerable additions and evolution steps envisioned in Chapter 6.

The language is described in a traditional way — by a context-free grammar. For the sake of structure, the grammar is partitioned in intuitively cohesive parts with some explanation, where necessary. Scope formation for every relevant entity of the language is explained after the grammar. Additional context-dependent constraints and well-formedness rules which are not captured within the grammar are formulated in a separate section. Finally, a set of examples of programs or fragments thereof concludes this chapter.

3.1 The Grammar

Prior to giving the grammar itself, some notational conventions are established first. Terminals which may appear in programs literally are underlined. Non-terminals start with a capital letter. Notation [A ;]* is equivalent to (A (; A)*)?; other separating characters instead of semicolon may occur in such expressions too. Similar abbreviation is also used for the proper repetition operator "+".

3.1.1 Programs

```
Program ::= program ID ;
            ( ADTDefinition )*
            ( { Definitions } )?
            ( ProcedureDeclaration )*
            [ var ID (, ID)* : Type ; ]+
            { Definitions? pre : Formula post : Formula Termination? }
            Statement
Termination ::= ⇓ | terminates
```

Every program has a name. A program may contain abstract data types definitions, global definitions for use in specifications and procedure declarations. Declarations of main

program variables are followed by the main program specification and program statements. A specification is enclosed in { } and consists of a precondition, a postcondition and an optional termination requirement. The token for termination is either the ASCII sequence or the Unicode arrow character entered by Alt+t as above; both forms are equivalent.

3.1.2 Abstract Data Type Definitions

```
ADTDefinition ::=  datatype ID is
                   Constructor ( [] Constructor )*
   Constructor ::=  ID ( [ ID : Type , ]* )
```

An inductive data-type in a program is identified by its name. Every data-type is defined by a set of constructors in which formal parameters may be of the type which is being defined. Symbol [] is used in the language as a separator of the "alternatives", in this case different constructors of an inductive data type.

3.1.3 Definitions For Specifications

```
          Definitions ::=  define Definition ( , Definition )*
           Definition ::=  ID ( [ ID : Type , ]* ) ( : Type )?
                           as DefinitionBody
       DefinitionBody ::=  Formula
       DefinitionBody ::=  match Formula with
                           MatchGExpression ( [] MatchGExpression )* .
  MatchGExpression ::=  MatchPattern –> Formula
        MatchPattern ::=  ID ( [ID , ]* )
```

Specification definitions may be non-recursive and recursive. Recursive functions are defined by means of a match expression. Every defined function has a type. If this type is omitted in the definition, it is assumed to be `bool`, i.e. the defined name is a predicate.

The arrow between a match pattern and a formula may be either the ASCII sequence `->` or the Unicode arrow character entered as Alt+-.

3.1.4 Procedure Declarations

```
ProcedureDeclaration ::=  procedure ID ( [ ProcedurePar , ]* )
                          { Definitions? pre : Formula post : Formula Termination? }
                          Statement
        ProcedurePar ::=  var? ID : Type
         Termination ::=  ⇓ | terminates
```

Procedures have names, parameters and their own specifications. Parameters can be marked to be Pascal-like variable parameters. Otherwise, parameters may be considered as local variables initialized with the argument value at the time of a procedure call.

3.1.5 Program Statements

```
          Statement ::= skip
          Statement ::= ID (, ID)* := Formula ( , Formula )*
          Statement ::= ID ( [ Formula , ]* )
          Statement ::= if GuardedStatement ( [] GuardedStatement )* fi
          Statement ::= { Definitions? inv : Formula ( dec : Formula )? }
                        do GuardedStatement ( [] GuardedStatement )* od
          Statement ::= match ID with
                        MatchGStatement ( [] MatchGStatement )* .
          Statement ::= Statements
GuardedStatement ::= Formula –> Statement
MatchGStatement ::= MatchPattern –> Statement
     Statements ::= begin
                    ( ( var ID (, ID)* : Type ; )+ )?
                    Statement ( ; Statement )*
                    end
```

Statements of the language are: a skip statement, multiple assignment (:=), procedure call, branching (if...fi) and repetition (do..od) statements with multiple alternatives, a pattern matching statement (match...with) and a block of statements. The repetition statement has its own specification which contains at least an invariant. Blocks are used to combine several statements and may contain auxiliary local variable declarations.

3.1.6 Types and Formulas

```
Type ::= int
Type ::= bool
Type ::= ID
```

Built-in types are integers (int) and booleans (bool). A type may also be one of the user-defined inductive data types.

```
Formula ::= ( ∀ | ∃ ) ID : Type . Formula
Formula ::= ( Formula )
Formula ::= true | false
Formula ::= integer number
Formula ::= ID
Formula ::= ID ( [ Formula , ]* )
Formula ::= expression
```

Valid formulas of the language are logical and arithmetic formulas. Natural numbers, variables and parametric constant (function) applications are also valid formulas. Parenthesis may be used to group parts of a formula and, as always, this is recommended. Expression in the last rule consists of formulas and the following operators grouped by increasing precedence:

$$\begin{array}{c} < \leq \geq > = \\ \hline \Rightarrow \\ \hline \wedge \vee \\ \hline + - \\ \hline * \\ \hline \neg \ -_{unary} \end{array}$$

The input of special operator symbols used in formulas is shown in the table.

$\forall$	Alt+a	$\exists$	Alt+e
$\leq$	<=	$\geq$	>=
$\wedge$	Alt+n	$\vee$	Alt+v
$\Rightarrow$	Alt+=	$\neg$	Alt+~

3.2 Scope Formation

3.2.1 Main Program Variables

The scope of each variable declared in the main program consists of:

1. the pre- and postcondition formulas of the main program specification,
2. the invariant and decrement formulas of repetition statements in the main program,
3. statements of the main program.

3.2.2 Abstract Data Type Definitions

These definitions bind user-defined inductive types to their names and provide constructors of the defined types. The scope of a name T and corresponding constructors is the whole program. Only data type definitions provide symbols which may be used both in specification formulas and program statements. Subsection 3.3.2 contains additional explanation about the use of T inside data type definitions themselves.

3.2.3 Definitions for Specifications

A sequence of definitions may appear at the beginning of a program ("global" definitions) or accompany a specification (see grammar). All defined symbols are subsequently used as functions. A predicate is considered to be a function which returns a value of type bool.

The general abstract form of a sequence of definitions is $f_1(P_1) : T_1 = E_1, \ldots, f_k(P_k) : T_k = E_k$, where f_i is a function name, P_i is a possibly empty set of parameters which may be used in E_i, T_i is a type and E_i is an expression. The scope of a non-recursive function symbol f_j from the sequence extends to:

1. expressions $E_{j+1}, \ldots, E_k$,
2. if f_j is among definitions accompanying a program or procedure specification, the pre- and postcondition formulas of this program or procedure and the invariant and decrement formulas of all repetition statements inside this program or procedure,

3. if f_j is among definitions accompanying a repetition statement specification, the invariant and decrement formulas of this repetition statement,

4. if f_j is defined globally, all specification formulas and definition expressions of the rest of the program.

A recursive function f_j has the same scope, but extended with E_j — the expression for this function itself, in order to allow recursion. This expression must be a pattern matching expression (match V with ...) and contains a number of $Pattern \rightarrow E$ pairs. $Pattern$ has a form of a constructor applied to some dummy arguments and E is an expression. The variable being matched (V) and the dummies may be used in E. Having said that, the k-th dummy in the pattern has the type of the k-th formal parameter of the constructor.

In conclusion, the main rule of thumb regarding definitions is that

> predicates and functions defined in the specification part of the language can not be used in program statements as guards or expressions.

3.2.4 Procedure Declarations

A declaration of procedure named $Proc$ with parameter a fixes the scope of these symbols. $Proc$ may be used in procedure calls inside of procedures declared after $Proc$ and in the main program. This means that direct or indirect procedural recursion is not possible. The scope of parameter a is:

1. the pre- and postcondition formulas of $Proc$,

2. the invariant and decrement formula of repetition statements inside $Proc$,

3. statements of $Proc$.

3.2.5 Blocks

If a block declares an auxiliary local variable v, it may be used inside this block only. If a nested block declares a variable with the same name, than v is shadowed by this inner variable.

3.3 Well-formedness Rules

3.3.1 Programs

At the high level of program structure, the following conditions must hold:

- every procedure has a unique name;
- every procedure must be declared before it can be used in the main program body;
- every variable must be declared before it can be used in the main program body;
- if a variable of inductive type T is declared, the program must contain a definition of data type T;
- the specification is well-formed (Subsection 3.3.4).

Furthermore, programs have the following properties:

- in the section with variable declarations, if a variable name is reused the corresponding variable will have the last associated type: var a:int; var a:bool ⊢ a:bool;
- the termination requirement inside the specification of the main program is optional.

3.3.2 Abstract Data Type Definitions

For a definition of an inductive data type T to be valid, the following conditions must hold:

- T is identified by a unique name not used by any other data type;
- every constructor of T has a unique name not used for any other constructor of any type;
- every parameter of every constructor of T is either of a built-in type, a previously defined type or of type T itself;
- T has at least one constructor which does not depend on T itself. This ensures the existence of finite terms over the signature of T.

3.3.3 Definitions for Specifications

For convenience, the general abstract form of a sequence of definitions is repeated here. In $f_1(P_1) : T_1 = E_1, \ldots, f_k(P_k) : T_k = E_k$, f_i is a auxiliary specification function name, P_i is a possibly empty set of parameters which may be used in E_i, T_i is a type and E_i is an expression. The well-formedness conditions for a non-recursive definition at position j in a sequence of definitions are:

- $f_j \neq f_i$, for $0 \leq i < j$;
- type T_j is either a built-in type or a previously defined data type;
- the type of expression E_j is T_j;
- expression E_j contains no free references with respect to the scope.

Recursive definitions always involve a pattern matching expression. According to the grammar, the body of such a definition is of the following form:

match V with
$Pattern_1 \rightarrow E_{j,1}$
[] $Pattern_2 \rightarrow E_{j,2}$
. . . .

Then, a recursive definition at position j is well-formed when the following conditions hold:

- $V \in P_j$ and $V : T$, where T is a previously defined inductive data type. This means that function f_j must depend on V;

- the match expression contains as many alternatives as the number of constructors of T. All possible decompositions of a member of an inductive type must be dealt with;
- for all l, $Pattern_l$ has a form of a function application with the function being the constructor of T at the l-th position in the definition of this data type;
- for all l, all parameters of the constructor in $Pattern_l$ are fresh names and have a form of a variable;
- for all l, the type of every expression $E_{j,l}$ is T_j;
- expression $E_{j,l}$ contains no free references with respect to the scope.

Note, that well-formedness of a recursive definition does not imply that it is also well-defined.

3.3.4 Program and Procedure Specifications

Specifications of programs and procedures are similar in structure. The general scheme of a specification is as follows:

{ D pre: F_1 post: F_2 T },

where D is a possibly empty sequence of definitions, F_1 and F_2 are formulas and T is an optional termination requirement. The latter is described in the well-formedness rules for procedure declarations (Subsection 3.3.5). A specification is well-formed when the following conditions are met:

- D is a well-formed sequence of definitions;
- F_1 and F_2 are first-order logic formulas;
- F_1 and F_2 contain no free references with respect to the scope.

The system offers an additional possibility related to procedure parameters. For every procedure parameter a in scope, an auxiliary specification variable a' is also automatically in scope. This variable represents the initial value of a at the moment of the procedure call. These variables can be useful for some specifications. For example, the following procedure uses multiple assignment to swap the values of two variables:

```
procedure swap(var a:int, var b:int)
{pre: true post: a=b' ∧ b=a'}
begin
 a,b:=b,a
end
```

3.3.5 Procedure Declarations

A declaration of procedure P is well formed under the following conditions:

- all parameters of P have distinct names;
- the specification of P is well formed;

- if P is called by the main program or a procedure P', which has a termination requirement in its specification, then P also has a termination requirement in its specification. Abstracting from the definition of the procedure, if the calling program or procedure is required to terminate, then termination of every procedure call must be guaranteed. The termination requirement in the specification of the called procedure represents this guarantee, from its user's point of view;
- statements of P do not use variables declared for the main program. This can be concluded from the scope formation rules;
- statements of P do not include a call of P. This no-recursion rule is also enforced by scope formation rules.

3.3.6 Program Statements

Every program statement, except for the skip statement, has its own restrictions. All of them are covered in this subsection for each statement separately.
A **multiple assignment** is well-formed when:

- the left-hand side contains only variables which are in the current scope;
- the number of variables on the left-hand side is the same as the number of expressions on the right-hand side;
- the type of the i-th variable on the left-hand side is the same as the type of the i-th expression on the right-hand side.

A **procedure call** is well-formed when:

- the procedure being called has been declared before the call;
- the number of actual parameters of the call is the same as the number of formal parameters of the procedure;
- the type of the i-th actual parameter is the same as the type of the i-th formal parameter of the procedure;
- if some j-th formal parameter of the procedure is a variable parameter, then the j-th actual parameter must be a variable of the same type and in the current scope.

A **branching statement** is well-formed when every of its guarded statements is well-formed. A guarded statement is considered well-formed when the following conditions hold:

- the guard is a formula of proposition logic involving variables or procedure parameters in the current scope;
- the statement is well-formed.

The **repetition statement** is arguably the most complex one because it has its own specification. Its most important part is the invariant of repetition, I. A repetition statement is well-formed when every of its guarded statements is well-formed as described above and when the following additional conditions are satisfied:

- if the specification of the statement contains a sequence of definitions, then they are well-formed;
- the invariant I of the specification is a first-order logic formula;
- I does not contain free references with respect to the scope;
- if provided, the type of the decrement function is int;
- if the specification of the enclosing procedure or program contains the termination requirement, then the decrement function is provided. Ultimately, termination depends only on repetition statements which may be written to loop indefinitely. A decrement function is used to describe the number of repetitions made. This is elaborated in detail in Subsection 4.2.5.

The **pattern matching statement** is in many ways similar to the pattern matching expression used to define recursive functions for specifications. The difference is that in the program part, as opposed to specification part, not expressions, but statements follow the patterns. Concerning the restrictions on the variable being matched with and on alternatives and their match patterns, refer to Subsection 3.3.3. Similarly to the scoping rules for the pattern matching expression, every alternative statement may use dummies from the corresponding pattern as if those where local variables. These variables exist between "their" match pattern and the following one or the end of the whole pattern matching statement. The copy semantics of the language allows for assignments to these variables if necessary, without changing the original term.

3.4 Examples

3.4.1 A Simple Program

```
program sum;

var a,b,c:int

{pre: true post: c=a+b}
begin
 c:=a+b
end
```

This simple program demonstrates several elements which are always present in any program: program name, declaration of program variables, the main program specification and the main program body. The three variables are of the built-in type int. The specification contains a precondition and a postcondition with obvious meanings. The only statement of the program establishes the postcondition. Note, that according to the grammar, the whole section starting with begin up to and including end is a statement. Nevertheless, in this example we may consider the assignment as the "only" meaningful statement which affects verification conditions of this program. Another example will illustrate a more sophisticated character of blocks, namely when a local variable comes in play.

3.4.2 Integer Division

```
program intdiv;

procedure div(var q:int,var r:int,A:int,B:int)
{define P(a:int,b:int,c:int,d:int) as a=b*c+d
 pre: 0<A ∧ 0<B post: P(A,q,B,r) ∧ (r<B)}
begin
  q,r:=0,A;
  { inv: P(A,q,B,r)}
  do r>B -> q,r:=q+1,r-B od
end

var a,b:int

{pre: true post: a=3 ∧ b=3}
begin
  div(a,b,15,4)
end
```

This program contains one procedure declaration, a declaration of two main program variables a and b and a sole procedure call in the main program body. Procedure div has two variable parameters q and r and two value parameters A and B. The specification of the procedure contains one auxiliary definition of predicate P. The point of defining it is that P by used both in the specification of div as the postcondition and as the invariant of the repetition because those are the same expression, namely $A = q * B + r$. This makes the program more readable and the annotations more modular. Note, that according to scope formation and well-formedness rules, parameters of div may not be used in the definition of P but may be used in the postcondition formula of the procedure.

3.4.3 Defining Lists and Sums

Let us make a more interesting program. We start by defining a data type List, which represents possibly empty sequences of integer numbers. As usual, such a list can be defined inductively by two constructors. In the language of the tool this definition looks like

```
datatype List is empty() [] cons(x:int,l:List)
```

with `[]` separating the two alternatives. Constructor empty immediately "creates" a list with no elements and constructor cons depends on the Type list itself, because it creates a new list from a number and an existing list. So, empty() is a list, cons(1,cons(2,empty())) is a list and so on.

Suppose that we want to write a program, which returns the sum of all elements in a given list. For the sake of specification, one has to define the sum of elements in the list first. In our tool, it is possible to do so with the help of recursive specification functions. The following is a global definition of a recursive function which defines the sum:

```
{define
 sum(l:List):int as
```

```
 match l with
    empty() -> 0
 [] cons(h,tail) -> h+sum(tail).
}
```

The type of the function is stated explicitly; leaving it out would result in a type error, because the alternative expressions are not of default type bool. Inductive data type List has two constructors, therefore the pattern matching expression with variable $l : \text{List}$ has all these constructors in the same order. Note, that in the match pattern with cons two dummies h and *tail* are introduced and used in the expression which follows.

3.4.4 Programming with Lists and Sums

We now proceed with the program itself:

```
program listsum;

datatype List is empty() [] cons(x:int,l:List)

{define
 sum(l:List):int as
 match l with
    empty() -> 0
 [] cons(h,tail) -> h+sum(tail).
}

var m:List;
var s:int

{pre: true post: s=sum(m)}
begin
 m:=cons(1,cons(2,cons(3,empty())));
 s:=0;
 begin var k:List;
  k:=m;
  {inv: s+sum(k)=sum(m)}
  do ¬ (k=empty()) ->
    match k with
       empty() -> skip
    [] cons(x,t) -> s,k:=s+x,t .
  od
 end
end
```

The postcondition of the specification says that the program implements the (abstract) function sum. Function sum can be used in the specification as-is because it is defined globally. After the execution of the program variable s will hold the sum of the elements of list m, which is initialized to the example value. The program also contains a block with a local

variable k which is needed for the invariant of the repetition. The assignment $k{:=}m$ means that the whole list in m is copied and this copy is bound to k. Therefore, assignments to k will not affect the original list in m, otherwise the invariant would be useless. The repetition shortens the list by removing its first element while at the same time updating the sum. Although nothing has to be done for empty lists, constructor empty() still must be present in the pattern matching statement so the skip statement proves is usefulness here. After list k becomes empty, the invariant establishes the postcondition of the program.

To prove termination one needs to define a function which returns the length of a list and provide an expression involving this function in the specification of the repetition. In our example list k, if non-empty, becomes shorter with every cycle of the repetition, so expression $length(k)$ would be a suitable decrement function.

Chapter 4

Verification Conditions

Generating verification conditions from annotated programs is the main task of the system. In this chapter both the procedure of generating verification conditions and the conditions themselves are explained. Here, under the procedure we mean how exactly a program with its structure and different language constructs results in different conditions which capture correctness of the program. How this procedure is implemented will be described in a later section.

First, the program-level general scheme of the generating procedure is described. After that, for every possible language statement it is explained, with motivation, how it participates in the procedure. Calculating weakest preconditions is the major activity of the generating algorithm. The alternative inductive definition of wp mentioned in Subsection 2.2.2 is given along with the discussion of statements. The possibility to define auxiliary functions for specifications adds another aspect to the whole matter: how these definitions influence the generated verification conditions. A separate section is devoted to this matter.

4.1 Programs and Verification Conditions

4.1.1 Programs as Sequences

A program which is put into the system consists of two parts: a specification and program statements. Specifications of programs are composite and divided between the main program itself, its procedures and its repetition statements. Additionally, auxiliary definitions may be present. However, when considering a program as a whole (and after some heavy abstraction), only two components of its specification are really essential: the pre- and postcondition.

Thus, in a very abstract form, a program S, its precondition P and postcondition Q combine, from the system's point of view, in

$$S : P \triangleright Q.$$

This notation is equivalent to the Hoare-triple

$$\{P\}S\{Q\}.$$

From the components of this combination the system will generate the following verification condition:

$$P \Rightarrow wp(S, Q),$$

where $wp(S, Q)$ is the weakest precondition of program S, given its postcondition Q. This is in direct correspondence with the rules of Hoare logic.

Both P and Q are supplied by the user, so the real work for the system is to calculate $wp(S, Q)$. This calculation depends on the structure of S: we must make S less abstract in order to make this dependance clear. All "interesting" non-trivial programs are sequences of program statements. In our language, as in many others, programs (statements) can be combined into bigger programs with a ";" (catenation) operator. Under this concretization $S : P \rhd Q$ becomes

$$S_1; S_2; \ldots; S_n : P \rhd Q,$$

with S_i being program statements, at the lowest level of structure. The composition rule of the Hoare logic states that the last combination is equivalent to

$$S_1; S_2; \ldots; S_{n-1} : P \rhd wp(S_n, Q).$$

By repeatedly calculating the weakest precondition of the last program statement and moving backward in this way, the same verification condition as for the abstract S will be generated. Clearly, we have to consider exactly which statement each S_i is, to be able to calculate its weakest precondition. The is done in the next section (Section 4.2).

4.1.2 Programs with Procedures

Beside the compositional nature of programs, the language allows another structural element of S: procedures. Procedures are first declared and then (possibly) called. The latter is a kind of program statement, so we will take a look at the former. By considering S to be only the main program, one can augment it with k procedure declarations:

$$Proc_1, \ldots, Proc_k, S : P \rhd Q.$$

A procedure has its own specification with a pre- and postcondition and a body. A procedure body is essentially a program with the procedure's specification as its own. This makes it possible to compute a verification condition for each procedure in the program in exactly the same way as in the previous subsection. The verification conditions of each procedure are independent of those of other procedures or the main program.

Summarizing, for a program of the form as above with procedure $Proc_j$ of the form $S_j : P_j \rhd Q_j$, the system will generate $k + 1$ verification conditions, one for the main program and k for its procedures, regardless of whether they are called or not:

$$\begin{gathered} P \Rightarrow wp(S, Q) \\ \ldots \\ P_j \Rightarrow wp(S_j, Q_j) \\ \ldots \end{gathered}$$

It is important to note, that when generated, these verification conditions are represented by separate formulas to comply with the requirement about correspondence of program fragments and proof obligations imposed by them. For ease of identification, the generated formulas will be marked with the names of the respective procedures or the program name.

4.2 Statements and Verification Conditions

Program statements participate in verification conditions generation in two ways. First, the calculation of the weakest precondition, while moving backward through the sequence of statements, differs for each kind of statement. Second, some statements of the language emit additional verification conditions. As these additional proof obligations are directly associated to these statements, they are output as separate formulas too.

Several subsections that follow deal with each kind of statements of the language, their weakest preconditions and, where applicable, additional verification conditions.

4.2.1 The Skip Statement

The skip statement does not change the state of the program. Therefore, for all Q:

$$wp(\mathsf{skip}, Q) = Q.$$

4.2.2 Multiple Assignment

Multiple assignment directly manipulates the state of the program. Using the assignment axiom of Hoare logic we get, for all Q:

$$wp(\Theta, Q) = \Theta(Q),$$

where $\Theta = v_1, \ldots, v_n := E_1, \ldots, E_n$. This means that the weakest precondition of a multiple assignment is its postcondition after application of the substitution as defined in the assignment.

4.2.3 Procedure Calls

Procedure calls in our language are not just simple macro expansions. For one thing, procedures have their own preconditions, so a procedure call can not be placed just anywhere in the program if one expects the program to be correct. Also, procedures have parameters, more importantly Pascal-like variable parameters. Procedures can not use global variables directly (by design), so variable parameters is the only mechanism of the language by which procedure calls can change the state of the program.

Verification conditions for procedure calls capture these two aspects: whether a procedure can be called at a given place and what does this call mean for the rest of the program which comes after it. Let $Proc$ be a procedure with specification $P^* \rhd Q^*$. At some point, after having repeatedly calculated the weakest preconditions of program statements, the algorithm will reach the call of $Proc$:

$\ldots$
call $Proc$
$\{Q\}$
$\ldots$

Operationally seen, if $Proc$ is an interesting procedure, i.e. has variable parameters, then depending on which actual parameters – variables in current scope – are passed to this call, the state after the call will change. The state after the call should satisfy Q, however this

formula does not necessarily depend only on the variables passed to $Proc$ during the call. Therefore, it makes sense to calculate $Keeps$ and $Alters$ for Q and use them accordingly.

Given this, the weakest precondition of the call to $Proc$ can be defined and motivated. Operationally, when control flow enters the procedure body its precondition P^* must hold. Additionally, by definition $Keeps(\text{call } Proc, Q)$ is not affected by *this particular* procedure call, so it can and must be passed on along the chain of statements. Therefore we arrive at the following definition:

$$wp(\text{call } Proc, Q) = Keeps(\text{call } Proc, Q) \wedge P^*(\mathcal{S}),$$

where $\mathcal{S}$ is a substitution of references to formal parameters of $Proc$ by the actual parameters of the call. It is important to remind here, that the verification condition concerning the consistency of P^* (and for that matter Q^*) with the body of $Proc$ (Section 4.1.2) is separated from the calls to this procedure.

We now turn the the second aspect of procedure calls: program state after the call. The state after the call should satisfy Q, however Q has been computed independently of $Proc$ or its particular call because of the bottom-up nature of the algorithm. Thus, in order to connect the two ends an additional verification condition is emitted for each procedure call. This is only necessary if $Q \neq Keeps(\text{call } Proc, Q)$, i.e. this verification condition will not be generated in case when the call does not alter the state at all. Note that the entire procedure call is then unnecessary and useless, unless intended so by the programmer. Operationally seen, when control flow has just passed the body of $Proc$ its postcondition Q^* must hold. Intuitively, Q^* resulting from *this particular* call of $Proc$ must be stronger than $Alters(\text{call } Proc, Q)$. Finally, it is possible and necessary to strengthen the antecedent with the part of Q not violated by the call. We arrive at the following verification condition for the call of $Proc$:

$$Keeps(\text{call } Proc, Q) \wedge Q^*(\mathcal{S}) \Rightarrow Alters(\text{call } Proc, Q),$$

where $\mathcal{S}$ is the aforementioned substitution. The respective formulas generated by the system will be marked as procedure calls, their identifying names including.

4.2.4 The Branching Statement

Branching statements do not emit any additional verification conditions and participate only in the calculation of weakest preconditions. Weakest precondition for this statement is implemented exactly as defined in [12]. Nevertheless, its explanation is repeated here for completeness.

The following situation, in which the weakest precondition must be computed, is assumed:

...
if $G_1 \rightarrow S_1$ [] $G_2 \rightarrow S_2$ [] ... fi (abbr. S_{if})
$\{Q\}$

The weakest precondition of the branching statement S_{if} is then defined as

$$wp(S_{\text{if}}, Q) = (G_1 \vee G_2 \vee \ldots) \wedge (G_1 \Rightarrow wp(S_1, Q)) \wedge (G_2 \Rightarrow wp(S_2, Q)) \wedge \ldots.$$

4.2.5 The Repetition Statement

A repetition statements has an attached specification: its invariant and possibly a decrement function, if termination is required. These elements produce additional characteristic proof obligations: invariance, finalization and termination, if required. A repetition statement may or may not touch variables in current scope. For this reason auxiliary calculations using $Keeps$ and $Alters$ are used here too.

We assume the following situation:

...
{inv: I dec: D}
do $G_1 \to S_1$ [] $G_2 \to S_2$ [] ... od (abbr. S_{do})
$\{Q\}$

By definition, the invariant of repetition must hold before the very first repetition step. Additionally, a portion of the state (description) not modified by S_{do} must be passed further "up" to be established by other statements. This suggests the following weakest precondition:

$$wp(S_{\text{do}}, Q) = Keeps(S_{\text{do}}, Q) \wedge I.$$

Invariance. For each alternative guarded command the system will generate one verification condition. Let $G_j \to S_j$ be a guarded command of S_{do}. The complete verification condition for invariance of I for the j-th guarded statement is

$$Keeps(S_{\text{do}}, Q) \wedge I \wedge G_j \Rightarrow wp(S_j, I).$$

Finalization. When control flow exits the repetition completely, the resulting state must satisfy Q. The verification condition for finalization captures exactly this:

$$Keeps(S_{\text{do}}, Q) \wedge I \wedge (\neg G_1 \wedge \neg G_2 \wedge \ldots) \Rightarrow Alters(S_{\text{do}}, Q).$$

Termination. The verification condition for termination is based on the usual idea of a decrement function bound from below which "counts" each repetition cycle. Clearly, the cycles can not be counted infinitely because of the bound: the repetition terminates as result. In our example this function is given by expression D.

The idea to be expressed in the verification condition is that every execution of any alternative statement S_j of S_{do} contributes to the decrease of D. Consequently, for each guarded statement one separate verification condition formula is generated. In terms of Hoare-triples, $\{D = C\}S_j\{D < C\}$ must be valid, for some logical constant C. This can be expressed through the weakest precondition of S_j. The full verification condition of termination for the j-th guarded statement of S_{do} is

$$Keeps(S_{\text{do}}, Q) \wedge I \wedge G_j \Rightarrow \big(wp(S_j, D < C)\big)(C := D),$$

with a fresh C.

The decrease of D by itself is not enough because the actual bound must be ensured too. Therefore, the whole repetition statement emits also a second component of the termination condition:

$$Keeps(S_{\text{do}}, Q) \wedge I \wedge (G_1 \vee G_2 \vee \ldots) \Rightarrow 0 \leq D.$$

These verification conditions are only output when a decrement function is included in the specification of the repetition.

4.2.6 Blocks

Simple blocks which do not contain local variables just group several program statements together. Therefore, the weakest precondition of a simple block is defined as the weakest precondition of its sequence of statements:

$$wp(\mathsf{begin}\ S_1, S_2, \ldots, S_k\ \mathsf{end}, Q) = wp(S_1, S_2, \ldots, S_k, Q).$$

Blocks may also include local variables which exist only inside of the defining block. The language allows to declare more than one local variable inside of a block. We will discuss cases with only one variable, as they are easily generalized. A local variable does not matter outside of the block and its (initial) value inside the block is not known. Therefore, the weakest precondition is the universal quantification over the local variable:

$$wp(\mathsf{begin\ var}\ v : T;\ S\ \mathsf{end}, Q) = \forall v : T.wp(S, Q).$$

Note, that the weakest precondition of simple blocks is a special case of this formula.

4.2.7 The Pattern Matching Statement

Pattern matching statements produce only weakest preconditions and do not emit additional verification conditions. An example situation with two alternatives is

...
match V with
$cons_1(a, b) \to S_1$
[] $cons_2(c, d) \to S_2$. (abbr. S_{match})
$\{Q\}$

Due to no junk property of the inductive types in the system and given the well-formedness rules imposed on the match statement, exactly one alternative will be chosen (of course, operationally V must be initialized). This means, that the alternative "guarded" statements can be viewed in isolation. Concerning the dummies used in the constructors, S_1, for example, may be considered to be inside of a block with local variables a and b. Thus, the weakest precondition of the entire statement is based on that for blocks:

$$\begin{aligned} wp(S_{\text{match}}, Q) = &(\forall a : T_a, b : T_b.V = cons_1(a, b) \Rightarrow wp(S_1, Q)) \\ &\wedge (\forall c : T_c, d : T_d.V = cons_2(c, d) \Rightarrow wp(S_2, Q)). \end{aligned}$$

4.3 Verification Conditions and Definitions

Definitions in the specification part of a program are obviously meant to be used in specification formulas. While recursive functions are the only way to specify some problems, simple macro-like definitions are mainly meant to simplify the formulas for the user by shortening large ones and avoiding to repeat almost the same formulas. Consequently, the system prints all generated verification conditions with the defined functions used in them. From the point of view of a system which processes these conditions (for example the proof repository), these simple definitions could be expanded prior to passing the formulas on. However the tool follows the opposite way and passes the generated formulas as they are — including auxiliary

functions. This means that auxiliary functions must be incorporated in the formulas. This is not the case with global definitions because they are passed on in a global context, which acts as the prelude and contains some standard language operators like + and < which are "predefined" from the users perspective.

Combining definitions and formulas is possible with a special form of formulas: if F is a formula and f is a definition of an auxiliary function, then $(f\ in\ F)$ is also a formula. Which functions can be combined (and used) in F depends on the current scope, as defined earlier in Section 3.2. In this section everything in scope except for auxiliary functions is ignored. Several formulas can share a context which consists of all symbols in scope. For example, pre- and postconditions from the same specification always share a context. A repetition invariant extends this context with its own definitions. Moreover, functions in the context may depend on other functions, so F can also stand for auxiliary functions. If several formulas share a context, each individual formula does not necessarily depend on all functions in the context; this has consequences for the generated formulas.

If F is (a sub-formula of) a verification condition with a user-supplied context $\Gamma = f_1, f_2, \ldots, f_k$, globally defined functions not counting, the system will generate a formula $(f_i\ in\ (\ldots\ in\ (f_j\ in\ F)))$, where $f_i, \ldots, f_j$ is a valid sub-context of Γ. The main property of this sub-context is that if some $f^* \in \Gamma$ is not included, then no element of $\{f_i, \ldots, f_j, F\}$ depends on f^*, directly or indirectly. This sub-context is minimal under preservation of this property, that is, unused auxiliary functions are not incorporated in the formula. The following somewhat artificial example of a pseudo-procedure illustrates this:

```
procedure loop();
{define X as A, Y as B
 pre: true post: Y}

 begin
  {define Z as X
   inv:Z}
  do G -> ... od
 end
```

The context of the pre- and postcondition formulas is X, Y; the context of the invariant formula is X, Y, Z with Z depending on X. Ignoring the $Keeps$ for the repetition, the following verification condition for the invariance will be generated:

$$(X = A\ in\ ((Z = X\ in\ Z) \wedge G \Rightarrow (Z = X\ in\ Z))).$$

Clearly, the invariant formula uses only two out of three functions in its context. Therefore, only these two are included: Z as the invariant itself and X because Z depends on X.

Chapter 5

Implementation

This chapter is a discussion of some implementation details about the tool. In particular term representation, parsing, type checking and the verification conditions generator are explained in separate sections.

The system builds upon readily available portions of Cocktail's code base by reusing and extending a lot of classes. Because of the different approach and totally new workflow, the user interface is completely new and not reused. As Cocktail is written in Java, this tool is obviously too.

5.1 Representation of Terms

A particular term representation had been extensively implemented in Cocktail and is reused in this tool. In Cocktail a signature for the Pure Type System (PTS) $\lambda P-$ was implemented and extended to also represent programs. Basic building blocks of this signature are terms (class Term) and names (class Name). Class Term has many sub-classes which represent more complex terms of the PTS. An important composed class is Item which represents the pair (Name, Term). The signature has facilities to represent terms with bound references and manipulate them by copying, substitutions, additional calculations and so on. First-order logic, needed for specifications and verification conditions, can be encoded by the implemented PTS and its formulas are expressible within the signature of Cocktail.

In the system everything, program statements and specifications, auxiliary specification functions, inductive types, is represented as trees. The nodes of these trees are (descendants of) Terms and Items which are, in turn, descendants of class Node — a general class for tree representation, traversal and annotation. Class Node is designed to work with the visitor pattern. A program and almost everything syntactically within it is represented as one tree, built by the parser. Because the signature explicitly supports bound variables, the tree has many nodes which contain back-references to nodes closer to the root. For example, a BindingTerm contains an Item, which may represent a bound variable, and a Term, which in turn can have a RefTerm child pointing to the Item. In programs every reference to a variable is represented by such a pointer to its declaration and so on. An elaborate explanation of this feature can be found in [8].

A special class outside the Node/Term hierarchy called ItemCollection is used to store contexts as a sequence of Items. There is a separate global context which contains built-in operators of the language and the theory (a set of axioms about the operators and built-in

types). Globally defined specification functions also reside in this context and not in the program tree. This preamble is passed on to the repository along with generated verification conditions.

Through inheritance, every program statement has its own representing term. Most of the corresponding classes are in package cocktail.gcl: GCL_MAssign, GCL_PCall, GCL_Do, GCL_If and some others. Following a tradition of Cocktail, a variant of the factory pattern is used to hide the details of creation of these and other classes. This is useful in producers of composite terms (tree nodes) like the parser and verification condition generator: these parts of the program do not have to be updated in case the signature is changed. The main factory class is GCLCreator, which inherits from other factories from Cocktail.

Specifications are stored in a special node Term_Spec which contains pre- and postconditions and optional termination requirements. This node is also a part of the program tree.

Signature for inductive definitions and the pattern matching statement consists of classes in package cocktail.gcl.inductive. Every data type definition is stored in the node of type Item_ADT which is a pair (data type name, Term_Constructors). Term_Constructors is a sequence of Item_Constructors. The latter extends a class which represents parametric constants from the PTS. Both pattern matching statement and pattern matching expression used for definitions of recursive specification functions are represented by GCL_Match. The former has program statements as subtrees and the latter expressions. Match patterns are parametric constant application nodes provided by the existing signature of Cocktail. These nodes combine (references to) parametric constants with (references to) arguments and represent actual instantiations of parametric constants.

For user-defined specification functions the system provides the Item_CDef class which is a pair (parametric constant, term). Pre- and postconditions from specifications which use auxiliary functions are "packaged" together with them in a special kind of term: Term_Delta. These terms encode formulas of the form $(f\ in\ F)$.

5.2 Parsing and Type Checking

Programs produced by the grammar from Section 3.1 are parsed by a top-down $LL(k)$ parser generated by JavaCC — a parser generator for Java [3]. Two things are produced by the parser: a program tree and an extension of the global context mentioned before. This extension contains only globally defined specification functions. Construction of the tree nodes is delegated to the GCLCreator factory class.

During the parsing process the parser actually maintains a local context. It is used to enforce the correct use of various scopes both for program part and specification part of the input. The most important reason for this non context-free nature of the parser are back-references between the nodes. Tree nodes which are closer to the root are constructed before the more remote nodes they are referenced from, so the parser has to remember them.

Although some checks are carried out by the parser, the majority of well-formedness condition tests are implemented in the type checker. The type checker is an inheritance hierarchy of classes. Extending the existing syntax-directed, PTS-based and some other type checkers of Cocktail, the system adds two of its own classes: Typer_GCL and Typer_ADT. The type checker is invoked after a program tree has been completely constructed by the parser and no well-formedness errors have been found during the parsing. The first invocation deals with the global context extension. The second one type-checks the program tree. The type

checker implements the visitor pattern and maintains a local context during operation.

5.3 Verification Condition Generator

The verification condition generator is implemented by the GCL_VCG class. The generating algorithm follows the idea explained in Subsection 4.1.1: starting with the last program or procedure statement it moves backwards through the sequence of statements while calculating the weakest precondition for the current statement given its previously calculated postcondition. Auxiliary calculations of $Keeps$ and $Alters$ are used when necessary. As a side effect, additional verification conditions are emitted when the corresponding statements are encountered by the algorithm. Although the generated formulas are generated directly from the nodes of the program tree, they are actually composed of copies of those nodes.

Chapter 6

Conclusions and Future Work

This master thesis describes a yet unnamed tool which aids in full formal verification of complete annotated programs written in the language supported by the tool. This verification is based on some well-known formalisms. During my work on the tool I extended it with a possibility to verify partial correctness, in addition to total correctness, define and use auxiliary functions in specification part of the the language and define and use inductive data-types in programs and specifications.

The tool works with complete programs, so there is no need for the user to calculate with logic. This is in contrast with the step-wise refinement method, where even computer-aided derivations are mostly unappealing to a programmer. This should make the system more accessible to and acceptable by a programmer. The only "difficult" part for a programmer is annotating programs. However, specifications are expressed in first-order logic which should be quite familiar to the intended audience.

The system is entirely formal with a small core based on Hoare logic and the weakest precondition calculus and is sound. Generated verification conditions are separate theorems which clearly correspond to program fragments they originate from. Integration with a proof repository adds a powerful possibility to use different automated theorem provers or, if needed, proof assistants, configure the proving environment for parallel and/or distributed operation and, more importantly, adds non-trivial memory of previous proofs might they be needed again. These features are meant to offer advantages over other (formal) software verification and construction tools.

The language supported by the tool is an extension of the Guarded Command Language. At this moment it is quite simple, yet more useful than the traditional GCL due to procedures and user-defined inductive data types which allow to program and verify a much broader class of algorithms.

The authors and contributors to the tool believe and hope that the design decisions and purposely devised features should ultimately make it possible for the system to reach the status of useful, real-world, production-capable software engineering tools. Approaching this goal means constant improvements and evolution in two main directions: the language of the tool and the usability of the system.

The envisioned development of the language includes (mutually) recursive procedures, to utilize inductive data types more fully, "native" procedures with their guaranteed correct implementation provided by some (imaginary or real) run-time system, pointers and objects with reference semantics, opposite to copy semantics of the current language, proof planning

by adding additional assertions or instructions for the prover and many more. All new features of the language must have a formal foundation to make generation of verification conditions possible and to preserve soundness of the tool. Some of the listed features will be implemented in the near future, others require more research, like pointers and objects.

Improving usability of the system, except for some more fundamental considerations about suitability and acceptability of the method and required knowledge, is possible by further development of the user interface. Ultimately, the system could become a Verifying IDE, especially if the language is to be compilable. Two basic steps are possible in the near future: improved presentation of verification conditions, for example almost next to the statements they are produced by and syntax highlighting for programs. Also, at this point error reporting is not consistent and sometimes not informative enough. Yet another improvement could be in better presentation of proof results with more information about each verification condition with meta-data received from the proof repository. More additions and improvements can be thought of both for the tool itself and for its interaction with the repository, especially because these projects are still in their early phase.

Bibliography

[1] *The Coq proof assistant.* http://coq.inria.fr.

[2] *ESC/Java2.* http://kind.ucd.ie/products/opensource/ESCJava2.

[3] *JavaCC, a parsers/scanner generator for Java.* https://javacc.dev.java.net/.

[4] *Perfect Developer.* http://www.eschertech.com/products/.

[5] *Spec#.* http://research.microsoft.com/specsharp/.

[6] Krzysztof R. Apt. Ten years of hoare's logic: A survey – part i. *ACM Transactions on Programming Languages and Systems, vol. 3, no. 4*, pages 431–483, 1981.

[7] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM, vol. 18, no. 8*, 1975.

[8] M. Franssen. *Cocktail: A Tool for Deriving Correct Programs.* IPA dissertation series 2000-07. Eindhoven University of Technology, 2000.

[9] M. Franssen and M. van den Brand. Design of a proof repository architecture. Technical report, Eindhoven University of Technology, 2009.

[10] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM, vol. 12, no. 10*, 1969.

[11] M. Huth and M. Ryan. *Logic in Computer Science.* Cambdridge University Press, 2006.

[12] A. Kaldewaij. *Programming: The Derivation of Algorithms.* Prentice Hall, 1990.

[13] P. Morris, T. Altenkirch, and N. Ghani. Constructing stictly positive families. In *Conferences in Research and Practice in Information Technology (CRPIT), vol. 65*, 2007.

[14] David A. Watt. *Programming Language Design Concepts.* Wiley, 2004.