An Introduction to Formal Specification and Verification Using EHDM

John Rushby, Friedrich von Henke, and Sam Owre

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025

Technical Report CSL-91-2
February 1991

Abstract

This report is a tutorial on formal specification and verification using EHDM. The EHDM specification language is very expressive, based on a strongly-typed higher-order logic, enriched with elements of the Hoare (relational) calculus. The type system provides subtypes, dependent types, and certain forms of type-polymorphism. Modules are used to structure large specifications and support hierarchical development. The language has a complete formal semantic characterization and is supported by a fully mechanized specification and verification environment that has been used to develop large specifications and perform very hard formal verifications.

The tutorial uses simple examples to describe the EHDM language, methodology, and tools. The first examples illustrate the basic ideas of specification and theorem proving in EHDM. We then introduce the ideas of testing specifications, of horizontal and vertical hierarchy, and of consistency and conservative extension. Later chapters cover more advanced topics including subtypes, higher-order logic, proofs by induction, and program verification using Hoare logic. The tutorial is illustrated throughout with self-contained examples of EHDM specifications and proofs, all of which have been mechanically checked.

Contents

List of Figures

How to get EHDM

EHDM is being developed by the Computer Science Laboratory (CSL) of SRI International for the US Government and is in the Public Domain. SRI International has no wish to restrict the availability of EHDM, but distribution of the EHDM system and its documentation [4–6, 27] is subject to controls imposed by the US Government. Permission to obtain copies of the EHDM system and its documentation is generally routine for Agencies of the US Government, and for US Corporations working on US Government contracts. Policies on wider distribution are unclear at present.

The present report and some of those describing applications of EHDM [20–22] can be distributed without restriction. Those interested in using EHDM should contact John Rushby at the address below.

John Rushby
Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park CA 94025 USA

Tel. (415) 859-5456
FAX (415) 859-2844
Email: Rushby@csl.sri.com

Acknowledgments

The original conception of the EHDM language and system was due to Michael Melliar-Smith, Richard Schwartz, and Rob Shostak. The early implementations were developed by Judith Crow, Friedrich von Henke, Stan Jefferson, Rosanna Lee, Michael Melliar-Smith, John Rushby, Richard Schwartz, Rob Shostak, and Mark Stickel. The further language and system developments that constitute the current EHDM system are the work of Friedrich von Henke, Sam Owre, John Rushby, and Natarajan Shankar. A revised language design and a new implementation of EHDM are currently under development by Sam Owre, John Rushby, and Natarajan Shankar with assistance from David Cyrluk and Steven Phillips.

In addition to those named above, valuable contributions were made by Dorothy Denning, Brian Fromme, Allen van Gelder, Dwight Hare, Peter Ladkin, Sheralyn Listgarten, Jeff Miner, Paul Oppenheimer, Jeff Reninger, and Lorna Shinkle.

Chapter 1

Introduction

Welcome to EHDM! EHDM is a formal specification and verification environment that provides a specification language and comprehensive mechanized support for syntactic and semantic analysis, theorem proving, and report generation. You can use EHDM to state, develop, prove, and document properties about specifications and abstract programs. This short tutorial will introduce you to writing specifications in the EHDM language and using the EHDM system.

A specification language for use in mechanical verification must strike a delicate balance between the expressiveness and convenience required for effective communication among its human users, and the austere and strict logical foundation required for mechanical analysis and effective theorem proving. We think EHDM strikes this balance better than most. Attractive specification languages, such as Z [24,25] and VDM [12], have little mechanical support; systems that support effective verification, such as the Boyer-Moore prover [2], typically require specifications to be written in a raw logic. EHDM, on the other hand, has an expressive specification language and is supported by a fully mechanized specification and verification system that has been used to perform some substantial and hard formal verifications [20–22]. We hope you will find that EHDM enables you to write specifications that are clear and useful, and also to prove the theorems required to gain confidence in them.

Before we introduce EHDM, let's review the background knowledge that is assumed, and the documents and facilities you should have available. First of all, note that this report is *not* a primer on formal specification and verification; it is an introduction to a particular system that supports these activities. Readers who require a more general introduction are referred to texts such as [9,11,14]. Next, the specification language of EHDM is based on first-order predicate calculus, enriched with elements of richer logics such as higher-order logic (which is also known as type-theory), lambda-calculus, and the Hoare, or relational, calculus. As a minimum, therefore , you should be familiar with first-order predicate calculus and its model-theoretic semantics. The first volume of the book by Manna and Waldinger [15] provides an introduction to these topics that is especially suitable for computer scientists. The second volume [16] provides good coverage of more specialized topics (such as Skolemization) that are needed for really productive use of EHDM, but which are not covered in the usual logic texts. Another overview oriented towards computer scientists is the recent book by Davis [7]. Standard textbooks on Mathematical Logic include Shoenfield's [23] and Enderton's [8]; Andrews [1] gives a more recent treatment, including a good discussion of higher-order logic.

EHDM implementations are available for Sun-3 and Sun-4 (SPARC) workstations. To use EHDM, you need to have a basic understanding of the UNIX operating system and of the keyboard, mouse, and file-naming conventions. You should also ask the person who installed EHDM to tell you the names of the directories where the EHDM examples and libraries have been installed: all the EHDM modules described in this tutorial are supplied on the system distribution tape. The EHDM language and system are described in two manuals [4,5] and you should have copies of these available.[1] When you are ready to make serious use of EHDM, you should also obtain a copy of its formal semantic description [27].[2]

The interface to the EHDM system is provided by the Gnu EMACS screen editor, and you will find it very frustrating if you are not already familiar with this editor before you attempt to use EHDM. At the very least, you must know how to type a command like `C-M-P` or `M-X pp`. The EHDM system manual includes a brief overview of EMACS; complete details can be found in the GNU EMACS manual [26].

A general caveat is in order before we start our introduction to EHDM. It must be understood that writing formal specifications and performing verifications that really mean something is a serious engineering endeavor. And although formal specification and verification are often recommended for systems that perform functions critical to human safety or national security, it should be recognized that formal analysis alone cannot provide assurance that systems are fit for such critical functions. Certifying a system as "safe" or "secure" is a responsibility that calls for the highest technical experience, skill, and judgment—and the consideration of multiple forms of evidence. Other important forms of analysis and evidence that should be considered for critical systems are systematic testing, quantitative reliability measurement, software safety analysis, and risk assessment. A companion document [18] provides an introduction to these topics and techniques. Also, it should be understood that the purpose of formal verification is not to provide unequivocal evidence that some aspects of a system design and implementation are "correct," but to help *you* the user convince yourself and others of that fact; the verification system does not act as an oracle, but as an implacable

[1] This tutorial describes EHDM System Version 5.2. The system version number is displayed on the welcome screen when EHDM starts up. The User Guide [4] has not yet been updated to reflect the system changes between Versions 4.1 and 5.2; these are currently documented in a supplementary volume [6].

[2] All the system documents mentioned above are supplied on the EHDM distribution tape.

skeptic that insists on you explaining and justifying every step of your reasoning—thereby helping you to reach a deeper and more complete understanding of your design.

So much for the general introduction, now let's get started.

Chapter 2

First Steps

In this chapter we will develop a simple specification, introduce some of the EHDM tools, and prove our first theorem.

2.1 Writing Specifications in EHDM

The specification language of EHDM is based on first-order predicate calculus. We will introduce the language by means of a simple example, namely the "family" theory from Manna and Waldinger [15, page 191]. The following description of the theory is taken almost verbatim from Manna and Waldinger.

> The intent of this example is to define a theory of family relationships. The intended domain is the set of people (or "persons" as we shall call them), and the theory is defined in terms of two functions and three predicates over this domain. Intuitively:
>
> | | |
> |---|---|
> | $f(x)$ | is the father of x |
> | $g(x)$ | is the mother of x |
> | $p(x, y)$ | means y is a parent of x[1] |
> | $q(x, y)$ | means y is a grandfather of x |
> | $r(x, y)$ | means y is a grandmother of x |
>
> Thus the vocabulary of the theory consists of the function symbols f and g, the predicate symbols, p, q, and r, and no constant symbols at all. The axioms of the theory are the following closed sentences:

[1] Note, the arguments here are reversed from their "usual" order: $p(x, y)$ means y is a parent of x, *not* that x is a parent of y.

F_1: $(\forall x)\, p(x, f(x))$
(Everyone's father is their parent.)

F_2: $(\forall x)\, p(x, g(x))$
(Everyone's mother is their parent.)

F_3: $(\forall x, y)\,[\text{ if } p(x, y) \text{ then } q(x, f(y))]$
(The father of one's parent is one's grandfather.)

F_4: $(\forall x, y)\,[\text{ if } p(x, y) \text{ then } r(x, g(y))]$
(The mother of one's parent is one's grandmother.)

Now let's turn this into a specification in EHDM. Unlike the first-order predicate calculus used by Manna and Waldinger, EHDM uses a *typed* (or *multisorted*) logic. The advantages of a typed logic for specification are similar to those of user-defined types in programming languages. EHDM has several "built-in" or *interpreted* types, such as `number` (the *rational* numbers), `integer` (which can be abbreviated to `int`), `naturalnumber` (which can be abbreviated to `nat`), and `boolean` (which can be abbreviated to `bool`), and it also allows the user to introduce new *uninterpreted* types. An uninterpreted type introduces a new domain of values (which, unless the user explicitly indicates otherwise, is assumed to be disjoint from any other domain). For the family example, it is appropriate to introduce "person" as a new type, which is accomplished by the declaration

```
person: TYPE
```

In this declaration, `TYPE` is a keyword; by convention, EHDM keywords are written in uppercase, but the system recognizes them no matter how they are written—`TYPE, Type, type`, and even `tYpE`, will all be recognized as the same keyword. Identifiers, however, are case-sensitive: `person` and `Person` are different identifiers. EHDM identifiers consist of a letter (upper or lower case), followed by any sequence of letters, digits, and underscore characters. As with many programming languages, adjacent EHDM keywords and/or identifiers must be separated from each other by whitespace characters (space, tab, newline, or formfeed). Unlike most programming languages, however, EHDM declarations and expressions are *not* terminated by semi-colons.

Next, we need to declare the "signatures" of the functions and predicates that we will use. The "father" function is one that takes a value of type `person` and returns a value of the same type. In EHDM this is expressed as:

```
f: function[person -> person]
```

It is generally inadvisable, however, to use short, meaningless identifiers in specifications, just as it is in programs; we should try and choose longer, mnemonic names. In this case, a much better declaration would be

```
father: function[person -> person]
```

It's clear that we need a function `mother` of the same signature, and we can combine the two declarations into one:

```
father, mother: function[person -> person]
```

Next, we need the predicates corresponding to p, q, and r. EHDM does not use a special notation for predicates: a predicate is simply a function with return type `bool`. It's often a good idea to use identifiers beginning with `is_` for predicates—simply as a mnemonic convention. Thus we have the following declaration:

```
is_parent, is_grandfather, is_grandmother:
                            function[person, person -> bool]
```

Before we can state the axioms of our family theory, we need to introduce some variables of type `person`. Unlike constants and functions (and we will learn later that functions are simply a special kind of constant) it is often appropriate to use short identifiers for variables:

```
x, y: VAR person
```

The keyword `VAR` indicates that the identifiers being declared are variables, as opposed to constants. Now we can state the first axiom:

```
F_1: AXIOM (FORALL x: is_parent(x, father(x)))
```

The keyword `AXIOM` introduces an axiom; the name of the axiom (`F_1`) appears to the left, separated by a colon, and a boolean-valued expression appears to the right. The keyword `FORALL` introduces a universally quantified expression; the variable(s) being quantified over appear after the keyword and are separated by a colon from the expression in the scope of the quantification. The whole quantified expression is enclosed in parentheses. Existentially quantified expressions have exactly the same form, but use the keyword `EXISTS`. Notice that the function and predicate applications appearing in the expression use ordinary round parentheses (e.g., `father(x)`), whereas the declarations that introduced the functions and predicates used square brackets (e.g., `function[person -> person]`). There is a perfectly logical reason for this apparent anomaly, but the explanation will have to wait until you have learned more of the language.

EHDM provides a shorthand for universally quantified formulas that reduces typing and makes formulas easier to read: all formulas are implicitly universally quantified over their free variables. (That is, formulas denote their universal closure.) Thus, the formula given above can be replaced by the simpler one:

```
F_1: AXIOM is_parent(x, father(x))
```

This form is generally preferred. Similarly, the second axiom is expressed in EHDM as:

```
F_2: AXIOM is_parent(x, mother(x))
```

To convert the third and fourth axioms into EHDM, note that the `if...then` notation used by Manna and Waldinger is rather non-standard. These axioms would normally be expressed as logical implications, generally denoted by the symbol $\supset$. Thus the third and fourth axioms would normally be written as follows:

$$\text{F\_3:} \quad (\forall x, y)\, p(x, y) \supset q(x, f(y))$$

$$\text{F\_4:} \quad (\forall x, y)\, p(x, y) \supset r(x, g(y)).$$

In EHDM, the propositional connectives are spelled out as keywords: `AND, OR, NOT` etc. In particular, implication is indicated by the keyword `IMPLIES`. Thus the axioms above become:

```
F_3: AXIOM (FORALL x, y:
        is_parent(x, y) IMPLIES is_grandfather(x, father(y)))
F_4: AXIOM (FORALL x, y:
        is_parent(x, y) IMPLIES is_grandmother(x, mother(y)))
```

These can be rewritten, using the universal closure convention, as

```
F_3: AXIOM is_parent(x, y) IMPLIES is_grandfather(x, father(y))
F_4: AXIOM is_parent(x, y) IMPLIES is_grandmother(x, mother(y))
```

We can now put all this together into the EHDM module called "family" shown in Figure 2.1.

An EHDM module is rather like a module or package in some modern programming languages: it serves to group related things together into a unit that can be used many times over, and it serves to delimit the scope of identifiers. By default the identifiers declared in a module are not visible outside the module unless they are explicitly "exported." Only types and constants may be exported (this includes functions and predicates, which are constants of a certain "higher" type). The `EXPORTING` construct is used to list the identifiers that are to be visible outside the module, while the `THEORY` keyword introduces the types, variables, constants, and axioms that make up the *theory* defined by the module.

In order to introduce as much of the language and system as possible, we will embellish this example by adding a "theorem" that can be proved from the family theory. The theorem concerned can be stated informally as "everybody has a grandmother." In EHDM, this becomes:

```
F: THEOREM (FORALL x: (EXISTS y: is_grandmother(x, y)))
```

Note that by the universal closure convention this is equivalent to:

```
family: MODULE

EXPORTING person, father, mother, is_parent,
          is_grandfather, is_grandmother

THEORY

  person: TYPE

  father, mother: function[person -> person]

  is_parent, is_grandfather, is_grandmother:
        function[person, person -> bool]

  x, y: VAR person

  F_1: AXIOM is_parent(x, father(x))
  F_2: AXIOM is_parent(x, mother(x))
  F_3: AXIOM is_parent(x, y) IMPLIES is_grandfather(x, father(y))
  F_4: AXIOM is_parent(x, y) IMPLIES is_grandmother(x, mother(y))

END family
```

Figure 2.1: The “Family” Example

```
F: THEOREM (EXISTS y: is_grandmother(x, y))
```

but very different from

```
F: THEOREM (EXISTS y: (FORALL x: is_grandmother(x, y)))
```

(this latter expression can be expressed informally as "there is a person who is everybody's grandmother").

The syntactic form of a `THEOREM` is identical to that of an `AXIOM` and the two constructs are treated almost identically by the EHDM system. The main difference is that an `AXIOM` introduces something we wish to take as primitive, whereas a `THEOREM` introduces something that we intend to prove from other `THEOREMS` and `AXIOMS`. EHDM contains a tool called the "Proof-Chain Checker" that checks whether we have discharged our obligation to ultimately justify all `THEOREMS` on the basis of `AXIOMS` alone. For improved readability, EHDM allows a number of synonyms for the keyword `THEOREM`. These include `CLAIM`, `CONJECTURE`, `COROLLARY`, `FACT`, `FORMULA`, `LEMMA`, `PROPOSITION` and `SUBLEMMA`. Theorems that are intended to be visible outside a module should be included in its *theory* part, whereas those that are purely "internal" appear in its *proof* part, which is separated from the theory part by the keyword `PROOF`. The proof part of a module is similar to the theory part, except that it may not contain axioms, and it may contain proofs (which are not allowed in the theory part). A proof declaration instructs the EHDM theorem prover to prove the named theorem. Thus, the declaration

```
F_proof: PROVE F
```

is a minimal proof declaration that instructs the system to prove the theorem F. (The identifier `F_proof` is simply the name of this particular proof declaration and can be chosen arbitrarily).

A mechanical theorem prover must be given some base set of facts (axioms and other theorems) that it may use in trying to find a proof. This base set could be indicated implicitly (e.g., all axioms and theorems in the current context), but EHDM actually requires it to be stated explicitly. Consequently, an EHDM proof declaration must also include a list of the other theorems and axioms that are needed to construct the proof (these axioms and theorems are called the *premises* of the proof). In the case of theorem F, it should be fairly clear that this can be proved from the axioms `F_2` and `F_4`, so an adequate proof declaration is the following:

```
F_proof: PROVE F FROM F_2, F_4
```

In general, the EHDM theorem prover cannot prove theorems without help from you, the user. This help indicates the explicit *instantiations* that are required for the premises and conclusion. We will cover this in a later chapter. For the time being, we will restrict ourselves to theorems that are sufficiently simple that the theorem prover can prove them without help.

We have now assembled all the parts of the expanded specification that is shown in Figure 2.2 and the next step is to sit down at a workstation and use this example to gain some experience with the EHDM system.

2.2 Using the EHDM System

Now that we have developed a specification, let's try it out on the EHDM system. Begin by logging in to the workstation of your choice. It's recommended that you create a fresh directory for your EHDM specifications. Next, you need to choose which version of EHDM to start up. EHDM comes in two incarnations: the "tool" version runs under SunView and allows pointing and selection from pop-up menus using the mouse, but can only be used when you are directly logged in on the workstation console; the "nontool" version lacks mouse support and can be used when you are logged into a workstation remotely (or are using X-Windows). Type the command `ehdmtool` if you are using the workstation directly and want to use the "tool" version, or `ehdm` if you are logged in remotely. (You may need to prefix these commands by the pathname of the EHDM system directory if this is not in your search path. You will have to ask the person who installed EHDM for this information). Wait a minute or so while the system starts up. It keeps you informed of its progress as it loads, and finally displays a welcome screen. The EHDM system uses a heavily customized version of Gnu EMACS as its user interface. Because it might contain commands that interfere with its customizations, EHDM does *not* load the `.emacs` initialization file from your home directory. If you are certain that this file will do no harm, you can load it (or some other file) manually at this point with the command `M-X load-file`. The welcome screen reminds you that you can always get help using the command `C-X C-H`. You might want to try this before going further.

The SunView version of EHDM can be "driven" using the mouse (use the right button to bring up a menu and then select entries in the way natural to the system you are using), whereas the non-tool Sun version must be controlled from the keyboard. Since the other version can also be controlled from the keyboard, and all use exactly the same keyboard commands, unless otherwise noted we will describe the use of the system in terms of this "lowest common denominator" keyboard command set. If you are fortunate enough to be using a mouse-sensitive version and wish to avoid typing commands, just experiment with the menus until you find the command you need. You should find that the menus are organized very logically.

Now that EHDM is ready for action, your next step should be to establish your *context* in the directory you created for this purpose. Do this using the `reset-context` command (`M-X rc`)[2] and entering the name of the directory concerned. (You can omit

[2]Each EHDM command has a full name and one or more short names. It can be invoked by `M-X` followed by either its full name or one of its short forms. In addition, some very frequently used

```
family: MODULE

EXPORTING person, father, mother, is_parent,
          is_grandfather, is_grandmother

THEORY

  person: TYPE

  father, mother: function[person -> person]

  is_parent, is_grandfather, is_grandmother:
        function[person, person -> bool]

  x, y: VAR person

  F_1: AXIOM is_parent(x, father(x))
  F_2: AXIOM is_parent(x, mother(x))
  F_3: AXIOM is_parent(x, y) IMPLIES is_grandfather(x, father(y))
  F_4: AXIOM is_parent(x, y) IMPLIES is_grandmother(x, mother(y))

  F: THEOREM (EXISTS y: is_grandmother(x, y))

PROOF

  F_proof: PROVE F FROM F_2, F_4

END family
```

Figure 2.2: The “Family” Example—Expanded

this step if you started the system from the directory concerned). Next, you should obtain a copy of the "family" example module that we developed in the previous section. This is done using the `import-module` command (`M-X im`) and giving it the module name `family` and the pathname of the file containing the "family" example. You will have to ask the person who installed EHDM which directory this file is in; its name should be `family.spec`. If you cannot locate the file, use the `new-module` command (`M-X nm`) to create a new module with the name `family`, and type in the text from Figure 2.2.

The analysis of an EHDM specification involves four main steps: parsing, typechecking, proving, and proof-chain analysis. Let's try the first one: parsing. To parse the specification displayed in the current window, use the `parse` command (`M-X ps`, or `C-X C-P`). Try it now. The system should tell you that it is parsing the specification and then display the message `parsed`. Now let's see what happens when errors are present. Introduce an error into the specification (delete the keyword `MODULE`, for example), and parse again. You should see the system move the cursor to the location of the error and open up a second window containing an error message. Correct the error and parse once more. The error window will disappear and the specification should parse successfully. (You can get rid of the error window manually by `C-X 1`. If you need to get an error window back again, do `M-X replay`.) Now give the parse command once more. The system will tell you that the module is already parsed. You can discover the state of a module (whether it has been parsed etc.) manually by the `module-status` command (`M-X stat`). Try it.

Now that the module is parsed, we can *typecheck* it. Typechecking performs a semantic analysis of the module and is invoked by the `typecheck` command (`M-X tc`, or `C-X C-T`). Try it now. This usually takes a little longer than parsing, but interacts in the same way. Next, introduce a semantic error into the specification by changing the spelling of "father" in axiom `F_1` to `fathr` and start the typechecker again. Notice that the system realizes that the module has changed and automatically invokes the parser first. Just like the parser, the typechecker places the cursor at the point where the error was discovered and prints its error message in a second window. Since misspelled identifiers are a common source of errors, the typechecker performs a limited search for plausible misspellings and offers suggested alternatives in its error display. Correct the deliberate misspelling and invoke the typechecker again to ensure everything is correct.

Before we start the next step—proving—we need to make a short digression to learn about the EHDM *variables*. These variables (there are about 30 of them) control some of the fine points of the system's behavior and allow you to establish your own preferences. The variables can be examined and set using the `variable-status` command (`M-X vs`). This opens a new window and displays a list of all the variables, together with their

---

commands are also bound to special key combinations. We give all the alternative forms when we introduce a command; the help command provides the same information. When using the mouse to issue commands, the correct menu item should normally be obvious.

current and possible values. The current values are indicated by asterisks (the standard default values are the first ones in each list). You can find out what a variable does by moving the cursor to its line and hitting the `h` key. To set a variable, position the cursor on its line, just in front of the value you want to select, and type the `s` key (for numeric values, you will be prompted to enter a value). To exit this mode, use the `q` key.[3]

Use the `variable-status` command to set the following values of the variables affecting the theorem prover (prover variables all begin with `pr`):

prbetareduce	*yes no
prchain	*terse verbose
prdefaultsubs	*no yes
prhalt	*error yes no
prlambdafree	*everywhere yes no
prmode	checking interactive *automatic heuristic
prrecordsubs	ask yes *no
prsave	*no yes
prsavesubs	ask yes *no
prsavesubsversion	*new same
prtrace	*mixed terse verbose no
prtried	*ask report continue repeat

Don't worry at the moment about what these variables actually do; the purpose of the settings given is to put the theorem prover into its most automatic mode.

Now that the prover is set up to our liking (don't forget to type `q` at the end), we can attempt the proof of theorem `F`. Move the cursor until it is somewhere earlier in the specification text than the end of the line containing `F_proof` and start the prover with the `prove` command (`M-X pr`, or `C-X P`). The display will switch to a different buffer in which a commentary on the progress of the proof will appear (probably much too fast for you to read), then the display should switch back to the original buffer, place the cursor in front of the `prove` command that has just been attempted, and display the result (and the time taken) at the bottom of the screen. This proof should succeed. You can take a more leisurely look at the prover commentary by manually switching to the buffer `*ehdm-instantiator*`.

Before we end this first exercise, try invoking the prover again. It should tell you that this proof has already been done. You can check up on the status of proofs with a variety of status commands. Try `module-status` (`M-X stat`), `proof-status` (`M-X prs`), and `moduleproof-status` (`M-X mprs`).

To exit EHDM, type `C-X C-C`. Your working "context" (including settings of all specification variables) will be saved and restored when you revisit the context, either

[3]Setting variables is much easier using the mouse—you don't need to remember any of what you've just been told if you have a mouse.

by restarting EHDM in the same directory or by using the `reset-context` (`M-X rc`) command.

Chapter 3

Hierarchies

An EHDM specification generally comprises many modules. These modules are linked together "horizontally" to form a theory, and "vertically" to construct refinements. In this chapter we'll learn how to link modules together in both vertical and horizontal hierarchies. Once again, we take our examples from Manna and Waldinger [15] so that you can relate the points that they make about logic to the points we make about EHDM. This does mean that our examples are rather far removed from any specifications one might reasonably want to write—we remedy this in later chapters.

3.1 Horizontal Hierarchy

A horizontal hierarchy is a set of modules that elaborate some large theory from a collection of smaller theories, in much the same way as a large program is built from smaller modules.

The EHDM construct that links modules in horizontal hierarchy is the `USING` clause. We will use this construct to embellish the "family" theory with the notion of "sibling." The module `siblings` shown in Figure 3.1 imports the "family" theory by means of the statement

```
USING family
```

This makes all the types and constants that are *explicitly* exported from `family` visible inside the `THEORY` part of `siblings`.[1] The module `siblings` adds a new predicate `is_sibling` to the "family" theory, and makes it visible outside by means of the `EXPORTING` clause. This clause also *re-exports* everything imported from `family` by means of the additional qualifier

[1]Although we will not demonstrate it in this example, *all* the types and constants of an imported module, whether explicitly exported or not, are visible inside the `PROOF` part of the module doing the importing. Types and constants declared *inside* a `PROOF` clause, however, are *never* visible outside that module.

```
siblings: MODULE

USING family

EXPORTING is_sibling WITH family

THEORY

  a, b, c: VAR person

  is_sibling: function[person, person -> bool] ==
    (LAMBDA a, b :
       father(a) = father(b) AND mother(a) = mother(b))

  reflexive: THEOREM is_sibling(a, a)

  symmetric: THEOREM is_sibling(a, b) IMPLIES is_sibling(b, a)

  transitive: THEOREM
    is_sibling(a, b) AND is_sibling(b, c) IMPLIES is_sibling(a, c)

PROOF

  r_proof: PROVE reflexive

  s_proof: PROVE symmetric

  t_proof: PROVE transitive

END siblings
```

Figure 3.1: The “Siblings” Example

```
WITH family
```

This causes modules that import `siblings` to automatically import `family` as well.

The declaration of `is_sibling` begins in the manner we have seen already:

```
is_sibling: function[person, person -> bool]
```

but then continues in an unfamiliar manner:

```
== (LAMBDA a, b : father(a) = father(b) AND mother(a) = mother(b))
```

What we are doing here is providing a *literal definition* for a constant by using `==`. It is similar in effect and meaning to a declaration such as

```
dozen: nat == 12
```

that assigns a value to an "ordinary" constant, but is more complex because `is_sibling` is a function and we must give it a value that denotes a function. We do this with the `LAMBDA` construct, which is used to introduce values denoting functions. In syntactic form, a `LAMBDA` expression is rather like a quantified expression, with parentheses surrounding the whole expression, the keyword `LAMBDA` following the open parenthesis, and a colon separating the bound variable part from the body. The bound variable part consists of a list of variables, whose types determine the domain of the function being defined. The body of the lambda definition is an expression (whose type must agree with the return type of the function being defined[2]) composed of constants and the variables from the bound variable list. Notice that the literal definition of a constant is introduced by a *double* equals sign `==`; you can also use a single equals here, but this has slightly different pragmatic effects (the logical meaning is the same) and should be avoided for now.

An alternative to specifying the interpretation for `is_sibling` as part of its declaration would be to provide it as an axiom: for example:

```
is_sibling: function[person, person -> bool]

sib_def: AXIOM: is_sibling(a, b) =
   (father(a) = father(b) AND mother(a) = mother(b))
```

There are two advantages in using literal definitions rather than axioms. The first is that definitions are guaranteed not to introduce inconsistencies into the theory being specified; the second is that *literal* definitions are used automatically in proofs. If we had instead used the axiom given above, we would have to cite it (and possibly instantiate it explicitly) in proofs involving the `is_sibling` predicate. On the other hand, this

[2]The bound variable list may optionally be followed by an arrow `->` and the name of the return type, but this is necessary only for subtype coercions—a rather advanced topic.

automatic use of literal definitions can often be a nuisance. A theorem may follow from certain properties that have been established for a function, and replacing the function by its interpretation will only complicate the expression that the theorem prover has to work on. The decision whether a function should be defined by literal definition or not is a subtle one learned only by experience.

The rest of the `siblings` module consists of theorems that state the properties of reflexivity, symmetry, and transitivity for the `is_sibling` predicate (and thereby establish that it is an equivalence relation). The `PROOF` part simply consists of proof declarations for these theorems.

Now that we understand the `siblings` module, let's try it out. First of all, we must create a fresh module using the `new-module` (`M-X nm`) command. This command will prompt for a module name, to which you should reply "siblings." Then type the text from Figure 3.1 directly into this module. Don't worry much about layout, nor about putting the keywords in uppercase—we'll use the EHDM prettyprinter to sort that out. When you have finished typing the module in, use the `parse` command to check that you haven't introduced any syntactic errors in typing. Correct them if you have. Notice that there is no need to explicitly save the text into a file, the system takes care of that for you. Then use the `prettyprint` command (`M-X pp`, or `C-M-P`) to reformat the module text in the approved fashion. There are several variables that fine-tune the behavior of the prettyprinter; you might want to experiment with changing them (their names all begin with `pp`). When you have the module formatted as you like, use the typecheck and prover commands you learned in the previous chapter to perform all the proofs in the module. If this fails and the system tells you that the `family` module has not been parsed or typechecked, don't worry—read on.

Since `siblings` depends on `family`, it is important that the latest version of `family` is available in a parsed and typechecked form before analysis of `siblings` is performed. EHDM contains a *version checker* that automatically keeps track of this. To try it out, switch to the `family` module using the `find-module` command (`C-X C-F`)[3], and make a semantically neutral change to its buffer (e.g., add a space and take it out again). Then switch back to the `siblings` module (use `find-module` again, or use the fact that it was the previous buffer and just type `C-X B` and `return`), and then tell the system to try one of the proofs in `siblings`. You should get an error message telling you that `family` has not been parsed since it was changed. You could then go back to `family` and parse and typecheck it, then return to `siblings` and do the same there, but there is a simpler way: the `typecheckall` command (`M-X tca`) will descend the using-chain from the current module and automatically bring everything up to date. The `provemodule` command (`prm`) automatically does a `typecheckall` if it finds that it is necessary, and then does all the proofs in the current module. There is an even

[3]This command supports module-name completion. Simply type in the beginning of the module-name and hit the space-bar; the system will complete as far as it can uniquely. If you hit the space bar again the set of possible completions will be displayed. Most of the EHDM commands have this facility.

more muscular command called `proveall` (`M-X pra`) that not only does all the proofs in the current module, but all the proofs in all the modules that it uses as well. Try out these commands, and also the status commands you learned in the previous chapter. Also try the `using-status` (`M-X us`) and `usedby-status` (`M-X ys`) commands to look at the "using-chain" that links `siblings` and `family`.

A more complex status check is performed by the *Proof-Chain Checker*. To demonstrate this, we need to construct another example. This example, called `grandparents` is shown in Figure 3.2 and should be self-explanatory.

```
grandparents: MODULE

USING family

THEORY

  x, y: VAR person

  is_grandparent: function[person, person -> bool] ==
    (LAMBDA x, y :
       is_grandfather(x, y) OR is_grandmother(x, y))

  G: THEOREM (EXISTS y : is_grandparent(x, y))

PROOF

  G_proof: PROVE G FROM F

END grandparents
```

Figure 3.2: The "Grandparents" Example

The proof of the formula `G` ("everybody has a grandparent") from `F` ("everybody has a grandmother") is trivial, and the theorem prover will prove it instantly. The proof-status command will then tell us that the theorem has been proved. But this notion of "proved" is a local one: it means only that the conclusion of the proof follows from the premises. We do not know whether the premises are axioms or other theorems—and if the latter, we do not know whether they have been proved or not. Consequently, we cannot be sure, without a lot of manual checking, whether a theorem and all the theorems it depends upon have been proved down to base axioms. This is where the Proof-Chain Checker comes in. Move the cursor to point to `G_proof` in the `grandparents` module and invoke the `proofchain-status` command (`M-X pcs`). When prompted for the "name of the module at the top of the proof tree," just type a `return`. If both the necessary proofs have been performed, you will be told that the proof-chain is sound and all the necessary proofs have been done. If you change the value of the `prchain` system variable to `verbose` and repeat the `proofchain-status` command, you will see a more elaborate display that chases the proof of `G` through its dependence on the theorem `F`, and its dependence, in turn, on the axioms `F_2` and `F_4`. Experiment with the output when one of the proofs have not been done, and when `F_proof` is actually deleted from its module. There is a more muscular command called `allproofchain-status` (`M-X aprs`) that checks the proof-chains of *all* the proofs in a module. Try it.

Instead of asking for the proof-chain status of a proof, you can also ask for the status of a formula (i.e., `AXIOM`, `FORMULA`, `THEOREM`, or `LEMMA`). Use the cursor to point to a formula and use the `formula-status` command (`M-X fs`). There is also an `allformula-status` command (`M-X afs`). Try them.

3.2 Testing Specifications

The generally accepted notion of program correctness is *consistency* between the program and its specification. But then how do we know that the specification is right? And what does it even mean for a specification to be right? There are no easy answers to those questions. In some cases, the correctness of a specification can be interpreted in the same way as for programs: a specification is correct if it is consistent with a yet higher-level specification. This is the notion of vertical hierarchy and is discussed in the next section. At the top of any specification hierarchy, however, must come the specifications whose only superior specifications are informal statements of requirements. How can we know that these top-level specifications capture the real requirements accurately and completely? The answer is that we cannot expect to justify or prove the appropriateness of these top-level specifications in any formal way: we must rely on peer review and the social process to provide the assurance we need. We should also apply good software engineering practice to our specifications—making extensive use of horizontal hierarchy to build our top-level specifications so that just a few, simple concepts are introduced in each module, and then gradually combined to yield the overall specification. In this way we can reduce the complexity of the task of comprehending the overall specification.

There is something else we can do to help our understanding of specifications: we can test them. Because EHDM uses a very rich logic, it is not an executable specification language, and so we cannot "run" our specifications[4]; however, we can test them indirectly by posing and attempting to prove putative theorems. In effect, we say: "if I've formalized this concept correctly, then the following ought to be true" ...then we

[4]High-level specifications are generally (deliberately) incomplete, and so it is infeasible to directly execute them anyway. However, there is some support for generating Ada code from EHDM specifications; see Chapter 7.

write it down and try to prove it. If we succeed, then our confidence in the specification is increased. If we fail, then we will probably have gained insight into some inadequacy in our specification. A contrary form of test would take the form: "if I've formalized this concept correctly, then the following ought *not* to be true."

Let's try this out with our "family" example. If we've formalized the notion of family relationships correctly, then surely it shouldn't admit theorems like the following (from Manna and Waldinger [15, Problem 4.1, page 212]):

> "If Alice is the parent of her own father, then Alice's father is his own grandfather."

This example is so artificial and simple that you can probably see it through in your head, but it will be a useful exercise to express it in EHDM. First, we need to introduce "Alice" as an uninterpreted constant of type person:

```
Alice: person
```

Then we need to express the putative theorem (which we'll call `X`):

```
X: THEOREM is_parent(father(Alice), Alice)
     IMPLIES is_grandfather(father(Alice), father(Alice))
```

Notice that `father(Alice)` appears three times in this expression. It might be nice to introduce a name for Alice's father, so that we can abbreviate things a little. Let's call him William:

```
William: person == father(Alice)

X: THEOREM is_parent(William, Alice)
     IMPLIES is_grandfather(William, William)
```

A little thought should convince you that this result is indeed a theorem of our "family" example, and that all that is needed to prove it is the `F_3 AXIOM`. Putting it all together, we construct the module `california` (so called because families such as these are only found in California) shown in Figure 3.3.

Notice that there is no `THEORY` part in this example. We could have put the first three declarations (but not the `PROVE`) into a theory part if we so wished, but there would have been little point since we do not expect to make them publicly available. Notice also that there is a `USING` clause following the `PROOF` part. Any modules imported through a `USING` clause at the head of the module are available throughout the module, but additional modules can be imported into just the proof part by a `USING` clause immediately following the keyword `PROOF`.

What does the fact that `X` is provable tell us? Since `X` could not be true of family relationships in the real world, our axiomatization in the `family` module must be wrong

```
california: MODULE

PROOF

USING family

  Alice: person

  William: person == father(Alice)

  X: THEOREM is_parent(William, Alice)
       IMPLIES is_grandfather(William, William)

  X_proof: PROVE X FROM F_3

END california
```

Figure 3.3: The "California" Example

or incomplete—right? Actually this is partly right and partly wrong. Certainly our `family` theory is very incomplete (we haven't identified the gender distinction between mother and father, nor the fact that nobody can be their own parent, to name but a few deficiencies), but the truth of `X` doesn't necessarily invalidate our axiomatization since `X` is an *implication* and an implication is true when its antecedent is false. So it's possible that `X` could be true even in a formulation of family that did agree with our intuition. What should worry us is the possibility that we could have a consistent theory in which Alice *is* the parent of her own father—for then we would know that our theory has unintended models. We will see how to check this in the next section.

"Alice" and theorem `X` raise the possibility that our family theory has too many models—the theory is incomplete since it does not uniquely characterize the properties of real families. Sometimes such incompleteness is intended and desirable: we need to axiomatize only the properties that matter to our particular application.

The "opposite" possibility—too *few* (i.e., zero) models—is never a good thing since it indicates an inconsistent theory. And from an inconsistent theory you can prove *anything*. We will demonstrate this using another example from Manna and Waldinger [15, page 197]. This example invites us to consider the case of the Biblical Adam, who had no father:

```
  Adam: person

  y: VAR person
```

```
  a: AXIOM NOT (EXISTS y: is_parent(Adam, y))
```

Incidentally, by a first-order transformation, the axiom could also be written as:

```
  a: AXIOM NOT is_parent(Adam, y)
```

The problem, of course, is that the negation of this axiom can easily be proved from the `F_1` axiom of the `family` module. Once we have established both a formula and its negation, we can prove anything—including `true = false` as shown in Figure 3.4.

```
eden: MODULE

USING family

THEORY

  Adam: person

  y: VAR person

  a: AXIOM NOT (EXISTS y : is_parent(Adam, y))

PROOF

  b: LEMMA (EXISTS y : is_parent(Adam, y))

  b_proof: PROVE b FROM F_1

  inconsistency: THEOREM true = false

  ouch: PROVE inconsistency FROM a, b

END eden
```

Figure 3.4: The "Eden" Example

Try typing this module into the system and performing the proofs. You should find that the system detects the inconsistency in the premises of the proof `ouch` and reports an error. (If you look at the `*ehdm-instantiator*` buffer, you will see that the theorem prover succeeded in proving the theorem before it detected the error.) While it is comforting to know that EHDM performs this safety check, it is not a complete safeguard. The theorem prover of EHDM is a refutation-based prover: that is, it tries to prove a theorem by showing that the conjunction of its premises and its negated conclusion is unsatisfiable. If it finds that the conjunction of the premises alone is unsatisfiable, then it raises the error message as in the `eden` example. Because of the incomplete heuristics employed, it may be possible for the prover to prove a theorem but not the inconsistency in its premises. More importantly, one may derive all sorts of conclusions from an inconsistent theory without doing anything so gross as to prove an outright contradiction. The only ways to ensure that a theory is consistent are to severely limit the forms of axioms employed (e.g., all Horn clauses, or all equations, with no interpreted functions), or to demonstrate the existence of a model. We consider this latter alternative in the following section.

Before concluding our brief discussion of hazards in specifications, there is one final point that is worth considering. The `eden` module extended the theory introduced in the family `module`. If we accept that the `family` theory is consistent, it must be that the inconsistency was introduced by `eden`. How do we know that the `california` module doesn't also introduce an inconsistency? The answer is that because `california` contains no axioms, it must be a *conservative* extension of `family`: anything that is true in `family+california` is also true in `family` alone.[5] The EHDM definition constructs (including the recursive definitions and constrained subtypes introduced in Chapter 6) are carefully designed to ensure conservative extension. For this reason, it is usually best to introduce new concepts by means of definitions rather than axioms whenever it is possible to do so.

3.3 Vertical Hierarchy

Horizontal hierarchy is used to incrementally build a specification from small, mind-sized chunks; *vertical* hierarchy, on the other hand, is the hierarchy of *refinement*. In vertical hierarchy we refine (or "implement") a specification at one level of abstraction in terms of objects and operations from a more detailed, concrete specification. The intent is not to add functionality, but to bring the description closer to an efficient, practical realization. An example is the implementation of a stack by means of an array and a pointer—we will actually carry out this example in the next chapter. The "H" in EHDM stands for "hierarchy" and it is the vertical form that is meant here. EHDM's ability to support true vertical hierarchy is one of its unique strengths.

The notion of refinement used in a vertical hierarchy of specifications is somewhat similar to the concept of *model* in logic. It is probably worth recapitulating the terminology of logic in this context. A model for a theory is an algebra whose objects and operations can be placed in correspondence with the individuals, functions, and predicates of the theory (this correspondence is called an *interpretation*) in such a way

[5]A theory A is an *extension* of a theory B if its language includes that of B and every theorem of B is also a theorem of A; A is a *conservative* extension of B if, in addition, every theorem of A that is in the language of B is also a theorem of B.

that the axioms of the theory are true statements of the model. A logical sentence is *valid* if it is true in every model of the theory; a logic is *sound* if only valid sentences are provable, it is *complete* if every valid sentence is provable. A theory is *inconsistent* if it has no models.

In a specification language like EHDM, we can only describe theories, and hence cannot directly address models and interpretations. However, there is a notion of *theory interpretation* that connects two theories in much the same way as a standard interpretation connects a theory to a model. A theory interpretation, or *mapping* as it is called in EHDM, is an association between the types, constants and functions of one theory and another. The mapping describes how to interpret the concepts of the higher, or more abstract theory, in terms of those of the lower or more concrete theory. In order to establish that the mapping is sound, we must prove that the (mapped) axioms of the higher theory become *theorems* of the lower theory. If these theorems can be proved, then we know that the lower theory correctly implements the higher one. In addition, if we know that the lower theory is consistent, then the higher one is also.

Let's do an example to see how this works. We will show that the `family` theory is consistent by mapping it onto a fragment of integer arithmetic. As before, this example is taken directly from Manna and Waldinger [15, page 192].

To perform the mapping, we need to construct a theory with which the types, constants and functions of family can be placed in one-to-one correspondence (actually many-to-one will do). Thus, we need an interpretation for the type `person`, and the functions `father, mother, is_parent, is_grandfather` and `is_grandmother`. The signatures of these new functions must correspond exactly to those of the family theory. The interpretation given by Manna and Waldinger associates the `integers` with `person`, the function that multiplies by two with `father`, the function that multiplies by three with `mother`, and so on (thus it is probably not the intended model for this axiomatization of families). To remind ourselves of the connection to be established between this new theory and `family`, we will use the same identifiers as in `family`, but with `interp_` added at the front.

Our first job is to define the type `interp_person`, but we want this to be the same as the integers. We do this with the declaration

```
interp_person: TYPE IS int
```

The keyword `IS` indicates that the new type `interp_person` is simply a synonym for the built-in type `int`: everywhere `interp_person` appears, it will be just as if we had written `int` instead. Next, since the signature of `father` is `person -> person`, we need a function `interp_father` of signature `interp_person -> interp_person` and we will give it the interpretation of multiplying its argument by two:

```
x: VAR interp_person

interp_father: function[interp_person -> interp_person] ==
    (LAMBDA x : 2 * x)
```

```
model: MODULE

EXPORTING ALL

THEORY

  interp_person: TYPE IS int

  x, y: VAR int

  interp_father:
    function[int -> int] ==
       (LAMBDA x : 2 * x)

  interp_mother:
    function[int -> int] ==
      (LAMBDA x : 3 * x)

  interp_is_parent:
    function[int, int -> bool] ==
      (LAMBDA x, y : y = 2 * x OR y = 3 * x)

  interp_is_grandfather:
    function[int, int -> bool] ==
      (LAMBDA x, y : y = 4 * x OR y = 6 * x)

  interp_is_grandmother:
    function[int, int -> bool] ==
      (LAMBDA x, y : y = 6 * x OR y = 9 * x)

END model
```

Figure 3.5: The “Model” Example

Proceeding in this fashion, we arrive at the module `model` shown in Figure 3.5. This module also introduces a new variant of the `EXPORTING` clause: `EXPORTING ALL` simply indicates that all types, constants and functions declared in the theory part of the module are to be made visible outside.

Next, we need to construct the *mapping module* that connects `family` to `model`. In this module we must establish the *associations* between the types, constants, and functions from `family` and the corresponding entities of `model`. The mapping module (which is called `interpretation`) from `family` to `model` is shown in Figure 3.6.

```
interpretation: MODULE

MAPPING family ONTO model

  person -> interp_person
  father -> interp_father
  mother -> interp_mother
  is_parent -> interp_is_parent
  is_grandfather -> interp_is_grandfather
  is_grandmother -> interp_is_grandmother

END interpretation
```

Figure 3.6: The "Interpretation" Mapping Module

The `interpretation` module begins with a `MAPPING` clause that tells the system that this module is a mapping module that provides an interpretation of `family` in terms of `model`. Instead of a theory part, a mapping module lists *associations* such as

```
  person -> interp_person
```

This indicates that the "source" type `person` from the module `family` is to be interpreted by the "target" type `interp_person` from the module `model`. The remaining associations establish the interpretations for the functions defined in the `family` module. Associations can be omitted when source and target names are the same.

When you typecheck the `interpretation` module, the system will tell you that it has generated a `MAPPED` module called `interpretation_map`. This module is shown in Figure 3.7.

EHDM creates a mapped module whenever it typechecks a mapping module. Each axiom from the source module(s) is translated according to the associations specified in the mapping module and the resulting "mapped axiom" is recorded as a formula in the mapped module under the same name as the axiom from which it was derived. If you can prove all the formulas generated in a mapped module, then the interpretation

```
interpretation_map: MODULE

USING model

EXPORTING ALL WITH model

THEORY

  y: VAR integer
  x: VAR integer

  F_1: FORMULA interp_is_parent(x, interp_father(x))

  F_2: FORMULA interp_is_parent(x, interp_mother(x))

  F_3: FORMULA interp_is_parent(x, y)
      IMPLIES interp_is_grandfather(x, interp_father(y))

  F_4: FORMULA interp_is_parent(x, y)
      IMPLIES interp_is_grandmother(x, interp_mother(y))

PROOF

  F_1_PROOF: PROVE F_1

  F_2_PROOF: PROVE F_2

  F_3_PROOF: PROVE F_3

  F_4_PROOF: PROVE F_4

END interpretation_map
```

Figure 3.7: The Mapped Module “Interpretation_Map”

specified by the mapping module is sound. EHDM automatically places an elementary proof declaration for each formula in the mapped module itself. These suffice in the case of `interpretation_map`, but in general you will need to construct more sophisticated proofs that cite any necessary axioms and lemmas. You must construct such proofs in a *different* module; EHDM will not allow modification to a mapped module, since the soundness of the analysis could be compromised thereby. You will notice that EHDM displays mapped modules in read-only buffers (indicated by `%%` at the left of the Emacs mode line); if you override this protection and alter a mapped module, EHDM will henceforth treat the module as a user-written module and it will lose its special status as a mapped module (as indicated by the `module-status` command).

Chapter 4

More About Theorem Proving

EHDM contains four components that perform theorem proving functions: the *Skolemizer*, the *Ground Prover*, the *Instantiator*, and the *Hoare Sentence Prover*.[1] The last of these is concerned with proving properties about operational programs and specifications and is described in a later chapter; this chapter is concerned with the Skolemizer, Ground Prover and Instantiator. We will explain these by example, using a modified version of the `siblings` module shown in Figure 4.1, that differs from the earlier one in that here the `is_sibling` function is defined axiomatically.

If we consider the first proof, we see that we are seeking to prove a formula whose full form is

```
(FORALL a: is_sibling(a, a))
```

from another formula whose full form is:

```
(FORALL a, b: is_sibling(a, b) IFF
    father(a) = father(b) AND mother(a) = mother(b))
```

where `IFF` is the propositional connective "if and only if."[2]

Now the basic theorem proving strategy in EHDM is based on seeking a *refutation* of the *negation* of the theorem. That is, we conjoin (i.e., "and" together) the premises and the negated conclusion and seek to show that the resulting formula is unsatisfiable. In the present case, this means we are trying to refute the following formula:

```
(NOT (FORALL a: is_sibling(a, a)))
     AND (FORALL a, b: is_sibling(a, b) IFF
       father(a) = father(b) AND mother(a) = mother(b))
```

[1]Versions of EHDM later than 5.1.1 contain an experimental *Heuristic* Instantiator in addition to the regular one. This second instantiator, which is used when the EHDM variable `prmode` has the value `heuristic`, is generally the most effective way to do "production" proofs. It is documented in [6, Appendix A].

[2]The "equal" symbol on the booleans is translated internally to `IFF`.

```
alt_siblings: MODULE

USING family

EXPORTING is_sibling

THEORY

  a, b, c: VAR person

  is_sibling: function[person, person -> bool]

  sib_def: AXIOM
    is_sibling(a, b)
      = (father(a) = father(b) AND mother(a) = mother(b))

  reflexive: THEOREM is_sibling(a, a)

  symmetric: THEOREM is_sibling(a, b) IMPLIES is_sibling(b, a)

  transitive: THEOREM
    is_sibling(a, b) AND is_sibling(b, c) IMPLIES is_sibling(a, c)

PROOF

  r_proof: PROVE reflexive FROM sib_def
  s_proof: PROVE symmetric FROM sib_def, sib_def
  t_proof: PROVE transitive FROM sib_def, sib_def, sib_def

END alt_siblings
```

Figure 4.1: The Alternative “Siblings” Example

The next step is to get rid of the quantifiers by Skolemization. There is a description of Skolemization in the User Guide [4, Section 5.1.1], but for our present purposes it suffices to say that all variables inside the scope of an *odd* number of negations or existential quantifiers are replaced by constants, and all quantifiers are deleted.[3] If we use `*a` to denote the Skolem constant corresponding to variable `a`, then our formula becomes the following:

```
(NOT is_sibling(*a, *a))
    AND (is_sibling(a, b) IFF
       father(a) = father(b) AND mother(a) = mother(b))
```

The next task is to provide constants as *substitutions* for the variables that remain in this *quantifier-free* formula, so that the resulting *ground* expression is propositionally unsatisfiable. It is fairly easy to see that if we substitute the constant `*a` for both the variables `a` and `b`, then we obtain the propositionally unsatisfiable formula:

```
(NOT is_sibling(*a, *a))
    AND (is_sibling(*a, *a) IFF
       father(*a) = father(*a) AND mother(*a) = mother(*a))
```

The (un)satisfiability of propositional calculus is a decidable problem—i.e., it can be done by a computer program.

Summarizing:

Skolemization reduces quantified formulas to unquantified ones.

Substitution of constants for variables reduces unquantified formulas to *ground* ones.

Decidability of propositional calculus ensures that we can tell whether the resulting ground formula is unsatisfiable. If it is unsatisfiable, then the original theorem is proved. Otherwise, either the theorem is false, or it is true but we chose the wrong substitutions.

The three steps above are performed by the three theorem proving components identified earlier: namely, the Skolemizer, the Instantiator, and the Ground Prover respectively. It should be clear that both the first and last steps are algorithmic—they cannot affect our ability to perform a proof (providing they are implemented correctly). All the "skill" is in the middle step, in the choice of the right substitutions to make for the free variables in the Skolem form. By the Herbrand-Skolem-Gödel theorem, there must be *some* set of substitutions that will do the trick, but by the semi-decidability of first-order predicate calculus (another one of Gödel's theorems), there is no algorithm

[3] There are several gross simplifications in this description. For a more precise explanation, see the description in the User Guide, in particular, the discussion of "Skolem *Functions*" (they are not constants, in general), "parity," and "governing variables." See also [16, Chapter 10].

that can be guaranteed to find them. Since the Instantiator of EHDM is a program, it cannot always find the correct substitutions to complete a proof. Consequently, EHDM provides two methods by which you, the user, can help the system to find the right substitutions. The first of these methods requires you to indicate the required substitutions directly in the `PROVE` declaration; the second method involves interaction with the Instantiator. This latter method is available only when the Instantiator is in *interactive* mode (which is controlled by the variable `prmode`). Interactive mode is available only in the SunView version of the system (i.e., it is not available in the non-tool version), and is not described here.

When the Instantiator succeeds in finding a successful set of substitutions (possibly found through interaction with the user), it can write the appropriate substitutions back into the specification text. This means that if the theorem needs to be proved again (e.g., because we have changed some—hopefully irrelevant—part of the specification), the proof will go *much* faster, since there will be no need to search for the substitutions. In fact, it is then possible to turn the Instantiator off altogether, which results in much increased prover speed. Some experts always use EHDM in this mode, since they find it easier or faster to provide the substitutions themselves.

Now let's try some of this out. First of all, we need to change the settings of some of the EHDM variables to the asterisked values shown below:

prbetareduce	*yes no
prchain	*terse verbose
prdefaultsubs	*no yes
prhalt	*error yes no
prlambdafree	*everywhere yes no
prmode	checking interactive *automatic heuristic
prrecordsubs	ask *yes no
prsave	*no yes
prsavesubs	ask *yes no
prsavesubsversion	*new same
prtrace	mixed terse *verbose no
prtried	ask report continue *repeat

Next, move the cursor to `r_proof` in the `alt_siblings` module and start the prover. The Instantiator will find a substitution to prove this theorem in just a few seconds and will write the substitution back into the specification text like this:

```
r_proof: PROVE reflexive FROM sib_def {b <- a@CS, a <- a@CS}
```

The substitution is the text in braces. What it says is that the variable `b` in the premise `sib_def` should be replaced by the Skolemized instance of the variable `a` from the conclusion (i.e., the formula `reflexive`) and that the variable `b` in the premise should be replaced by the Skolem instance of `a` from the conclusion. It is the appearance of the

`S` and `C` in the qualifiers `@CS` attached to the variables that indicate "conclusion" and "Skolem instance" respectively. With the substitutions now embedded in the specification text, try doing the proof again. You should find it goes a bit quicker than before (though the need to reparse and typecheck the modified specification slows down the total operation). Next, switch to the `*ehdm-info*` buffer and study the "trace" of the process of Skolemization and substitution that is recorded there. Try and understand how the substitutions serve to link the formulas together.

Now try the same sequence of actions on the next proof in the module and on the third. Notice that the second proof needs two instances of the `sib_def` premise, while the third needs three. See what happens if you provide too many or too few instances of this premise. Also see what happens if you provide some of the substitutions in the specification, and let the Instantiator find the rest. What happens if you give some wrong substitutions in the specification text? Also see what happens if you set the variable `prmode` to `checking` and invoke the theorem prover on proofs with full and partial explicit substitutions.[4] Finally, see if you can construct the necessary substitutions on your own.

Note that you can generate the Skolemized form of a proof without having to go through the full proof process using the `skolemize` command (`M-X sko`). Try it—note that the value `mixed` for `prtrace` behaves like `terse` for proofs and like `verbose` when you use the skolemize command. Also try the `show-parity` command that tells you which variables in the formula indicated by the cursor will become Skolemized and which will be available for substitution.

[4] The Instantiator is not used in checking mode. In checking mode, the formula is reduced to ground form using any explicit substitutions given in the text, together with a set of default bindings that are described in the User Guide.

Chapter 5

Putting It Together

In this chapter, we'll put what we've learned so far, together with a few new things, to use on a computer science application: the canonical "stacks" example. Later we show how to verify the implementation of a stack that uses an array and a pointer.

5.1 Specification of Stacks

Given what we know already, we can easily construct the specification for a simple "stack of integers" module shown in Figure 5.1. A deficiency of this specification is that it is specific to the particular case of a stack of integers. If we wanted a stack of booleans, or a stack of some user-defined type, we would have to define a new stacks module. A similar situation occurs in older programming languages, such as Pascal, which does not allow arrays of variable size as parameters of subprograms, so that different subprograms had to be written for different sizes of arrays. This is, obviously, rather unproductive because the code of all those subprograms would be almost identical. What is needed, in both programming and specification languages, is some way to parameterize modules—and subprograms—so that they can be reused in different incarnations. Ada, for example, provides *generic packages* for this purpose; the equivalent construct in EHDM is the *parameterized module.*

EHDM allows modules to be parameterized by types and by constants, so the natural way to rewrite our simple stacks module is as a parameterized module that takes the type of the elements of the stack as a parameter. We should also add the particular value of the element type that is to be used as an "error flag" as a second parameter (it's hardwired as 0 in the simple stacks example). Before we construct a parameterized stacks module, however, there is an additional point that is worth introducing. We explained earlier that the EHDM theorem prover needed to be given an explicit list of the premises it could use in each proof. In the particular case of a module whose axioms consist solely of *equations*, it is possible to tell the theorem prover that the whole theory defined by that module is available, and the theorem prover will use those equations in

```
simple_stacks: MODULE

EXPORTING ALL

THEORY

  stack: TYPE

  newstack: stack

  s: VAR stack
  e: VAR int

  push: function[stack, int -> stack]
  pop: function[stack -> stack]
  top: function[stack -> int]
  isnewstack: function[stack -> bool]

  replace: function[stack, int -> stack] ==
    (LAMBDA s, e : push(pop(s), e))

  popnew: AXIOM pop(newstack) = newstack
  topnew: AXIOM top(newstack) = 0
  poppush: AXIOM pop(push(s, e)) = s
  toppush: AXIOM top(push(s, e)) = e
  isnewnew: AXIOM isnewstack(newstack)
  isnewpush: AXIOM NOT isnewstack(push(s, e))

END simple_stacks
```

Figure 5.1: The “Simple Stacks” Example

a very efficient manner. Looking at our simple stacks example, we see that all but the last two axioms are simple equations. In fact we could rewrite these as equations:

```
isnewnew: AXIOM isnewstack(newstack) = true
isnewpush: AXIOM isnewstack(push(s, e)) = false
```

but this option will not always be available in other specifications that we write. A more general approach to specifications that have many equations, plus a few non-equational axioms, is to split the specification into two modules: one for the equations, and the other for the non-equational axioms. Hence we arrive at the specification shown in the two modules of Figures 5.2 and 5.3. [1]

Since `stack_eqns` is a parameterized module, any uses of it need to indicate which *instantiation* is required. For example, the instantiation of `stacks_eqns` in which the integers are used as the actual `elem` type, and 0 is used us the `undef` flag would be indicated by

```
stack_eqns[int, 0]
```

while a stack of type `person`, with `Alice` as the undefined flag would be indicated by

```
stack_eqns[person, Alice]
```

What if we needed to have both of these instantiations available simultaneously? How would we indicate which version of `push` we meant in each particular context? The answer is that we would have to *qualify* each use of an identifier from `stack_eqns` by the particular instantiation intended. This is done by preceding the name by a dot and the specific instantiation of the module that is intended:

```
stack_eqns[int, 0].push
```

As a convenient abbreviation, EHDM allows you to delete the module name and move the identifier to the front if the result is unambiguous. Thus the following form is equivalent to the one given previously.

```
push[int, 0]
```

As a further abbreviation, you can often get away with just saying `push`—the type-checker will try to resolve the specific instance on the basis of the argument types that are used.

Turning to the case of module `stacks`, notice that it is itself a parameterized module and that it imports `stack_eqns` instantiated with its own formal parameters.

We can use the theory defined by `stack_eqns` and `stacks` to prove some simple theorems. This is done in the module `stack_thms` shown in Figure 5.4. Notice that

```
stack_eqns: MODULE [elem: TYPE, undef: elem]

EXPORTING stack, newstack, push, pop, top, replace

THEORY

  stack: TYPE
  newstack: stack
  s: VAR stack

  e: VAR elem

  push: function[stack, elem -> stack]
  pop: function[stack -> stack]
  top: function[stack -> elem]

  replace: function[stack, elem -> stack] ==
    (LAMBDA s, e -> stack : push(pop(s), e))

  popnew: AXIOM pop(newstack) = newstack
  topnew: AXIOM top(newstack) = undef
  poppush: AXIOM pop(push(s, e)) = s
  toppush: AXIOM top(push(s, e)) = e

END stack_eqns
```

Figure 5.2: The “Stack_Eqns” Example

```
stacks: MODULE [elem: TYPE, undef: elem]

USING stack_eqns[elem, undef]

EXPORTING isnewstack WITH stack_eqns[elem, undef]

THEORY

  s: VAR stack
  e: VAR elem

  isnewstack: function[stack -> bool]

  isnewnew: AXIOM isnewstack(newstack)
  isnewpush: AXIOM NOT isnewstack(push(s, e))

END stacks
```

Figure 5.3: The "Stacks" Example

since `stacks` re-exports `stack_eqns`, it is not necessary for `stack_thms` to explicitly import `stack_eqns`.
Notice that the proof part of `stack_thms` contains a new construct:

```
  WITH stack_eqns
```

This tells EHDM that the theorem prover can make use of all the axioms in the module `stack_eqns` in performing the proofs that follow.[2] This use of the `WITH` clause may refer only to modules whose axioms are all simple equations.

Notice also that the final proof in `stack_thms` does not follow from the equational axioms alone and that we need to indicate the additional axiom that is required to perform this proof. You might want to try deleting the `WITH` clause from `stack_thms` and trying to provide the full `FROM` clauses that will then be required. Beware that if an axiom needs to be used more than once in a proof, then it must be cited the requisite number of times in the `FROM` clause.

[1] These modules have been constructed specifically to demonstrate various features of the EHDM system and we deliberately ignore issues such as stack overflow. Appendix A provides a more realistic alternative specification.

[2] This applies only when the Instantiator is in use—that is, when the variable `prmode` has the value `automatic` or `interactive`.

```
stack_thms: MODULE

USING stacks[int, 0]

THEORY

  s: stack
  e, x: VAR nat

  top_replace: THEOREM top(replace(s, e)) = e
  pop_replace: THEOREM pop(replace(s, e)) = pop(s)
  push_replace: THEOREM replace(push(s, e), x) = push(s, x)
  isnewreplace: THEOREM NOT isnewstack(replace(s, e))

PROOF

WITH stack_eqns

  topr_proof: PROVE top_replace
  popr_proof: PROVE pop_replace
  pushr_proof: PROVE push_replace
  newr_proof: PROVE isnewreplace FROM isnewpush

END stack_thms
```

Figure 5.4: The “Stack_Thms” Example

We are now in a position to clear up the mystery regarding the syntax for function declarations. The explanation is that `function` is a type exported from a parameterized module called `functions`[3]. Thus

```
function[stack, elem -> stack]
```

is really an abbreviation for

```
functions[stack, elem -> stack].function
```

where the arrow `->` can be regarded as simply a special type of comma.

5.2 Implementation of Stacks

In this section we describe the standard implementation of stacks by a pair consisting of an array and a pointer, and we prove the "correctness" of this implementation (i.e., its consistency with the axiomatization of stacks just presented).

The stack implementation is specified in the module `stackrep` shown in Figure 5.5. In this module we define types, constants, and functions corresponding to each of those defined in the basic stacks theory (defined by `stacks` and `stack_eqns`). To aid comprehension, the names of the implementation-level objects are the same as those of their abstract counterparts, with `rep_` stuck on the front.

This module introduces a number of new features. The first is the declaration

```
linarray: TYPE = array [nat] OF elemtype
```

This is an instance of a *type definition*; in contrast to a type *declaration* (such as `foo: TYPE`) that introduces an uninterpreted type, a definition introduces an interpreted type whose interpretation is specified by the *type expression* given to the right of the equals sign. EHDM type expressions provide for three kinds of interpreted types: arrays, records and enumerations. Here we are introducing an array type; the array elements are of type `elemtype` and are indexed by the type `nat`. Semantically, arrays are identical to functions, and array access is indicated in the same way as function application (e.g., `x(4)`). New values are assigned to array elements by a `WITH` construct as follows:

```
x, y: linarray
i: VAR nat
xdef: FORMULA x = y WITH [i := undef]
```

The formula `xdef` indicates that the array `x` has the same elements as the array `y`, except that its `i`'th element has the value `undef`.

Next, the declaration

[3]The `functions` module is defined in the prelude, but is not really a module, since it has a variable number of parameters. EHDM does not currently support user-defined modules with a variable number of parameters

```
stackrep: MODULE [elemtype: TYPE, undef: elemtype]

EXPORTING linarray, rep_stack, concrete_equality, rep_newstack,
  rep_push, rep_pop, rep_top, rep_isnewstack

THEORY

  linarray: TYPE = ARRAY [nat] OF elemtype

  rep_stack: TYPE = RECORD astack: linarray,
                           pointer: nat
                    END RECORD

  p, p1, p2: VAR rep_stack
  e: VAR elemtype
  i: VAR nat

  concrete_equality: function[rep_stack, rep_stack -> bool] =
    (LAMBDA p1, p2 :
       (p1.pointer = p2.pointer)
         AND (FORALL i :
            (1 <= i AND i <= p1.pointer)
              IMPLIES (p1.astack(i) = p2.astack(i))))

  rep_newstack: rep_stack

  rep_new_ax: AXIOM rep_newstack.pointer = 0

  rep_push: function[rep_stack, elemtype -> rep_stack] =
    (LAMBDA p, e : p
       WITH [pointer := p.pointer + 1, astack := p.astack
             WITH [(p.pointer + 1) := e]])

  rep_pop: function[rep_stack -> rep_stack] =
    (LAMBDA p : p
       WITH [pointer :=
             IF p.pointer = 0 THEN 0 ELSE pred(p.pointer) END IF])

  rep_top: LITERAL function[rep_stack -> elemtype] ==
    (LAMBDA p :
       IF p.pointer = 0 THEN undef ELSE p.astack(p.pointer) END IF)

  rep_isnewstack: LITERAL function[rep_stack -> bool] ==
    (LAMBDA p : p.pointer = 0)

END stackrep
```

Figure 5.5: The “Stackrep” Example

```
rep_stack: TYPE = RECORD astack: linarray,
                         pointer: nat
                  END RECORD
```

introduces a *record* type; records are similar to arrays, but field access is indicated by the usual "dot" notation (e.g., `x.stack`). Update is indicated by the same `WITH` construct as arrays (e.g., `x WITH [pointer := 5]`).

The function `concrete_equality` introduces an important, but subtle point concerning the correctness of implementations. We need to show that the axioms of the `stacks` theory hold in the `stackrep` implementation. Consider the axiom

```
poppush: AXIOM pop(push(s, e)) = s
```

In the implementation, the array-pointer pair corresponding to `pop(push(s, e))` will *not* be identical to that corresponding to `s`, because the `push` operation will have (possibly) changed the value stored in the array at the location one beyond those used to represent the values stored in `s`. Since the `pop` operation will then have reduced the value of the pointer below that of this changed location, the fact that its value has changed is irrelevant to the value of the stack represented by the array-pointer combination. But the fact remains that the array-pointer combination corresponding to `pop(push(s, e))` may *not* be identical to that corresponding to `s`. The explanation, of course, is that only the part of the array below the pointer is relevant to the value of the stack it represents, and so we need a notion of equality on the array-pointer stack representation that takes this into account. This modified equality function is generally called a *concrete equality* and is specified in the present case by the definition:

```
p1, p2: VAR rep_stack

concrete_equality: function[rep_stack, rep_stack -> bool] =
  (LAMBDA p1, p2 :
     (p1.pointer = p2.pointer)
       AND (FORALL i :
           (1 <= i AND i <= p1.pointer)
             IMPLIES (p1.astack(i) = p2.astack(i))))
```

This simply specifies that two stack representations are considered to represent the same stack if their pointers have the same values, and their arrays contain the same elements up to the location indicated by the pointer.

There is another point about this declaration that needs some explanation. Until now we have been introducing interpreted constants by *literal* definitions, indicated by the symbol `==`. Such declarations cause all appearances of the defined identifier to be replaced by the associated definition—which can be undesirable for complex definitions (and for recursive definitions, which we will encounter in the next chapter). EHDM has another sort of definition that is not expanded automatically. This form uses a single `=`

symbol, and is referred to as simply a *definition* (as opposed to a *literal* definition). To clarify what sort of definition is being used, you can place the optional keyword `LITERAL` after the colon in a literal definition, and the optional keyword `DEFINITION` in the same place in an ordinary definition, thus:

```
plus1: DEFINITION function[nat -> nat] = (LAMBDA n : n+1)
```

A definition such as this is equivalent to a declaration followed by an axiom:

```
plus1: function[nat -> nat]
plus1: AXIOM sigma(n) = n + 1
```

Notice that the symbol `plus1` serves double duty as the name of both the identifier being defined and its defining equation. These two uses cannot be confused, since the former can appear only in expressions (or exporting lists), while the latter can appear only in proofs. Unlike literal definitions, (ordinary) definitions must be cited where needed in proofs.

In the present example, the functions `concrete_equality`, `rep_pop` and `rep_push` are given by definitions, `rep_top` and `rep_isnewstack` are defined literally, and `rep_newstack` is defined by an axiom. It is usually best, for pragmatic reasons associated with theorem proving, to use literal definitions only for simple constructions (remember, these will be expanded in-place everywhere). When both are feasible, definitions are generally preferable to axioms because they cannot introduce (new) inconsistencies. In the present case, the specification is entirely definitional, with the exception of the `rep_new_ax` axiom.

The next step in this analysis is to connect the abstract and the concrete levels of specification together using the mapping module `stackmap` shown in figure 5.6. Note the special association

```
=[stack] -> concrete_equality
```

that is used to indicate the interpretation for equality on the type `stack`. (If this association is omitted, then it is assumed that the interpretation for equality on `stack` is the standard equality on the type that interprets stack—i.e., `rep_stack` in this case.)

Typechecking `stackmap` will produce the mapped module `stackmap_map` shown in Figure 5.7. As we expect, this module contains mapped instances of all the axioms from the source modules. But it also contains three formulas (the first three) that we might not have expected. These are generated because we indicated that the `concrete_equality` predicate was to be used as the interpretation for equality on stacks. We must ensure that this predicate really is a reasonable equality—i.e., that it satisfies the properties of an equivalence relation. For this reason, EHDM automatically inserts appropriate instances of the reflexivity, symmetry, and transitivity into the mapped module whenever an interpretation is specified for an equality predicate.[4]

[4]To guarantee soundness, substitutivity should also be enforced, but this is not done at present.

```
stackmap: MODULE [etype: TYPE, udef: etype]

MAPPING stack_eqns[etype, udef], stacks[etype, udef]
  ONTO stackrep[etype, udef]

  stack -> rep_stack
  newstack -> rep_newstack
  pop -> rep_pop
  top -> rep_top
  push -> rep_push
  isnewstack -> rep_isnewstack
  =[stack] -> concrete_equality

END stackmap
```

Figure 5.6: The "Stackmap" Mapping Module

The trivial proof declarations automatically supplied in `stackmap_map` are inadequate to the task, so we must construct more complex proof declarations in order to discharge our proof obligations. A suitable module `stackmap_proofs` is shown in Figure 5.8.

Note that these (and all subsequent) proofs should be performed using the EHDM prover in checking mode (i.e., set the value of the variable `prmode` to `checking`). Observe that three of the proofs use a premise `nat_invariant` and a variable `nat_var` that do not appear to be defined anywhere. These are the standard names for the subtype predicate and variable associated with the built-in type `nat` (recall the description on Page 56). The built-in predecessor function `pred` on the naturals is also referenced. The relevant definitions from the "prelude" module `naturalnumbers` are:

```
nat_var: VAR nat
nat_invariant: AXIOM (FORALL nat_var: nat_var >= 0)

pred: function[nat -> nat] ==
  (LAMBDA nat_var -> nat : IF nat_var > 0 THEN nat_var - 1 ELSE 0 END IF)
```

```
stackmap_map: MODULE [etype: TYPE, udef: etype]

USING stackrep[etype, udef]

EXPORTING ALL WITH stackrep[etype, udef]

THEORY

  e: VAR etype
  s: VAR stackrep[etype, udef].rep_stack
  x1: VAR stackrep[etype, udef].rep_stack
  x2: VAR stackrep[etype, udef].rep_stack
  x3: VAR stackrep[etype, udef].rep_stack

  concrete_equality_isreflexive: FORMULA concrete_equality(x1, x1)

  concrete_equality_issymmetric: FORMULA
    concrete_equality(x1, x2) IMPLIES concrete_equality(x2, x1)

  concrete_equality_istransitive: FORMULA
    concrete_equality(x1, x2) AND concrete_equality(x2, x3)
      IMPLIES concrete_equality(x1, x3)

  isnewnew: FORMULA rep_isnewstack(rep_newstack)
  isnewpush: FORMULA NOT rep_isnewstack(rep_push(s, e))
  popnew: FORMULA concrete_equality(rep_pop(rep_newstack), rep_newstack)
  topnew: FORMULA rep_top(rep_newstack) = udef
  poppush: FORMULA concrete_equality(rep_pop(rep_push(s, e)), s)
  toppush: FORMULA rep_top(rep_push(s, e)) = e

PROOF

  concrete_equality_isreflexive_PROOF: PROVE
    concrete_equality_isreflexive

  concrete_equality_issymmetric_PROOF: PROVE
    concrete_equality_issymmetric

  concrete_equality_istransitive_PROOF: PROVE
    concrete_equality_istransitive

  isnewnew_PROOF: PROVE isnewnew
  isnewpush_PROOF: PROVE isnewpush
  popnew_PROOF: PROVE popnew
  topnew_PROOF: PROVE topnew
  poppush_PROOF: PROVE poppush
  toppush_PROOF: PROVE toppush

END stackmap_map
```

Figure 5.7: The Mapped Module “Stackmap_map”

```
stackmap_proofs: MODULE [etype: TYPE, udef: etype]

USING stackmap_map[etype, udef]

PROOF (* Requires checking mode only *)

  e: VAR etype

  s: VAR rep_stack

  ref_proof: PROVE concrete_equality_isreflexive FROM
    concrete_equality {p1 <- x1@c, p2 <- x1@c}

  sym_proof: PROVE concrete_equality_issymmetric FROM
    concrete_equality {p1 <- x1@c, p2 <- x2@c, i <- i@p2s},
    concrete_equality {p1 <- x2@c, p2 <- x1@c}

  trans_proof: PROVE concrete_equality_istransitive FROM
    concrete_equality {p1 <- x1@c, p2 <- x2@c, i <- i@p3},
    concrete_equality {p1 <- x2@c, p2 <- x3@c, i <- i@p3},
    concrete_equality {p1 <- x1@c, p2 <- x3@c}

  isnewnew_proof: PROVE isnewnew FROM rep_new_ax

  isnewpush_proof: PROVE isnewpush FROM
    rep_push {p <- s}, nat_invariant {nat_var <- (s@c).pointer}

  popnew_proof: PROVE popnew FROM
    concrete_equality {p1 <- rep_pop(rep_newstack), p2 <- rep_newstack},
    rep_new_ax,
    rep_pop {p <- rep_newstack}

  topnew_proof: PROVE topnew FROM rep_new_ax

  poppush_proof: PROVE poppush FROM
    concrete_equality {p1 <- rep_pop(rep_push(s, e)), p2 <- s},
    rep_pop {p <- rep_push(s, e)},
    rep_push {p <- s},
    nat_invariant {nat_var <- (s@c).pointer}

  toppush_proof: PROVE toppush FROM
    rep_push {p <- s}, nat_invariant {nat_var <- (s@c).pointer}

END stackmap_proofs
```

Figure 5.8: The "Stackmap_proofs" Module

Chapter 6

Some Advanced Topics

In this chapter we will meet some rather more advanced topics: recursive definitions, subtypes, type-correctness conditions, higher-order logic, and module assumptions. We'll take as our example the problem of establishing the closed-form expression for the sum of the first n natural numbers:

$$\sum_{i=1}^{n} i = \frac{n \times (n+1)}{2}$$

To begin, we need to specify the summation appearing on the left of this equation. It is possible, using higher-order logic, to specify a general summation operator, but for the present we will content ourselves with specifying a more specific function `sigma` that denotes the particular sum we are interested in. The obvious way to define `sigma` uses a recursion, something like this:

```
sigma: function[nat -> nat] ==
  (LAMBDA n : IF n = 1 THEN 1 ELSE sigma(n-1) + n END IF)
```

However, there are a couple of things wrong with this definition as it stands. First of all, a literal definition (one using `==`) such as this causes all appearances of the defined identifier to be replaced by its definition. The recursion in the definition for `sigma` would therefore lead to endless rewriting, and is prohibited in EHDM literal definitions for this reason. The obvious repair is to use an (ordinary) definition instead of a literal definition, thus:

```
sigma: DEFINITION function[nat -> nat] =
  (LAMBDA n : IF n = 1 THEN 1 ELSE sigma(n-1) + n END IF)
```

Replacing the literal definition of `sigma` by an ordinary one has dealt with the pragmatic issue of uncontrolled rewrites, but a serious logical problem remains. Careless use of recursion can introduce inconsistencies into a specification, for example:

```
nono: function[bool -> bool] = (LAMBDA b: NOT nono(b))
```

A recursively defined function is acceptable only if its "computation" terminates for all arguments. The usual way to prove termination is to establish that at each recursive call the "size" of the arguments decreases according to some well-founded relation. In EHDM, the well-founded relation `<` on `nat` is used and we require that a *measure function* be specified with every recursive definition. The measure function must have the same domain type as the recursive function being defined, but must return a value of type `nat`.[1] The name of the measure function is indicated in a `BY` clause following the recursive definition thus:

```
n: VAR nat
identity: LITERAL function[nat -> nat] == (LAMBDA n : n)

sigma: RECURSIVE function[nat -> nat] =
    (LAMBDA n : IF n = 1 THEN 1 ELSE sigma(n-1) + n END IF)
  BY identity
```

Notice that the optional keyword `RECURSIVE` may be given after the colon in order to assist readability.[2]

EHDM analyzes recursive definitions and produces formulas involving the measure function that must be proved in order to establish the well-foundedness of the definition concerned. In the present case, the formula concerned is

```
sigma_TCC1: FORMULA
  n >= 0 IMPLIES NOT (n = 1) IMPLIES identity(n) > identity(n - 1)
```

which is easily seen to be true (the `n >= 0` condition comes from the *subtype predicate* on the type `nat`, which is discussed below, and `n = 1` comes from the condition in the recursive definition itself). We will explain shortly how this formula is generated and presented to the user, but first we need to press on with our examination of the inadequacies in our basic definition for `sigma`.

Notice that the definition of `sigma` does not terminate if it is given an argument of zero—even though the well-foundedness condition `sigma_TCC1` is true. We must be missing something.

Indeed we are, and it is a check on the type-correctness of the recursive call to `sigma` appearing in its definition. The formal argument `n` to `sigma` has been declared to be of type `nat`—so we must ensure that the `n-1` in the recursive call `sigma(n-1)` is indeed of type `nat`. But what exactly is the type `nat`?

In EHDM, the natural numbers are declared as a *subtype* of the integers:

[1]Only the primitive recursive higher-order functions can be defined this way.

[2]The earlier definition of `sigma` on page 48 is syntactically unacceptable to EHDM on two grounds: a) a recursive definition must have a `BY` clause, and b) the optional keyword `DEFINITION` can only be used in nonrecursive definitions; in a recursive definition the optional keyword is `RECURSIVE`.

```
x: VAR int
nat: TYPE FROM int WITH (LAMBDA x: x >= 0)
```

This declaration is built-in to the "prelude" of standard theories, but would have the same form in a user-written module. It declares the type `nat` to be a subtype of the type `int` with defining predicate `(LAMBDA x : x >=0)`. This means that wherever an expression of type `nat` is required, it is necessary to ensure that the expression actually supplied does satisfy the subtype predicate for `nat`. In the case of the definition of `sigma`, this means that it is necessary to ensure that the `n-1` in the recursive call `sigma(n-1)` satisfies the subtype predicate on `nat` (because the argument to `sigma` has been declared to be of type `nat`). We know that `n` is of type `nat`, since it was declared so, and the constant 1 is known to be of type `nat`, so what is the type of `n-1`? There is no subtraction operator defined on the type `nat` in EHDM, but there is a subtraction operator defined in the type `int`, which is the parent type of `nat`. Consequently, the system promotes `n` and 1 to the parent type `int`, interprets the `-` sign as the subtraction operator on `int`, and concludes that `n-1` has type `int`. However, the context demands something of type `nat`. Since `nat` is known to be a subtype of `int`, and `n-1` is now known to be an `int`, this expression will also be a `nat` provided it satisfies the defining predicate for `nat`. Thus, we will have to prove `(n-1) >= 0`. Now this particular appearance of the expression `n-1` occurs in the `ELSE` part of an `IF` with condition `n=1`. Hence, we know `NOT (n = 1)` in this context. We also know that `n` is a `nat`, and so it satisfies the subtype predicate `n >= 0`. Hence, the *typecheck-consistency condition*, or `TCC` for the appearance of `n-1` in `sigma(n-1)` is:

```
sigma_TCC2: FORMULA
  n >= 0 IMPLIES NOT (n = 1) IMPLIES n - 1 >= 0
```

This is false, and so our putative recursive definition for `sigma` is not type-correct and will not be accepted by EHDM.

There are two plausible corrections that we could make. The first would redefine `sigma` so that the recursion bottoms out at 0 rather than 1:

```
sigma: RECURSIVE function[nat -> nat] =
 (LAMBDA n : IF n = 0 THEN 0 ELSE sigma(n-1) + n END IF)
BY identity
```

This is probably the best solution. For pedagogical purposes, however, we will adopt the other approach: redefining `sigma` to take a *strictly positive* integer as its argument. This is shown in Figure 6.1. We define `posint` to be a subtype of `nat` comprising just the strictly positive numbers. We then define `sigma` as before, and introduce the theorem `closed_form` that we wish to prove. Notice that the literal definition of the `identity` function has to include the return type indicator `-> nat` in its `LAMBDA` expression. If this indicator is omitted, the type computed for the `LAMBDA` expression

```
sum: MODULE

EXPORTING ALL

THEORY

  n: VAR nat

  posint: TYPE FROM nat WITH (LAMBDA n : n > 0)

  m, q, z: VAR posint

  identity: function[posint -> nat] == (LAMBDA q -> nat : q)

  sigma: RECURSIVE function[posint -> nat] =
    (LAMBDA q : IF q = 1 THEN 1 ELSE sigma(q - 1) + q END IF)
  BY identity

  square: function[nat -> nat] == (LAMBDA n : n * n)

  closed_form: THEOREM 2 * sigma(q) = square(q) + q

END sum
```

Figure 6.1: Theory Section for the “Sum” Example

will be `function[posint -> posint]` and this is type-incompatible with the declared signature of `identity` as `function[posint -> nat]`.[3] By including the return type indicator in the `LAMBDA` expression, we coerce its type to that required.

The definition of `closed_form` in Figure 6.1 also raises a few interesting points. The straightforward transliteration of the formula we gave at the beginning would lead us to write:

```
closed_form: THEOREM sigma(p) = p * (p + 1) /2
```

and this would be perfectly acceptable. Pragmatically, however, it is best to avoid division and also non-linear multiplication since these present difficulties in theorem proving (the theories concerned are undecidable). By multiplying both sides by two we avoid the division, and by expanding the product on the right hand side we can capture the nonlinear multiplication inside the function `square`, where it will be easier to deal with if it proves necessary to assist the theorem prover. (As it happens, these precautions are unnecessary in this case, but you should be aware of the issues.)

If we now typecheck the module `sum`, we will be told that the system has generated a "`TCC` module" `sum_tcc`. This module is shown in Figure 6.2.

[3]Subtyping in the present version of EHDM applies only to ground types and does not induce a subtype hierarchy on the function types.

```
sum_tcc: MODULE

USING sum

EXPORTING ALL WITH sum

THEORY

  n: VAR naturalnumber
  q: VAR posint
  p: VAR function[posint -> boolean]
  m: VAR posint

(* Existence TCC generated for posint *)

  posint_TCC1: FORMULA (EXISTS n : n > 0)

(* Subtype TCC generated for the first argument to sigma in sigma *)

  sigma_TCC1: FORMULA (NOT (q = 1)) IMPLIES (q - 1 >= 0) AND (q - 1 > 0)

(* Termination TCC generated for sigma *)

  sigma_TCC2: FORMULA (NOT (q = 1)) IMPLIES identity(q) > identity(q - 1)

(* Subtype TCC generated for the first argument to p in induction  AND
   Subtype TCC generated for the first argument to sigma in basis  AND
   Subtype TCC generated for basis_proof *)

  induction_TCC1: FORMULA (1 > 0)

(* Subtype TCC generated for the first argument to p in induction *)

  induction_TCC2: FORMULA (p(m)) AND (p(1)) IMPLIES (m + 1 > 0)

(* Subtype TCC generated for the first argument to sigma in inductive_step *)

  inductive_step_TCC1: FORMULA
    (2 * sigma(q) = square(q) + q) IMPLIES (q + 1 > 0)

(* Subtype TCC generated for ind_step_proof *)

  ind_step_proof_TCC1: FORMULA (q + 1 > 0)
```

Figure 6.2: The TCC Module “Sum_Tcc” (continues)

```
PROOF

  posint_TCC1_PROOF: PROVE posint_TCC1

  sigma_TCC1_PROOF: PROVE sigma_TCC1

  sigma_TCC2_PROOF: PROVE sigma_TCC2

  induction_TCC1_PROOF: PROVE induction_TCC1

  induction_TCC2_PROOF: PROVE induction_TCC2

  inductive_step_TCC1_PROOF: PROVE inductive_step_TCC1

  ind_step_proof_TCC1_PROOF: PROVE ind_step_proof_TCC1

END sum_tcc
```

Figure 6.2: The TCC Module "Sum_Tcc"

TCC modules contain typecheck-consistency conditions: formulas that must be proven in order to ensure the soundness of their parent modules. TCC modules are protected from modification just like MAP modules, but the relationship between a TCC module and its parent is more intimate than that between a MAP module and its parent. A MAP module is an auxiliary to its parent mapping module; the meaning of the mapping module is not affected by the presence or absence of its MAP module, nor by whether its proofs have been performed successfully. A module having a TCC module, however, is not fully admitted into a specification unless its TCC module is present and all its formulas are proven—since these are necessary to ensure the soundness of the parent module. The EHDM system permits users a certain flexibility in choosing when to attack the formulas in a TCC module: the system insists on the TCC module being available and unaltered whenever any operation is performed that depends on the semantics of its parent module, but the check that the formulas in the TCC have been proved is delayed until the user invokes a proof-chain analysis that involves formulas from the parent module.

The TCCs shown in Figure 6.2 include two similar to those we saw earlier, and a third one that is new. This is an *existence* TCC for the subtype posint; whenever a subtype is introduced, it is necessary to prove that the type has a member, since the standard rules for reasoning with quantified formulas depend on the nonemptiness of the ranges of quantification.

Notice that it is necessary for a parent module to export the types and constants that appear in its TCC module—if a TCC module fails to typecheck, the probable explanation is that you forgot to export the necessary identifiers from its parent module. Also notice in Figure 6.2 that trivial proof declarations (i.e., declarations without premises or substitutions) for the formulas are generated automatically in the TCC module itself. These often suffice, but do not do so for the existence TCC here.[4] Consequently, we must provide the additional module `sum_tcc_proofs` shown in Figure 6.3 that supplies the necessary proof. The presence of multiple proof declarations for a single formula (as here for `posint_TCC1`) is not a problem; the proof-chain analyzer will always prefer a successful proof over an unsuccessful one.

```
sum_tcc_proofs: MODULE

PROOF

USING sum_tcc

  posint_TCC1_PROOF: PROVE posint_TCC1 {n <- 1}

END sum_tcc_proofs
```

Figure 6.3: The "Sum_Tcc_Proofs" Proof Module

TCC modules are generated in three circumstances: when subtypes are used, when a definition is recursive, and when a state invariant is specified. The third of these possibilities is not discussed here. For subtypes, the TCC's are of two types: the nonemptiness TCC, and those needed to ensure type-correctness in places where a coercion from the parent to the subtype is required. The termination TCC for a recursive definition has already been described. Note that the proof of a termination TCC must not use, either directly or indirectly, the recursive function concerned. This requirement is enforced by the EHDM proof-chain checker.

The EHDM system automatically inserts instances of subtype predicates in proofs when necessary. Occasionally, however, it is necessary to cite the necessary predicate explicitly. This is done using standard names that are introduced automatically whenever a subtype is declared. For the declaration

```
    sub:  TYPE FROM t WITH (LAMBDA x:  inv(x))
```

[4]Trivial declarations never can suffice for existence TCCs, since explicit substitutions will be necessary to provide witnesses to the existential quantifications.

the system automatically generates a variable of type `sub` and an axiom that expresses the invariant property:

```
sub_var:  VAR sub
sub_invariant:  AXIOM (FORALL sub_var:  inv(sub_var))
```

The names of the variable and axiom are always formed with the canonical suffixes `_var` and `_invariant`, respectively.

Now that we have finally succeeded in defining our problem, we can proceed to solve it. Since `sigma` is defined recursively, the natural way to prove the `closed_form` theorem is by induction. The specification language of EHDM contains elements of higher-order logic; that is, it allows quantification over function types. This means that we can define induction axioms directly in EHDM. For example, the induction axiom we need here can be written as follows:

```
n, m: VAR posint

p: VAR function[posint -> bool]

induction: AXIOM
  (FORALL p: p(1) AND (FORALL m : p(m) IMPLIES p(m + 1))
    IMPLIES (FORALL n : p(n)))
```

We will worry about justifying this later, but first let's look at what it says and consider how to use it. This specification fragment introduces `p` as a predicate *variable* and then asserts that for any such predicate, if `p` is true for 1, and for any `posint m`, `p` true for `m` implies `p` true for `m+1`, then we may conclude `p` is true for all `n` of type `posint`. Our goal is to establish `closed_form` using the `induction` axiom—which means that we will need to substitute for `p` in `induction` the predicate implicitly appearing in `closed_form`. That is, our proof will contain a premise like:

```
induction {p <- (LAMBDA q : 2 * sigma(q) = square(q) + q)}
```

Looking at the `induction` axiom, we can see that in order to obtain the result we want, we will have to establish the two conjuncts in its antecedent. We can break these out in instantiated form as the following two lemmas:

```
basis: LEMMA 2 * sigma(1) = square(1) + 1

inductive_step: LEMMA
  2 * sigma(q) = square(q) + q
    IMPLIES 2 * sigma(q + 1) = square(q + 1) + q + 1
```

In this way we construct the full form of the `sum` module shown in Figure 6.4. This form of the module generates a `TCC` module with 7 formulas; all but the last of which require nothing beyond the automatically supplied proof declarations. The remaining TCC is the one we saw earlier that asserts the nonemptiness of the posint type.

At this point, we recommend trying out the proofs in the modules sum, sum_tcc, and sum_tcc_proofs and then invoking the proof-chain checker in verbose mode on the formula closed_form (point to the formula and do M-X fs; when asked for the name of the module at the top of the proof tree, respond with sum_tcc_proofs). Notice how the proof-chain checker tracks the dependencies on TCCs. In particular, notice that it checks that a recursively defined function is not used in its own termination proof.

We have now succeeded in proving the closed_form theorem, but to do so we introduced an induction axiom in a rather ad-hoc manner. It is always necessary to scrutinize axioms with great care, since they can potentially introduce inconsistencies. Often, it is preferable to axiomatize a standard and well-understood theory, and then derive the particular results needed as lemmas, rather than assert those results directly as axioms. The benefits of the recommended approach are that you can refer to textbooks for appropriate formulations of the theories of interest, and that you will build up a library of generally useful specifications that can be reused many times.

In the present case, we have introduced an induction axiom that is specific to the type posint, and that is not justified by reference to external authority. We can remedy both deficiencies by deriving our specific induction scheme as an instance of a more general one. The most general induction scheme is that known as Noetherian[5] or well-founded induction. If < is a well-founded relation over a type dom, then Noetherian induction can be stated as follows [16, page 6]

```
p: VAR function[dom -> bool]

d, d1, d2: VAR dom

general_induction: AXIOM
  (FORALL p :
    (FORALL d1 :
        (FORALL d2 : d2 < d1 IMPLIES p(d2)) IMPLIES p(d1))
      IMPLIES (FORALL d : p(d)))
```

There are some subtleties in this definition. For example, if d1 is a minimal element (i.e., if there is no d2 such that d2 < d1), then the antecedent to the first implication is false, and the implication is therefore true. Now this implication is the antecedent to the second—and so it will be necessary to establish that p holds for such a minimal element.

Since this axiom applies to any type dom and well-founded relation <, we should enclose it in a module having these as parameters. A suitable declaration would be:

[5] Named after the German mathematician Emmy Noether; an alternative spelling of her name is Nöther; we use the simple form to accommodate the limited character set of EHDM.

```
sum: MODULE

EXPORTING ALL

THEORY

  n: VAR nat

  posint: TYPE FROM nat WITH (LAMBDA n : n > 0)

  m, q, z: VAR posint

  identity: function[posint -> nat] == (LAMBDA q -> nat : q)

  sigma: RECURSIVE function[posint -> nat] =
    (LAMBDA q : IF q = 1 THEN 1 ELSE sigma(q - 1) + q END IF)
  BY identity

  square: function[nat -> nat] == (LAMBDA n : n * n)

  closed_form: THEOREM 2 * sigma(q) = square(q) + q

  p: VAR function[posint -> bool]

  induction: AXIOM (FORALL p :
       p(1) AND (FORALL m : p(m) IMPLIES p(m + 1))
         IMPLIES (FORALL q : p(q)))

PROOF

  basis: LEMMA 2 * sigma(1) = square(1) + 1

  basis_proof: PROVE basis FROM sigma {q <- 1}

  inductive_step: LEMMA
    2 * sigma(q) = square(q) + q
      IMPLIES 2 * sigma(q + 1) = square(q + 1) + q + 1

  ind_step_proof: PROVE inductive_step FROM sigma {q <- q + 1}

  the_proof: PROVE closed_form FROM
    induction {p <- (LAMBDA z : 2 * sigma(z) = square(z) + z)},
    basis,
    inductive_step {q <- m@p1}

END sum
```

Figure 6.4: The Full “Sum” Example

```
noetherian: MODULE [dom: TYPE, <: function[dom, dom -> bool]]
```

Notice that EHDM allows the standard arithmetic relations and operators to be *overloaded* with new definitions. The EHDM typechecker resolves such uses on the basis of the argument types supplied.

The only thing wrong with this specification is that we have not taken care of the requirement that the relation `<` be *well-founded*: there is nothing to stop us using instantiations like

```
noetherian[int, <]
```

(here `<` is the standard less-than relation on the integers) and deriving contradictions—since less-than is not well-founded on the integers.

We need some way to ensure that any use of the `noetherian` module respects the requirement that the relation in question should be well-founded. This is accomplished in EHDM by the `ASSUMING` clause. This clause may precede the `THEORY` part of any parameterized module and serves to introduce formulas that are *assumed* inside the module and that must be *discharged* whenever the module is used. In the present case, the `ASSUMING` clause needs to state the requirement that the relation supplied as the second argument to the module be well-founded. It is a little tedious to give a full specification for well-foundedness (it requires infinite sequences), so we will deal only with a simple form of well-foundedness corresponding to the less-than relation on the natural numbers. It is well-known that a relation can be shown to be well-founded by establishing a correspondence with another relation that is already known to be well-founded [16, pp 8–9]. Now the ordinary less-than relation is known to be well-founded on the naturals, so that we can state the requirement for well-foundedness in terms of the existence of a *measure function*. This is accomplished in the full `Noetherian` module shown in Figure 6.5. (Notice we have dropped the outermost quantifier on `p` in the induction axiom).

Now let's consider how we should use the `Noetherian` module to provide the induction needed in the `sum` module. There are two ways to proceed: either we can try and prove `closed_form` directly from `general_induction`, or we can use `general_induction` to prove our original `induction` axiom (which would then become a lemma). The second course of action is likely to be easier, and we could also then place the `induction` lemma in a module of its own and make it generally available as a proven, simplified induction scheme for those who do not require the full power of `general_induction`. But perhaps the simple `induction` scheme is a little *too* simple for this purpose; it would be better to generalize it a little so that it is less specific to the `posint` type and the "step by 1" inductive step. A suitably generalized module is shown in Figure 6.6.

The `simple_induction` module introduces an induction scheme that is suitable for those cases where we have a type `dom` with a single base element, and a "step function"

```
noetherian: MODULE [dom: TYPE, <: function[dom, dom -> bool]]

ASSUMING

  measure: VAR function[dom -> nat]

  a, b: VAR dom

  well_founded: FORMULA
    (EXISTS measure : a < b IMPLIES measure(a) < measure(b))

THEORY

  p: VAR function[dom -> bool]

  d, d1, d2: VAR dom

  general_induction: AXIOM
    (FORALL d1 :
         (FORALL d2 : d2 < d1 IMPLIES p(d2)) IMPLIES p(d1))
      IMPLIES (FORALL d : p(d))

END noetherian
```

Figure 6.5: The "Noetherian" Example

```
simple_induction: MODULE [dom: TYPE, base_elem: dom,
                          step: function[dom -> dom]]

ASSUMING

  x, y: VAR dom

  measure: VAR function[dom -> nat]

  well_ordered: FORMULA (EXISTS measure : measure(x) < measure(step(x)))

  reachability: FORMULA x /= base_elem IFF (EXISTS y : step(y) = x)

THEORY

  d1, d2: VAR dom

  lessp: function[dom, dom -> bool] == (LAMBDA d1, d2 : d2 = step(d1))

  n, m: VAR dom

  p: VAR function[dom -> bool]

  induction: THEOREM
    (p(base_elem) AND (FORALL m : p(m) IMPLIES p(step(m))))
      IMPLIES p(n)

PROOF

USING noetherian[dom, lessp]

  ind_proof: PROVE induction {m <- y@p2} FROM
    general_induction {d <- n, d2 <- m}, reachability {x <- d1@p1}

  discharge: PROVE well_founded {measure <- measure@p1} FROM
    well_ordered {x <- a@c}

END simple_induction
```

Figure 6.6: The “Simple_Induction” Example

with the property that any member of the type `dom` can be reached by successive applications of the step function, starting from the base element. Notice that this new module also has an assuming clause of it own, and that it discharges the assumption on the Noetherian module.

Now we are finally in a position to perform an "elegant" proof of the `closed_form` theorem. This is given in the `alt_sum` module shown in Figure 6.7

The module synonym declaration on the third line of the proof clause of this module is worth a mention. Observe that it is necessary to construct the `next` function before the proper instance of the `simple_induction` module can be cited. Since the `USING` clause must come before any declarations, we have a potential chicken and egg situation. In this particular case, we could avoid it by defining the `next` function in the `THEORY` section, ahead of the `USING` clause in the `PROOF` section. Sometimes, however, this is not possible (as, for example, when the module instance is needed in the `THEORY` section itself). There are four solutions to this difficulty.

- Cite the generic (unparameterized) instance of the module in the `using` clause, and use the fully-qualified versions of any names taken from that module—e.g., `induction[posint, 1, next]`.

- A variant on the above: cite the generic instance in the `USING` clause, then introduce a module synonym for the instantiated instance of the module as soon as the other necessary terms have been defined. The module synonym can then be used to abbreviate the fully qualified names (e.g., `instance.induction`) or, if there is only one such module synonym, it will automatically establish the correct instances of names from the module concerned. This is the approach illustrated in Figure 6.7.

- Define the necessary terms in a separate module that exports their names. Cite this module earlier in the `USING` clause than the module whose instance needs to use those names.

- Use a nested module. This is discouraged, since nested modules will be removed from the language in Version 6.

Try out the proofs in `alt_sum` (you will have to supply a module containing a proof for one of the TCCs generated by `alt_sum`), and then invoke the `formulastatus` command in verbose mode on the formula `closed_form`. Notice how the proof-chain checker tracks the obligations to discharge assumptions whenever a module having an assuming clause is used.

LaTeX-Printing Specifications Specifications are not intended solely for mechanical analysis; humans are expected to read and review them as well. EHDM provides a LaTeX-print command that generates LaTeX source files suitable for use in reports or viewgraph

```
alt_sum: MODULE

EXPORTING ALL

THEORY

  n: VAR nat
  posint: TYPE FROM nat WITH (LAMBDA n : n > 0)
  m, q, z: VAR posint

  identity: function[posint -> nat] == (LAMBDA q -> nat : q)

  sigma: RECURSIVE function[posint -> nat] =
      (LAMBDA q : IF q = 1 THEN 1 ELSE sigma(q - 1) + q END IF)
    BY identity

  square: function[nat -> nat] == (LAMBDA n : n * n)

  closed_form: THEOREM 2 * sigma(q) = square(q) + q

PROOF  USING simple_induction

  next: function[posint -> posint] == (LAMBDA q -> posint : q+1)

  instance: MODULE IS simple_induction[posint, 1, next]

  basis: LEMMA 2 * sigma(1) = square(1) + 1

  basis_proof: PROVE basis FROM sigma {q <- 1}

  inductive_step: LEMMA
    2 * sigma(q) = square(q) + q
      IMPLIES 2 * sigma(q + 1) = square(q + 1) + q + 1

  ind_step_proof: PROVE inductive_step FROM sigma {q <- q + 1}

  the_proof: PROVE closed_form FROM
    induction {n <- q, p <- (LAMBDA z : 2 * sigma(z) = square(z) + z)},
    basis,
    inductive_step {q <- m@p1}

  discharge_well_ordered: PROVE well_ordered {measure <- identity}

  discharge_reachability: PROVE
          reachability {y <-if x>1 then x-1 else 1 end if}

END alt_sum
```

Figure 6.7: The “Alternative Sum” Example

presentations. This facility pretty-prints the specification, substituting standard mathematical notation for keywords and special symbols (so that `AND`, for example, becomes $\wedge$). In addition, it substitutes Greek characters for their names (e.g., `delta` becomes δ). To use this facility, go to the `sum` module, and type `M-X latex-print`. After a few moments, the latex file `sum.tex` will be generated, along with the file `ehdm-modules.tex`. The `sum.tex` file is expected to be `\input` to another file, and cannot be LaTeXed on its own. The `ehdm-modules.tex` file is a complete skeleton file that `\input`s the file just generated: to see the LaTeX-printed module, simply LaTeX the file `ehdm-modules.tex` and print (or preview) the result. The result in this case should look something like Figure 6.8, (the exact result depends on the settings of the EHDM variables controlling the prettyprinter).

The default substitutions that are applied when LaTeX-printing an EHDM module can be augmented or overridden by user-supplied substitutions. The details require a good understanding of LaTeX and we won't go into them here (see [6]), but Figure 6.9 shows the theory part of the `sum` module, when $\mathcal{N}$, $\mathcal{N}^+$, $(\star 1)^2$ and $\sum_{i=1}^{\star 1} i$ have been specified as the substitutions for `nat`, `posint`, `square` and `sigma`, respectively.

sum: **Module**

Exporting all

Theory

n: **Var** nat
posint: **Type from** nat **with** $(\lambda\, n : n > 0)$
m, q, z: **Var** posint
identity: function[posint $\rightarrow$ nat] == $(\lambda\, q \rightarrow \text{nat} : q)$
σ : **Recursive** function[posint $\rightarrow$ nat] =
  $(\lambda\, q :$ **if** $q = 1$ **then** 1 **else** $\sigma(q-1) + q$ **end if**) **by** identity
square: function[nat $\rightarrow$ nat] == $(\lambda\, n : n * n)$

closed_form: **Theorem** $2 * \sigma(q) = \text{square}(q) + q$

p: **Var** function[posint $\rightarrow$ bool]

induction: **Axiom** $(\forall\, p : p(1) \wedge (\forall\, m : p(m) \supset p(m+1)) \supset (\forall\, q : p(q)))$

Proof

basis: **Lemma** $2 * \sigma(1) = \text{square}(1) + 1$

basis_proof: **Prove** basis **from** σ $\{q \leftarrow 1\}$

inductive_step: **Lemma**
  $2 * \sigma(q) = \text{square}(q) + q \supset 2 * \sigma(q+1) = \text{square}(q+1) + q + 1$

ind_step_proof: **Prove** inductive_step **from** σ $\{q \leftarrow q + 1\}$

the_proof: **Prove** closed_form **from**
  induction $\{p \leftarrow (\lambda\, z : 2 * \sigma(z) = \text{square}(z) + z)\}$,
  basis,
  inductive_step $\{q \leftarrow m@p1\}$

End sum

Figure 6.8: LaTeX-printed Theory part of "Sum" Module

sum: **Module**

Exporting all

Theory

n: **Var** $\mathcal{N}$
$\mathcal{N}^+$: **Type from** $\mathcal{N}$ **with** $(\lambda\, n : n > 0)$
m, q, z: **Var** $\mathcal{N}^+$
identity: function$[\mathcal{N}^+ \rightarrow \mathcal{N}] == (\lambda\, q \rightarrow \mathcal{N} : q)$
$\sum_{i=1}^{\star 1} i$: **Recursive** function$[\mathcal{N}^+ \rightarrow \mathcal{N}] =$
$(\lambda\, q :$ **if** $q = 1$ **then** 1 **else** $\sum_{i=1}^{q-1} i + q$ **end if**) **by** identity
$(\star 1)^2$: function$[\mathcal{N} \rightarrow \mathcal{N}] == (\lambda\, n : n * n)$

closed_form: **Theorem** $2 * \sum_{i=1}^{q} i = (q)^2 + q$

p: **Var** function$[\mathcal{N}^+ \rightarrow \text{bool}]$

induction: **Axiom** $(\forall\, p : p(1) \wedge (\forall\, m : p(m) \supset p(m+1)) \supset (\forall\, q : p(q)))$

End sum

Figure 6.9: LaTeX-printed Theory part of "Sum" Module after Substitutions

Chapter 7

Specifying and Reasoning about State-dependent Behavior

In this chapter, we look at those aspects of the EHDM system that deal with abstract programs. By "program" we mean a program in a procedural programming language, such as Ada or Pascal.[1] A main characteristic of those languages is that computation is expressed by means of program variables and manipulation of their values: values are stored in program variables by assignment statements and subsequently retrieved for use in further computation. We can view the collection of program variables and their values at a given point in a computation as the *state* of the computation, and view computations themselves as manipulations or transformations of states. It is these notions of state and state transformation that are modeled in EHDM by corresponding concepts of *state object* and *operation*.

7.1 State Objects

A program variable in, say, Ada is an object that can hold values of a certain type; for example,

```
x:  integer;
```

is the declaration of a variable that can hold values of type `integer`. If the value of the variable is meant to be constant, one uses a declaration like

```
c:  CONSTANT integer = 10;
```

In EHDM, the corresponding declaration of the constant is

```
c:  integer == 10
```

[1]We here use "procedural language" in contrast to "functional language."

You might think that the corresponding EHDM declaration for the program variable `x` is

```
x:  VAR integer
```

However, such a declaration introduces a *logical* variable, that is, the sort of variable used in quantified formulas and lambda expressions. What we need here is a completely different kind of "variable." In EHDM, we have a special class of types, called *state types*, for objects whose values are state-dependent and which we call *state objects*. We can form the state type for an arbitrary type. For example, `state[int]` is the type of integer state objects, so the EHDM declaration for `x` is

```
x:  state[int]
```

We will see in a moment that state objects are used in EHDM just like program variables in Ada. The use of the term "variable" in programming languages is rather unfortunate because it conflicts with its use in logic. The two concepts are quite distinct, and maintaining the distinction is very important (even though not all specification languages make it). In order to avoid confusion, we adopted the term "state objects," which is meant to suggest the connection with state dependence.

7.2 Operations

Now that we have state objects, how do we associate values with them? In Ada, we assign a value to `x` by a statement like

```
x := 10;
```

or

```
x := x + 2;
```

In EHDM, we can do exactly the same. In fact, EHDM includes a number of constructs which together form a simple "programming language," with assignments and various control structures; because it uses Ada syntax, this sublanguage looks like a fragment of the Ada statement language.

However, when developing specifications, we are usually not so much interested in actual code, at least not in the beginning. Instead, we want to specify and reason about state transformations in a more abstract setting. A state transformation is expressed in EHDM by an *operation*. An operation is simply an element of the primitive type `operation`. An operation can have parameters. You can think of an operation as an abstract form of a *procedure* in Ada—abstract in the sense that no body has been given for it.

There are several ways to specify the meaning of an operation. We can give an operation an interpretation by declaring that it stands for a piece of "code" in the sublanguage just mentioned. This is similar to declaring a constant or giving a function interpretation; we will come back to this later.

More interesting is an abstract description of the effect of an operation. Remember that the meaning of an operation is a state transformation. "State" is expressed by one or more state objects, and a state is modified by changing the value(s) of the state object(s). So the effect of an operation can be described by specifying the relationship between the states before and after modification. Such relationships are conveniently expressed by *Hoare formulas.*[2]

Let us look at a few simple examples to make all this more concrete. Consider the following declarations:

```
sqrt_op:  operation[number,state[number]]
n:  VAR number
r:  VAR state[number]
sqrt_ax:  AXIOM {n > 0} sqrt_op(n,r) {r*r = n}
```

The first declaration introduces an operation `sqrt_op` that takes two parameters: a number and a state object of type `state[number]`. The distinction between these two kinds of parameters is essentially the same as between *in* and *in out* parameters in Ada (or between value parameters and variable parameters in Pascal): with the first kind, values are passed to the operation; parameters of the second kind can have their values modified by the operation, providing a means for passing values out of the operation.

The formula `sqrt_ax` is a Hoare formula. It consists of three parts: a *precondition*, an operation part, and a *postcondition*. The operation part is some expression of type `operation`; in the example it is a "call" to `sqrt_op`—think of it as a procedure call. The precondition and the postcondition are expressions of type `boolean` that typically refer to objects appearing in the operation part. Both describe states abstractly, that is, they do not specify *particular* states, but *classes* of states. For example, `n > 0` means "some state in which `n > 0` is true." The meaning of the Hoare formula `sqrt_ax` is then

> A state satisfying the precondition `n > 0` is transformed by the operation `sqrt_op(n,r)` into a state satisfying the postcondition `r*r = n`.

Of course, `n` itself is merely a variable and thus does not depend on states; but the formula can be instantiated in a way that `n` is replaced by an expression that does involve state objects. Hoare formulas are treated much like other formulas; in particular, they can be quantified and, just as in first-order formulas, universal quantifiers can be dropped at the outermost level. Thus `sqrt_ax` is the same as

```
sqrt_ax:  AXIOM
   (FORALL n, r:  {n > 0} sqrt_op(n,r) {r*r = n})
```

[2]Hoare formulas are named after C. A. R. Hoare, who introduced the basic concept and also the logic for reasoning with them [10].

Note that sqrt_ax does not say anything about the effect of sqrt_op when the precondition is not satisfied. It is quite common that a Hoare formula gives only a partial specification of an operation.

Now, let us look at another example.

```
swap:  operation[state[int],state[int]]
x, y:  VAR state[int]
a, b:  VAR int
swap_1:  AXIOM
   {x = a AND y = b} swap(x,y) {x = b AND y = a}
```

The operation swap is declared as taking two state objects of type state[int] as arguments. The intended meaning of the operation swap is that it swaps the values of its arguments: in the new state, x should have the old value of y and y the old value of x. In the formula swap_1, this is expressed as "a state in which x has some value a and y some value b is transformed by swap(x,y) into a state in which x has the value b and y the value a." In this formulation the link between old and new values is rather indirect; we can express it more conspicuously by using *primed* names. An occurrence of x in a postcondition (or precondition) stands for the *current* value of the state object, whereas x' denotes the *previous* value of x, more precisely, the value of x in the state associated with the precondition. (Use of primed names of state objects is restricted to postconditions of Hoare formulas because only there do they have a meaning.) Thus, formula swap_1 can be restated as

```
swap_2:  AXIOM
   {true} swap(x,y) {x = y' AND y = x'}
```

(The trivial precondition, true, can be interpreted as "any state.") The formula swap_2 is preferred over swap_1 since it avoids the auxiliary variables.

Changes Clauses. A Hoare formula specifies how an operation transforms a state. However, as we pointed out before, a Hoare formula need not be a *complete* specification of an operation; it need not describe *all* possible effects of the operation. Thus, when we see a formula like swap_2, we cannot assume that no other state object is affected by swap. EHDM provides a construct that lets you specify explicitly which state objects can possibly be affected by an operation; we call this construct a CHANGES clause. A CHANGES clause may be used in a separate axiom or formula declaration; alternatively, it can be attached to the declaration of an operation with the reserved word WHERE. For example, we would specify for the operation swap:

```
swap_ch:  AXIOM swap(x,y) CHANGES x, y
```

or, alternatively,

```
swap:  operation[state[int],state[int]]
   WHERE swap(x,y) CHANGES x, y
```

Either form says that the operation `swap` may modify the values of its arguments and has no effect on any other state object. (In either form, the variables `x` and `y` must have been declared beforehand.) A `CHANGES` clause permits users to delimit the possible extent of effects of an operation without having to be specific about *how* the state is changed. The most important aspect of a `CHANGES` clause is the complement of what it states, namely that every state object not listed in the clause remains unchanged.[3] In proofs with Hoare formulas—which will be discussed shortly—it is often necessary to state such invariance by providing `CHANGES` clauses for abstract operations.

[3]Such statements are often referred to as *frame axioms* or *frame statements*.

```
stackops: MODULE [t: TYPE, undefined: t]

  USING stacks[t, undefined]
  EXPORTING ALL

THEORY

  v: VAR t
  x: VAR state[t]

  stack_obj: state[stack]

  init_op: operation
WHERE init_op CHANGES stack_obj

  push_op: operation [t]
WHERE push_op(v) CHANGES stack_obj

  pop_op:  operation [state[t]]
WHERE pop_op(x) CHANGES stack_obj, x

  init_ax:  AXIOM {true} init_op {stack_obj = newstack}

  push_ax: AXIOM {true} push_op(v) {stack_obj = push(stack_obj', v)}

  pop_ax: AXIOM
    {true}
      pop_op(x)
    {x = top(stack_obj') AND stack_obj = pop(stack_obj')}
```

Figure 7.1: The “Stackops” Example (continues)

7.3 State Machines

So far we have talked only about individual operations and Hoare formulas. We now look at how they are used in the larger context of specifying an abstract state machine. We use the simple example of a stack module, displayed in Figure 7.1. It defines an object, `stack_obj`, and operations on that object. Each of the operations modifies the stack object. The Hoare formulas specify the effects of the operations; in essence, they say that the operations are procedural counterparts of the stack functions of module `stacks`.

Each operation is specified by a Hoare formula and a `CHANGES` clause. The operation `pop_op` returns the top value of the stack in its argument and also pops the stack, thus it combines the functionality of both `top` and `pop`.

7.4 Proving Properties of Operations

Now let us prove some simple properties about operations. Properties are stated (mainly) by Hoare formulas, so this means that we have to prove Hoare formulas. EHDM provides a special theorem prover component, usually called the *Hoare Sentence Prover* (HSP), which has the semantics of Hoare formulas and the operation constructs built into it.

The proof part of module `stackops` (Figure 7.1) declares three lemmas and their proofs. The first thing you may notice in the proof declarations is that the reserved word `PROVE` is replaced by `VERIFY`. The latter tells the EHDM theorem prover to invoke the Hoare Sentence Prover component, rather than the prover you have been using so far (the ground prover or the instantiator).

The first lemma, `L1`, asserts that a push operation followed by a pop operation leaves the stack object unchanged. You may think of it as a test whether the specifications of the operations are meaningful: at the least, the basic properties of stack structures defined in module `stacks` should be preserved. It is also about the simplest Hoare formula that mentions more than one operation. How does the prover prove `L1`? The HSP uses a forward strategy that reduces a Hoare formula proof to a sequence of proofs in predicate logic. Recall the basic meaning of a Hoare formula, which we can rephrase like this: Assuming that the precondition is true in the state at the start, the postcondition will be true for the final state after a "hypothetical execution" of the operation part. The HSP simulates this process exactly:

Assume that the precondition is true, i.e., assume the precondition. In our case, we get to assume `true`—big deal! The first operation is `push_op(v)`. The premise `push_ax` tells us that if its precondition is true then we can assert its postcondition for the state corresponding to the ";" in `L1`. If `push_ax` had a non-trivial precondition, the HSP would generate a subproof to ascertain that the condition is true. Here, nothing needs to be done, and we assert

```
PROOF  (* checking mode *)

  USING associativity
  w: VAR t
  y, z: VAR state[t]

  L1: LEMMA
{true} push_op(v); pop_op(x) {stack_obj = stack_obj'}

  pr1: VERIFY L1 FROM
    push_ax, pop_ax,
    poppush@C' {s <- stack_obj, e <- v}

  op: VAR operation

  L2: LEMMA
  {true} op {stack_obj = stack_obj'}
        IMPLIES
          {true}
            push_op(x); op; pop_op(x)
          {stack_obj = stack_obj' AND x=x'}

  pr2: VERIFY L2 FROM
      push_ax, pop_ax,
      poppush@C' {s <- stack_obj, e <- x},
      toppush@C' {s <- stack_obj, e <- x}

  L3: LEMMA
  {true}
    push_op(y); push_op(w); pop_op(z); pop_op(y)
  {stack_obj = stack_obj' AND y = y'}

  pr3: PROVE L3 FROM
     L2 {op <- (push_op(w); pop_op(z)), x <- y},
     L1 {v <- w, x <- z},
     assoc {op1 <- push_op(w), op2 <- pop_op(z), op3 <- pop_op(y)}

END stackops
```

Figure 7.1: The “Stackops” Example

```
stack_obj₁ = push(stack_obj₀, v)
```

The indices of `stack_obj` indicate the state change.

Next, we make use of the premise `pop_ax`. Again, no subproof is needed, and we get to assert

```
x = top(stack_obj₁) AND stack_obj₂ = pop(stack_obj₁)
```

If we combine this with the first assertion, we get

```
x = top(push(stack_obj₀, v)) AND
stack_obj₂ = pop(push(stack_obj₀, v))
```

The formula describes what we have derived to be true for the final state. The last step is to prove that this assertion implies the postcondition of `L1`. This is where the last premise comes into play. Formula `poppush`, from the module `stack_eqns` (see Figure 5.2) is a first-order formula. If its instantiation is to involve a state object, we need to indicate at what point its value is to be used. This is done by instantiating the formula with a qualified state object: The qualification is syntactically very similar to the `@`-suffixes used in ordinary proofs, but it has a very different meaning: here, a suffix refers to either the postcondition or the precondition of a Hoare formula, where the formula itself is indicated by a `C` (for conclusion) or a `Pn` (for premise *n*). The precondition of a formula is indicated by a prime ('), and the postcondition by the lack of a prime. Thus, `stack_obj@C'` means "the value of `stack_obj` in the state associated with the precondition of the conclusion," which we already gave the name `stack_obj`$_0$. With the help of this instantiation of `poppush`, the predicate logic prover can deduce that the postcondition of `L1` implies the assertion for the final state.

The user does not see anything of this process, but it is important to understand the basic principle. The system writes out a trace of all the subproofs generated by the HSP unless the variable `prtrace` is set to `no`. The HSP is invoked by the same commands as the ordinary prover; the reserved word `VERIFY` tells the system which prover component to use. The trace is written into the `*ehdm-info*` buffer; try out a proof and look at the subproofs. If the proof fails, the subproofs trace usually reveals where a subproof did not succeed. There is also a special command `vcg` that invokes the HSP just to generate the subproofs without actually submitting them to the predicate logic prover; this can be quite helpful in figuring out what kind of predicate-logic premises are required for a Hoare formula proof.

Of course, the reduction process is much more involved for more complex formulas. For example, case splits are generated for conditional operations, and special rules apply to loops. However, the basic principle of forward reduction of the operation part and accumulation of assertions remains the same.

Lemma `L2` shows a formula that has Hoare formulas embedded in boolean structure. As in all formulas, the free variable `op` is universally quantified. The proof of `L2` proceeds essentially like the one described above: the conclusion of the implication is proved using the antecedent as the specification of `op`.

Lemma `L3` is a slight generalization of lemma `L1`. We use it here to demonstrate how ordinary first-order proof techniques can be used with operations and Hoare formulas. Remember that Hoare formulas are essentially just predicates. The ground prover treats them as uninterpreted predicates. The proof `pr3` is declared with the reserved word `PROVE`, thus it invokes the ordinary prover. The proof is based on simple equational reasoning, using a lemma—displayed in Figure 7.2—that states associativity of operation composition. (Composition is expressed by "`;`" as an infix operator on operations; the ground prover has no built-in knowledge about this operator.) Note also how lemma `L2` is used as a premise: the operation variable `op` is instantiated to the inner composition of operations.

```
associativity: MODULE
THEORY
  op1, op2, op3: VAR operation
  assoc: LEMMA ((op1; op2); op3) = (op1; (op2; op3))
END associativity
```

Figure 7.2: Associativity of ";"

7.5 Implementing the Stack Operations

The module `stackops` specifies the stack operations abstractly. As a next step, we give them interpretations that look more like real program code.

```
stackops2: MODULE [etype: TYPE, undef: etype]

USING stackrep[etype, undef], makrec[etype, undef]

EXPORTING init_op, push_op, pop_op

THEORY

  v: VAR etype
  x: VAR state[etype]
  istack: state[linarray]
  index: state[nat]
  init_op: operation == BEGIN index := 0 END
  push_op: operation[etype]

  push_def: FORMULA
    push_op(v) = BEGIN index := index + 1;
          istack(index) := v
          END

  pop_op: operation[state[etype]]

  pop_def: FORMULA pop_op(x)
      = BEGIN
          IF index = 0
            THEN x := undef
            ELSE x := istack(index);
            index := pred(index)
            END IF
          END

  init_lm: LEMMA {true} init_op {index = 0}

  push_lm: LEMMA {true}
              push_op(v)
              {istack = (rep_push(rec(istack', index'), v)).astack
                AND index = (rep_push(rec(istack', index'), v)).pointer}

  pop_lm: LEMMA {true}
              pop_op(x)
              {x = rep_top(rec(istack', index'))
                AND istack = (rep_pop(rec(istack', index'))).astack
                  AND index = (rep_pop(rec(istack', index'))).pointer}
```

Figure 7.3: Implementation of "Stackops" (continues)

```
PROOF

  pr1: VERIFY init_lm

  pr2: VERIFY push_lm FROM
    push_def,
    rep_push@c' {p <- rec(istack, index), e <- v},
    recax1@c' {ll <- istack, n <- index},
    recax2@c' {ll <- istack, n <- index}

  pr3: VERIFY pop_lm FROM
    pop_def,
    rep_pop@c' {p <- rec(istack, index), e <- x},
    recax1@c' {ll <- istack, n <- index},
    recax2@c' {ll <- istack, n <- index}

END stackops2
```

Figure 7.3: Implementation of "Stackops"

The module stackops2, displayed in Figure 7.3, contains an implementation of stackops. It is modeled closely after the implementation of stacks by stackrep. In fact, we use stackrep to state the properties we want to prove about the implementation, as we will see below. Instead of the abstract state object stack_obj, we now have two state objects, an array istack and a naturalnumber-valued object index. Each operation has been given an interpretation in the form of "concrete" operations (assignments, conditional statements etc.) on istack and index. The code for the operations is quite straightforward.

The figure above displays the continuation of the module stackops2. The properties that are stated as lemmas and proved in that module correspond exactly to the axioms in module stackops, where they define the meaning of the stack operations. The correspondence becomes clear when, following the approach taken in module stackrep, we regard the record consisting of (the values of) the two objects as the representation (or "implementation") of (the value of) the abstract state object stack_obj. The current version of the EHDM language does not provide a built-in constructor for record constants with given components;[4] we have defined one in the module makrec displayed in Figure 7.4. The postconditions of the lemmas in stackops2 are derived from those of the axioms in stackops by applying the mapping defined by the module stackmap:

[4]This deficiency is corrected in EHDM Version 6.

```
makrec: MODULE [etype: TYPE, undef: etype]

USING stackrep[etype, undef]

EXPORTING ALL

THEORY

  ll: VAR linarray

  n: VAR nat

  cc: rep_stack

  rec: function[linarray, nat -> rep_stack]

  recax1: AXIOM (rec(ll, n)).astack = ll

  recax2: AXIOM (rec(ll, n)).pointer = n

END makrec
```

Figure 7.4: The Record Constructor Module "Makrec"

`stack_obj` is replaced by `rec(istack,index)`, `push` by `rep_push` etc. To make the conditions more readable and the proofs more manageable, we have replaced the equalities between records by equalities among the record components. (We cannot express the mapping from `stackops` onto `stackops2` as in module `stackmap` because the current version of EHDM does not support the mapping of one state object onto an aggregate of several.)

The proofs of the lemmas are straightforward; they require only the definitions of the operations, the basic properties of `rep_push` and `rep_pop` (from module `stackrep`), and the meaning of the record constructor function as expressed in module `makrec`. Try to figure them out for yourself by following the reduction process described above.

7.6 Generating Ada Text

EHDM provides an experimental facility for translating procedural code-level specifications to Ada. The translator extracts the "computational" content of a specification module and converts it into Ada text. Figure 7.5 shows the result of translating the module `stackops2`. The translator is invoked by the command `Translate to Ada` (`M-X tr`); it writes the Ada text into a new buffer and file with the extension `.ada`.

In this example, it is quite obvious that the generated Ada text is computationally equivalent to the operations specified in the module `stackops2` and satisfies the intent of the specification in module `stackops`. Given a formal semantics for the fragment of Ada that we employ, it should, in principle, be possible to develop an argument for the correctness of the resulting code. At present, however, informal reasoning and inspection must be used to determine the equivalence of a specification and the executable Ada program generated from it.[5] The purpose of the Ada translator is simply to allow practical experiment with this approach to program development.

The Ada translator cannot handle all the constructs of EHDM. For example, it cannot handle higher-order functions, lambda expressions or quantified expressions. This is reasonable, since the translator simply converts from EHDM to Ada syntax; it is not a code synthesizer. Therefore, you should develop a specification all the way down to simple imperative operations before invoking the Ada translator. However, even low-level specification modules often contain these untranslatable constructs—for example in declarations needed to state properties required during verification. The Ada translator cannot tell which declarations in a specification are intended as part of the "computational" component. Consequently, it attempts to translate every declaration; constructs that it cannot translate are replaced by placeholders and an explanation is printed in the form of an Ada comment. Examples appear in the translation of `stackrep` shown in Figure 7.6. Most of this module, including all the untranslated constructions,

[5] This "equivalence" is not strict: for example, the Ada type `Natural` and the EHDM type `nat` are rather different.

```
WITH stackrep;
WITH makrec;

GENERIC
  TYPE etype IS PRIVATE;
  undef : etype;

PACKAGE stackops2 IS
  PROCEDURE push_op (v : etype);
  PROCEDURE pop_op (x : in out etype);
END stackops2;

PACKAGE BODY stackops2 IS

  PACKAGE makrec_2 IS NEW makrec(etype, undef);
  USE makrec_2;

  PACKAGE stackrep_1 IS NEW stackrep(etype, undef);
  USE stackrep_1;
  istack : linarray;
  index : Natural;

  PROCEDURE push_op (v : etype) IS
  BEGIN

    index := index + 1;
    istack(index) := v;

  END push_op;

  PROCEDURE pop_op (x : in out etype) IS
  BEGIN

    IF index = 0 THEN
      x := undef;
    ELSE
      x := istack(index);
      index := pred(index);
    END IF;

  END pop_op;

END stackops2;
```

Figure 7.5: The Ada code generated by “stackops2”

is irrelevant to the "computational" component of `stackops2`, being needed only in the statement and proof of its lemmas—in fact, only the type definition `linarray` is needed for the Ada code. It is up to the user to recognize this fact and to delete all the irrelevant parts of translated modules.[6]

[6] It is, in fact, possible to use the Ada translator without having developed the specification to a very low level. In this case, the translated "code" will consist mostly of procedure stubs and comments containing specification fragments. These can then serve as the starting point for a program development conducted in Ada.

```
GENERIC
  TYPE elemtype IS PRIVATE;
  undef : elemtype;

PACKAGE stackrep IS
  TYPE linarray IS ARRAY (Natural) OF elemtype;
  TYPE rep_stack IS RECORD astack : linarray; pointer : Natural; END RECORD;
  FUNCTION concrete_equality (p1, p2 : rep_stack) RETURN Boolean;
  rep_newstack : CONSTANT rep_stack := a_3619;
  FUNCTION rep_push (p : rep_stack; e : elemtype) RETURN rep_stack;
  FUNCTION rep_pop (p : rep_stack) RETURN rep_stack;
  FUNCTION rep_top (p : rep_stack) RETURN elemtype;
  FUNCTION rep_isnewstack (p : rep_stack) RETURN Boolean;
END stackrep;

PACKAGE BODY stackrep IS
-- The following EHDM syntax is not currently translatable
    -- (FORALL i : (1 <= i AND i <= p1.pointer)
    --       IMPLIES (p1.astack(i) = p2.astack(i)))
  -- It is replaced by QUANEXPR in the following ADA code

  FUNCTION concrete_equality (p1, p2 : rep_stack) RETURN Boolean IS
  BEGIN

    RETURN (p1.pointer = p2.pointer) and QUANEXPR;

  END concrete_equality;
```

Figure 7.6: The Ada code generated by “Stackrep” (continues)

```
-- The following EHDM syntax is not currently translatable
    -- p WITH [pointer := p.pointer + 1, astack := p.astack
    --          WITH [(p.pointer + 1) := e]]
  -- It is replaced by WITHEXPR in the following ADA code

  FUNCTION rep_push (p : rep_stack; e : elemtype) RETURN rep_stack IS
  BEGIN

    RETURN WITHEXPR;

  END rep_push;

-- The following EHDM syntax is not currently translatable
    -- p WITH [pointer := IF p.pointer = 0 THEN 0 ELSE pred(p.pointer) END IF]
  -- It is replaced by WITHEXPR in the following ADA code

  FUNCTION rep_pop (p : rep_stack) RETURN rep_stack IS
  BEGIN

    RETURN WITHEXPR;

  END rep_pop;

  FUNCTION rep_top (p : rep_stack) RETURN elemtype IS
  BEGIN

    RETURN IFEXPR(p.pointer = 0, undef, p.astack(p.pointer));

  END rep_top;

  FUNCTION rep_isnewstack (p : rep_stack) RETURN Boolean IS
  BEGIN

    RETURN p.pointer = 0;

  END rep_isnewstack;

END stackrep;
```

Figure 7.6: The Ada code generated by "Stackrep"

Chapter 8

Final Words

This tutorial has concentrated on the mechanics of the EHDM language and system and we have not said much about style and clarity in formal specifications, nor about using verification for maximum benefit. We plan to publish a companion document the near future that will exemplify the approach to formal specification and verification that we advocate. In the meantime, three full-scale applications demonstrate the use of EHDM in earnest [20–22]. One of these [21] provides some interesting evidence for the value of formal analysis. The journal proof that the Interactive Convergence Clock Synchronization Algorithm maintains synchronization despite the occurrence of Byzantine faults comprises five lemmas and a main theorem [13]. Our analysis, using EHDM, demonstrated that four of the five lemmas, and the main theorem, were all false as stated. As far as we know, these flaws had not previously been detected by the "social process" of informal peer scrutiny to which the journal paper has been subjected since its publication. We corrected the flaws in the journal proof (which were nontrivial and required changes to the external specification of the algorithm, not just to the proofs themselves), formally verified the corrected proof, and extracted a revised journal-level proof from the formal verification that is much simpler and easier to follow than the original, incorrect, proof. Applications undertaken in EHDM by others are also available in reports and papers [3, 17, 28].

Appendix A

Alternative Stacks Example

The stacks example given in section 5.1 is somewhat artificial—it was designed to show features of the language without getting bogged down in detail. Here we provide a more complete and realistic specification for stacks, consisting of the module `alt_stacks` (Figure A.1) and an implementation `alt_stackrep` (Figure A.2).

The `alt_stacks` module extends the `stacks` module by the addition of the predicates `isempty` and `isfull`, and some new axioms. If `isfull` is always false, then this new specification is equivalent to the earlier `stacks` and `stack_eqns` modules. The `alt_stackrep` module is extended to provide interpretations for these new predicates. In addition, there is a new module parameter, `max`, that bounds the depth of stack and is used in the interpretation of the `isfull` predicate. The `alt_stackmap` and `alt_stackmap_proofs` modules are not provided here; you are encouraged to specify and prove these as an exercise.

```
alt_stacks: MODULE [elem: TYPE, undef: elem]

EXPORTING ALL

THEORY

  stack: TYPE
  s: VAR stack
  e: VAR elem

  newstack: stack
  push: function[stack, elem -> stack]
  pop: function[stack -> stack]
  top: function[stack -> elem]

  replace: function[stack, elem -> stack] ==
    (LAMBDA s, e -> stack : push(pop(s), e))

  isempty: function[stack -> bool] == (LAMBDA s : s = newstack)
  isfull: function[stack -> bool]

  fullnewstack: AXIOM NOT isfull(newstack)

  popempty: AXIOM isempty(s) IMPLIES pop(s) = s

  pushempty: AXIOM NOT isempty(push(s, e))

  topempty: AXIOM isempty(s) IMPLIES top(s) = undef

  popfull: AXIOM NOT isfull(pop(s))

  pushfull: AXIOM isfull(s) IMPLIES push(s, e) = s

  poppush: AXIOM NOT isfull(s) IMPLIES pop(push(s, e)) = s

  pushpop: AXIOM NOT isempty(s) IMPLIES push(pop(s),top(s)) = s

  toppush: AXIOM NOT isfull(s) IMPLIES top(push(s, e)) = e

END alt_stacks
```

Figure A.1: The Alternative “Stacks” Example

```
alt_stackrep: MODULE [elemtype: TYPE, undef: elemtype, max: nat]

EXPORTING range, linarray, rep_stack, concrete_equality, rep_newstack,
  rep_push, rep_pop, rep_top, rep_isempty, rep_isfull

ASSUMING nonzero_max: FORMULA max > 0

THEORY

  n: VAR nat
  range: TYPE FROM nat WITH (LAMBDA n : n <= max)
  linarray: TYPE = ARRAY [range] OF elemtype
  rep_stack: TYPE = RECORD astack: linarray,
                           pointer: range
                    END RECORD
  p, p1, p2: VAR rep_stack
  e: VAR elemtype
  i: VAR range
  concrete_equality: function[rep_stack, rep_stack -> bool] =
    (LAMBDA p1, p2 :
       (p1.pointer = p2.pointer)
         AND (FORALL i :
            (1 <= i AND i <= p1.pointer)
              IMPLIES (p1.astack(i) = p2.astack(i))))

  rep_newstack: rep_stack

  rep_new_ax: AXIOM rep_newstack.pointer = 0

  rep_isempty: LITERAL function[rep_stack -> bool] ==
    (LAMBDA p : p.pointer = 0)
  rep_isfull: LITERAL function[rep_stack -> bool] ==
    (LAMBDA p : p.pointer = max)

  rep_push: function[rep_stack, elemtype -> rep_stack] =
    (LAMBDA p, e :
      IF NOT rep_isfull(p)
         THEN p WITH [pointer := p.pointer + 1,
                      astack := p.astack WITH [(p.pointer + 1) := e]]
         ELSE p
         END IF)
  rep_pop: function[rep_stack -> rep_stack] =
    (LAMBDA p : p WITH [pointer := pred(p.pointer)])
  rep_top: LITERAL function[rep_stack -> elemtype] ==
    (LAMBDA p : IF rep_isempty(p) THEN undef ELSE p.astack(p.pointer) END IF)

END alt_stackrep
```

Figure A.2: The Alternative “Stackrep” Example

Bibliography

[1] Peter B. Andrews. *An Introduction to Logic and Type Theory: To Truth through Proof.* Academic press, 1986.

[2] R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, 1988.

[3] R. W. Butler and J. A. Sjogren. Hardware proofs using EHDM and the RSRE verification methodology. NASA Technical Memorandum 100669, NASA Langley Research Center, 1988.

[4] EHDM *Specification and Verification System Version 4.1—User's Guide.* Computer Science Laboratory, SRI International, Menlo Park, CA, November 1988. See [6] for the updates to Version 5.2.

[5] EHDM *Specification and Verification System Version 5.0—Description of the* EHDM *Specification Language.* Computer Science Laboratory, SRI International, Menlo Park, CA, January 1990. See [6] for the updates to Version 5.2.

[6] EHDM *Specification and Verification System Version 5. X—Supplement to User's and Language Manuals.* Computer Science Laboratory, SRI International, Menlo Park, CA, August 1991. Current version number is 5.2.

[7] Ruth E. Davis. *Truth, Deduction, and Computation.* Principles of Computer Science Series. Computer Science Press, an imprint of W. H. Freeman and Company, New York, 1989.

[8] H. B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, New York, 1972.

[9] Ian Hayes, editor. *Specification Case Studies.* Prentice-Hall International (UK) Ltd., Hemel Hempstead, UK, 1987.

[10] C. A. R. Hoare. An axiomatic basis of computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[11] Cliff B. Jones. *Software Development: A Rigorous Approach*. Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, 1986.

[12] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, second edition, 1990.

[13] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[14] David C. Luckham, Friedrich W. von Henke, Bernd Krieg-Brückner, and Olaf Owe. *ANNA: A Language for Annotating Ada Programs*, volume 260 of *Lecture Notes in Computer Science*. Springer Verlag, 1987.

[15] Zohar Manna and Richard Waldinger. *The Logical Basis for Computer Programming*, volume 1: Deductive Reasoning. Addison-Wesley, 1985.

[16] Zohar Manna and Richard Waldinger. *The Logical Basis for Computer Programming*, volume 2: Deductive Systems. Addison-Wesley, 1990.

[17] Andrew P. Moore. The specification and verified decomposition of system requirements using CSP. *IEEE Transactions on Software Engineering*, 16(9):932–948, September 1990.

[18] John Rushby. *Measures and Techniques for Software Quality Assurance*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1988. (The first 60 pages of [19]).

[19] John Rushby. Quality measures and assurance for AI software. Technical Report SRI-CSL-88-7R, Computer Science Laboratory, SRI International, Menlo Park, CA, September 1988. Also available as NASA Contractor Report 4187.

[20] John Rushby. Formal specification and verification of a fault-masking and transient-recovery model for digital flight-control systems. Technical Report SRI-CSL-91-3, Computer Science Laboratory, SRI International, Menlo Park, CA, January 1991. Also available as NASA Contractor Report 4384.

[21] John Rushby and Friedrich von Henke. Formal verification of the Interactive Convergence clock synchronization algorithm using EHDM. Technical Report SRI-CSL-89-3R, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1989 (Revised August 1991). Original version also available as NASA Contractor Report 4239.

[22] Natarajan Shankar. Mechanical verification of a schematic Byzantine fault-tolerant clock synchronization algorithm. Technical Report SRI-CSL-91-4, Computer Science Laboratory, SRI International, Menlo Park, CA, January 1991. Also available as NASA Contractor Report 4386.

[23] Joseph R. Shoenfield. *Mathematical Logic.* Addison-Wesley, Reading, MA, 1967.

[24] J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics.* Cambridge Tracts in Theoretical Computer Science 3. Cambridge University Press, Cambridge, UK, 1988.

[25] J. M. Spivey, editor. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, 1989.

[26] Richard Stallman. *GNU Emacs Manual.* Free Software Foundation, 675 Massachusetts Ave., Cambridge, MA, 6th edition, March 1987.

[27] Friedrich von Henke, Natarajan Shankar, and John Rushby. *Formal Semantics of* EHDM. Computer Science Laboratory, SRI International, Menlo Park, CA, January 1990. This document describes EHDM Version 5.0; see [6] for informal descriptions of the changes in Version 5.2.

[28] R. Alan Whitehurst and T. F. Lunt. The SeaView verification. In *Proceedings of the Computer Security Foundations Workshop II*, pages 125–132, Franconia, NH, June 1989. IEEE Computer Society.