STeP

The Stanford Temporal Prover
Educational Release

Version 1.0

USER'S MANUAL

Nikolaj Bjørner, Anca Browne, Eddie Chang, Michael Colón,
Arjun Kapur, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe

November, 1995

Computer Science Department
Stanford University
Stanford, California 94305

Contents

Appendix

Chapter 1

Introduction

The Stanford Temporal Prover, STeP, supports the computer-aided formal verification of concurrent and reactive systems based on temporal specifications. Reactive systems maintain an ongoing interaction with their environment, and their specifications are typically expressed as constraints on their behavior over time. STeP is not restricted to finite-state systems, but combines *model checking* with *deductive methods* to allow for the verification of a broad class of systems, including parameterized (N-component) circuit designs, parameterized (N-process) programs, and programs with infinite data domains.

Finite-state systems can be automatically verified using model checking alone. For large finite-state or for infinite-state systems, for which model checking is not possible or feasible, STeP also uses deductive methods: *Verification rules* [Manna and Pnueli, 1995] reduce temporal properties to first-order verification conditions. *Verification diagrams* [Manna and Pnueli, 1994] are a visual language for guiding, organizing and displaying proofs. They can be used to construct proofs hierarchically, starting from a high-level sketch and proceeding incrementally through layers of greater detail.

STeP also includes techniques for automatic *invariant generation* [Bjørner *et al.*, 1995]. Deductive verification almost always relies on finding, for a given program and specification, suitably strong invariants and intermediate assertions. The system generates bottom-up invariants automatically by analyzing the program text. Combining them with the property to be proved, sufficiently detailed invariants can often be obtained to carry through the verification process.

Finally, STeP provides a collection of simplification and decision procedures that automatically check the validity of a large class of first-order and temporal formulas. This degree of automated deduction can handle many of the verification conditions that arise in deductive verification.

Figure 1.1 shows an overview of STeP. The main inputs are a reactive system (which can be a description of hardware or software) and a property to be proven about the system, represented by a temporal logic formula. Verification can be performed by the model checker or by deductive means. Model checking is automatic; deductive verification is to a large extent automatic for simple *safety* properties, while *progress* properties require more user guidance, usually given in the form of a verification diagram. In all cases, the automatic prover is responsible for generating and proving the required verification conditions. An interactive Gentzen-style theorem prover is used to establish those verification conditions

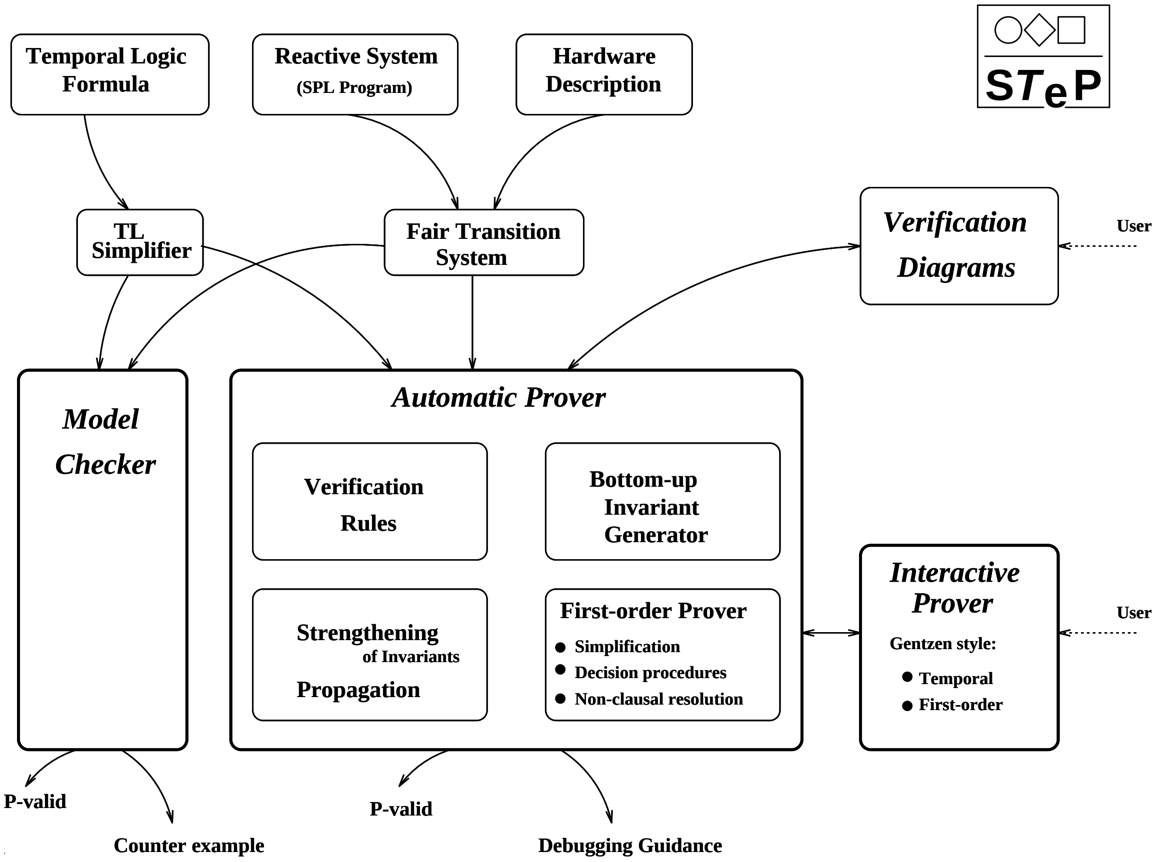


Figure 1.1: An overview of the STeP system.

that are not proved automatically.

STeP: The Educational Version

The current release of STeP is the *educational version*, and is a companion to the textbook [Manna and Pnueli, 1995]. Like the book, it concentrates on safety properties, and is intended for use by students, instructors and researchers of formal methods.

1.1 Reading this Manual

This manual is a complete reference guide for using STeP. For more background on STeP's framework for temporal specification and program verification, see [Manna and Pnueli, 1991] and [Manna and Pnueli, 1995].

Following are some suggestions on reading this manual:

- **Getting started:** Section 1.2 describes how to get STeP up and running. Section 1.3 gives an overview of STeP and a flavor of how a typical session proceeds. Section 1.4 defines the main terms used throughout this manual.

- **The Tutorial:** This manual includes a basic tutorial on STeP, in Chapter 9. This tutorial is the recommended method for readers to familiarize themselves with the STeP interface and the different styles of verification in STeP.

  After the tutorial, you should be able to verify some of the other programs included in the `examples` directory as well as your own simple programs, using the rest of this manual as a reference.

Some sections are marked with a "⋆" to indicate that they should be read only after the unmarked sections are understood. For instance, first-time users can postpone studying features such as tactics (Section 6.3) and user-defined rewrite and simplification rules (Section 2.4.4), until they feel comfortable with the rest of the system. Sections marked with "⋆⋆" are intended for advanced users and may be safely omitted by the beginning student.

Chapters 2-8 are the main reference for using the system. For those readers unfamiliar with [Manna and Pnueli, 1995], Appendixes A and B summarize STeP's program and property specification languages. Appendix C presents the most important differences between STeP and [Manna and Pnueli, 1995].

1.2 Starting STeP

The following are the steps for starting STeP:

1. Set your `STEP_DIR` environment variable to the full directory path where STeP is installed.[1] If this variable is not set, you will not be able to run STeP. (If you are installing the system, you can edit the `bin/STeP` file to correctly set this value for all users.)

2. The `DISPLAY` environment variable is used by X-windows to identify the screen where programs should run, and is usually of the form *machine-name*`:0.0`. Due to the inner workings of eXene (the SML X-windows library), this should be *changed* to the form *ip-address*`:0.0`, where *ip-address* is the corresponding numerical IP-address.[2]

   Alternatively, the display can be specified with the `-display` command-line option when running STeP, as in "`STeP -display` *ip-address*`:0.0`".

3. Make sure that the machine where you plan to run STeP has permission to open windows on your display. To make sure this is the case, execute, in your X-windows console, the command "`xhost +`".

4. Add `STEP_DIR` to your *path*. In this way, when executing STeP you don't have to type `STEP_DIR/STeP`, but just `STeP`.

5. Execute STeP, by running `STeP`. If your `DISPLAY` variable and `xhost` permissions are set correctly, the STeP main window will appear on your screen.

[1] Environment variables are set with "`setenv` *variable value*" Unix commands, which you can add to your `.cshrc` file for convenience.

[2] On `UNIX` systems, you can usually obtain this address with the commands "`nslookup` *machine-name*" or "`ping -l` *machine-name*".

A number of other environment variables can be optionally set according to each user's preferences; these are described in Appendix E.

Appendix D gives details on the installation of STeP. If you have problems installing or running STeP, please send mail to `step-bugs@cs.stanford.edu`.

1.3 Interacting with STeP

STeP has three main interface components: the *Top-level Prover*, from which verification sessions are managed and verification rules invoked; the *Interactive Prover*, used to prove the validity of first-order and temporal-logic formulas; and the *Verification Diagram Editor*, for the creation of Verification Diagrams. Figure 1.2 shows all STeP's interfaces with the program MUX-SEM loaded.

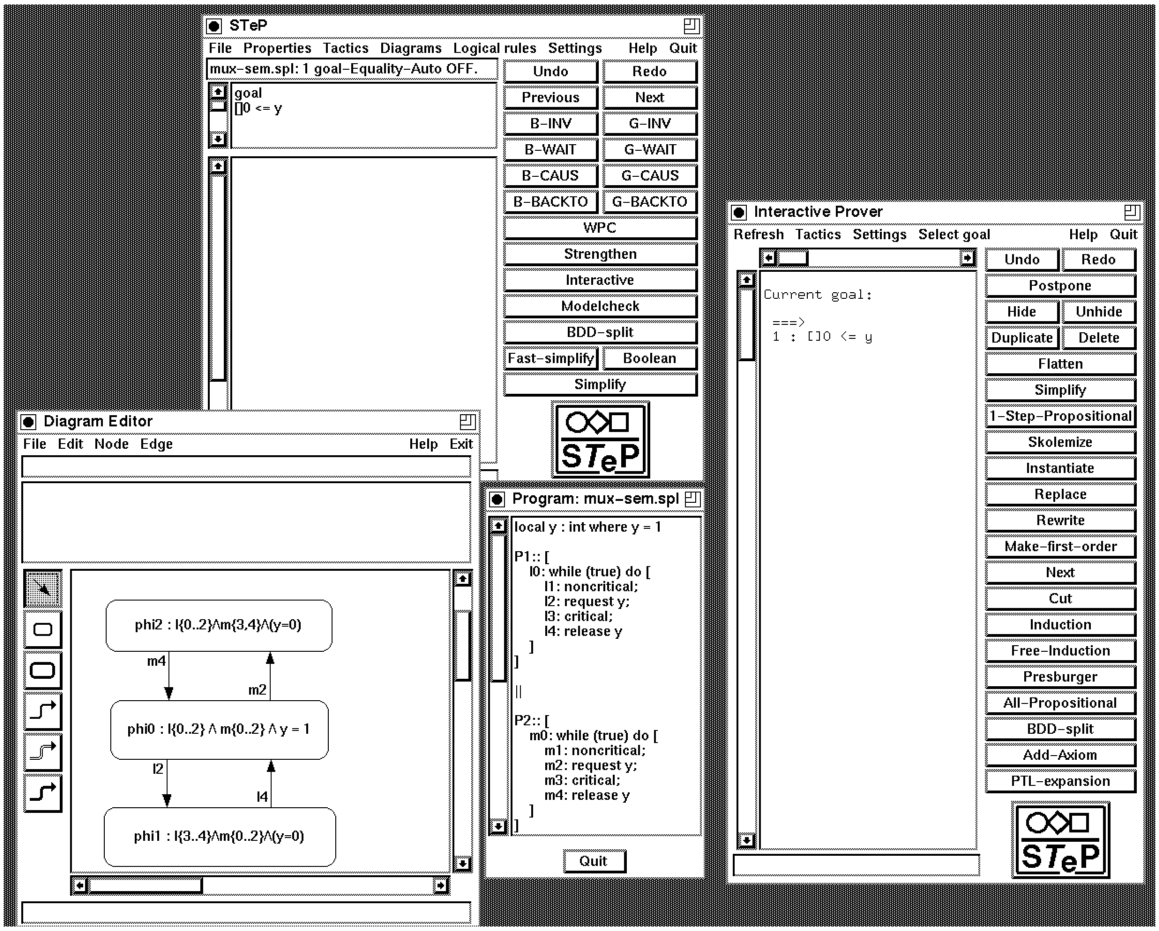


Figure 1.2: Overview of STeP's interfaces

Verification Session—An Overview

A verification session begins by loading a program or a transition system that describes the system of interest and entering a temporal-logic formula that expresses one or more system properties to be proved.

The formula becomes the *top* or *root goal* of a *proof tree*. There are now several ways to proceed. The most common route is to apply a *verification rule* (Section 4.2), which reduces the formula to simpler first-order or temporal formulas, thus generating the first level of *subgoals* of the proof tree. These subgoals are numbered accordingly: 1.1, 1.2, 1.3, etc. If all subgoals are established automatically by the Simplifier (i.e., the Simplifier can reduce them to *true*), all branches are *closed* and the proof is finished.

If the Simplifier is not able to prove one of the subgoals, we have one of the following possibilities:

1. The subgoal is not valid for the given program. In this case, the proof cannot succeed.

2. The property is valid for the program, but not generally valid. In this case you will have to (a) strengthen the original formula, or (b) first prove some auxiliary properties of the program, relative to which the problematic verification condition becomes valid. In the latter case you may start a new proof tree for the auxiliary property and return to the original property after the auxiliary proof is finished. This approach reflects the incremental proof style advocated in [Manna and Pnueli, 1995].

   Another way to establish auxiliary properties is through the automatic generation of invariants. STeP includes three such methods, which generate properties that often help in proving verification conditions (Chapter 7).

3. The property is generally valid, but the simplifier is unable to reduce it to *true*. In this case, the Interactive Prover (Chapter 8) can be used to establish the given property with user guidance. If this is successful, the given subgoal is closed. Partial results may also be returned to the Top-level Prover and incorporated in the current proof tree as a new subgoal.

Another way to verify a program property is to use a *verification diagram*. A verification diagram represents a set of verification conditions sufficient to establish a given property. Diagrams are created by the Verification Diagram Editor (Chapter 5) and subsequently transferred to the Top-level Prover.

Finally, any property or subgoal may be established by the Model Checker, through the exploration of the state space of the system (Section 4.5). Although model checking is only guaranteed to succeed for finite-state programs whose state space is not too large, it is also generally useful to identify *erroneous* properties, since it provides a counterexample computation if the property is not valid for the given system.[3]

STeP can also be used as a theorem prover, independently of any particular system verification task. In this case you only need to load a specification, and only the Simplifier and the Interactive Prover are used.

[3] Restrictions may be placed on the system to make model checking more feasible—see Section 4.5.

Verification Session—A Quick Tour

To make things more concrete we will give an example of an actual STeP session. For a step-by-step description of the commands needed to run this example, see the basic STeP tutorial in Chapter 9.

$$\textbf{local} \quad y \quad : \textbf{integer where } y = 1$$

$$P_1 :: \left[\begin{array}{l} \ell_0\text{: } \textbf{loop forever do} \\ \quad \left[\begin{array}{l} \ell_1\text{: } \textbf{noncritical} \\ \ell_2\text{: } \textbf{request } y \\ \ell_3\text{: } \textbf{critical} \\ \ell_4\text{: } \textbf{release } y \end{array}\right] \end{array}\right] \quad || \quad P_2 :: \left[\begin{array}{l} m_0\text{: } \textbf{loop forever do} \\ \quad \left[\begin{array}{l} m_1\text{: } \textbf{noncritical} \\ m_2\text{: } \textbf{request } y \\ m_3\text{: } \textbf{critical} \\ m_4\text{: } \textbf{release } y \end{array}\right] \end{array}\right]$$

Figure 1.3: Program MUX-SEM (mutual exclusion by semaphores)

Consider program MUX-SEM, shown in Figure 1.3. It implements mutual exclusion using the semaphore y, **request** decrements the semaphore, when it is positive, and **release** imcrements the semaphore. We want to prove the following properties of this program:

$$\varphi_1 : \Box(y \geq 0)$$

$$\varphi_2 : \Box(\neg(\ell_3 \wedge m_3))$$

The first property, stating that y is always greater than or equal to 0, is proved automatically using rule B-INV (Section 4.2) and decision procedures for linear arithmetic. The property is then added to the list of background properties and may be used in subsequent proofs.

The second property expresses mutual exclusion and is not *inductive*, that is, it is not preserved by all transitions. In particular, the verification conditions for transitions ℓ_2 and m_2 are not valid, and indeed when we apply rule B-INV the Simplifier does not reduce them to *true*. We now have several options. Since this is a finite-state program with a small number of states, model checking has a good chance of succeeding; indeed, the Model Checker can quickly verify this property.

Another option is to generate some auxiliary invariants. One of the *linear invariants* automatically generated for this system is:

$$I : \Box(\ell_0 + \ell_1 + \ell_2 = m_3 + m_4 + y)$$

Unfortunately, property φ_2 is not inductive relative to this invariant, so the Simplifier is still unable to prove the verification conditions for ℓ_2 and m_2. Repeating the application of B-INV with all linear invariants active for simplification, you will notice that simplification is significantly slower, since the background properties are being used in simplification.

However, property φ_2 is directly implied by the invariant I and the property φ_1 we proved earlier. Therefore, we can apply rule MON-I (Section 4.4) directly to the top-level goal, and finish the proof after one simplification step.

Another option is to use rule G-INV (Section 4.2), supplementing the verification with a strengthened assertion, e.g.,

$$\ell_3 + \ell_4 + m_3 + m_4 + y = 1$$

This assertion implies mutual exclusion and is inductive (the locations have been *arethmetized*, e.g., the value of ℓ_3 is 1 if control resides at ℓ_3, 0 otherwise). All verification conditions are reduced to *true* by the Simplifier, thus completing the proof.

Yet another option is a proof by verification diagram. The lower left corner of Figure 1.2 shows a state-partition diagram for program MUX-SEM that implies mutual exclusion; the property $\Box(\neg(\ell_3 \wedge m_3))$ is proved automatically by invoking the *Verification Diagram Rule* on this diagram. This produces 35 different verification conditions, all of which are reduced to *true*.

1.4 Terms and Definitions

STeP Sub-systems

In STeP we identify the following sub-systems, each of which operates relatively independently of the rest:

Top-level Prover (Chapter 3) The Top-level Prover is the main interface to a STeP session, from which all other activities are initiated.

Interactive Prover (Chapter 8) The Interactive Prover is invoked from the Top-level Prover to prove a particular goal or subgoal. When the Interactive Prover is active the Top-level Prover is suspended. The Interactive Prover provides a Gentzen-style proof system, which is guided by the user. While the Interactive Prover is normally used to complete the proof of a given subgoal, partial results from the Interactive Prover can also be transferred back to the Top-level Prover.

Verification Diagram Editor (Chapter 5) The Verification Diagram Editor is also invoked from the Top-level Prover, but can be run concurrently with it. It is used to create and edit verification diagrams, which can be given to the Top-level Prover to use in a proof.

Model Checker (Section 4.5) The Model Checker can be invoked from the Top-level Prover on any given temporal subgoal. If the Model Checker establishes the P-validity of the formula, the corresponding goal is proved. Otherwise, a counterexample computation (i.e. one that violates the formula) is returned.

Simplifier (Section 6.1) The Simplifier is a collection of decision procedures and rewrite rules used to reduce and prove general verification conditions. The Simplifier may be invoked by the user, both from the Top-level Prover and the Interactive Prover, to simplify single goals or subgoals. In the Top-level Prover, the Simplifier uses all axioms, invariants, and previously proven properties that are currently active. The power of the Simplifier, i.e., the trade-off between power and speed, is controlled by a number of global settings.

STeP Activities

We identify a few (nested) levels of activities, each with its own environment:

STeP session At the top level is the STeP session, which lasts for the entire period STeP is running. Simplification settings are part of the STeP session environment. They can be changed at any time, but are not affected by loading new systems or moving on to new goals.

System verification session The next level down is the *system verification session*, which lasts from the time a program or system is loaded until a new one is loaded. Proven properties are accumulated as long as the session lasts. Thus both the system and all the properties proven about it, as well as the axioms loaded for it, are part of the system verification session environment. All properties proven are valid relative to the current system. A system verification session may be terminated explicitly by loading a new system or resetting STeP.

Theorem-proving session At the same level as the system verification session is the *theorem-proving session*. The difference is that a theorem-proving session environment does not include a system. Thus, properties are proved to be generally valid in this case, and only the Simplifier and the Interactive Prover can be used to prove formulas.

Goal session A system verification or theorem-proving session may contain multiple *goal sessions*. Each unfinished proof of a goal is represented by a tree with the goal as its root. At any given time, only one tree is being worked on. The user can move from one tree to the next in the middle of a proof, and later return to the unfinished proof, e.g., when some auxiliary properties have been established. A goal session ends when the goal is proved valid or is abandoned by the user.

Properties

We use different terms for properties depending on their status in a verification or theorem-proving session. We list the terms here in alphabetical order:

Axiom An *axiom* is provided by the user and is assumed to be valid throughout a system verification or theorem-proving session. An axiom is considered to be a background property.

Background property A *background property* is assumed or proven to be valid for the program in a system verification session, and assumed or proven to be generally valid in a theorem-proving session. The user controls their use in simplification by activating and de-activating them. Background properties may also be deleted altogether from a verification session.

If active, background properties are used by the Simplifier in the simplification process of goals and subgoals at the Top-level Prover, and can be added to the current sequent in the Interactive Prover.

Background properties include axioms entered by the user, properties already proved in the same verification session, and automatically generated invariants.

Note that the set of background properties is lost upon termination of a system-verification or theorem-proving session.

Goal A *goal* is a formula that we want to prove generally valid or valid over a given program, and becomes the root of a proof tree.

Invariant An *invariant* is a formula of the form $\Box\, p$, stating that p is true at every state of a program or system computation. Invariants can be independently proved, or automatically generated by one of three methods: local, linear, or polyhedral invariant generation. Proven invariants are added to the set of background properties, and will be used by the simplifier when simplifying new formulas. They can also be used as axioms in the interactive prover.

Subgoal A *subgoal* is a formula that results from the application of a verification rule to a goal or another subgoal. It is a node of the proof tree. An *open* subgoal is a leaf of the proof tree that has not been reduced to *true*.

Verification condition A *verification condition* is a formula that results from the application of a verification rule to a goal or a subgoal. Verification conditions are a subset of the subgoals.

System Description and Specification

STeP accepts a system description in two different formats, namely as an SPL program or as a fair transition system. Specifications are entered as linear-time temporal logic formulas.

SPL program A system description may be given in the form of an SPL program (*Simple Programming Language* program). The syntax of well-formed SPL programs is given in Section 2.5. SPL programs are parsed into fair transition systems. The semantics of SPL are given in Appendix A.

Fair transition system An alternative way to provide a system description is by means of a fair transition system. A *fair transition system* is a tuple with the following components:

- V, a set of system variables. These include all program variables.
- Θ, the initial condition.
- $\mathcal{T}$, a set of transitions.

Each transition is a relation between states, specifying how the system can move from one state to the next. Section 2.6 describes how transition systems are specified in STeP. Appendix A describes fair transition systems in more detail.

Auxiliary and system variables Auxiliary variables are declared in specifications. System variables get declared in system descriptions and participate in the transition relations of the described transition system.

Flexible and rigid variables The value of a *flexible* variable may change over time, while that of a *rigid* variable is assumed to be fixed throughout a computation. Auxiliary variables in STeP can be declared to be flexible or rigid. System variables declared as `in` are rigid, while those declared as `out` and `local` are flexible.

Note that the flexible or rigid status of variables has a significant effect on whether terms under the scope of temporal operators can be replaced by others during interactive and automatic theorem-proving.[4]

Future operators Future temporal operators are $\square$ (Henceforth), $\diamond$ (Eventually), $\mathcal{U}$ (Until), $\mathcal{W}$ (Waiting-for also called Unless) and $\bigcirc$ (Next).

In STeP's ASCII representation, these are `[]`, `<>`, `Until`, `Awaits`, and `()` (see Appendix B), respectively.

Past operators Past operators are $\boxminus$ (So-far), $\diamondminus$ (Once), $\mathcal{S}$ (Since), $\mathcal{B}$ (Back-to), and $\ominus$ (Previously).

In STeP's ASCII representation, these are `[-]`, `<->`, `Since`, `Backto`, and `(-)` (see Appendix B), respectively.

Past formula A *past formula* is a temporal-logic formula that does not contain any future operators, that is, it only contains state formulas and past operators.

1.5 Feedback

If you have problems running STeP or find bugs in the system, please send mail to

`step-bugs@cs.stanford.edu`.

If you have questions or feedback, send mail to

`step-comments@cs.stanford.edu`.

There is a mailing list for STeP users, which will be used to distribute updated information about the system. To be added or removed from the list, send mail to

`step-request@cs.stanford.edu`.

1.6 Acknowledgements

STeP has been designed and implemented by Nikolaj Bjørner, Anca Browne, Eddie Chang and Tomás Uribe. Henny Sipma and Arjun Kapur have collaborated with the development and documentation of STeP.

The Verification Diagram editor was developed by Michael Colón. The interface tools were developed by Michael Colón and Hsiao-Lan Liao, and the SPL compiler was implemented by Jaejin Lee. Mark Stickel provided the AC-matching and unification facilities.

[4]For example it is in general not possible to instantiate an existentially bound rigid variable by a term containing flexible variables.

Harish Devarajan implemented the decision procedure for full Presburger arithmetic. Assistance and feedback has been provided by Anuchit Anuchitanukul, Luca de Alfaro, Jeff Kamerer, and Yoshihiro Yamada. We would also like to thank Paul Neves and the students of CS256L at Stanford. The STeP logo was designed by Anuchit Anuchitanukul.

STeP has mainly been implemented in Standard ML of New Jersey, using its X-windows utilities eXene for the user interface. David McQueen and John Reppy have been helpful assisting us in using SML/NJ and eXene. Xavier Leroy introduced us to ML.

The polyhedral invariant generation uses the polyhedral manipulation package developed at VERIMAG/CNRS (Grenoble) by Nicolas Halbwachs, Yann-Eric Proy and Herve Leverge. The model checker uses the PTL satisfiability tester by Hugh McGuire at Stanford.

The STeP project has been supported in part by the National Science Foundation under grant CCR-92-23226, by the Defense Advanced Research Projects Agency under grant NAG2-892, and by the United States Air Force Office of Scientific Research under grant F49620-93-1-0139.

The STeP project is supervised by Professor Zohar Manna.

Chapter 2

Systems and Specifications

A system can be input to STeP as an SPL program or directly as a transition system; SPL programs are parsed into transition systems. The computational model associated with transition systems and SPL programs is presented in Appendix A. The specification language is linear-time temporal logic. Its semantics is presented in Appendix B.

An SPL program or transition system is always loaded from a file. The properties to be proven may be entered directly or loaded from a file.

This chapter presents the STeP format for describing systems and their specifications. Since systems and specifications share the same syntax for variable and type declarations and logical expressions, we first describe the general syntax for types (Section 2.1) and expressions (Section 2.2). We then describe the overall style of specifications (Section 2.3) and the syntax for declarations (Section 2.4). Finally, we present the syntax for describing systems either as SPL programs (Section 2.5) or as transition systems (Section 2.6).

We use extended Backus-Naur form (EBNF) to describe the syntax of STeP's input. Terminal symbols are written in `typefont`, non-terminals in *italics*. Alternatives are separated by |, optional parts enclosed in [ ], zero or more occurrences of a construct are indicated by enclosing in { }, and { }+ is used for one or more occurrences.

Comments can be included in a file using the `%` character (indicating a comment until the end of the line), or using `(*` and `*)` for multi-line comments.

2.1 Types

The *basic types* in STeP are booleans, integers, rationals, range types, underspecified types, and datatypes. New compound types can be created using *type constructors*. The type constructors in STeP, listed below, define types for arrays, channels, and tuples of types in the usual way. A type can be enclosed in parenthesis to distinguish for instance a tuple of three elements, e.g., `int * rat * bool`, from a nested tuple of pairs, `(int * rat) * bool`.

type	::=	*id*	type identifier
	|	*basety*	base type
	|	`array [` *range* { `,` *range* } `] of` *type*	array type
	|	`channel of` *type*	synchronous channel
	|	`channel [1..] of` *type*	unbounded asynchronous channel

	|	`channel [1 ..` *int-const* `] of` *type*	bounded asynchronous channel
	|	`[` *range* `]`	range type
	|	*type* { `*` *type*}+	tuple type
	|	`(` *type* `)`	parenthesized type
range	::=	*expn* `..` *expn*	range
basety	::=	`bool`	boolean type
	|	`int`	integer type
	|	`rat`	rational type

The type constructor $*$ has higher precedence than all other type constructors. Thus, the type

```
channel [1..]  of bool * int * int
```

is parsed as

```
channel [1..]  of (bool * int * int).
```

Note that `(bool * int * int)`, `((bool * int) * int)` and `(bool * (int * int))` are three different types.

New underspecified types, datatypes and enumeration types can be declared in specification files; see Section 2.4.1.

2.2 Expressions

Expressions are constructed from constants, identifiers, function application, array references, tuple expressions, binary infix expressions, prefix expressions and quantification. The following well-formedness constraints are imposed:

- The boolean type is treated as a subtype of the integers, where the boolean value `false` corresponds to the integer value 0, and `true` corresponds to 1. Therefore any expression of boolean type can be used in an integer context. This is referred to as *arithmetization.*[1] Similarly, the integers (type `int`) are treated as a subtype of rationals (type `rat`).

- The operators `/\` (conjunction), `\/` (disjunction), `-->` (implication), `<-->`, (bi-implication), `!`, `~` (negation), and the temporal operators `Until`, `Awaits`, `Since`, `Backto`, `==>`, `<==>`, `()`, `[]`, `<>`, `(-)`, `[-]`, `<->` take boolean arguments, and the resulting expression is of type boolean.

- The operators `+`,`-`,`*` require integer or rational arguments and produce a value of the corresponding type. Division `/` requires arguments of type integer or rational, and results in a value of type rational. The operators `mod` and `div` require arguments of type integer and result in a value of type integer.

[1] Users should note that this arithmetization is very convenient, but at times confusing, see Section 2.7.

- Variables bound by quantifiers may not be array types.

The full grammar for expressions that can be used in specifications is as follows:

```
expn        ::=  bool-const                             boolean value, true or false
             |   int-const                              integer value
             |   id                                     identifier
             |   id { index { , index } }               disjunction of locations
                                                        id can be an array identifier
             |   id(expns)                              function application
             |   expr[expns]                            array reference
             |   (expn)                                 parenthesized expression
             |   (expn,expns)                           a tuple of expressions
             |   # int-const expn                       int-const'th projection of tuple
             |   expn infix expn                        infix operator
             |   prefix expn                            unary prefix operator
             |   if expn then expn else expn            if then else
             |   bind bv . expn                         binding
             |   expn'                                  primed expression

expns       ::=  expn | expn , expns

bv          ::=  id                                     globally or locally declared
             |   id : type [kind]                         bound variable

index       ::=  int-const | int-const .. int-const     indexing locations

infix       ::=  /\ | \/ | --> | <-->                   binary boolean operators
             |   Until | Awaits | Since | Backto        binary temporal operators
             |   ==> | <==>                             entailment and congruence
             |   + | * | - | / | mod | div              arithmetical operators
             |   = | != | < | > | >= | <=               predicate operators

prefix      ::=  ! | ~                                  negation
             |   -                                      minus
             |   () | [] | <>                           future operators
             |   (-) | [-] | <->                        past operators
             |   append | head | tail | length          channel operations
             |   assign                                 array update

bind        ::=  Forall | Exists | Exists! | Sum | Prod binding operators

kind        ::=  Rigid | Flexible                       flexibility of variable

bool-const  ::=  true | false
int-const   ::=  digit { digit }
```

id	::=	*alpha* { *alpha* | *digit* | _ }
alpha	::=	*upper or lower case letter*
digit	::=	*number between 0 and 9*

location disjunction: As an abbreviated notation for the disjunction of consecutive locations, the specification may contain expressions of the form

```
l{1..5,7,9..11}[i]
```

which is shorthand for

```
   l1[i] \/ l2[i] \/ l3[i]  \/ l4[i] \/ l5[i]
\/ l7[i] \/ l9[i] \/ l10[i] \/ l11[i]
```

channel operations are described separately in Section 2.4.5.

array assignment: Arrays are multidimensional tables. Individual table entries can be updated by the built-in function `assign`.

Example: The expression entered below:

```
in    N, M : int where N > 0 /\ M > 0
local A    : array [1..N, 1 ..M] of int
  assign(A,5,1,1)
```

is identical to the array `A`, except for position `[1,1]`, where it has been updated to the value `5`.

The simplifier uses the following simplification rule automatically (so it really does not have to be declared):

```
variable a,x : [1..N]
variable b,y : [1..M]
variable e   : int
SIMPLIFY : (assign(A,e,a,b))[x,y] --->
                   if a = x /\ b = y then e else A[x,y]
```

See section 2.4.4 for how to declare your own simplification rules.

tuple projection: The i'th element of a tuple t can be accessed as:

$$\# \; i \; t$$

Tuple indices are counted from 1, i.e., the first element in a tuple has index 1.

Symbols	*Associativity*
`of`	*left*
`array`	*left*
bind bv `.`	*nonassoc*
`then`	*nonassoc*
`else`	*nonassoc*
`<-->, <==>`	*right*
`-->, ==>`	*right*
`\/`	*right*
`/\`	*right*
`Until Awaits Since Backto`	*right*
`() [] <> (-) <-> [-]`	*nonassoc*
`= < > >= <= !=`	*left*
`~ !`	*nonassoc*
`mod div`	*left*
binary `-`, `+`	*left*
`/ *`	*left*
unary `-`	*right*
id `head tail append length assign`	*left*

Table 2.1: Operator precedences from lowest to highest

binding: The binding operators `Forall`, `Exists`, and `Exists!` correspond to $\forall$, $\exists$ and $\exists!$ (unique existence) respectively. They bind the variables in their scope, so for instance in

```
Forall x : int .  p(x)
```

the variable `x` is bound by the quantifier `Forall`. The binding operators `Sum` and `Prod` correspond to the arithmetic operators Σ and Π and bind variables similarly.

As the syntax indicates, the type field of the bound variables is optional. If no type is given, where the variable is bound, the variable must have been declared before using a variable declaration.

priming: The primed verision of a flexible variable refers to its value at the next state. The parser accepts primed expressions, and automatically distributes the prime inwards to all its flexible variables. Double priming is not supported and results in a parser error.

The precedence of the various operators is given in Table 2.1.

2.3 Specification Files

A specification file usually contains three parts: declarations, axioms and properties.

1. The *declarations* define new types, values, macros, rewrite rules, and auxiliary variables.

2. The *axioms* define properties that may be assumed by STeP in a verification session. Each axiom becomes a background property, and may be activated or deactivated from the Top-level Prover.

3. The *properties* are those formulas you want to prove true for a program or system, or are state-valid in general. These become top-level *goals* in the Top-level Prover.

The format of the specification file is described by the following grammar:

specification	::=	{ *declaration* } *expn*	
	|	`SPEC` {*declaration* | *axiom* | *property*}	
axiom	::=	`AXIOM` *desc : expn*	named axiom
property	::=	`PROPERTY` *desc : expn*	named property specification

The nonterminal *desc* is an informal description of the corresponding axiom or property, and can be any sequence of characters except colon (:).

Note that STeP does not guarantee the consistency of the given set of axioms, so the specification of axioms should be done with caution.

A specification file may contain a single formula, which by default becomes the property to be proven, or may contain multiple axioms and properties. In addition, the specification file may contain declarations of macros, values, and auxiliary variables. An example of a file with multiple axioms and properties is shown in Figure 2.1.

```
SPEC (* Greatest common divisor spec file *)

value COMMUTATIVE gcd :  int*int --> int

AXIOM gcd1:  []Forall m,n:int .  (m != n --> gcd(m,n) = gcd(m-n,n))

AXIOM gcd2:  []Forall m:int .  (m > 0 --> gcd(m,m) = m)
AXIOM gcd3:  []Forall m:int .  (m < 0 --> gcd(m,m) = -m)

PROPERTY aux1:  [](y1 > 0)
PROPERTY aux2:  [](y2 > 0)
PROPERTY aux3:  [](gcd(y1,y2) = gcd(a,b))

PROPERTY partial correctness:  l8 ==> g = gcd(a,b)
```

Figure 2.1: Specification file with multiple axioms and properties

2.4 Declarations

Declarations can be in one of the following forms, which we describe below:

declaration	::=	*type-decl* | *datatype-decl*	declaration of types
	|	*value-decl* | *macro-decl*	declaration of constructs
	|	*aux-var-decl* | *system-var-decl*	declaration of variables
	|	*simplify* | *rewrite* | *order*	rewrite rules

2.4.1 Type Declarations

Type declarations are used to define abbreviations for compound types, new underspecified types, and new *datatypes*.

Type abbreviations are declared as `type` *id* `=` *type*, where *type* is a *type expression* (see Section 2.1). Omitting `=` and the type expression declares *id* as a new, underspecified type:

type-decl	::=	`type` *id* `=` *type*	type declaration
	|	`type` *id* `= {` *ids* `}`	enumeration type
	|	`type` *id*	underspecified type identifier

Datatype Declarations ⋆

Datatypes are declared using `==` instead of a single `=`, with the following syntax:

datatype-decl	::=	`type` *id* `==` *constructors*	
		{ `and` *id* `==` *constructors* }	datatype declaration
constructors	::=	*constructor* { | *constructor* }	
constructor	::=	*id* [`::` *deconstructors*]	constructor *id* and fields
deconstructors	::=	*id* `:` *type* { `,` *id* `:` *type* }	datatype deconstructors

The enumeration type declaration

`type` *id* `=` $\{id_1, \ldots, id_n\}$

is shorthand for the datatype declaration

`type` *id* `==` $id_1 \mid \ldots \mid id_n$.

Example: A *tree* datatype can be defined recursively with a *forest* datatype as follows:

```
type ident
type tree   ==  Node ::  id: ident, children: forest
and forest  ==  Nil
            |   Cons ::  first: tree, next: forest
```

The type `ident` is defined as a new underspecified type. A forest is a set of trees represented as a list—either `Nil` or the `Cons` of a tree and a forest. A tree itself is a `Node` containing an id and a set of children, represented as a forest.

The above datatype definition *implicitly* declares the following function symbols:

```
value Node: ident * forest --> tree
value id: tree -> ident
value children: tree -> forest
value Nil: forest
value Cons: tree * forest --> forest
value first: forest --> tree
value next: forest --> forest
```

The *constructors* (`Node`, `Nil`, `Cons`) are used to create new elements of the corresponding type, while the *deconstructors* (`id`, `children`, `first`, `next`) are used to access the different components of the datatypes.

For the deconstructors, the following simplification rules, explained in Section 2.4.4, are automatically inferred, and do not have to be declared:

```
variable i:  ident
variable ch,n:  forest
variable t:  tree
SIMPLIFY : id(Node(i,ch))         --->  i
SIMPLIFY : children(Node(i,ch))   --->  ch
SIMPLIFY : first(Cons(t,n))       --->  t
SIMPLIFY : next(Cons(t,n))        --->  n
```

These rules are used automatically by the simplification procedures. Note that no simplification rules are given for `first(Nil)` and `next(Nil)` although these expressions are well typed. Their interpretation is left *underspecified*, but they are not erroneous.

For the constructors, axioms asserting injectivity are also asserted automatically and used by the simplifier, so they don't have to be declared:

```
Forall t:tree .  Forall f:forest .  Nil != Cons(t,f)
Forall i,j:ident .  Forall c1,c2:forest .
     Node(i,c1) = Node(j,c2) --> i = j /\ c1 = c2
Forall t1,t2:tree .  Forall sib1,sib2:forest .
     Cons(t1,sib1) = Cons(t2,sib2) --> t1 = t2 /\ sib1 = sib2
```

The complete axiomatization of datatypes also includes *well-founded induction schemas* (see [Manna and Waldinger, 1993].) These are not automatically provided in the current release of STeP, so the user may have to provide suitable instances.

2.4.2 Value and Macro Declarations

Program and specification files may contain value and macro declarations. Their syntax is given by the following grammar:

value-decl	::=	`value` *[ac] ids* : *[type* `-->`*] type*	value declaration
macro-decl	::=	`macro` *id* : *[type* `-->`*] type* `where` *id [*`(` *ids* `)`*]* = *expn*	macro declaration

ac	::=	`AC` | `COMMUTATIVE` | `ASSOCIATIVE`	
ids	::=	*id* {, *id* }	identifier list

Uninterpreted function symbols are declared as values. Binary function symbols can have additional attributes used by the Simplifier: the keyword `AC` states that the function is both associative and commutative; `COMMUTATIVE` and `ASSOCIATIVE` state that the function is only commutative or associative, respectively. For example, in Figure 2.1 the `gcd` function was defined to be commutative. Thus the simplifier assumes that `Forall a,b: int . gcd(a,b) = gcd(b,a)` besides the other axioms.

Macros, which may take optional arguments, are declared with the keyword `macro`. A `where` clause defines the expression corresponding to the macro. Each macro takes a fixed number of arguments and is expanded by the parser through purely syntactic substitution. A macro definition can use other previously defined macros, which will be expanded in turn.

Example: Figure 2.2 shows an example of value and macro declarations as they may appear in a program or a specification file. The value declarations define the integer constant c and function symbol f, which takes a pair of arguments of boolean type and returns an integer. The macro declarations indicate that the symbol m should be expanded to the expression $c < c + 1$, and the symbol g with integer arguments (i, j) should be expanded into $i + i + j + 5$.

```
value c:int
value f:bool * bool --> int
macro m:bool where m = c < c+1
macro g:int * int --> int where g(i,j) = i+i+j+5
```

Figure 2.2: Value and macro definitions

Example: Figure 2.3 shows a specification file to prove 1-bounded overtaking for program `mux-pet1` (Figure 3.4). The proof by rule G-WAIT requires the input of four auxiliary assertions. Having defined these assertions in the specification file allows the user to enter them by name, `phi0`, `phi1`, `phi2` or `phi3`, in the dialog window, reducing the chance of typing errors and saving time if the proof needs to be repeated multiple times.

2.4.3 Variable Declarations

aux-var-decl	::=	`variable` *ids* : *type* [*kind*]	auxiliary variable declaration
kind	::=	`Rigid` | `Flexible`	
system-var-decl	::=	*mode* *ids* : *type* [`where` *expn*]	system variable declaration
	|	`clock` *ids* [`where` *expn*]	system clock
mode	::=	`in` | `out` | `local`	
ids	::=	*id* {, *id* }	list of identifiers

```
% 1-bounded overtaking for mux-pet1
%
% for proof by rule G-WAIT use the following intermediate assertions:
%
% phi0:  l4
% phi1:  l3 /\ ((m0\/m1\/m2\/m5) \/ (m3 /\ !(s=1)))
% phi2:  l3 /\ m4
% phi3:  l3 /\ m3 /\ s=1
SPEC
macro phi0:  bool where phi0 = l4
macro phi1:  bool where phi1 = l3 /\ (m{0..2,5} \/ (m3 /\ !(s=1)))
macro phi2:  bool where phi2 = l3 /\ m4
macro phi3:  bool where phi3 = l3 /\ m3 /\ s=1

PROPERTY 1-bounded overtaking:
    l3 ==> ((!m4) Awaits ((m4) Awaits ((!m4) Awaits l4)))
```

Figure 2.3: Specification file for proving `mux-pet1` 1-bounded overtaking

Auxiliary variables are declared with the keyword `variable`. Auxiliary variables can only be declared in the specification file, or in a directly entered goal, and are not part of the system being verified. Each auxiliary variable can be optionally declared to be `Rigid` (its value does not change over time) or `Flexible` (its value may change from one point in time to another). By default, auxiliary variables are rigid. Auxiliary variables are often used as part of macros and rewrite rules.

For convenience, a specification may also include declarations for system variables; these are further described in Section 2.5.

Example: The declarations

```
variable x,y,z : int
variable a,b,c : bool Flexible
variable i     : [1..N]
```

declare the rigid integer-type variables x, y and z, the flexible boolean variables a, b and c, and the rigid variable i of type range [1..N].

Previously declared auxiliary variables can be used in quantification, so the type of the quantified variable type does not have to be declared within the quantifier (see Section 2.2).

2.4.4 Rewrite and Simplification Rules $\star$

Specifications can also include *simplification and rewrite rules* to efficiently build in equational axioms, as well as *ordering relations* to control automatic rewriting.

User-defined rewrite rules can be specified in one of two ways: rewrite rules declared with the `REWRITE` keyword are applied *under the user's control*, step-by-step, in the Interactive Prover. *Simplification rules*, declared with the `SIMPLIFY` keyword, are applied automatically and exhaustively whenever the Simplifier is invoked from either the Top-level Prover or the Interactive Prover. Thus, the set of user-defined simplification rules is expected to *terminate*, while the set of rewrite rules, under user control, does not have to be terminating.

An ordering relation, given by the `ORDER` keyword, specifies a linear order over uninterpreted function symbols, as a list. This order is used when converting equalities into rewrite rules during simplification; roughly speaking, if $t_1 = t_2$ or $t_2 = t_1$ is an equality encountered in contextual simplification, t_1 will be rewritten to t_2 if t_1 is built from a function symbol that appears earlier in the order.[2]

The syntax for simplification and rewrite rules is given by the following grammar:

rewrite	::=	`REWRITE` *desc* : *expn* `--->` *expn*	rewrite rule
simplify	::=	`SIMPLIFY` *desc* : *expn* `--->` *expn*	simplification rule
order	::=	`ORDER` *ids*	

The nonterminal *desc* can be any sequence of characters except colon (`:`), and is used as an informal description of the simplification or rewrite rule. The auxiliary variables free in the expression to the left of `--->` should be a superset of the free variables of the right hand side expression. In connection with rewrite and simplifcation rules, system variables are treated as constants (constructors).

Example:

Figure 2.4 shows an example of two rewrite rules. The rule `Peano 1`, rewrites $6 + 0$ to 6, and the rule `ad hoc` rewrites, $f(4 + 2)$ to `if y then 4+2 else -(4+2)`.

```
variable x:int
local y:bool
value f:int --> int
REWRITE Peano 1:  x + 0 ---> x
REWRITE ad hoc:  f(x) ---> if y then x else -x
```

Figure 2.4: Rewrite rules in a specification file

As with axiomatizations, the user is responsible for the consistency of the set of rewrite and simplification rules. The reader interested in learning more about rewrite rules and their applications should consult, for example, [Dershowitz and Jouannaud, 1990]. An example of simplification rules appears in Section 2.4.1.

[2]The Simplifier uses the *recursive path ordering on terms* to orient equalities, see [Dershowitz and Jouannaud, 1990].

2.4.5 Channel Operations ⋆

Asynchronous channels are modeled as queues, i.e., new elements are appended to the tail of the queue, and elements are extracted from the head.

The basic operations on channels, `head`, `tail`, `append`, and `length` are best described in the specification file that users should provide when they want to establish properties of channels. That is, STeP does not include the partial axiomatization given below; instead, specifications should provide separate axiomatizations for each type of asynchronous channel used.

```
%% Sample channel specification
SPEC
type t
type chan : channel [1..] of t
local ch : chan
value pick : chan * int --> t
variable x : t

AXIOM : [] Forall i : int .
            pick(append(ch,x),i) =
                 if i-1 = length(ch) then x else pick(ch,i)
AXIOM : [] Forall i : int . pick(tail(ch),i) = pick(ch,i+1)
SIMPLIFY : head(ch)             ---> pick(ch,1)
SIMPLIFY : length(append(ch,x)) ---> 1 + length(ch)
SIMPLIFY : length(tail(ch))     --->
                        if length(ch) = 0 then 0 else (length(ch)) - 1
```

Hence the operation `head` extracts the first element from the buffer, `tail` returns the buffer excluding its first element, `append` inserts a new element to the end of the buffer, and `length` measures the number of elements residing in the buffer. The auxiliary function `pick`, axiomatized in the specification, provides us with a formal definition of the `head`, `tail`, and `append` operations. We notice that taking `head` of an empty buffer only simplifies to some expression involving `pick` and its value is therefore underspecified.

2.5 SPL Programs

The syntax of SPL programs follows that of traditional imperative languages such as Pascal. In addition to the basic constructs found in these languages, SPL supports nondeterminism by means of the *selection* statement `or` and parallel composition by means of the *cooperation* statement `||`. Parallel processes can interact through shared variables such as semaphores, as well as by synchronous and asynchronous channels. Execution of parallel processes is assumed to proceed by interleaving (see Appendix A for a complete account of SPL semantics).

```
in N : int where N >= 2
local f :  array [1..N] of int where Forall i:[1..N] . (f[i] = 1)
local r :  int where r = N - 1

|| P[i:[1..N]] ::
        l0:  loop forever do [
              l1 :  noncritical;
              l2 :  request r;
              l3 :  request f[i];
              l4 :  request f[(i mod N) + 1];
              l5 :  critical;
              l6 :  release f[i];
              l7 :  release f[(i mod N) + 1];
              l8 :  release r
        ]
```

Figure 2.5: Program DINE (dining philosophers)

Program variables

As in most imperative languages, program variables must be declared prior to their use, in the appropriate scope. SPL provides an additional *mode* associated with the declaration of program variables. A variable declared as `in` is an input to the program; it may not be modified by any of the program statements, and is assumed to have a fixed value throughout the program execution. Variables declared as `local` or `out` may be changed by the program.

Variables may be initialized or constrained with an optional `where` clause. The given condition is assumed to hold only at the start of the program, not each time the program enters the block where the variable is declared.

Parameterized programs

SPL allows you to write *parameterized* (N-process) programs, in which an arbitrary number of identical processes are declared. Both parameterized composition and parameterized selection are supported.

Example:

Figure 2.5 shows a deadlock-free solution to the N-process *Dining Philosophers* problem. The program contains `N` instances of the infinite loop `l0-l8`, each with its unique value of `i`.

Grouped statements

A *grouped statement* is a set of statements enclosed by `<<` and `>>`. A grouped statement is translated into a single transition, that is, one atomically executable unit. Grouped statements may only contain basic statements, and may only be combined into composite statements by selection and concatenation. A grouped statement may contain at most one `send` or `receive` statement on the same channel.

Example: To request resources `r` and `s` simultaneously one can enter

```
<< request r; request s >>
```

which can also be encoded as

```
guard r > 0 /\ s > 0 do (r,s) := (r-1,s-1)
```

Program statements

The following grammar provides the syntax of SPL programs:

<table>
<tr><td>program</td><td>::=</td><td>{ decl } composite_stmt [; label]</td><td></td></tr>
<tr><td>composite_stmt</td><td>::=</td><td>composite_stmt ; composite_stmt</td><td>concatenation</td></tr>
<tr><td></td><td>|</td><td>composite_stmt <code>||</code> composite_stmt</td><td>cooperation statement</td></tr>
<tr><td></td><td>|</td><td>composite_stmt <code>or</code> composite_stmt</td><td>selection statement</td></tr>
<tr><td></td><td>|</td><td>[label] stmt</td><td></td></tr>
<tr><td>stmt</td><td>::=</td><td>basic_group_stmt</td><td></td></tr>
<tr><td></td><td>|</td><td><code>request</code> variable</td><td>request statement</td></tr>
<tr><td></td><td>|</td><td><code>release</code> variable</td><td>release statement</td></tr>
<tr><td></td><td>|</td><td><code>noncritical</code></td><td>noncritical statement</td></tr>
<tr><td></td><td>|</td><td><code>critical</code></td><td>critical statement</td></tr>
<tr><td></td><td>|</td><td><code>choose</code> variable</td><td>choose</td></tr>
<tr><td></td><td>|</td><td><code>produce</code> variable</td><td>produce statement</td></tr>
<tr><td></td><td>|</td><td><code>consume</code> variable</td><td>consume statement</td></tr>
<tr><td></td><td>|</td><td><code>guard</code> p_expn <code>do</code> assignment</td><td>guarded assignment</td></tr>
<tr><td></td><td>|</td><td><code>if</code> p_expn <code>then</code> composite_stmt</td><td>1-way conditional</td></tr>
<tr><td></td><td>|</td><td><code>if</code> p_expn <code>then</code> composite_stmt
<code>else</code> composite_stmt</td><td>2-way conditional</td></tr>
<tr><td></td><td>|</td><td><code>when</code> p_expn <code>do</code> composite_stmt</td><td>when statement</td></tr>
<tr><td></td><td>|</td><td><code>while</code> p_expn <code>do</code> composite_stmt</td><td>while statement</td></tr>
<tr><td></td><td>|</td><td><code>repeat</code> composite_stmt <code>until</code> p_expn</td><td>repeat statement</td></tr>
<tr><td></td><td>|</td><td><code>loop forever do</code> composite_stmt</td><td>infinite loop</td></tr>
<tr><td></td><td>|</td><td>[id ::] <code>[</code> program <code>]</code></td><td>block statement</td></tr>
<tr><td></td><td>|</td><td><code><<</code> comp_group_stmt <code>>></code></td><td>grouped statement</td></tr>
<tr><td></td><td>|</td><td><code>or</code> binding composite_stmt</td><td>parameterized selection</td></tr>
<tr><td></td><td>|</td><td><code>||</code> binding composite_stmt</td><td>parameterized cooperation</td></tr>
<tr><td>comp_group_stmt</td><td>::=</td><td>basic_group_stmt</td><td></td></tr>
<tr><td></td><td>|</td><td><code>if</code> p_expn <code>then</code> comp_group_stmt</td><td>1-way conditional</td></tr>
</table>

	|	`if` *p_expn* `then` *comp_group_stmt* `else` *comp_group_stmt*	2-way conditional
	|	`when` *p_expn* `do` *comp_group_stmt*	when statement
	|	*comp_group_stmt* `or` *comp_group_stmt*	selection statement
	|	*comp_group_stmt* `;` *comp_group_stmt*	concatenation
basic_group_stmt	::=	`skip`	skip statement
	|	*assignment*	
	|	`await` *p_expn*	await statement
	|	*variable* `<==` *p_expn*	send statement
	|	*variable* `==>` *variable*	receive statement
assignment	::=	*left-val* `:=` *p_expn*	
left-val	::=	*p_expn*	assignable value
variable	::=	*id*	variable
	|	*id* `[` *p_expns* `]`	array variable
binding	::=	*id* `[` *params* `]` `::`	named binding
	|	*params* `.`	unnamed binding
params	::=	*param* {`,` *param*}	list of parameters
param	::=	*id* `:` `[` *range* `]`	parameter
label	::=	*id* `:`	label
ids	::=	*id* { , *id* }	list of identifiers

decl	::=	*type-decl* | *datatype-decl*	SPL declaration
	|	*value-decl* | *macro-decl*	
	|	*system-var-decl*	
system-var-decl	::=	*mode ids* `:` *type [* `where` *expn]*	basic variable declaration
mode	::=	`in` | `out` | `local`	mode of variable

Assignable expressions are built from system variables declared as `local` or `out`, and array references to `local` or `out` arrays. Composite assignable expressions are obtained by collecting simpler assignable expressions into tuples.

Example: An example of a composite assignment swaps the values of `x` and `y` as follows:

```
(x,y) := (y,x)
```

Expressions

The expressions allowed in SPL programs are a subset of the expressions allowed in specifications. In particular, the following are *not* allowed in SPL programs:

- primed variables
- `Prod` and `Sum`
- temporal operators
- location expressions

Furthermore, when binding variables by quantifiers inside an SPL program, their types must always be declared locally, and the optional *kind* is not allowed (they are always `Rigid`).

The syntax of SPL expressions is then as follows:

<table>
<tr><td>p_expn</td><td>::=</td><td>bool-const</td><td>boolean value, true or false</td></tr>
<tr><td></td><td>|</td><td>int-const</td><td>integer value</td></tr>
<tr><td></td><td>|</td><td>id</td><td>identifier</td></tr>
<tr><td></td><td>|</td><td>id(p_expns)</td><td>function application</td></tr>
<tr><td></td><td>|</td><td>expr[p_expns]</td><td>array reference</td></tr>
<tr><td></td><td>|</td><td>(p_expn)</td><td>parenthesized expression</td></tr>
<tr><td></td><td>|</td><td>(p_expn,p_expns)</td><td>a tuple of expressions</td></tr>
<tr><td></td><td>|</td><td># int-const p_expn</td><td>int-const'th projection of tuple</td></tr>
<tr><td></td><td>|</td><td>p_expn infix p_expn</td><td>infix operator</td></tr>
<tr><td></td><td>|</td><td>prefix p_expn</td><td>unary prefix operator</td></tr>
<tr><td></td><td>|</td><td>if p_expn then p_expn else p_expn</td><td>if then else</td></tr>
<tr><td></td><td>|</td><td>bind id : type . p_expn</td><td>binding</td></tr>
<tr><td>p_expns</td><td>::=</td><td>p_expn | p_expn , p_expns</td><td></td></tr>
<tr><td>infix</td><td>::=</td><td>/\ | \/ | --> | <--></td><td>binary boolean operators</td></tr>
<tr><td></td><td>|</td><td>+ | * | - | / | mod | div</td><td>arithmetical operators</td></tr>
<tr><td></td><td>|</td><td>= | != | < | > | >= | <= | <></td><td>predicate operators</td></tr>
<tr><td>prefix</td><td>::=</td><td>! | ~ | -</td><td>negation and minus</td></tr>
<tr><td></td><td>|</td><td>append | head | tail | length</td><td>channel operations</td></tr>
<tr><td></td><td>|</td><td>assign</td><td>array update</td></tr>
<tr><td>bind</td><td>::=</td><td>Forall | Exists | Exists!</td><td>binding operators</td></tr>
<tr><td>bool-const</td><td>::=</td><td>true | false</td><td></td></tr>
<tr><td>int-const</td><td>::=</td><td>digit { digit }</td><td></td></tr>
<tr><td>id</td><td>::=</td><td>alpha { alpha | digit | _ }</td><td></td></tr>
</table>

2.6 Transition Systems

Transition systems are the basic system representation in STeP. SPL programs are parsed into fair transition systems; an alternative to SPL is to describe transition systems directly. Appendix A includes the basic definitions of transition systems. For a more leisurely introduction, see [Manna and Pnueli, 1991].

2.6.1 Fair Transition Systems

A transition system specification contains:

- Declarations of types and free variables.
- An optional *initial condition*, which is assumed to hold at the initial state of every computation.
- A set of transitions.

Transitions are expressed in terms of, and are usually equivalent to (see below), *transition relations*. A transition relation describes how the system can change from one state to the next. Strictly speaking, a transition relation can be expressed by a formula over the set of primed and unprimed system variables; however, for added convenience, transition relations in STeP can be specified by a combination of four different fields. Each field is optional, and STeP inserts default values for absent ones. The four fields are the following:

- The `modvar` field: contains a set of variables that are changed when the transition is taken. To indicate that the transition relation allows a variable to change value nondeterministically, one can include it in the `modvar` field and not mention it elsewhere.

  By default the only variables that are changed by the transition relation are the ones primed in the `modrel` field and the ones to the left-hand side of assignments in the `assign` field described below. The `modvar` field does not need to contain those variables.

- The `enable` field: the *enabling condition*, a formula that may not contain any primed variables. The transition can only be taken if the enabling condition is satisfied. By default, the enabling condition is *true*.

- The `modrel` field: contains the *modifying relation*, an arbitrary relation over primed and unprimed variables that is assumed to hold when the transition is taken. Its default value is *true*.

- The `assign` field: lists the assignments for those variables whose next-state values can be described as a function of the current-state (unprimed) variables.

An assignment `x := t` (in the assign field) can be equivalently expressed as a conjunct `x' = t` in the modifying relation, but assignment relations need not also appear in the `modrel` field. In fact, it is preferable to only include them in the `assign` field. For example, rather than having `x'=x+1` as a conjunct in the `modrel` field, you should include `x:=x+1` in the `assign` field.

In general, variables that appear on the left-hand side of an assignment statement should not appear primed in the modifying relation (there is the risk of unwillingly specifying an unsatisfiable transition relation). Still, there is the possibility that the user will include a conjunct of the form `x'=x+1` in the `modrel` field, or will mention a primed variable that is also assigned to. The parser will automatically discover these cases and normalize the transition relation, while printing a warning to the console.

A *transition* consists of one or more *transition relations*. This partition allows a more convenient representation of certain transition relations, e.g., those for the `while` and `if-then-else` SPL constructs.[3] A transition is considered enabled if at least one of its constituent transition relations is enabled. It is considered taken when one of its transition relations is taken. A transition can be taken only if it is enabled.

Transitions that only differ with respect to some parameters can be collected in one *parameterized* transition. The optional parameters are listed after the transition's name and declare one transition for each value assignment to the parameters.

A transition contains an optional *fairness requirement*, which can be `Just`, `Compassionate` or `NoFairness`. The default fairness requirement is `NoFairness`.

Example: The transition system in Figure 2.6 implements Euclid's algorithm for computing the greatest common divisor of two positive integers x and y.

```
Transition System % Euclid's algorithm
in a,b:  int where a > 0 /\ b > 0
local x,y:  int
out gcd:  int
Initially x = a /\ y = b
Transition t1 Just:
    enable x > y
    assign x := x - y
Transition t2 Just:
    enable x < y
    assign y := y - x
Transition t3 Just:
    enable x = y
    assign gcd := x
```

Figure 2.6: Euclid's algorithm for greatest common divisor

The system illustrates that no control locations need to be associated with the transitions when it is entered directly as a transition system. This contrasts with the transitions generated from SPL programs. An SPL program corresponding to the above

[3]The verification rules generate one verification condition for each transition relation, which may be simpler to manipulate than the verification condition for the entire transition. See Appendix A for the semantics of `while` and conditional statements. See [Manna and Pnueli, 1995] for an example where transition relations are used to simplify verification conditions.

transition system is shown in Figure 2.7. It generates the transition system shown in Figure 2.8.

```
in a,b:  int where a > 0, b > 0
local x,y:  int where x = a, y = b
out gcd:  int
l0:  [loop forever do
      [
        t1:  guard x > y do x := x - y
      or
        t2:  guard x < y do y := y - x
      or
        t3:  guard x = y do gcd := x
      ]
    ]
```

Figure 2.7: SPL program for Euclid's algorithm

```
Transition System
in a,b:  int
local x,y:  int
out gcd:  int
control pi0 :  [0..0]
Initially a > 0 /\ b > 0 /\ x = a /\ y = b /\ pi0 = 0
Transition t1 Just:
    enable pi0 = 0 /\ x > y
    assign x := x - y
Transition t2 Just:
    enable pi0 = 0 /\ x < y
    assign y := y - x
Transition t3 Just:
    enable pi0 = 0 /\ x = y
    assign gcd := x
```

Figure 2.8: Transition system obtained from the Euclid SPL program (Figure 2.7)

2.6.2 Transition System Syntax

The syntax of transition systems is given by the following grammar:

transition-system ::= `Transition System`

		{ *decl* } *[initial]* { *transition* }+
	|	`Clocked Transition System`
		{ *decl* } *[initial] [progress]* { *transition* }+
decl	*::=*	*type-decl* | *datatype-decl*
	|	*value-decl* | *macro-decl*
	|	*system-var-decl*
system-var-decl	*::=*	*mode ids* : *type [*`where` *expn]*
	|	`clock` *ids [*`where` *expn]*
mode	*::=*	`in` | `out` | `local`
initial	*::=*	`Initially` *expn*
progress	*::=*	`Progress` *expn*
transition	*::=*	`Transition` *name [* `[` *params* `]` *] [fairness]* : *relation* { ; *relation* }
relation	*::=*	*[* `modvar :` *ids]*
		[`enable :` *expn]*
		[`modrel :` *expn]*
		[`assign :` *assignment* { , *assignment* }*]*
name	*::=*	*id*
params	*::=*	*id* : *type* { , *id* : *type* }
fairness	*::=*	`Justice` | `Compassionate` | `NoFairness`
assignment	*::=*	*left-val* `:=` *expn*
left-val	*::=*	*expn*

Assignable expressions are built from system variables declared as `local`, `out` or `clock`, as well as array references to `local` or `out` arrays.

2.6.3 Clocked Transition Systems ⋆⋆

Clocked transition systems [Kesten *et al.*, 1996] can be used to model discrete or continuous real-time systems, and are partially supported in the educational release of STeP. Clocked transition systems differ from fair transition systems in the following ways:

- There are designated clock variables `T` (global time) and `tick` (time progress measure) which may not be changed by the standard transitions. Only `T` may be used in the standard transitions. Initially `T` is set to 0. These variables are declared automatically when a clocked transition system is specified.
- Fairness conditions on transitions are irrelevant, and therefore only the condition `NoFairness` is allowed.
- The user can specify a special *progress condition*. It restricts the amount that time is allowed to progress between the standard transitions.

The progress condition is written as:

`Progress` *expn*

where *expn* is assumed to be a first-order formula $F(clock_1, \ldots, clock_n, T)$, with the auxiliary clocks $clock_1, \ldots, clock_n$ and the global clock `T` as free variables. The progress condition corresponds to a transition of the form:

```
Transition tick:
    modvar      tick
    enable      tick > 0
           /\   Forall t :  rat .
                     (0 <= t /\ t <= tick -->  F(clock1+t,...,clockn+t,T+t))
    assign      (clock1,...,clockn,T) := (clock1+tick,...,clockn+tick,T+tick)
```

which is generated automatically by the parser. It is helpful to think about this transition as one that allows time to progress from `T` to `T + tick`, whenever the progress condition F is satisfied for all intermediary values between `T` and `T + tick`.

Clock variables are always of type `rat`.

2.7 When Parsing Fails

It is common for programs and specifications to fail to parse on the first try. The error messages generated by the parser typically indicate the line number and character range where the error was detected, separated by a ":", and provide some extra information about the offending part.

Unfortunately, the error messages are not always as illuminating as we would like. Here are some hints for when things don't work as expected.

- The different kinds of arrows can be confusing:

`<->`	is the temporal operator ⟐ ("*once*");
`<-->`	is double implication
`<=`	is "less than or equals"
`-->`	is normal implication, but
`-->`	is also used in function types
`==>`	is the temporal "entails" relation (`p==>q` stands for `[](p-->q)`), but
`==>`	is also used for channel operations in SPL
`<==>`	is a built-in macro, expanded to `[](p<-->q)`
`--->`	is used to specify rewrite rules

- A period (".") is expected after quantifiers.

- Variables declared in the specification file and quantified variables are rigid by default. To specify a flexible variable in quantification, just write

  `Forall x : int Flexible .` *expn*.

Even after parsing a file, watch out for the following:

- Arithmetization:

  `1 <= i <= N` is parsed as `(1 <= i) <= N`, hence as "the truth value of `1 <= i` is less than or equal to `N`", and not as `1 <= i /\ i <= N`.

- Rigid vs. flexible variables (see above).

- Scope of quantifiers and declarations:

  Binding operators have maximal scope. Hence

  ```
  Forall x : int . p(x) --> q
  ```

  is parsed as

  ```
  Forall x : int . (p(x) --> q)
  ```

  and not as

  ```
  (Forall x : int . p(x)) --> q
  ```

  although `x` is not free in q.

- When in doubt, add parentheses!

Chapter 3

The Top-Level Interface

Starting STeP brings up the Top-level Prover window (Figure 3.1), a point-and-click X-windows interface. The Top-level Prover window has the following main components:

1. At the top of the main STeP window is the *main menu*, used to load systems and specifications, save proof searches, view systems, select properties to be used in proofs, online help, and exit STeP.

2. Under the main menu is the *status line*. It displays the name of the current system file loaded, if any; the number of unfinished proof trees, the number of open goals in the current proof tree, the current simplifier settings, and whether automatic simplification is ON or OFF.

3. Below the status line is the *current goal window*, where the current unproven goal or subgoal is displayed. This window has a scroll bar at the left. Verification rules are applied to the formula that appears in this window.

4. Below the current goal window is the *output window*, which displays the results of applying rules and generating invariants. It will also state when a proof is complete, and generally leave a record of the system's activities. This window can be reset with the Clear text window item under the Settings pull-down menu.

5. Under the output window, at the bottom of the top-level prover, is STeP's *message/help line*. In this window, STeP indicates what action is being done, which dialog window is waiting for input, or if an action has terminated successfully or not. It will also display a brief description of any feature that is being selected. For example, if Undo is selected, this window will display "`Backtrack to parent`".

6. At the right side of the top-level window is the *action region*, a double column of buttons that invoke verification rules and decision procedures and apply them to the the current goal.

   Besides the (temporal) verification rules, buttons in this column invoke the automatic Simplifier or start up the Interactive Prover. The Previous and Next buttons cycle through the open goals in the current search.

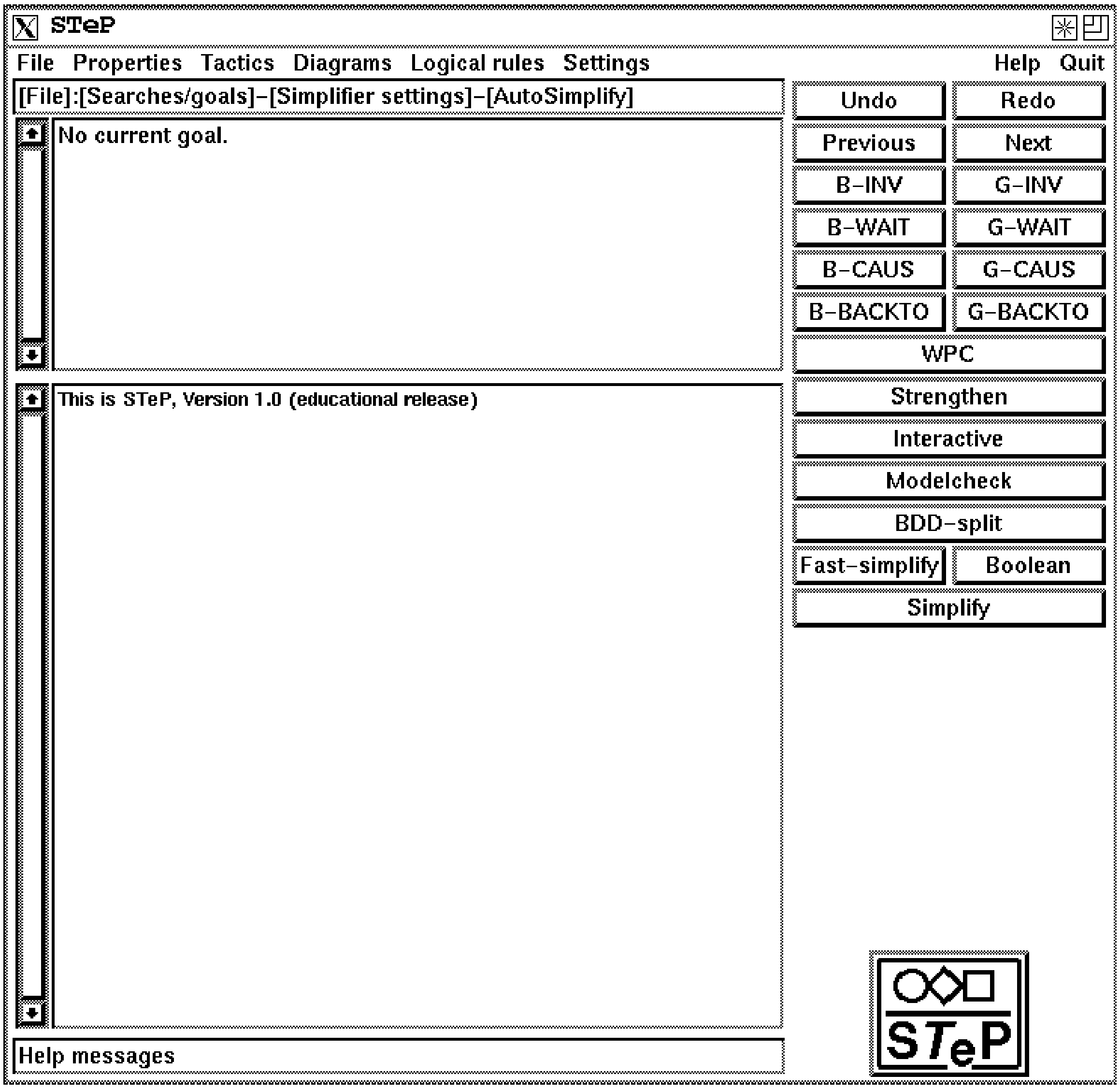


Figure 3.1: Top-level Prover window

7. In the lower right-hand corner of the window is a button with the STeP logo. This button functions as the *interrupt button*, and can be used to halt most of the automatic operations performed by STeP.

3.1 Menu Options

The File Menu

The File pull-down menu (Figure 3.2) is used to load system description and specification files, and to save finished and unfinished proofs.

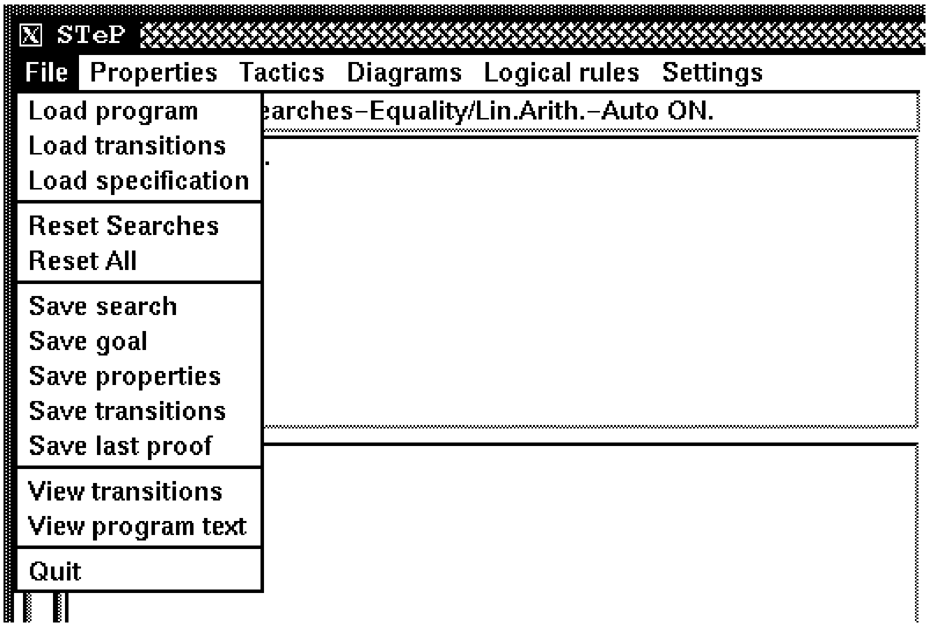


Figure 3.2: Top-Level Prover File Menu

Several options in this menu activate the *File Browser*, a utility that allows you to move around in the directory hierarchy and select files.

Activating the File Browser brings up a window with several panels as shown in Figure 3.3. The name of the window reflects the function that called it (in the figure, **Load program**). The window under `File name` shows the current file selected, and you can enter the filename here directly. The two buttons below allow you to display all files (*), or only those with the default extension for the current function (*.spl, in this case, since the default extension for programs is `.spl`). The left panel below these buttons shows the files in the selected directory, and the right panel shows the directory hierarchy. To move to another directory, click on the directory name in the right panel. To select a file, click on the filename in the left panel. To load or save a file, click OK after selecting the file or entering its name.

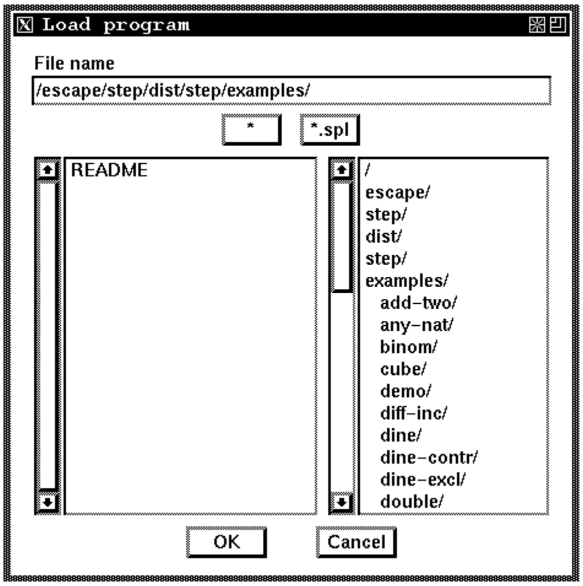


Figure 3.3: File Browser

If a file does not parse correctly, a **Read Error** window will appear with an indication of the first parsing error encountered. The file can be corrected and read again by clicking Try same file again, or the operation canceled with Cancel.

Load program Loads an SPL program from a file. The default extension for a program file is `.spl`. The program will be parsed into a fair transition system. The program text is displayed in a separate window (setting the `STEP_SHOW_LOADED` environment variable to OFF disables this—see Appendix E). The program text window may be deleted at any time by clicking Quit, and redisplayed with the View program text option below.

Figure 3.4 shows the program window for program MUX-PET1. The program window also displays the correspondence between the program locations and the value of the internal control location variable. Control locations are explained in detail in Appendix C.2.

If Load program is selected while another verification session is still in progress, a confirmation window will first appear. The entire verification session environment is reset when a new program is loaded, i.e., all background properties and unproven goals are deleted.

Load transitions The Load transitions option works like Load program, but expects a transition system file. The default extension for transition system files is `.trans`. The format for transition systems was given in Section 2.6.

As with programs, the transition system is displayed in a separate window (unless the `STEP_SHOW_LOADED` environment variable is `OFF`). Loading a new transition system

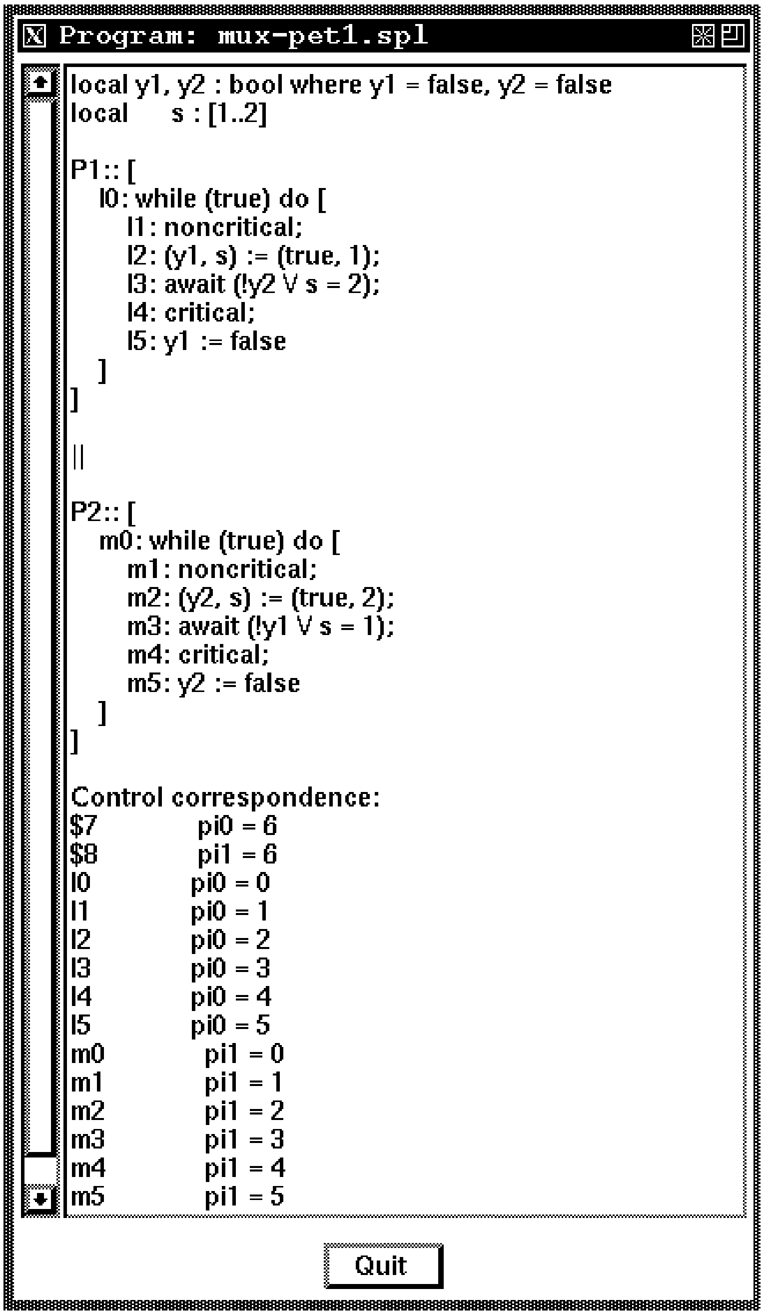


Figure 3.4: Program Window for program MUX-PET1

terminates the current verification session.

Load specification Loads a specification (a list of axioms and properties to be proved) from a file. The default extension of a specification file is `.spec`. The input format of specification files was given in Section 2.3. Each property becomes the root of a separate proof tree, and the first one appears in the current goal window. If a specification file is loaded during another goal session, the current proof search is suspended and the first property in the specification file will become the new current goal, as the root of a new proof search. The old proof search, however, remains in the system, and may be returned to later (e.g. with the select search option).

Reset Searches Selecting Reset Searches deletes all pending proof trees and goals, as well as all background properties. However, the current program or transition system is retained. Thus, it resets a verification session.

Reset All Terminates a system-verification or theorem-proving session: it removes all current proof trees, background properties, and the current system description, if any.

Save search Saves the current proof tree to a file, in ASCII format, for off-line inspection. Note that the current proof tree is necessarily unfinished, i.e., it still has open subgoals. Once all subgoals are closed the proof tree is removed from the list, and the next proof tree, if any, becomes the current one. To save the last complete proof, use Save last proof, described below.

Note: the current version of STeP does not support reloading this file to resume the proof. We plan to add this capability in future releases.

Save goal Saves the current goal to a file. The file will contain declarations for all free variables.

Save properties Saves all background properties to a file. The background properties include all properties proven and all invariants generated during the current program verification session, as well as all axioms asserted by the user.

Save transitions Saves the current transition system to a file; this is useful to see how SPL programs are converted to transition systems.

Save last proof Saves the finished proof of the most recently established goal.

View transitions Displays a separate window containing the current transition system. It can also be selected when the system description is loaded as an SPL program. The transition window can be deleted at any time by clicking on the $\boxed{\text{Quit}}$ button of this window.

View program text Displays a separate window with the text of the current SPL program, similar to View transitions above.

Quit Terminates the STeP session, after confirmation from the user.

The Properties Menu

The Properties pull-down menu (Figure 3.5) is used to manage the background properties, move between different proof trees, generate invariants, and enter new goals and axioms directly.

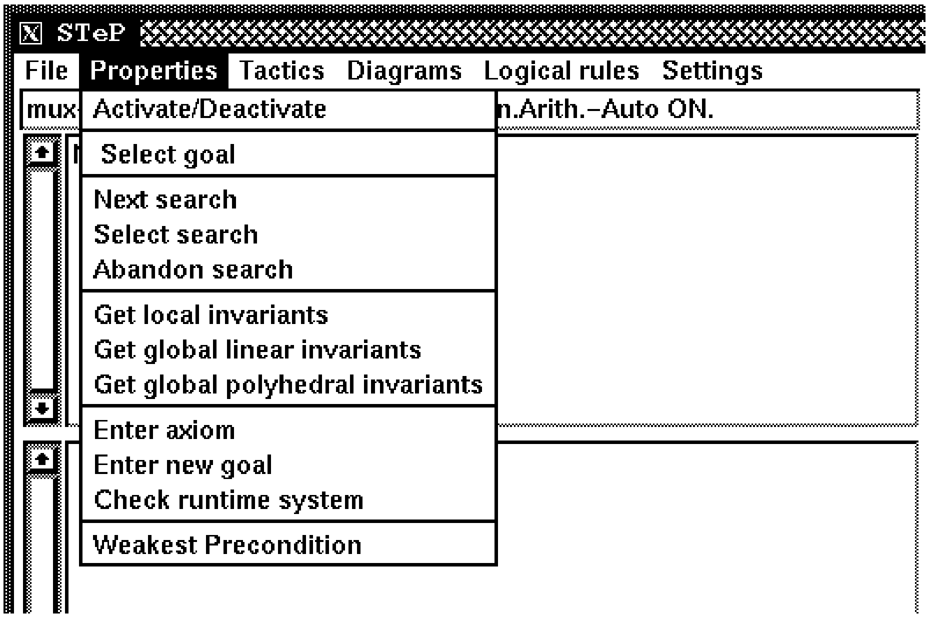


Figure 3.5: Properties menu

Activate/Deactivate This option displays a window with all the background properties, i.e., all axioms entered, invariants generated, and properties proven during the current system verification session. (See for example Figure 3.6, which displays the axioms, some invariants and a proven property `aux1` of the program GCD.) By default all properties are active, which means that they are used by the Simplifier. This window allows you to deactivate properties, or delete them altogether. A property is deactivated by clicking on the switch, and removed by clicking on the skull icon. The buttons Select All and Deselect All select or deselect all the listed properties.

No window will appear when there are no background properties.

Select goal This menu option allows you to move to another open goal in the current proof tree. When selected, it displays a window listing all open goals in the current proof tree (see, for example, Figure 3.7). Each goal has a select button. By clicking on Select the corresponding goal is made the current one. The Cancel button returns to the current goal.

Next search Moves to the next proof tree with open subgoals. The previous proof tree is kept in its current state, and may be returned to later. This option is useful if you reach a dead-end in a proof and realize you need to prove another property first.

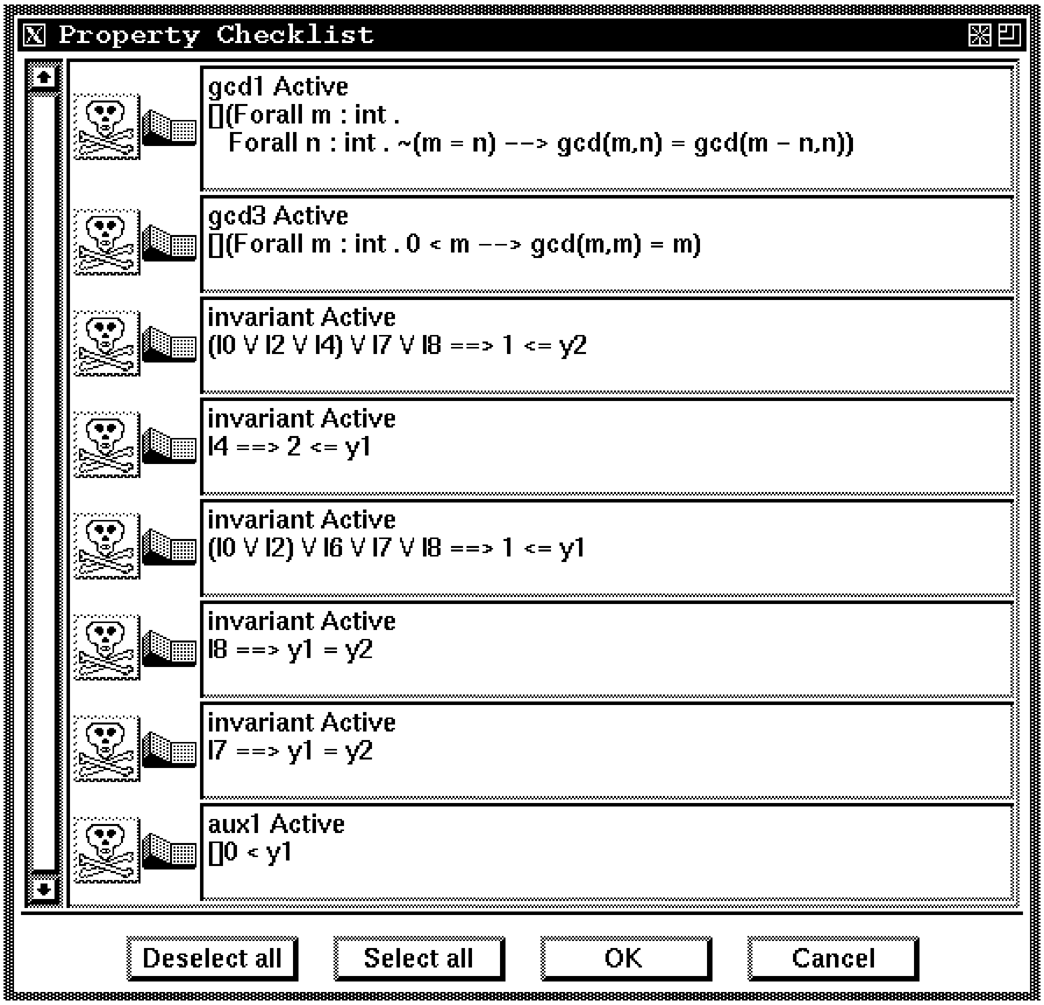


Figure 3.6: Activate/Deactivate Window

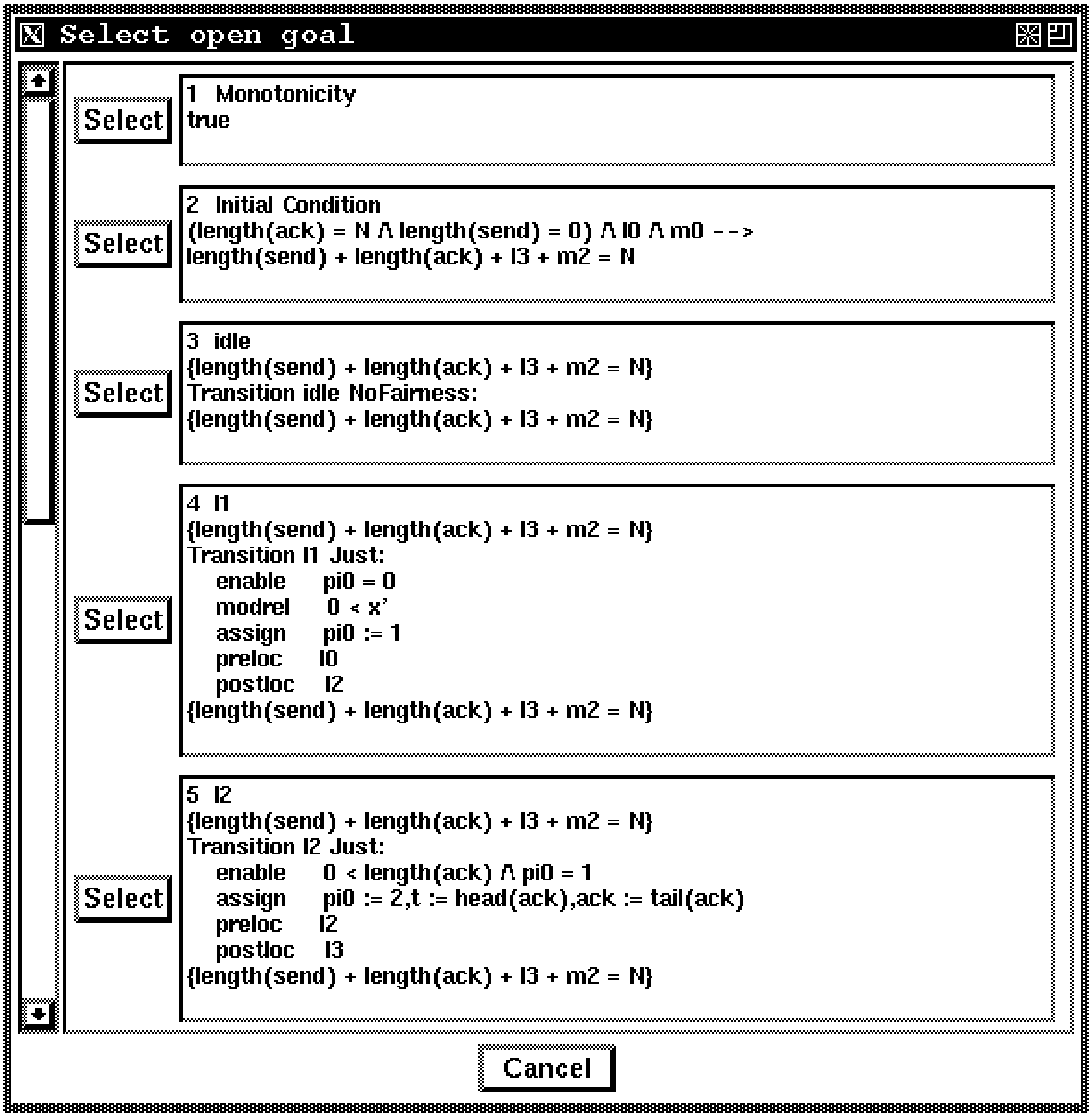


Figure 3.7: Goal selection window

Select search Generalizes Next search above to move to any other proof tree. A window listing the root nodes of all proof trees is displayed. See Figure 3.8 for an example. By clicking on the Select button, the corresponding proof tree is made the current proof tree.

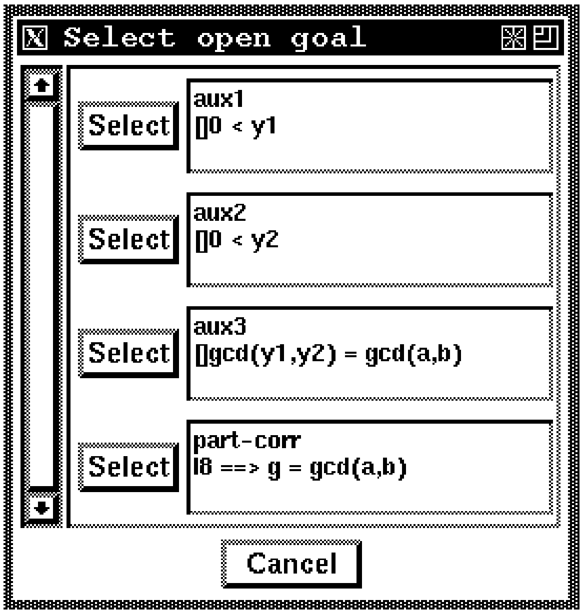


Figure 3.8: Proof tree selection window

No window will appear if there is nothing to select.

Abandon search Deletes the current proof tree and makes the next proof tree in the list, if any, the current proof tree. A confirmation window appears before this action is taken.

The following options generate program invariants. When the invariant generation is done, the invariants are displayed in the output window and added to the list of background properties. By default they are active, and thus used in the simplification of goals. They may be deactivated with the Activate/Deactivate menu option (above). When the invariant generation is finished a message will appear in the message window.

Newly generated invariants are simplified using the basic simplification and decision procedures, independently of previous background properties. Invariant generation and simplification can be halted by clicking on the interrupt button (the STeP logo).

Get local invariants Activates the local invariant generator; requires an SPL program.

Get global linear invariants Activates the linear invariant generator; requires an SPL program or fair transition system.

Get global polyhedral invariants Activates the polyhedral invariant generator, after prompting the user for a number of parameters that control the generation method. See Chapter 7 for details. Polyhedral invariants are generated for SPL programs and fair transition systems.

Enter axiom Used to enter a property that the system may assume to be true during the current program verification session, i.e., it is added to the list of background properties. By default, the new axiom is active, and will thus be used by the Simplifier.

Note: STeP does not guarantee the consistency of the set of background axioms, nor does it require you to prove any of them, so this feature should be used with caution.

Enter new goal Used to enter a new goal to be proven. If the formula parses correctly it will become the root of a new proof tree, which is made the current proof tree while the previous one is put on hold.

Check runtime system Generates a set of goals that, if proved valid, ensure that the transition system will not generate any runtime errors such as division by 0 or array references out of range.

Weakest precondition This option allows you to calculate *weakest precondition* for an arbitrary formula and transition in the current system (see Section 4.3).

The dialog window shown in Figure 3.9 (upper window) will appear; after entering the transition name and formula, clicking on the OK button will display the weakest precondition in a separate window (shown in Figure 3.9's lower window). The formula may be saved to a file with the Save button.

This function has no effect on the proof tree or on the background properties; it serves a purely informative role. Weakest precondition is also provided as an inference rule, useful for strengthening invariants, and is described in Section 4.3.

The Tactics Menu

The Tactics pull-down menu is used to run tactics loaded from a file or entered directly. Tactics can be run in two modes: *batchmode* and *interactive*. Batchmode tactics are applied without any user interaction; they may not always terminate, but can be interrupted by clicking on the STeP icon (the *interrupt button*).

For interactive tactics, a window appears with the first atomic step of the tactic. The Apply button applies the displayed step, and the next one will appear. In this way, you can apply tactic steps until the proof is finished, or at any point interrupt the tactic by clicking on the Quit button.

Tactics and their syntax are presented in Section 6.3.

Load batch tactic Reads a tactic from a file using the File Browser and applies it in batch mode.

Load interactive tactic Reads a tactic from a file and runs it in interactive mode.

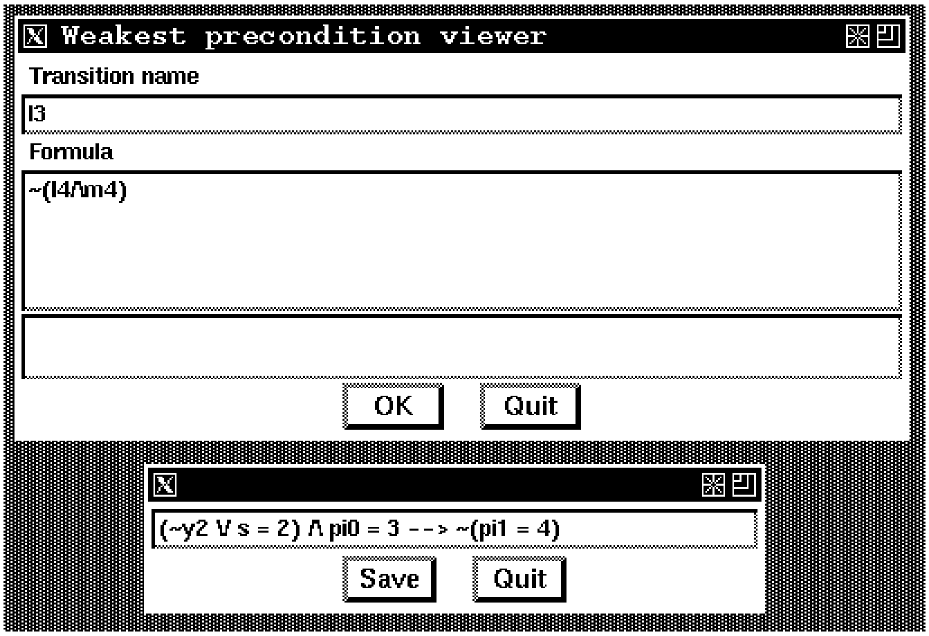


Figure 3.9: Weakest Precondition: entry and output window

Enter batch tactic Prompts the user to enter a tactic, which is then parsed and run in batch mode. Error messages from parsing the tactic are displayed in the bottom of the input window.

Enter interactive tactic Prompts the user to enter a tactic, which is then parsed and run in interactive mode.

The Diagrams Menu

The Diagrams pull-down menu is used to start up the Verification Diagram Editor and use verification diagrams in the current verification session. The Verification Diagram Editor may be used in parallel with the Top-level Prover, and is described in more detail in Chapter 5.

Edit diagram Starts the Verification Diagram Editor in a separate window.

Verification Diagram rule Uses the current verification diagram as a verification rule for the current goal or subgoal. It generates subgoals corresponding to all the verification conditions implicit in the diagram, if the diagram is applicable. Thus, the verification diagram can be seen as a specialized verification rule.

A verification diagram is chosen to be the current one by the Current-to-verify menu option in the Verification Diagram Editor.

The Logical rules Menu

The Logical rules pull-down menu (Figure 3.10) provides inference rules that simplify temporal properties. Thus, they can be used to derive new properties from previously proven ones. Unlike verification rules, these rules are independent of the system being verified. Section 4.4 presents a full description of these rules.

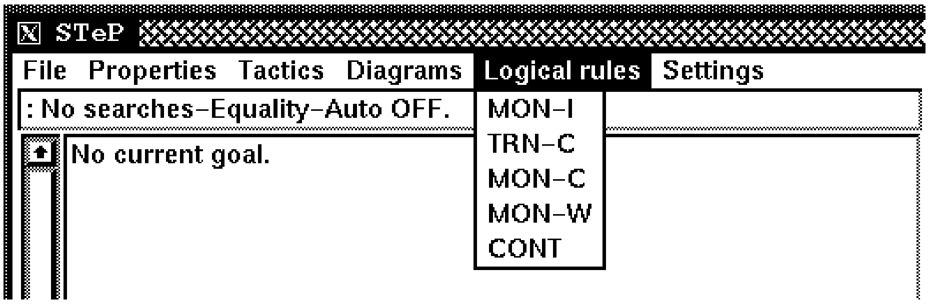


Figure 3.10: Logical Rules menu

The Settings Menu

The Settings pull-down menu (Figure 3.11) controls some of the global STeP environment variables, such as automatic simplification of subgoals and the strength of the Simplifier.

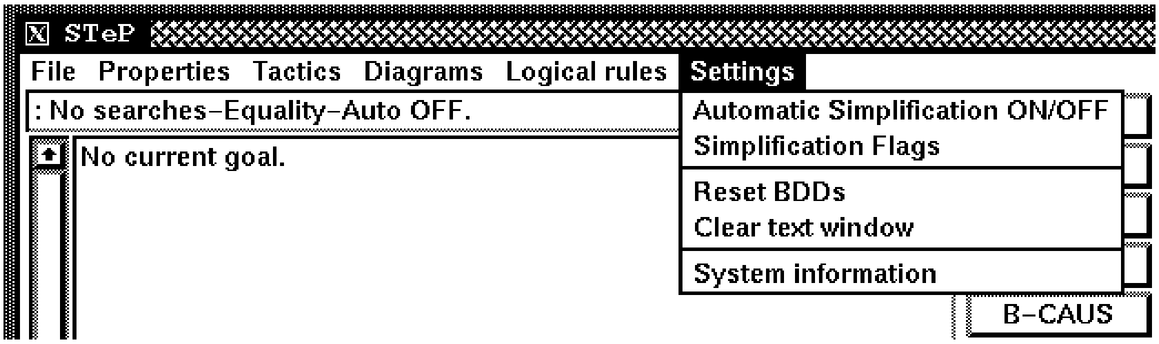


Figure 3.11: Settings menu

Automatic Simplification ON/OFF Sets and resets automatic simplification; see Section 4.1.

Simplification Flags Brings up a window used to set the default strength of the Simplifier. See Section 6.1 for more details.

Reset BDDs Resets the internal BDD package; if the package is used often, this option can be used sporadically to save memory, at some (usually small) short-term efficiency loss (see Section 6.2). This option should also be used if the number of BDD nodes reaches the maximum given by the `STEP_BDD_NODES` environment variable (see Section E).

Clear text window Clears the output window, but has no other effect on the system.

System information Prints out miscellaneous statistics (in the output window) for the current verification session, mostly related to the BDD package (see Section 6.2).

The Help menu

On-line help is available via your favorite WWW browser when clicking on the Help menu near the right corner of the top-level interface. The hyper-text help pages give an alternative and often quicker introduction to STeP's facilities.

3.2 Action Buttons

The right-hand column of the Top-level Prover contains rules to manipulate and construct the main proof search. The first four buttons manipulate the structure of the proof tree:

Undo This button makes the parent of the current subgoal into the current goal. The previous goal is remembered, to enable a Redo, until another verification rule is applied to the new goal.

Redo Redoes a previous Undo, that is, restores the part of the proof tree eliminated by Undo.

Previous Postpones the current goal, and turns the previous subgoal in the subgoal queue into the current subgoal.

Next Postpones the current subgoal and moves to the next subgoal in the subgoal queue.

The next set of buttons invoke verification rules as described in [Manna and Pnueli, 1995]. These rules are not applicable if there is no current program or transition system.

Verification rules:

B-INV	G-INV
B-WAIT	G-WAIT
B-CAUS	G-CAUS
B-BACKTO	G-BACKTO

See Section 4.2 for a description of the main verification rules in their basic and general versions. Section 4.2 also describes the WPC and Strengthen proof rules.

The remaining buttons in the action region invoke the Interactive Prover and the Simplifier as proof rules, applied to the current goal:

$\boxed{\text{Interactive}}$ Invokes the Interactive Prover on the current goal. The Interactive Prover window will appear with the current goal as the single consequent in the sequent. Once the Interactive Prover is active, the Top-level Prover is suspended, to be reactivated when the Interactive Prover session is finished.

Upon exiting the Interactive Prover you have the option to return the result to the Top-level Prover. If the goal was proved, then the current goal is closed, and the next goal in the proof tree, if any, becomes the current goal. If the goal was not proved, a formula equivalent to the remaining subgoals is returned to the Top-level Prover as a new subgoal.

The Interactive Prover is described in Chapter 8.

$\boxed{\text{Modelcheck}}$ Invokes the Model Checker on the current goal. Before the Model Checker starts, you can enter additional assumptions to limit the search space. If the goal is proved without any additional assumptions, the current goal is closed, and the next goal in the proof tree, if any, becomes the current goal. If the goal is not proved, it remains unchanged.

If the goal is proved under additional assumptions, it is not added to the set of background properties.

The Model Checker results are reported in a logfile. See Section 4.5 for a more detailed description of the modelchecker.

$\boxed{\text{BDD-split}}$, $\boxed{\text{Fast-simplify}}$, $\boxed{\text{Boolean}}$, $\boxed{\text{Simplify}}$ These buttons invoke various versions of the automatic Simplifier; while $\boxed{\text{BDD-split}}$ can generate a set of subgoals, the others can only reduce the formula to a simpler one. See Section 6.1 for details.

STeP **logo** The STeP logo serves as the interrupt button, and can be used to exit from each of the following time-consuming operations:

- Simplification of verification conditions
- Generation and simplification of invariants
- Model checking

Chapter 4

Verification and Logical Rules

The main function of the Top-level Prover is to apply *verification rules* to temporal properties, reducing them to simpler verification conditions. These verification conditions become subgoals in the top-level proof tree. The verification rules depend on the system being verified. In contrast, a set of *logical rules* reduces temporal properties independent of the system being verified.

This chapter describes the general structure of the proof search and the rules used to construct it in the Top-level Prover.

4.1 The Proof Search

The structure of the proof search is similar in the Top-level Prover and the Interactive Prover: the proof is a tree, with the original goal as its root node. Nodes are labeled with subgoals, called *verification conditions* in the Top-level Prover and *sequents* in the Interactive Prover, and the application of different rules generates new subgoals (or children nodes) for the current node.

Since more than one proof rule can be applied at each node, the process is a *search*. The proof is finished when there are no subgoals left; the number of subgoals is reduced whenever a simplification step reduces a subgoal to *true*, which is the only formula that requires no proof.

In the Top-level Prover, the rules applied are temporal verification rules, logical rules, or simplification steps. In addition, rules that can strengthen a verification condition are available, such as weakest precondition. Finally, goals in the Top-level Prover can also be closed by sessions of the Interactive Prover.

The Background Properties:

Background properties include axioms, automatically generated invariants, and previously established properties. When a top-level proof is completed, the property is added to the list of background properties.

In the Top-level Prover, simplification is carried out relative to the properties and axioms that are active at the time (except for the $\boxed{\text{Fast-simplify}}$ option, discussed in Section 6.1).

In the Interactive Prover, these background properties have to be explicitly added by the user to the current sequent before the simplification rule is called.

Navigating the Proof Tree:

The options Postpone (in the Interactive Prover), Previous, Next (in the Top-level Prover), and Select goal (in both provers) can be used to jump from one open node to another. The Undo and Redo buttons alternate between a given node and its children.

In the Top-level Prover, any number of proof searches can be conducted simultaneously, sharing the same set of background properties. The Next search button can be used to jump from one proof search to the next.

Automatic Simplification

A verification rule can generate a large number of verification conditions, depending on the size of the underlying transition system. After having applied a verification rule a message in the message window will display how many subgoals were generated. If automatic simplification is `OFF`the first subgoal will be displayed in the specification window. All the generated subgoals can be viewed by selecting the Select goal option of the Properties pull-down menu.

If automatic simplification is `ON`, the Simplifier will be applied to each subgoal. Only those subgoals that fail to simplify to *true* will remain. Simplifications steps which fail to simplify to *true* will be undone to enable subsequent application of rules such as Strengthen and WPC; these are only applicable to unsimplified verification conditions, since they require knowing the particular transition that generates the verification condition.

When automatic simplification is `OFF`, a verification rule only generates subgoals, but does not simplify them. These then have to be manually simplified by successive clicks on Simplify, or with the Interactive Prover. If you unintentionally invoke a verification rule with automatic simplification `OFF`, you can click on Undo, turn automatic simplification `ON`, and apply the rule again.

4.2 Verification Rules

This section describes the verification rules available in STeP. In the following description, Θ refers to the initial condition of the current program or transition system, and $\mathcal{T}$ refers to its set of transitions (see Appendix A). For a formula p, p_0 is the *initial version* of p, that is, the first-order formula equivalent to p at the initial state of the computation.

As usual, the notation $\{p\}\tau\{q\}$, stands for the verification condition $(p \wedge \rho_\tau) \Rightarrow q'$. The *inverse verification condition*, $\{p\}\tau^{-1}\{q\}$, is equivalent to $\{\neg q\}\tau\{\neg p\}$, giving an alternate presentation of verification conditions. Below, we try to use the most convenient one in each case. The notation $\{p\}\mathcal{T}\{q\}$ stands for the set of verification conditions $\{p\}\tau\{q\}$ for each transition τ in the system; similarly, $\{p\}\mathcal{T}^{-1}\{q\}$ is the set $\{p\}\tau^{-1}\{q\}$ for each τ.

Verification conditions of the form $\alpha \Rightarrow \beta$ are replaced by $\alpha \rightarrow \beta$ when α and β are *assertions*, that is, state-formulas with no temporal operators. In this case, the P-validity of $\alpha \rightarrow \beta$ implies that of $\alpha \Rightarrow \beta$. If this P-validity cannot be established, the invariant $\alpha \Rightarrow \beta$ may be proved separately, for an incremental proof.

- $\boxed{\textsf{B-INV}}$ Applies the *basic invariance rule* to the current subgoal, which has to be of the form $\square\, p$, for a past formula p. This rule generates the following subgoals:

$$\begin{array}{rl} p \Rightarrow p & \text{Monotonicity} \\ \Theta \rightarrow p_0 & \text{Initial condition} \\ \{p\}\tau\{p\}, & \text{for each } \tau \in \mathcal{T} \end{array}$$

  The monotonicity subgoal is generated because this rule is based on rule G-INV, below; this goal always trivially simplifies to *true*.

- $\boxed{\textsf{G-INV}}$ Applies the *general invariance rule* to the current subgoal. The current goal has to be of the form $\square\, q$, with q a past formula. After clicking, a dialog window will appear that allows you to enter a strengthened version p of q. The default is q. After entering the auxiliary assertion p, the following subgoals are generated:

$$\begin{array}{rl} p \Rightarrow q & \text{Monotonicity} \\ \Theta \rightarrow p_0 & \text{Initial condition} \\ \{p\}\tau\{p\} & \text{for each } \tau \in \mathcal{T} \end{array}$$

- $\boxed{\textsf{B-WAIT}}$ Applies the basic *nested wait-for* rule. This rule is applicable only if the current goal is a nested wait-for formula, that is, it is of the form

$$p \Rightarrow q_n \,\mathcal{W}\, q_{n-1} \,\mathcal{W}\, q_{n-2} \ldots q_1 \,\mathcal{W}\, q_0,$$

  where $p, q_n, \ldots q_0$ are past formulas. Informally, the nested wait-for formula asserts that whenever p becomes true, when there is a, possibly empty, interval where q_n holds, followed by a q_{n-1} interval, up to q_0 becomes true, unless one of the previous q's remains true infinitely after. After clicking on this button the following subgoals are generated:

$$\begin{array}{rl} p \Rightarrow \bigvee_{i=0}^{n} q_i & \text{Premise N1} \\ q_i \Rightarrow q_i & \text{Monotonicity } i\text{, for each } i = 0 \ldots n \\ \{q_i\}\mathcal{T}\{\bigvee_{j \leq i} q_j\} & \text{for each } i = 1 \ldots n \end{array}$$

  The monotonicity conditions are generated because the implementation of B-WAIT is based on that of G-WAIT, below. They always trivially simplify to *true*.

- $\boxed{\textsf{G-WAIT}}$ This button invokes the general *nested wait-for* rule. This rule is applicable only if the current goal is a nested wait-for formula, that is, it is of the form

$$p \Rightarrow q_n \,\mathcal{W}\, q_{n-1} \,\mathcal{W}\, q_{n-2} \ldots q_1 \,\mathcal{W}\, q_0,$$

  where $p, q_n, \ldots q_0$ are past formulas. A dialog window will appear which allows you to enter the auxiliary assertions $\varphi_n, \ldots, \varphi_0$. Their defaults are $q_n, \ldots, q_0$, respectively. After entering these assertions the following subgoals are generated:

$$\begin{array}{rl} p \Rightarrow \bigvee_{i=0}^{n} \varphi_i & \text{Premise N1} \\ \varphi_i \Rightarrow q_i & \text{Monotonicity } i\text{, for each } i = 0 \ldots n \\ \{\varphi_i\}\mathcal{T}\{\bigvee_{j \leq i} \varphi_j\} & \text{for each } i = 1 \ldots n \end{array}$$

- $\boxed{\text{B-CAUS}}$ The basic causality rule is a specialization of the basic invariance rule. It is applicable to goals of the form

$$p \Rightarrow \diamondminus q,$$

where p and q are past formulas. It usually generates simpler verification conditions than its counterpart B-INV, which is also applicable to goals of this form, by taking into account the special form of the goal.

This rule generates the following subgoals:

$$\begin{array}{rl} p \Rightarrow p \vee q & \text{Monotonicity} \\ \Theta \rightarrow \neg(p)_0 \vee (q)_0 & \text{Initial condition} \\ \{p\}\mathcal{T}^{-1}\{p \vee q\} & \end{array}$$

- $\boxed{\text{G-CAUS}}$ The general causality rule specializes the general invariance rule. It is applicable to goals of the form

$$p \Rightarrow \diamondminus q,$$

where p and q are past formulas. It usually generates simpler verification conditions than its counterpart G-INV, which is also applicable to goals of this form, by taking into account the special form of the formula to be proved.

After clicking on this button a dialog window will appear, which allows you to enter an intermediate formula φ (default q). After φ is entered, the following subgoals are generated:

$$\begin{array}{rl} p \Rightarrow \varphi \vee q & \text{Monotonicity} \\ \Theta \rightarrow \neg(\varphi)_0 \vee (q)_0 & \text{Initial condition} \\ \{\varphi\}\mathcal{T}^{-1}\{\varphi \vee q\} & \end{array}$$

- $\boxed{\text{B-BACKTO}}$ The *basic back-to rule* establishes properties of the form

$$p \Rightarrow q_n \,\mathcal{B}\, \ldots \,\mathcal{B}\, q_0$$

for past formulas $p, q_n, \ldots, q_0$. It generates the following subgoals:

$$\begin{array}{rl} p \Rightarrow \bigvee_{i=0}^{n} q_i & \text{Premise N1} \\ q_i \Rightarrow q_i & \text{Monotonicity for each } i \text{ in } 1 \ldots n \\ \{q_i\}\mathcal{T}^{-1}\{\bigvee_{j \leq i} q_j\} & \text{for each } i \text{ in } 1 \ldots n \end{array}$$

- $\boxed{\text{G-BACKTO}}$ The *general back-to* rule generalizes the above by allowing strengthening of the q_i's and weakening of p. To prove

$$p \Rightarrow q_n \,\mathcal{B}\, \ldots \,\mathcal{B}\, q_0,$$

given formulas $\varphi_0, \ldots, \varphi_n$, the following subgoals are generated:

$$\begin{array}{rl} p \Rightarrow \bigvee_{i=0}^{n} \varphi_i & \text{Premise N1} \\ \varphi_i \Rightarrow q_i & \text{Monotonicity for each } i \text{ in } 1 \ldots n \\ \{\varphi_i\}\mathcal{T}^{-1}\{\bigvee_{j \leq i} \varphi_j\} & \text{for each } i \text{ in } 1 \ldots n \end{array}$$

4.3 WPC and Strengthening

A basic and often successful heuristic for constructing inductive invariance properties is based on *weakest preconditions*.

- $\boxed{\textsf{WPC}}$ The weakest precondition verification rule can be used to prove a verification condition that does not itself simplify to *true*. The rule is applicable to subgoals of the form

  $$\{\varphi\}\tau\{\psi\}$$

  and when applied it generates the subgoal

  $$\text{Weakest Precondition} \qquad \Box(\rho_\tau \rightarrow \psi')$$

  which implies the original verification condition.

  The symbol ρ_τ refers to the transition relation, and is the logical form of the transition τ. The formula ψ' is the primed version of ψ, in which each system variable occurring free in ψ has been replaced by its primed version. Appendix A describes transitions and transition relations in more detail.

- $\boxed{\textsf{Strengthen}}$ The strengthening rule can be used to strengthen a property that is not inductive. The rule is applicable to subgoals of the form

  $$\{\varphi\}\tau\{\psi\}$$

  and when applied it generates the subgoal

  $$\Box(\varphi \wedge (\rho_\tau \rightarrow \psi')).$$

  while deleting the original verification condition and its siblings from the proof tree.

4.4 Logical Rules

The Logical rules pull-down menu provides inference rules that allow the derivation of new properties from previously proven properties.

If the current formula does not match the kind of formula that the given rule applies to, a message will appear in the bottom help window, and the given function will have no effect.

MON-I The *monotonicity rule for invariances* can be used to prove a property of the form $\Box\, p$, where p is a state formula. If the current goal is $\Box\, p$, the following subgoal is generated:

$$\bigwedge_i \varphi_i \;\rightarrow\; p$$

where φ_i ranges over all the currently active in the background properties.

TRN-C The *transitivity of causality rule* proves a causality property from two other causality properties. If the current goal has the form $p \Rightarrow \diamondminus r$, this option displays a dialog window for entering an auxiliary assertion. After a new assertion q is entered (default is r), the following two subgoals are generated:

$$\begin{array}{ll} p \Rightarrow \diamondminus q & \text{Left Transitivity} \\ q \Rightarrow \diamondminus r & \text{Right Transitivity} \end{array}$$

MON-C The *monotonicity rule for causality* can prove a causality property from a stronger causality property. If the current goal has the form $p \Rightarrow \diamondminus u$, for past formulas p and u, this option generates a dialog window for entering two auxiliary assertions. After entering the two assertions q and r (defaults are p and u) the following three subgoals are generated:

$$\begin{array}{rl} p \Rightarrow q & \text{Monotonicity 1} \\ q \Rightarrow \diamondminus r & \text{Causality} \\ r \Rightarrow u & \text{Monotonicity 2} \end{array}$$

MON-W The *monotonicity rule for wait-for formulas* can be used to prove a wait-for formula from a stronger wait-for formula. If the current goal has the form $p \Rightarrow q_n \mathcal{W} \ldots \mathcal{W} q_0$, a dialog window will appear for entering $n+2$ auxiliary assertions. The default assertions are p, q_n,..., q_0. After entering these assertions, p_{new}, $q_{n,new}$, ... $q_{0,new}$ and clicking the $\boxed{\textsf{OK}}$ button, the following $n+3$ subgoals are generated:

$$\begin{array}{rl} p \Rightarrow p_{new} & \text{Monotonicity} \\ q_{n,new} \Rightarrow q_n & \text{Monotonicity } n \\ \vdots & \\ q_{0,new} \Rightarrow q_0 & \text{Monotonicity } 0 \end{array}$$

and

$$p_{new} \Rightarrow q_{n,new} \mathcal{W} \ldots \mathcal{W} q_{0,new}$$

CONT The *contradiction rule* can be used to prove a general safety property by converting it into a causality formula.

If the current goal is of the form $\square \neg p$, for past formula p, the following subgoal will be generated:

$$p \Rightarrow \diamondminus \texttt{false} \qquad \text{Contradiction}$$

4.5 The Model Checker

Model checking determines if a given temporal formula holds for a given system by a systematic exploration of the state-space of the system. STeP includes a model checker for transition systems and formulas of linear-time temporal logic, which can be used to close subgoals or produce counterexamples to particular properties.

The STeP model checker checks the current goal under an optional additional first-order assumption. If the goal is found to be false, a counterexample is reported.

When the $\boxed{\textsf{Modelcheck}}$ proof rule is invoked, a dialog window appears requesting the input of an auxiliary first-order assumption, by default *true*. You may enter restrictions on parameters or system variables with infinite domains, in order to make the program finite-state. For example, for the program MPX-SEM you may enter `M=2`, indicating that the number of processes you want to consider is two.

The goal is closed only if the Model Checker proves it with assumption *true*.

The Model Checker Preprocessor

The Model Checker first applies a preprocessor to the given system and the goal. If possible, a finite number of problem instances with constant-size arrays and constant-bounded parameters equivalent to the original goal is produced. If this is not possible, the Model Checker terminates, control returns to the Top-level Prover and a message is displayed. Some of the tasks accomplished by the preprocessor are:

- simplification of data types;
- elimination of unnecessary variables;
- instantiation of parameters and other variables;
- introduction of additional transitions corresponding to possible run-time errors;
- general simplification of the transition system and of the property, using the Simplifier.

Note that the additional assumption can be used to restrict the range of a variable on which array or parameter bounds depend.

The Core Model Checker

For each problem instance, an external core model checker is invoked. It searches the state-space and stops when

1. a run-time error is found,
2. a counterexample computation is found, or
3. the entire state space has been explored.

The main algorithm used by the core model checker is adapted from [Hojati *et al.*, 1993].

Restrictions

Not all programs and properties accepted by STeP can be model checked. The datatypes handled by the core model checker are: boolean, integer, and channels, and arrays over booleans and integers. The preprocessor can reduce some other datatypes to these. The model checker can only handle basic logic, arithmetic, and channel operations applied to these variables. Formulas given to the Model Checker cannot have temporal operators in the scope of binding operators.

Log file

The model checking results are reported in a *log file*, selected by the user before model checking begins.

After calling the core model checker, the log file will contain: the list of relevant system variables, the preprocessed transition system, some statistical information, and, if the formula proved to be invalid, a counterexample computation.

Environment Variables

The search space may be infinite if there are system variables that range over an infinite domain, and in this case the Model Checker may or may not terminate. Even if the system is finite-state, the state-space may prove too large for the time and memory available. The following settings can be used to limit the Model Checker resources:

- `STEP_MC_SPACE`: Maximum amount of memory, in Megabytes.
  Default: unlimited.
- `STEP_MC_TIME`: Maximum amount of user time, in minutes.
  Default: unlimited.
- `STEP_MC_CPU`: Maximum amount of CPU time, in minutes.
  Default: unlimited.

The Model Checker can be interrupted with STeP's *interrupt button* (the STeP logo). NOTE: This may leave a UNIX process running in the background; see the UNIX `kill` man pages for details on how to kill runaway processes.

Chapter 5

The Verification Diagram Editor

Verification diagrams are a visual representation of a proof. A verification diagram describes sets of states given by assertions and possible transitions between them. A well-formed verification diagram represents a set of verification conditions that are sufficient to establish a given temporal formula. Verification diagrams are described in detail in [Manna and Pnueli, 1994] and [Manna and Pnueli, 1995].

STeP currently supports three types of verification diagrams:

- *Invariance diagrams*, to represent proofs of invariance formulas: $\square\, p$.
- *Wait-for diagrams*, to represent proofs of nested wait-for formulas:

$$p \Rightarrow q_n \mathcal{W} q_{n-1} \ldots \mathcal{W} q_0.$$

- *Chain diagrams*, to represent proofs of response formulas $p \Rightarrow \diamondsuit\, q$ that do not require induction.

Here, p, q, $q_0, \ldots, q_n$ are state assertions.

Hierarchical verification diagrams (Section 5.7) support distributing a large verification diagram over several files.

5.1 Definitions

A verification diagram is a directed labeled graph constructed as follows:

- *Nodes* in the graph are labeled by assertions.
- *Edges* in the graph represent transitions between nodes. Each edge connects one node to another and is labeled by the name of a transition in the program. We refer to an edge labeled by τ as a *τ-edge*.
- One of the nodes may be designated as a *terminal node*. This node is distinguished by having a boldface boundary. No edges may depart from a terminal node.

Verification diagrams represent sets of verification conditions as follows: For every non-terminal node (labeled by) φ and transition τ, let $\varphi_1, \ldots, \varphi_k$ be the nodes to which φ is connected by τ-edges. The *verification condition associated with φ and τ* is given by:

$$\{\varphi\}\ \tau\ \{\varphi \vee \varphi_1 \vee \ldots \vee \varphi_k\}.$$

Note that if $k = 0$, i.e., no τ-edges depart from φ, the verification condition is:

$$\{\varphi\}\tau\{\varphi\}.$$

No verification conditions are associated with terminal nodes. A diagram is called *valid over program P* (*P-valid*) if all the verification conditions associated with nodes of the diagram are P-state valid. Thus, a valid diagram provides a succinct representation of the verification conditions of the form $\{\varphi\}\tau\{\psi\}$, for all combinations of assertions φ and ψ and transitions τ that appear in the diagram.

5.2 Wait-for Diagrams

A verification diagram is called a WAIT-FOR diagram if there is only one terminal node and it is labeled by φ_0 and the diagram is weakly acyclic, i.e., whenever node φ_i is connected by an edge to node φ_j, $i \geq j$. P-valid WAIT-FOR diagrams can be used to establish the P-validity of a nested wait-for formula.

A P-valid WAIT-FOR diagram establishes that the formula

$$\left(\bigvee_{j=0}^{m} \varphi_j\right) \Rightarrow \varphi_m \mathcal{W} \varphi_{m-1} \ldots \varphi_1 \mathcal{W} \varphi_0$$

is P-valid. To establish the P-validity of

$$p \Rightarrow q_m \mathcal{W} q_{m-1} \ldots \mathcal{W} q_0$$

the following additional verification conditions (referred to as *side verification conditions*) have to be established:

$$p \rightarrow \textstyle\bigvee_{j=0}^{m} \varphi_j \qquad \text{(W1)}$$

$$\varphi_i \rightarrow q_i \quad \text{for } i = 0, 1, \ldots, m. \quad \text{(W2)}$$

5.3 Invariance Diagrams

A verification diagram is called an INVARIANCE diagram if it is exit-free, i.e., it has no terminal node. Unlike WAIT-FOR diagrams, INVARIANCE diagrams may contain cycles.

A P-valid INVARIANCE diagram with nodes $\varphi_1, \ldots, \varphi_m$ establishes that the formula

$$(\bigvee_{j=1}^{m} \varphi_j) \Rightarrow \Box(\bigvee_{j=1}^{m} \varphi_j)$$

is P-valid. To establish the validity of $\Box q$ the following two side verification conditions have to be established:

$$\Theta \rightarrow \bigvee_{j=1}^{m} \varphi_j \quad \text{(I1)}$$

$$(\bigvee_{j=1}^{m} \varphi_j) \rightarrow q \quad \text{(I2)}$$

5.4 Chain Diagrams ⋆

Verification diagrams can also represent fairness requirements. The main extension over the previous types of diagrams is the introduction of two additional types of edges, *double* and *solid*. A double edge corresponds to a helpful transition that is just, and a solid edge corresponds to a helpful transition that is compassionate. Thus, edges connecting nodes in a diagram can be single, double or solid.

A verification diagram is said to be a CHAIN diagram if its nodes are labeled by assertions $\varphi_0, \ldots, \varphi_m$, with φ_0 being the terminal node, and if it satisfies the following requirements:

- If a single edge connects node φ_i to node φ_j, then $i \geq j$.
- If a double or solid edge connects φ_i to φ_j, then $i > j$.
- Every node φ_i, $i > 0$, has at least one departing edge which is either double or solid. The transition labeling this edge is identified as helpful for assertion φ_i.
- Every transition can label at most one type of edge (single, double, or solid) departing from the same node.

The first two requirements ensure that the diagram is weakly acyclic in the same sense as defined for the WAIT-FOR diagram. The stronger second requirement ensures that the subgraph based on the double and solid edges is acyclic, forbidding self-connections by double or solid edges. The third requirement demands that every nonterminal node has at least one helpful transition associated with it.

Chain diagrams allow proving progress properties that do not require an infinite size well-founded domain. Later releases of STeP will allow the user to enter well-founded domains as part of the chain diagram.

Verification Conditions

Let φ be a nonterminal node and $\varphi_1, \ldots, \varphi_k$, $k \geq 0$, be the τ-successors of φ.

- If τ labels a single edge, we associate with φ and τ the usual verification condition

$$\{\varphi\}\tau\{\varphi \vee \varphi_1 \vee \ldots \vee \varphi_k\}.$$

- If τ labels a double or solid edge, we associate with φ and τ the verification condition

$$\{\varphi\}\tau\{\varphi_1 \vee \ldots \vee \varphi_k\}.$$

- If τ labels a double edge departing from φ, we associate with φ and τ the implication

$$\varphi \rightarrow En(\tau).$$

i.e., when φ holds, τ is enabled.

- If τ labels a solid edge departing from φ, we associate with φ and τ the compassion requirement in the form of the response formula

$$\varphi \Rightarrow \diamondsuit(\neg\varphi \vee En(\tau)).$$

Valid Chain Diagrams

A CHAIN diagram is *P-valid* if the first three types of verification conditions are P-state valid and the compassion requirements associated with solid edges, are P-valid.

A P-valid CHAIN diagram establishes that the response formula

$$(\bigvee_{j=0}^{m} \varphi_j) \Rightarrow \diamondsuit \varphi_0$$

is P-valid. To establish the P-validity of

$$p \Rightarrow \diamondsuit q$$

the following two additional side verification conditions have to be established:

$$p \rightarrow \bigvee_{j=0}^{m} \varphi_j \quad \text{(FC1)}$$

$$\varphi_0 \rightarrow q \qquad \text{(FC2)}.$$

5.5 Compound Nodes

To make verification diagrams more readable, STeP supports encapsulation conventions similar to those of Statecharts [Harel, 1987]. The basic encapsulation construct is the *compound node* containing *internal nodes.* We refer to the contained nodes as the *descendants* of the compound node. Nodes that are not compound are called *basic nodes.*

Departing edges An edge departing from a compound node is interpreted as though it departed from each of its descendants.

Arriving edges An edge arriving at a compound node is interpreted as though it arrived at each of its descendants. An assertion φ labeling a compound node is interpreted as though it were a conjunct added to each of its descendants.

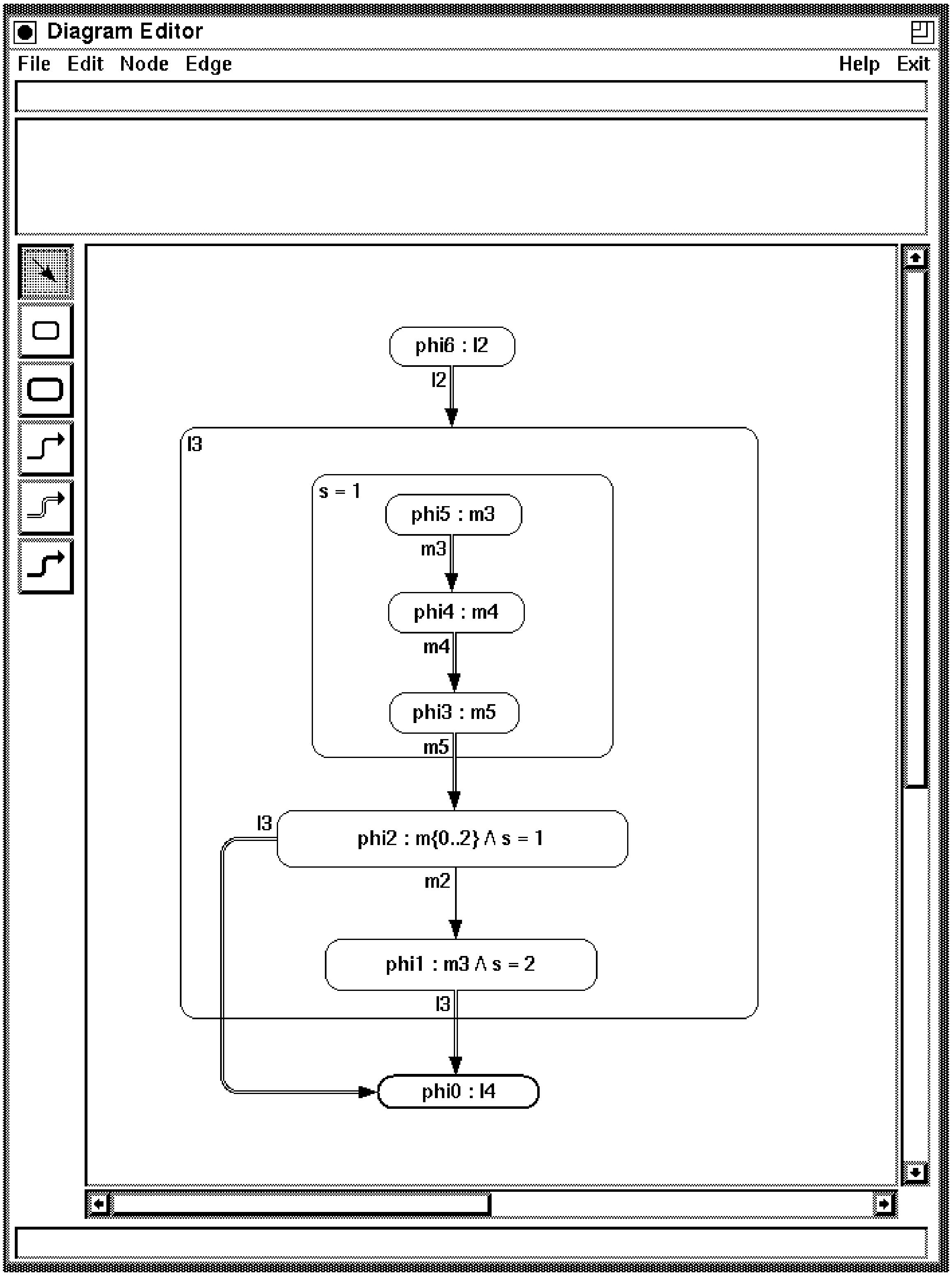


Figure 5.1: The Verification Diagram Editor

5.6 Verification Diagram Editor: Interface

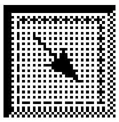
The Verification Diagram Editor allows you to load and manipulate verification diagrams. The interface is shown in Figure 5.1. The two top panels are used to edit the label and the assertions on nodes and edges. The diagram itself is drawn in the large panel, using the drawing tools displayed in the column to its left.

The verification diagram does not have to be related to the current system description, and you can use the Verification Diagram Editor independently of the STeP verification session in progress. By selecting the Current-to-Verify option under the File menu, you make the current diagram the one used by the verification session. In this way, a verification diagram can be applied as a verification rule to generate verification conditions that establish a given property.

The Verification Diagram Editor does not parse the expressions labeling the nodes, or check the existence of the transitions labeling the edges. These checks are performed when the verification diagram rule is invoked. If an assertion does not parse, or there is any other problem with the diagram, a message indicating the error will appear in the message window of the Top-level Prover. The verification diagram can then be corrected within the Verification Diagram Editor, made current again, and the verification diagram rule applied again. [1]

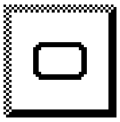
Drawing Tools

The leftmost column contains the tools for drawing basic and compound nodes, and edges corresponding to the different justice requirements.



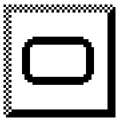
This tool sets the pointer to selection mode. It enables you to select nodes and edges by clicking on them. When an edge is selected, all the nodes to which it is attached are selected too. To select a single edge (e.g., to convert it into a different type of edge) you first have to move the edge away from the nodes. You can do this by pressing the left mouse button while pointing at the edge and then moving it quickly while keeping the button pressed. To select multiple edges and nodes, click on some location outside these nodes, and while keeping the left mouse button pressed, drag the mouse until the nodes and edges are enclosed in the region indicated by the dashed rectangle.

[1]An inquisitive user may wonder why the Verification Diagram Editor does not check for parsing errors. The reason is that to check parsing errors, one requires knowing the system for which a diagram is being applied to. However, STeP allows the Verification Diagram Editor to run independent to any system. This feature is useful when the same (or similar) diagrams uare being applied to different systems.

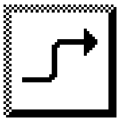


This tool is used to draw *non-terminal nodes*. After selecting it, move the mouse to the location where you want the upper left corner of the new node and click on the left button; while keeping it pressed, move the mouse down and to the right until the displayed shape has the proper dimensions. Releasing the button will then create the node. The node will be initially selected, and the two editable fields above the diagram will contain the default *name* and *label* for this node. You can edit the name field and enter a state assertion for this node.

To resize an existing node, select it with the left button (after first selecting the pointer tool), move the mouse to one of its corners, and resize the node while keeping the button pressed on the black square in the chosen corner.

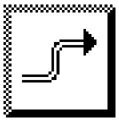


Works exactly as the tool for non-terminal nodes, but the generated node is a terminal node.

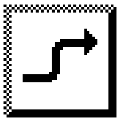


This tool draws an edge without an associated fairness requirement. An edge is drawn using the left mouse button: press the button at the location where you want the edge to start, keep the button pressed while moving the pointer and release the button at the desired endpoint. To draw an edge with corners, release the button momentarily at the location of the turn, press again, and move the mouse in the new direction. Several consecutive turns can be made in this way.

The label of the edge will appear in the top panel and can be edited there. The label can correspond to any transition of the system being analyzed.



Works exactly as the tool for drawing a single-lined edge, but draws a double-lined edge that has an associated `Justice` requirement. The label of such a Just edge must be the name of a transition declared as Just or Compassionate.



Works exactly as the tool for drawing a single-lined edge, but draws a solid-lined edge that has an associated `Compassion` requirement. The label of a Compassionate edge must be the name of a transition declared as Compassionate.

Menu Options

The File Menu

The **File** menu is used to load and save verification diagrams, as well as to make verification diagrams accessible to the Top-level Prover.

New Resets the verification diagram editor, deleting the currently edited diagram.

Load Loads a verification diagram from a file.

Save Saves the verification diagram currently being edited to a file.

Current-to-verify Selects the currently edited verification diagram as the one used by the Top-Level Prover.

Exit Quits the Verification Diagram Editor after confirmation by the user.

The Edit Menu

The **Edit** menu is used to cut and paste parts of the diagram.

Cut Deletes one or more selected items from the editing window. The deleted item(s) (node or edge) are put on a scratchpad and can later be recovered by Paste.

Copy Copies the selected item from the editing window to the scratchpad.

Paste Takes the item most recently put on the scratchpad (either by Cut or Copy) and puts it in the editing window in the same location it was before.

Duplicate Duplicates one or more selected items. This is equivalent to doing Copy followed by Paste.

The Node Menu

The Node menu is used to group nodes into compound nodes, change terminal into nonterminal nodes, and vice-versa.

Group/Compound By selecting a large node and the nodes enclosed by the large node, Group turns the large node into a compound node, containing the enclosed and selected sub-nodes (see Figure 5.1).

Subnodes of a compound node cannot be edited. They have to be released with Ungroup before you can change their labels or move them.

The label and assertion of the compound (enclosing) node is displayed in the upper left corner of the node.

Ungroup/Uncompound Converts a selected compound node into a regular node, and releases the subnodes such that they can be edited and moved.

Terminal Converts the selected node (only one node should be selected) into a terminal node. The borders of the node are drawn with thick lines and the node is declared as terminal.

Nonterminal Converts a selected terminal node into a nonterminal node.

The Edge Menu

The **Edge** menu lets you change the fairness requirements associated with a selected edge, and check that all edges are properly connected to the nodes.

Cut Head Cuts the head hop of a selected edge. The head hop is the hop with an arrow where an edge ends.

Cut Tail Cuts the tail hop of a selected edge. The tail hop is the hop where an edge starts.

Merge Merges a selected edge with the edges chained to it. Two edges are considered chained if one's ending point is close to the other's starting point.

Unjust Converts a selected edge into an Unjust edge. Unjust edges are drawn with thin black lines.

Just Converts a selected edge into a Just edge. Just edges are drawn with double lines.

Compassionate Converts a selected edge into a Compassionate edge. Compassionate edges are drawn with thick black lines.

Connectivity Highlights those edges whose ends are not connected to nodes. This is a convenient way to check whether all edges are connected properly in the verification diagram.

Global Hookup Checks and hooks up all loose edges. An edge will be hooked up to a node if its end is close enough to or inside the node.

Help

Activation of the on-line help pages, using Mosaic or your favorite `html` (WWW) browser. The Top-level Prover help pages are accessible from the Verification Diagram Editor help pages and vice-versa, so a single help session should generally suffice. The executable used for on-line help can be changed by setting the `STEP_BROWSER` environment variable.

5.7 Hierarchical Verification Diagrams ⋆

In order to allow decomposition of verification diagrams into smaller, more manageable parts, STeP supports hierarchical verification diagrams. With hierarchical diagrams, several nodes in a large diagram can be represented by a single node, making large diagrams more readable and easier to edit.

Import Nodes

A hierarchical verification diagram has one or more import nodes. An *import node* imports a verification diagram from a file. The imported diagram may itself have import nodes that import other files. The edges going out from the import node are matched up with outgoing edges from the imported diagram, and similarly for incoming edges.

An import node is created by giving it the label `#import` and giving it as assertion the filename (with full or relative path name) within angle brackets (`<>`), for example `<mux-pet2/mux-h1.diagram>`, of the file that contains the diagram to be imported. The assertion field may also contain parameters (see next paragraph) enclosed within "%". Any text in the assertion field of an import node that is not enclosed by angle brackets or percent signs is ignored.

Parameter Passing

Hierarchical verification diagrams allow macro-like text replacement when importing a verification diagram. Parameters (in the node to be imported) are of the form "@1", "@2", "@3", etc., and "@1" is replaced by the first argument passed by the importing node, "@2" is replaced by the second argument, etc.

When the top-level diagram is processed (i.e., when it is used as a verification rule by the Top-level Prover), STeP checks whether the number of parameters each imported diagram requires is equal to the number of arguments passed. The arguments passed by the import node are included in the assertion field; each argument is enclosed within percent signs ("%"). An example of the assertion field of an import node that passes parameters is

```
<mux-pet2/mux-h1.diagram> %m3% %l3% %s=1%
```

The arguments do not have to be on the same line, and white space is ignored.

Edges

The connection of edges between the import node and the imported diagram is done by matching unconnected outgoing edges in the imported diagram (also called dangling edges) with outgoing edges of the import nodes that have the same label, and similarly for incoming edges. If multiple edges of the imported diagram match multiple edges of the import node, then edges are created for all possible combinations (taking the product over all edges). Thus, one incoming edge in an imported diagram may translate into multiple edges with the same label entering that same node in the imported diagram.

If multiple edges with the same label entering or leaving an import node need to be distinguished (e.g., because they enter different nodes in the imported diagram), the labels may be provided with a secondary label. The secondary label follows the regular transition name, separated by a colon. Thus, if `l1` is the regular transition name, then `l1:1` labels an edge for transition `l1`, with secondary label `1`. Edges connected to the import node and dangling edges in the imported diagram only match when both the primary and the secondary label are identical. White space is ignored in secondary labels. Secondary labels default to `0` (zero) if left blank, but this value is not displayed in the diagram.

Edges can also be connected to import nodes as if the import node was a compound node (i.e., an incoming edge in the import node is connected to all nodes of the imported diagram, and similarly for an outgoing edge), by adding the secondary label **All** to the label of the edge entering the import node. Thus, an edge labeled with **All** in the importing diagram is connected to all nodes of the imported diagram without any name matching. It is not meaningful to add the secondary label **All** to dangling edges in an imported diagram, and this is therefore ignored.

If import nodes are within a compound node, then edges entering or leaving the compound node are interpreted, as usual, to represent edges entering or leaving each enclosed node. Therefore these edges are considered to be connected to the import nodes, and interpreted as usual.

Chapter 6

Automatic Theorem-Proving

A computer-aided verification system should try to make the task of proving verification conditions as easy as possible while still ensuring soundness.

To this effect, STeP provides powerful, usually fast, automatic simplification procedures. These are supplemented by the interactive Gentzen-style theorem prover, discussed in Chapter 8, used to establish properties that the automatic simplifier cannot.

6.1 The Automatic Simplifier

There is a basic trade-off between the strength and the speed of the automatic Simplifier. The more powerful the settings are, the slower the Simplifier is. The default settings are a good compromise: they are able to simplify most commonly occurring verification conditions in a reasonable time.

The impatient user may want to use the interrupt button to halt lengthy simplification operations and try alternative proof techniques.

Selecting the Simplification Flags menu option brings up the window shown in Figure 6.1. It contains the following settings:

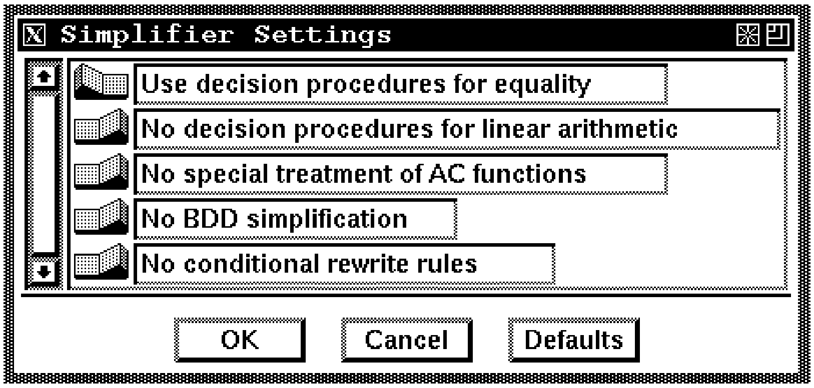


Figure 6.1: Simplifier settings

- **Equality and Inequality**
  If ON, the Simplifier will use properties of equalities and inequalities. For example it will simplify
  $$y > 1 \;\rightarrow\; y > 0$$
  to *true* when this switch is ON, but not when it is OFF.

  These decision procedures are quite efficient and useful in general, so they are ON by default.

- **Linear Arithmetic**
  If ON, the Simplifier will use decision procedures for linear arithmetic, which use properties of $>$, $+$, and $=$. For example, it will simplify
  $$y >= 0 \;\wedge\; y' = y + 1 \;\rightarrow\; y' > 0$$
  to *true* when the switch is ON, but will not be able to simplify it when the switch is OFF. The default is OFF.

  The performance of these procedures can vary widely from one formula to the next; the *interrupt button* can be used to halt the computation.

- **Associative and Commutative functions**
  If ON, STeP will take into account the associativity and commutativity of addition, multiplication, and boolean connectives, as well as those user-defined binary function symbols that have been declared as AC, ASSOCIATIVE or COMMUTATIVE (see Section 2.3).

  Note that since STeP rewrites all formulas into a normal form, it implicitly uses many instances of the associative and commutative laws even when this flag is OFF. The default is OFF.

- **BDD simplification**
  If ON, the (propositionally complete) BDD simplification method will be used as part of the simplifier, before other decision procedures are used (see Section 6.2). The default is OFF.

- **Conditional Rewrite Rules**
  If ON, the Simplifier will use a number of built-in conditional rewrite rules. For example, it will simplify
  $$(y > 0 \wedge x \cdot y > 0) \rightarrow x > 0$$
  to *true* when the switch is ON, but will not be able to simplify it when the switch is OFF.

  Conditional rewrite rules may slow down the Simplifier considerably, so they are switched OFF by default.

Commonly used Simplifier settings are also available as separate buttons in the action region of the Top-level Prover:

Fast-simplify Applies only the most basic simplification procedures, disregarding the background properties. This is useful to rewrite formulas into a more standard form, after which they can be compared more easily with each other.

Bool-simplify Applies a propositional satisfiability check, using the active background properties. If the given formula is a propositional tautology with respect to the active background properties, it will be simplified to *true*. This procedure is complete for propositional logic. The basic simplification procedures (see **Fast-simplify** above) are called before the propositional test.

This procedure uses STeP's BDD package (see Section 6.2).

Simplify Invokes the simplifier and decision procedures as given by the current simplification settings.

For material related to the decision procedures and indexing used in STeP, see [Bledsoe, 1975], [Shostak, 1979], [Nelson and Oppen, 1980], [Boyer and Moore, 1988], [Stickel, 1989], [McCune, 1992], [Zhang, 1993].

6.2 BDD Simplification

STeP includes a package for the creation and manipulation of propositional (i.e., boolean) formulas using Binary Decision Diagrams (BDDs). This package is used in the BDD-split proof rule as well as in boolean simplification steps.

Since the amount of memory used by the BDD package can grow indefinitely (in the form of accumulated *BDD nodes*), the package can be reset at any time with the Reset BDDs option under the Settings menu. This only incurs a (usually small) short-term efficiency loss, and does not affect the soundness of the prover.

This option should also be used if the number of BDD nodes reaches the maximum given by the `STEP_BDD_NODES` environment variable (see Appendix E). The number of existing BDD nodes can be obtained through the Sytem Information option.

A number of environment variables control the amount of memory that BDDs use:

- `STEP_BDD_NODES` The maximum number of BDD nodes kept around at one time.
- `STEP_BDD_CACHE` The size of the BDD *result cache*.
- `STEP_BDD_SPLIT` The maximum number of subgoals that can be generated by a BDD split operation in both the Top-level and Interactive Provers (see Section 6.2.1 below).

For more information on BDDs, see [Bryant, 1986, Bryant, 1992].

6.2.1 BDD split ⋆

The BDD-split proof rule uses BDDs to convert the current goal into a list of subgoals, each a disjunction of atomic formulas. The subgoals correspond to the negation of the propositional models that can falsify the formula. If all the subgoals are valid, the original goal is valid as well.

In the worst case, exponentially many children are produced using this expansion, but in most applications there is a good chance that it will work well. The acceptable maximum number of subgoals is given by the `STEP_BDD_SPLIT` environment variable.

In the Top-level Prover, the active background formulas are automatically used to reduce the number of subgoals generated; in the Interactive Prover, the background formulas must be added to the sequent before invoking BDD-split. This is analogous to the behavior of the Simplifier in the two provers.

6.3 Tactics ⋆

Tactics are a method to automate parts of the high-level proof search by encoding long or repetitive sequences of proof commands. An individual proof command is an *atomic tactic*.

The Top-level Prover and the Interactive Prover share the same tactic language, parameterized by their atomic tactics (since each has a different set of proof commands). The general syntax of a tactic is as follows:

<table>
<tr><td>tactic</td><td>::=</td><td>predefined</td><td>Predefined tactic</td></tr>
<tr><td></td><td>|</td><td>composite</td><td>Composite tactic</td></tr>
<tr><td></td><td>|</td><td>atomic</td><td>Atomic tactic</td></tr>
<tr><td>predefined</td><td>::=</td><td><code>skip</code></td><td>Identity tactic</td></tr>
<tr><td></td><td>|</td><td><code>fail</code></td><td>Failure tactic</td></tr>
<tr><td>composite</td><td>::=</td><td><code>try</code> (tactic , tactic , tactic)</td><td></td></tr>
<tr><td></td><td>|</td><td><code>then</code> [form] tactics</td><td></td></tr>
<tr><td></td><td>|</td><td><code>sequence</code> tactics</td><td></td></tr>
<tr><td></td><td>|</td><td><code>else</code> tactics</td><td></td></tr>
<tr><td></td><td>|</td><td><code>repeat</code> [form] tactic</td><td></td></tr>
<tr><td></td><td>|</td><td><code>spread</code> tactic tactics</td><td></td></tr>
<tr><td></td><td>|</td><td><code>branch</code> tactic tactics [<code>otherwise</code> tactic]</td><td></td></tr>
<tr><td></td><td>|</td><td>tactic ; tactic</td><td></td></tr>
<tr><td></td><td>|</td><td>(tactic)</td><td></td></tr>
<tr><td>tactics</td><td>::=</td><td><code>[</code> [tactic { , tactic }] <code>]</code></td><td></td></tr>
<tr><td>form</td><td>::=</td><td><code>depth first</code></td><td></td></tr>
<tr><td></td><td>|</td><td><code>breadth first</code></td><td></td></tr>
<tr><td>atomic</td><td>::=</td><td><code>{</code> atomic-tactic <code>}</code></td><td></td></tr>
</table>

The grammar for *atomic-tactic* for the Top-level Prover is given in Section 6.3.2; the grammar for the Interactive Prover atomic tactics is given in Section 6.3.3.

The curly brackets can be omitted if the atomic tactic can be written without spaces. In other cases, curly brackets have to enclose each atomic tactic.

6.3.1 Composition Options

When a verification rule is applied to a goal to which it does not syntactically make sense (for example $\boxed{\text{B-INV}}$ applied to a response formula) it *fails*, otherwise it *succeeds*. Tactics, which are composed from verification rules can also fail and succeed as described below.

try(*tac1*,*tac2*,*tac3*)

Try tactic *tac1*, if it succeeds, apply *tac2* to each subgoal generated. If *tac1* fails, apply tactic *tac3* to the original goal. The entire tactic fails if *tac2* fails on one of the generated subgoals of *tac1*, or *tac3* fails.

Figure 6.2 gives an idea of how the proof-search looks like after *tac1* and *tac2* have been applied successfully, or *tac1* failed and *tac3* has been applied. The lower-most dot represents the original goal, the first left-most layer of dots represent the subgoals generated from *tac1*, the second left-most layer, the subgoals generated from *tac2*. To the right is a schematic representation of the subgoals generated by *tac3*.

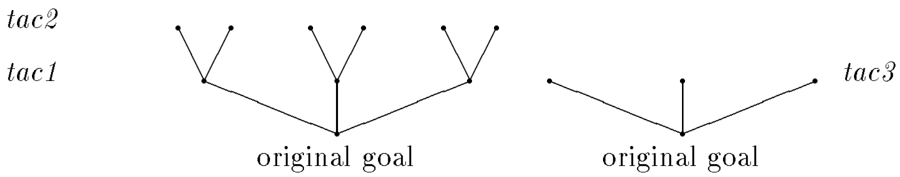


Figure 6.2: `try`(*tac1,tac2,tac3*)

then *[form] tactics*

`then []` has the same effect as `skip` - the original subgoal is reduced to itself. For non-empty lists we define inductively:

`then breadth first [` tac_1,tac_2,…,tac_n`]` applies tactic tac_1 first. If it succeeds, it applies the tactic `then breadth first [` tac_2,…,tac_n`]` to *each* of the generated subgoals. If it fails it applies the tactic `then breadth first [` tac_2,…,tac_n`]` to the original subgoal.

Figure 6.3 illustrates the levels obtained from successful completion of `then breadth first` [*tac1,tac2,tac3*]. So for instance, the nodes above the original goal in the bottom represent the result of applying *tac1*.

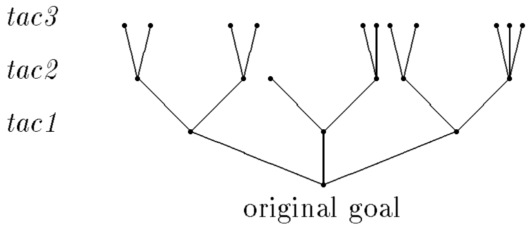


Figure 6.3: `then breadth first` [*tac1,tac2,tac3*]

`then depth first [` tac_1,tac_2,...,tac_n`]` applies the tactic tac_1 first. If it succeeds, it applies `then depth first [` tac_2,...,tac_n`]` to the first subgoal generated. If it fails, it applies `then depth first [` tac_2,...,tac_n`]` to the original goal. If any left-most subgoal is closed by a tactic tac_i, where $i < n$, then $tac_{i+1} \ldots tac_n$ are ignored.

Figure 6.4 illustrates the effect of successful completion of `then depth first` [*tac1,tac2,tac3*].

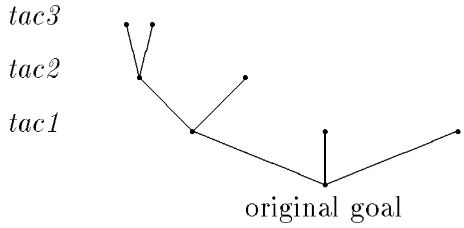


Figure 6.4: `then depth first` [*tac1,tac2,tac3*]

If no *form* is given the default is `breadth first`.

sequence *tactics*

Applies the tactics in left-to-right order breadth first until the first tactic fails. The constructor `sequence` can be defined in terms of `try`, namely `sequence` [tac_1, tac_2, tac_3] has the same effect as `try`(tac_1, `try`(tac_2, tac_3,`fail`), `fail`).

tactic ; *tactic*

tac_1 ; tac_2 is shorthand for `then [` tac_1, tac_2 `]`. The semicolon ; associates to the left and has the highest precedence.

else *tactics*

Applies the first tactic that succeeds from the list of tactics supplied.

Figure 6.5 illustrates the result of `else` [*tac1,tac2,...,*tac_i*,...,*tac_n], where tac_i is the first tactic that succeeds.

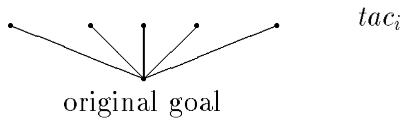


Figure 6.5: `else` [*tac1,tac2,...,*tac_i*,...,*tac_n]

repeat *[form] tactic*

`repeat breadth first` *tac* applies *tac* repeatedly until failure appears on one of the generated subgoals in a layer.

`repeat depth first` *tac* applies *tac* repeatedly on the left-most subgoal generated.

spread *tactic tactics*

`spread` *tac* `[` $tac_1, tac_2, \ldots, tac_n$ `]` applies first *tac*, and then applies tac_i to the i'th subgoal generated. The tactic fails if the number of subgoals generated from *tac* is different from n. Also if any of the tactics *tac* or tac_i fail, the entire tactic fails.

branch *tactic tactics [* **otherwise** *tactic]*

`branch` *tac* $[tac_1, \ldots, tac_n]$ works like `spread`, except that if the number of subgoals is $m \geq n$, then it applies tactic tac_n to the subgoals indexed $n, n+1, \ldots, m$.

`branch` *tac* $[tac_1, \ldots, tac_n]$ `otherwise` *tac'* first applies the tactic `branch` *tac* $[tac_1, \ldots, tac_n]$. If this fails, it applies *tac'* to the subgoal.

6.3.2 Top-level Prover Tactics

atomic-tactic	::=	*basic-rule*	Basic rule not requiring arguments
	|	*arg-rule expns*	Rule requiring argument expressions
basic-rule	::=	`B-INV`	
	|	`B-WAIT`	
	|	`B-BACKTO`	
	|	`B-CAUS`	
	|	`Simplify`	
	|	`Interactive`	
	|	`Undo`	
	|	`Redo`	
	|	`Postpone`	
	|	`BDD`	
arg-rule	::=	`G-INV`	
	|	`G-WAIT`	
	|	`G-BACKTO`	
	|	`G-CAUS`	
	|	`Modelcheck`	
expns	::=	{ `‘‘`*expn*`’’` }+	

6.3.3 Interactive Prover Tactics

To appreciate this section it is helpful to have read Chapter 8.

atomic-tactic	::=	*basic-rule*	Basic rule not requiring arguments
	|	*e-rule expns*	Rule taking expression argument
	|	*p-rule positions*	Rule taking position arguments
	|	*pe-rule expn-poss*	Rule taking position and expression arguments
	|	*p-rule*	Position rule without position arguments
	|	*replace*	Replacement of equals for equals
	|	*skolemize*	Skolemization

basic-rule	::=	`Undo`	Basic rules behave the
	$\mid$	`Redo`	same as their
	$\mid$	`Postpone`	pushbutton counterparts
	$\mid$	`Flatten`	
	$\mid$	`Simplify`	
	$\mid$	`Make-first-order`	
	$\mid$	`Next`	
	$\mid$	`Presburger`	
	$\mid$	`All-Propositional`	
	$\mid$	`BDD-split`	
	$\mid$	`PTL-expansion`	
e-rule	::=	`Cut`	adds argument expressions as cuts
	$\mid$	`Add-Axiom`	adds argument expressions as axioms
	$\mid$	`Rewrite`	arguments is a tuple of pairs
			each pair represents a rewrite rule
	$\mid$	`Free-Induction`	arguments are the integer variables
			in the induction
p-rule	::=	`Duplicate`	duplicates formulas at
			indicated positions
	$\mid$	`Delete`	duplicates formulas at
			indicated positions
	$\mid$	`Hide`	hides formulas at
			indicated positions
	$\mid$	`Unhide`	unhides formulas located at
			their hidden positions.
	$\mid$	`1-Step-Propositional`	
	$\mid$	`Induction`	
pe-rule	::=	`Instantiate`	Instantiate existential force
			quantifiers at indicated positions
			by the listed expressions
replace	::=	`Replace at` *i-val* `left by right`	
	$\mid$	`Replace at` *i-val* `right by left`	
			In both cases *i-val* must be < 0.
skolemize	::=	`Skolemize`	
	$\mid$	`Skolemize` `` ` ` `` *expn* `''` `at` *position*	
	$\mid$	`Skolemize` *position*	
expns	::=	{ `` ` ` `` *expn* `''` }+	
positions	::=	{ *position* }+	
position	::=	*i-val*	
	$\mid$	`[` *i-val* { `,` *i-val* } `]`	

expn-poss ::= { `` *expn* '' `at` *position* }+

If a *p-rule* is given without arguments, it is applied exhaustively in a breadth-first manner.

6.3.4 Interrupting Tactics

Tactics are not guaranteed to terminate, or can otherwise run for a very long time; the *interrupt button* (the STeP logo) is useful to stop the execution of a batchmode tactic.

Two things can actually happen if the interrupt button is pressed during the execution of a tactic: if the interrupt button is activated during a simplification step, the simplification will be canceled and the system will continue with the next one; if the system is interrupted when no simplification is going on, the entire tactic will be interrupted.

6.3.5 Commonly used Tactics

B-INV; Simplify

Applies the rule B-INV followed by simplification of each subgoal. This differs from the effect of setting Automatic Simplification ON by not backtracking to the unsimplified verification condition when simplification fails.

B-INV; Simplify; Undo

If simplification does not establish the given subgoal of B-INV; the current leaf in the proof search tree is undone, getting back to the unsimplified subgoal.

repeat(B-INV; Simplify; Undo; WPC)

Repeats application of B-INV to each unsimplified subgoal that has been strengthened by the WPC rule.

{ G-INV "`(l5 /\ m5) /\ (l4 /\ m5 --> s = 2 /\  y1)`" }; Simplify

Applies rule G-INV with a strengthening formula, and each subgoal is then simplified. G-INV only requires one strengthening formula; rules such as G-WAIT require more, so a list of expressions must be entered following the keyword "G-WAIT".

Skolemize; Simplify; BDD-split; Simplify

Called from the Interactive Prover, this tactic attempts to establish a subgoal by first skolemizing quantifiers of universal force, and then simplifying in the hope that simplification will reduce the boolean complexity of the verification condition. BDD-split is then invoked to split the goal into subgoals of atomic formulas; the simplifier is finally invoked to establish each of these subgoals.

1-Step-Propositional; Simplify

The Interactive Prover $\boxed{\textsf{1-Step-Propositional}}$ rule is used without a position. It is therefore used exhaustively until it cannot be applied. Then each generated subgoal is simplified by $\boxed{\textsf{Simplify}}$.

Chapter 7

Automatic Generation of Invariants

STeP includes three different methods that automatically generate *bottom-up invariants*, based solely on the system to be verified, of increasing power and complexity. Since these invariants are, by construction, guaranteed to be true of the system, they are automatically added to the list of active background properties.

Of the three methods, the first, *local invariants*, operates on an SPL program; the other two operate on the transition system directly. A description of the general principles behind STeP's invariant generation algorithms is [Bjørner *et al.*, 1995].

To obtain invariants, select the appropriate option under the Properties menu; when the invariant generation is done, each invariant is simplified, displayed in the output window, and added to the set of background properties. By default they are active, and thus used in the simplification of goals. They may be deactivated with the Activate/Deactivate menu option.

Each invariant is simplified on its own (i.e., no background properties are used), using the basic decision procedures for equality (see Section 6.1).

When the invariant generator is finished a message will appear in the message window. Invariant generation and simplification can be halted by clicking on the interrupt button (the STeP logo).

7.1 Local Invariants

Local invariant generation produces very simple but useful invariants, based on local analysis of the program text. The generated invariants can be classified in the following three groups:

1. *Finite-domain invariants.* Invariants of the form

$$l_i \Rightarrow x = c_1 \vee x = c_2 \vee \ldots \vee x = c_n$$

where l_i is a program location, $c_1, .., c_n$ are constants, and x is a program variable.

2. *Conditional invariants.* Invariants of the form $l_i \Rightarrow cond$, where l_i is a program location and *cond* is a formula arising from the initial condition or a branch statement,

e.g., the program fragment `if cond then l1: S1 else l2: S2` generates the conditional invariants `l1` $\Rightarrow$ `cond`, and `l2` $\Rightarrow$ $\neg$`cond`, if no variables in `cond` are modified by parallel statements.

3. *Range invariants.* Invariants of the form $l_i \Rightarrow lb \leq x \wedge x \leq ub$, where lb and ub are integer or rational constants (or infinity).

7.2 Linear Invariants

Linear invariant generation arises from analyzing the linear relationships between system variables in the course of a program.

The invariants generated are usually linear equalities of the form

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n = b$$

where $a_1, \ldots, a_n, b$ are constants. Each x_i can be (a) a variable, (b) the sum `Sum i:[m..M]. A[i]`, of the values of an array, or (c) a sum `Sum i:[m..M]. x[i]`, where `x` is a local variable of a parameterized process `P[i:[m..M]]`.

7.3 Polyhedral Invariants $\star$

The polyhedral invariant generator is the most powerful of the methods provided (currently in "experimental" state). The invariants generated depend on the set of variables under consideration, which can be controlled by the user. The more variables that are included, the more powerful the invariants that can be generated, but the time spent generating invariants will also increase (possibly in an exponential manner).

When the polyhedral invariant generation is selected, a dialog window appears where the user can enter the following:

1. *Variables to be ignored:* you may enter a list of program and auxiliary variables, separated by spaces or commas. These variables will be *ignored* by the invariant generator, that is, the invariant generation will not try to deduce polyhedral relationships which include them.

2. *Location variables to be considered:* By default, location variables are not considered by the polyhedral invariant generator. Here you may enter a list of location variables to be added to the vocabulary of the invariant generator.

The generator uses an external package, whose time and space limitations can be controlled by setting the following environment variables:

`STEP_POLY_SPACE`	(Megabytes)
`STEP_POLY_TIME`	(minutes)
`STEP_POLY_CPU`	(minutes)

The invariant generation can be interrupted by clicking on the interrupt button (the STeP logo); this may leave a separate UNIX process running, which users may have to terminate independently from a UNIX prompt (see the UNIX `kill` command man pages).

Chapter 8

The Interactive Prover $\star$

The Interactive Prover is based on a *Gentzen-style proof system*, specialized to linear-time temporal logic. For more background, see, e.g., [Gallier, 1987]. This chapter explains the basic functionality of the various rules provided in the Interactive Prover and gives examples of their use.

8.1 Proof Structure

A goal or subgoal in the Interactive Prover is a *sequent*, of the form

$$\Gamma \Longrightarrow \Delta$$

where Γ and Δ are *multisets* of formulas (i.e., formulas can be repeated). A sequent stands for the formula

$$\bigwedge \gamma_i \rightarrow \bigvee \delta_j$$

where γ_i are the elements of Γ and δ_j are the elements of Δ. Free variables are implicitly universally quantified.

The Interactive Prover creates a *proof tree* similar to that in the Top-level Prover. An inference rule applies to zero or more of the formulas in a sequent $\Gamma \Longrightarrow \Delta$, and produces a number of subgoals, the branches of the tree, again in the form of sequents. The proof is finished when all subgoals are closed.

A subgoal is closed when one of the following conditions is satisfied:

- The sequent contains `true` on the right-hand side of the long doubly lined implication.

$$\Gamma \Longrightarrow \Delta, \texttt{true}$$

- The sequent contains `false` on the left-hand side of the long doubly lined implication:

$$\Gamma, \texttt{false} \Longrightarrow \Delta$$

- The sequent contains a formula `p` in both the left and right sides of the long doubly lined implication:

$$\texttt{p}, \Gamma \Longrightarrow \Delta, \texttt{p}$$

- The sequent contains a formula `(-)p` on the left side of the long doubly lined implication, and no applications of $\boxed{\textsf{Next}}$ (see Section 8.3) separate the sequent from the root of the proof tree:

$$(-)\mathtt{p}, \Gamma \Longrightarrow \Delta$$

  This axiom reflects the well-foundedness of time. In the first time instance, the truth-value of `(-)p` is false for any formula `p`, because there is no past at this instance.

- The sequent $\Gamma \Longrightarrow \Delta$ is *duplicated* along a branch in the proof tree, i.e., the following conditions are satisfied:

  1. There is a sequent $\Gamma' \Longrightarrow \Delta'$ between the root and $\Gamma \Longrightarrow \Delta$, such that $\Gamma' \subseteq \Gamma$ and $\Delta' \subseteq \Delta$.
  2. The two sequents are separated by at least one application of the **Next**-rule.
  3. There is an *unfulfilled* formula, in the sense of the temporal tableau construction [Manna and Pnueli, 1995], for the sequents below applications of $\boxed{\textsf{Next}}$ and above $\Gamma' \Longrightarrow \Delta'$.

  It helps to think of duplication in the presence of unfulfilled formulas as corresponding to an induction step. The requirement also replaces inference rules formalizing induction on temporal operators.

Fortunately all these conditions are kept track of automatically and you do not have to worry about the detailed conditions, so that you can focus on the outline of the proof.

8.2 User Interface

Figure 8.1 shows the interface of the Interactive Prover. It has a menu bar on top and action buttons to the right. Most interaction in the Interactive Prover is accomplished by button clicks. If a rule is applicable to multiple formulas in the sequent, the candidates are highlighted (shown in reverse video) and you can select the one to which you intend to apply the rule. Some rules require additional information to be entered through dialog windows.

Menu Options

Refresh redraws the Interactive Prover windows.

Tactics is equivalent to the Tactics menu in the Top-level Prover (Section 3.1) except now it invokes tactics composed of Interactive Prover primitive commands (see Section 6.3).

Settings

Simplification Flags This menu option is equivalent to the Simplification Flags option in the Top-level Prover. It brings up a window used to set the power of the automatic Simplifier; see Section 6.1.

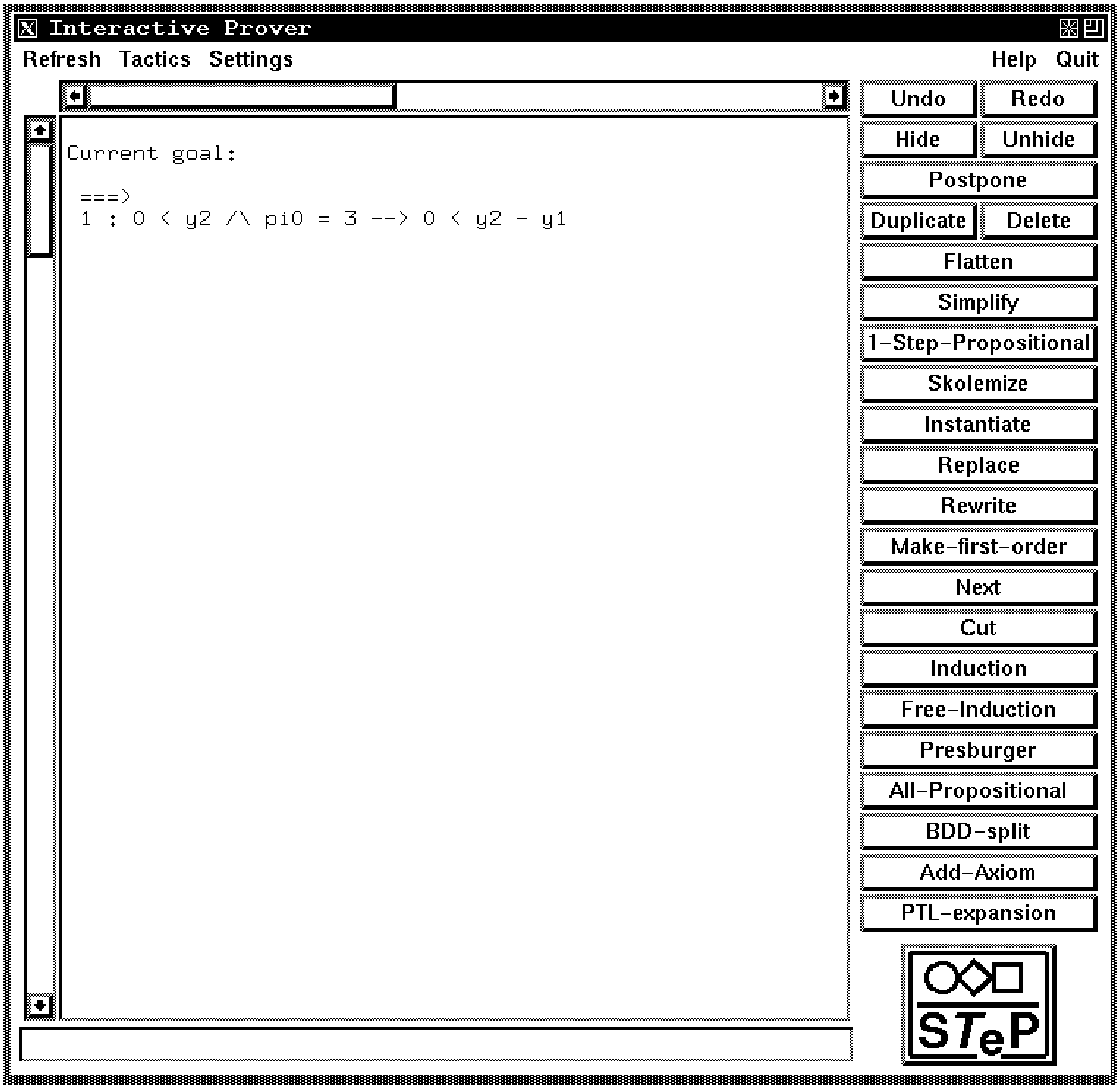


Figure 8.1: Interactive Prover Window

Select boxes This menu option allows the user to control whether boxes, i.e., the temporal operator $\Box$ known as "henceforth", in front of axioms added to the sequent during the proof should be kept or not. By default boxes are removed from axioms, when axioms are added to the sequent. See also the explanation of Add-Axiom below for an example of the usage of **Select boxes**.

Quit brings up the window shown in Figure 8.2.

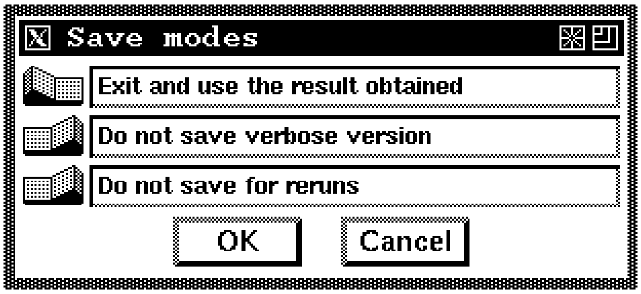


Figure 8.2: Exit Window for Interactive Prover

If you choose to use the results obtained, any unfinished goals are returned to the Top-level Prover as a new goal with the name `Interactive`. If there are no goals left, the current goal in the Top-level Prover is closed. If you choose to ignore the effect of the Interactive Prover the proof tree in the Top-level Prover stays as it was before invoking the Interactive Prover.

The button saving *verbose* versions allows you to save the entire proof in the format seen during verification. Each sequent is printed in its full length, and annotated with its position in the proof-search.

The button saving for reruns, saves a dense proof-script, which can be loaded as a tactic for later automatic recreation of the proof.

8.3 Interactive Prover Rules

The right column of the Interactive Prover contains buttons to manipulate the proof tree and invoke the Gentzen proof rules. The proof search proceeds as usual, with logical rules that reduce the current goal to a number of subgoals, and simplification steps that close or simplify the current goal.

A number of inference rules require the user to select subformulas in the sequent. The subformulas, that may be selected are highlighted when selecting the inference rule, and the user clicks on the relevant highlighted formula to invoke the rule on that formula.

Undo Makes the ancestor of the current subgoal into the current goal. The former current goal is retained to enable a Redo, unless another proof rule is applied to the new current goal.

Redo Redoes a previous Undo.

Hide Temporarily hides a formula from the current sequent. Hidden formulas do not participate in simplification steps, and cannot be acted upon by inference rules. Hidden formulas, however, can be recovered with the **Unhide** rule, below. Hiding formulas can make the sequent more readable and may speed up simplification.

After clicking this button all formulas are highlighted, and you can select the formula to hide. To hide several formulas you must use **Hide** multiple times.

Unhide Recovers a formula hidden with **Hide**.

Postpone The next subgoal in the subgoal queue becomes the current subgoal.

Duplicate Duplicates a formula in the sequent. This is useful if you need to instantiate a quantified variable more than once, using different values. Note that duplicated formulas can be hidden with **Hide**, above, until they are needed.

Delete This button implements the *weakening rule*. It allows you to delete formulas that are not needed to establish the goal. Deletion of unnecessary or redundant formulas makes the current goal easier to read and speeds up the simplification procedures.

Flatten Creates a new sequent in which all conjunctions in the antecedent and all disjunctions in the consequent are split into separate formulas.

Example: For formulas `a, b, c, d, e, f, g`, **Flatten** has the following effect:

$$\frac{\texttt{a, b, b \textbackslash/ c} \quad \Longrightarrow \quad \texttt{d /\textbackslash{} e, f, g}}{\texttt{a /\textbackslash{} b, b \textbackslash/ c} \quad \Longrightarrow \quad \texttt{d /\textbackslash{} e, f \textbackslash/ g}}$$

Simplify Invokes the Simplifier on the current goal. The strength of the Simplifier is controlled via the Simplification Flags option under the Settings menu.

1-Step-Propositional Application of propositional inference rules is controlled uniformly with this button.

After clicking this button, formulas which contain propositional connectives that can be analyzed are highlighted. Clicking on one of the highlighted formulas applies the appropriate propositional inference rule to the selected formula.

The boolean connectives analyzed are:

$$\wedge\,,\ \vee\,,\ \rightarrow,\ \leftrightarrow,\ \neg,\ \textbf{if-then-else},$$

which are entered as:

`/\ \/, -->, <-->, !, if-then-else`

The temporal connectives analyzed are:

$$\Box,\ \Diamond,\ \mathcal{U}\,,\ \mathcal{W}\,,\ \boxminus,\ \diamondminus,\ \mathcal{S}\,,\ \mathcal{B}$$

which are entered as:

`[], <>, Until, Awaits, [-], <->, Since, Backto`

Notice that $\bigcirc$ and $\ominus$ are not in this group.

Figure 8.3 shows all the rules that can be applied by $\boxed{\text{1-Step-Propositional}}$.

$\boxed{\text{Skolemize}}$ Quantifiers of universal force can be eliminated by skolemization, which replaces the bound variable by a new variable.

If there are multiple instances of universally quantified variables all of the corresponding formulas are highlighted upon clicking on this button, and you can select the one you want to skolemize by clicking on it.

$\boxed{\text{Skolemize}}$ implements the inference rule:

$$\frac{\Gamma,\ \texttt{f(b)} \quad\Longrightarrow\quad \Delta}{\Gamma,\ \texttt{Exists x . f(x)} \quad\Longrightarrow\quad \Delta}$$

where `b` is a fresh variable not occurring free in Γ, Δ or `Exists x . f(x)`.

$$\frac{\Gamma \quad\Longrightarrow\quad \Delta,\ \texttt{f(b)}}{\Gamma \quad\Longrightarrow\quad \Delta,\ \texttt{Forall x . f(x)}}$$

where again `b` is not free in either Γ, Δ or `Forall x . f(x)`.

Besides these cases, the Interactive Prover supports skolemization of quantifiers under any nesting of the boolean connectives `/\`, `\/`, `-->`, `~`, and the `then` and `else` branches in `if-then-else`.

The type of the new variable `b` is that of `x`, where range types have been replaced by `int`. Range constraints are reconstructed in a type condition that is added to the formula `f`.

Example: For formulas `e,f,g`, `Forall x : [1..N]. h(x)`, and `y` of type integer not free in these, $\boxed{\text{Skolemize}}$ eliminates the universal force quantifier by:

$$\frac{\texttt{f} \quad\Longrightarrow\quad \texttt{e --> g /\textbackslash\ (1 <= y /\textbackslash\ y <= N --> h(y))}}{\texttt{f} \quad\Longrightarrow\quad \texttt{e --> g /\textbackslash\ Forall x : [1..N]. h(x)}}$$

$\boxed{\text{Instantiate}}$ Quantifiers of existential force can be instantiated by terms.

After clicking on this button, a data entry window appears for entering the term you want to instantiate the bound variable with.

$\boxed{\text{Instantiate}}$ implements the inference rules:

$$\frac{\Gamma \quad\Longrightarrow\quad \Delta,\ \texttt{p(t)}}{\Gamma \quad\Longrightarrow\quad \Delta,\ \texttt{Exists x . p(x)}}$$

$$\frac{\Gamma,\ \texttt{p(t)} \quad\Longrightarrow\quad \Delta}{\Gamma,\ \texttt{Forall x . p(x)} \quad\Longrightarrow\quad \Delta}$$

Propositional Rules

Sequents and formulas in square brackets [] are only generated when the sequent below the line is above an application of the rule *Next*. This guarantees that statements such as $\Box\boxminus(\mathit{first})$ are not derivable, as opposed to $\boxminus\,\mathit{first}$.
(first is shorthand for $\neg\ominus$ `true`, which only holds at position 0.)

$$(\Rightarrow \neg)\qquad \frac{\Gamma, \varphi \Rightarrow \Delta}{\Gamma \Rightarrow \Delta, \neg\varphi} \qquad\qquad (\neg \Rightarrow)\qquad \frac{\Gamma \Rightarrow \Delta, \varphi}{\Gamma, \neg\varphi \Rightarrow \Delta}$$

$$(\Rightarrow \vee)\qquad \frac{\Gamma \Rightarrow \Delta, \varphi, \psi}{\Gamma \Rightarrow \Delta, \varphi \vee \psi} \qquad\qquad (\vee \Rightarrow)\qquad \frac{\Gamma, \psi \Rightarrow \Delta \quad \Gamma, \varphi \Rightarrow \Delta}{\Gamma, \varphi \vee \psi \Rightarrow \Delta}$$

$$(\Rightarrow\rightarrow)\qquad \frac{\Gamma, \varphi \Rightarrow \Delta, \psi}{\Gamma \Rightarrow \Delta, \varphi \rightarrow \psi} \qquad\qquad (\rightarrow\Rightarrow)\qquad \frac{\Gamma, \psi \Rightarrow \Delta \quad \Gamma \Rightarrow \Delta, \varphi}{\Gamma, \varphi \rightarrow \psi \Rightarrow \Delta}$$

$$(\Rightarrow\leftrightarrow)\qquad \frac{\Gamma, \varphi \Rightarrow \Delta, \psi \quad \Gamma, \psi \Rightarrow \Delta, \varphi}{\Gamma \Rightarrow \Delta, \varphi \leftrightarrow \psi} \qquad\qquad (\leftrightarrow\Rightarrow)\qquad \frac{\Gamma, \psi, \varphi \Rightarrow \Delta \quad \Gamma \Rightarrow \Delta, \psi, \varphi}{\Gamma, \varphi \leftrightarrow \psi \Rightarrow \Delta}$$

$$(\Rightarrow \wedge)\qquad \frac{\Gamma \Rightarrow \Delta, \psi \quad \Gamma \Rightarrow \Delta, \varphi}{\Gamma \Rightarrow \Delta, \varphi \wedge \psi} \qquad\qquad (\wedge \Rightarrow)\qquad \frac{\Gamma, \varphi, \psi \Rightarrow \Delta}{\Gamma \Rightarrow \Delta, \varphi \wedge \psi}$$

$$(\Rightarrow \Box)\qquad \frac{\Gamma \Rightarrow \Delta, \varphi \quad \varphi, \Gamma \Rightarrow \Delta, \bigcirc\Box\varphi}{\Gamma \Rightarrow \Delta, \Box\varphi} \qquad\qquad (\Box \Rightarrow)\qquad \frac{\Gamma, \varphi, \bigcirc\Box\varphi \Rightarrow \Delta}{\Gamma, \Box\varphi \Rightarrow \Delta}$$

$$(\Rightarrow \Diamond)\qquad \frac{\Gamma \Rightarrow \Delta, \varphi, \bigcirc\Diamond\varphi}{\Gamma \Rightarrow \Delta, \Diamond\varphi} \qquad\qquad (\Diamond \Rightarrow)\qquad \frac{\Gamma, \varphi \Rightarrow \Delta \quad \bigcirc\Diamond\varphi, \Gamma \Rightarrow \Delta, \varphi}{\Gamma, \Diamond\varphi \Rightarrow \Delta}$$

$$(\Rightarrow \mathcal{W})\qquad \frac{\Gamma, \varphi, \bigcirc(\varphi\,\mathcal{W}\,\psi) \Rightarrow \Delta \quad \Gamma, \psi \Rightarrow \Delta}{\Gamma, \varphi\,\mathcal{W}\,\psi \Rightarrow \Delta} \qquad\qquad (\mathcal{W} \Rightarrow)\qquad \frac{\Gamma \Rightarrow \Delta, \varphi, \psi \quad \varphi, \Gamma \Rightarrow \Delta, \bigcirc(\varphi\,\mathcal{W}\,\psi), \psi}{\Gamma \Rightarrow \Delta, \varphi\,\mathcal{W}\,\psi}$$

$$(\Rightarrow \mathcal{U})\qquad \frac{\Gamma, \varphi, \bigcirc(\varphi\,\mathcal{U}\,\psi) \Rightarrow \Delta, \psi \quad \Gamma, \psi \Rightarrow \Delta}{\Gamma, \varphi\,\mathcal{U}\,\psi \Rightarrow \Delta} \qquad\qquad (\mathcal{U} \Rightarrow)\qquad \frac{\Gamma \Rightarrow \Delta, \varphi, \psi \quad \Gamma \Rightarrow \Delta, \bigcirc(\varphi\,\mathcal{U}\,\psi), \psi}{\Gamma \Rightarrow \Delta, \varphi\,\mathcal{U}\,\psi}$$

$$(\Rightarrow \boxminus)\qquad \frac{\Gamma \Rightarrow \Delta, \varphi \quad [\varphi, \Gamma \Rightarrow \Delta, \ominus\boxminus\varphi]}{\Gamma \Rightarrow \Delta, \boxminus\varphi} \qquad\qquad (\boxminus \Rightarrow)\qquad \frac{\Gamma, \varphi[, \ominus\boxminus\varphi] \Rightarrow \Delta}{\Gamma, \boxminus\varphi \Rightarrow \Delta}$$

$$(\Rightarrow \diamondminus)\qquad \frac{\Gamma \Rightarrow \Delta, \varphi[, \ominus\diamondminus\varphi]}{\Gamma \Rightarrow \Delta, \diamondminus\varphi} \qquad\qquad (\diamondminus \Rightarrow)\qquad \frac{\Gamma, \varphi \Rightarrow \Delta \quad [\ominus\diamondminus\varphi, \Gamma \Rightarrow \Delta, \varphi]}{\Gamma, \diamondminus\varphi \Rightarrow \Delta}$$

$$(\Rightarrow \mathcal{B})\qquad \frac{\Gamma \Rightarrow \Delta, \varphi, \psi \quad [\varphi, \Gamma \Rightarrow \Delta, \ominus(\varphi\,\mathcal{B}\,\psi), \psi]}{\Gamma \Rightarrow \Delta, \varphi\,\mathcal{B}\,\psi} \qquad\qquad (\mathcal{B} \Rightarrow)\qquad \frac{\Gamma, \varphi[, \ominus(\varphi\,\mathcal{B}\,\psi)] \Rightarrow \Delta \quad \Gamma, \psi \Rightarrow \Delta}{\Gamma, \varphi\,\mathcal{B}\,\psi \Rightarrow \Delta}$$

$$(\Rightarrow \mathcal{S})\qquad \frac{[\Gamma \Rightarrow \Delta, \varphi, \psi] \quad \Gamma \Rightarrow \Delta[, \ominus(\varphi\,\mathcal{S}\,\psi)], \psi}{\Gamma \Rightarrow \Delta, \varphi\,\mathcal{S}\,\psi} \qquad\qquad (\mathcal{S} \Rightarrow)\qquad \frac{[\Gamma, \varphi, \ominus(\varphi\,\mathcal{S}\,\psi) \Rightarrow \Delta, \psi] \quad \Gamma, \psi \Rightarrow \Delta}{\Gamma, \varphi\,\mathcal{S}\,\psi \Rightarrow \Delta}$$

where `t` is a term and, `p(x)` is a formula containing `x`. Complementing skolemization, instantiation applies to any existential force quantifiers under
`-->`, `\/`, `/\`, `~`, `if-then-else`.

If the user-supplied term `t` is not obviously of the same type as `x`, Instantiate generates additional type constraint conditions along with the instantiated formula.

Example: For formulas `(N > 0)`, `Exists : [0..N-1]. h(x)`, and term `t`, Instantiate `x` with `t` eliminates the existential quantifier in the following way:

$$\frac{\texttt{N} > \texttt{0} \implies \texttt{0} <= \texttt{t} \wedge \texttt{t} <= \texttt{N} - \texttt{1} \wedge \texttt{h(t)}}{\texttt{N} > \texttt{0} \implies \texttt{Exists x : [0..N} - \texttt{1]. h(x)}}$$

Replace This button allows you to replace equals by equals, that is, any term that is asserted to be equal to another term in the antecedent may be replaced with that other term throughout the sequent.

Upon clicking on this button all terms that are candidates to be replaced are highlighted, and you can select the term that you want to replace with.

Example: Starting with the sequent

$$\texttt{x} = 3,\ \texttt{y} = 4 \implies \texttt{x} * \texttt{x} + \texttt{y} * \texttt{y} = 25$$

clicking on Replace highlights: `x`, `y`, 3, and 4. Clicking on 3 replaces `x` with 3 in the consequent, resulting in

$$\texttt{x} = 3,\ \texttt{y} = 4 \implies 3 * 3 + \texttt{y} * \texttt{y} = 25$$

Rewrite Brings up the **Select rewrite** window, which displays all rewrite rules that were declared in the specification during the current verification session via

`REWRITE` *desc* : *expn* `--->` *expn*

Rules can be selected and de-selected by clicking on the Select buttons. When clicking on the Apply button, the current goal is simplified according to the selected rewrite rules.

Make-first-order Upon clicking on this button with current goal sequent G, a first-order subgoal G' is generated, such that the first-order validity of G' implies the first-order temporal validity of G.

Next Upon clicking on this button a new subgoal is created in which the operator $\bigcirc$ is deleted from (sub)formulas preceded by it, and in which $\ominus$ is appended to (sub)formulas not preceded by $\bigcirc$.

$\boxed{\textsf{Cut}}$ With the **Cut** rule you can perform a case split analysis on the current goal. Upon clicking on this button, a dialog window will appear, where you can enter the formula to split on. Thus upon having entered the cut-formula `p`, $\boxed{\textsf{Cut}}$ implements the inference rule

$$\frac{\begin{array}{lcl}\Gamma, \mathtt{p} & \Longrightarrow & \Delta \\ \Gamma & \Longrightarrow & \Delta, \mathtt{p}\end{array}}{\begin{array}{lcl}\Gamma & \Longrightarrow & \Delta\end{array}}$$

$\boxed{\textsf{Induction}}$ Upon clicking on this button a dialog window will appear in which you can enter the variable you want to do induction on. The variable has to be a universally quantified integer variable. The button implements the inference rule:

$$\frac{\begin{array}{lcl}\Gamma & \Longrightarrow & \Delta,\ \mathtt{p(0)} \\ \Gamma & \Longrightarrow & \Delta,\ \mathtt{Forall\ x\ :\ int\ .\ (x\ >=\ 0\ /\backslash\ p(x)\ \ -->\ \ p(x+1))} \\ \Gamma & \Longrightarrow & \Delta,\ \mathtt{Forall\ x\ :\ int\ .\ (x\ <=\ 0\ /\backslash\ p(x)\ \ -->\ \ p(x-1))}\end{array}}{\begin{array}{lcl}\Gamma & \Longrightarrow & \Delta,\ \mathtt{Forall\ x\ :\ int\ .\ p(x)}\end{array}}$$

where `p` is a formula containing `x`.

$\boxed{\textsf{Free-Induction}}$ supports simple mathematical induction for free variables. Suppose that the integer variable `x` occurs free in the sequent $\Gamma \Longrightarrow \Delta$, which we recall corresponds to a formula:

$$\mathtt{p(x)} : \bigwedge_{\gamma(x)\in\Gamma} \gamma(x) \rightarrow \bigvee_{\delta(x)\in\Delta} \delta(x).$$

Since x is implicitly universally quantified it makes sense to formulate the induction principle:

$$\frac{\begin{array}{lcl} & \Longrightarrow & \mathtt{p(0)} \\ & \Longrightarrow & \mathtt{Forall\ x\ :\ int\ .\ (x\ >=\ 0\ /\backslash\ p(x)\ \ -->\ \ p(x+1))} \\ & \Longrightarrow & \mathtt{Forall\ x\ :\ int\ .\ (x\ <=\ 0\ /\backslash\ p(x)\ \ -->\ \ p(x-1))}\end{array}}{\begin{array}{lcl}\Gamma & \Longrightarrow & \Delta\end{array}}$$

$\boxed{\textsf{Presburger}}$ Decides formulas of Presburger arithmetic. Formulas belonging to Presburger arithmetic may involve integer valued variables, addition, multiplication by constants, boolean connectives, and quantifiers over integer variables.

The decision problem is super-exponential, and thus only works on small examples. It is seldom needed in practice, as the decision procedure cannot handle uninterpreted function symbols (e.g., arrays).

Example: The following are Presburger formulas:

```
Forall x : int . Exists y : int . (x + y > 0)
Exists x : int . Forall y : int . Exists z : int .
     (x > y \/ x < y \/ z < y - x /\ 5*z = x)
```

The following are not Presburger formulas:

```
Forall x : int . Exists y : int . (f(x) + y > 0)
Exists x : int . Forall y : rat . (x > y)
```

since the first contains an uninterpreted function symbol `f`, while the second includes the unsupported type `rat`.

All-Propositional Applies 1-Step-Propositional recursively until there are no new formulas to which it is applicable.

BDD-split Simplifies the purely propositional structure of the formulas in the sequent. Thus, a formula whose main connective is a temporal operator `[]`, `<>`, `Awaits`, `Until`, `Since`, `Backto`, `[-]`, `<->`, is treated as an atomic formula (as opposed to All-Propositional). See Section 6.2 for further explanations.

Add-Axiom A list of previously established properties or asserted axioms is displayed in a separate window. The user selects the previously proved property or asserted axiom to be inserted in the antecedent of the sequent.

The Select boxes menu-option under Settings allows the user to automatically remove or retain preceding boxes from the axioms added by Add-Axiom. By default boxes are removed from axioms added to the sequent.

Example: By default, adding the axiom `[](p(x) /\ q(y))` to the sequent $\Gamma \Longrightarrow \Delta$ generates

$$(\mathtt{p(x)}\ \ /\backslash\ \ \mathtt{q(y)}), \Gamma \Longrightarrow \Delta.$$

Selecting Select boxes keeps the boxes. Hence adding the same axiom generates the sequent:

$$[](\mathtt{p(x)}\ \ /\backslash\ \ \mathtt{q(y)}), \Gamma \Longrightarrow \Delta.$$

PTL-expansion The structure of the Gentzen sequent calculus resembles the one used in temporal tableau-like decision procedures for propositional linear-time temporal logic. The prover therefore supports an automatic decision procedure for propositional temporal logic.

Disclaimer: This implementation of PTL-expansion only applies well to small formulas, so its use should be limited to textbook examples. More efficient implementation are available, e.g., [Kesten *et al.*, 1993].

Chapter 9

Basic Tutorial

This chapter presents detailed verification sessions that can be followed as a tutorial for STeP and used as templates for other verifications. All input files mentioned in this chapter are available in the `examples` directory in the STeP distribution. We assume that you have successfully started STeP (see Section 1.2).

9.1 Preliminaries

The input to STeP will be an SPL program P and a temporal-logic formula φ that expresses a property of P to be verified. A verification session deals with a single program and one or more properties to be verified for that program. Thus, loading a new program reinitializes the system, and starts a new verification session.

Loading a program

To load an SPL program, select the File menu option in the upper left corner (see Figure 3.2), and select Load program. This will start up the File Browser with a Load program window, as shown in Figure 3.3. By clicking on the *.spl button, you can choose to see only the files with extension `.spl`, the standard extension for SPL program files. The right column shows the directory hierarchy, and the left column shows the files in the current directory. Clicking on a directory in the right column makes it the current directory. You can load a file by clicking on its filename in the left column. Alternatively, you may enter the full filename directly in the entry line under `File name`. After entry, click the OK button, and the selected file will be loaded.

Loading the specification

To load a specification, select the Load specification option from the File menu. Again, this activates the File Browser, which brings up a Load specification window. The default extension for specification files is `.spec`. If the specification file contains a single formula, this formula is understood to be the top-level goal, and is displayed in the top window as the new current goal. If the file contains multiple formulas to be proved, the first one is displayed as the current top-level goal. The other top-level goals can be accessed by selecting the Next search or Select search option under the Properties menu.

Specification files can also contain declarations, axioms and definitions—see Chapter 2.

Help

At any time, you can get online help by selecting the Help option from the menu. This will bring up a WWW browser with STeP's help pages.

Quitting STeP

To leave STeP, select the Quit option from the top-level menu or select the Quit option from the File menu option. Both will bring up a confirmation window asking if you really want to exit STeP.

9.2 MUX-SEM: Mutual Exclusion

We now assume that you have started STeP and are able to load program and specification files. The program and specifications mentioned in this section are available in the directory `examples/mux-sem`.

$$\textbf{local} \quad y \quad : \textbf{integer where } y = 1$$

$$P_1 :: \left[\begin{array}{l} \ell_0\text{: } \textbf{loop forever do} \\ \left[\begin{array}{ll} \ell_1\text{:} & \textbf{noncritical} \\ \ell_2\text{:} & \textbf{request } y \\ \ell_3\text{:} & \textbf{critical} \\ \ell_4\text{:} & \textbf{release } y \end{array}\right] \end{array}\right] \quad || \quad P_2 :: \left[\begin{array}{l} m_0\text{: } \textbf{loop forever do} \\ \left[\begin{array}{ll} m_1\text{:} & \textbf{noncritical} \\ m_2\text{:} & \textbf{request } y \\ m_3\text{:} & \textbf{critical} \\ m_4\text{:} & \textbf{release } y \end{array}\right] \end{array}\right]$$

Figure 9.1: Program MUX-SEM (mutual exclusion by semaphores)

We begin with the proof of two simple safety properties for the program MUX-SEM, shown in Figure 9.1: the first, φ_1, states that y is always greater than 0, and the second, φ_2, expresses mutual exclusion. These properties are expressed by the formulas

$$\varphi_1 : \Box(y \geq 0)$$

$$\varphi_2 : \Box(\neg(\ell_3 \ \wedge \ m_3))$$

or, in STeP's input format,

φ_1 : `[](y >= 0)`

φ_2 : `[]!(l3 /\ m3).`

To illustrate the various methods available for verification we will prove these properties in a number of different ways.

In this example we will load each property from a separate file. Subsequent examples will illustrate specification files with multiple properties, as well as entering goals directly.

9.2.1 Using B-INV

The first property, $\square(y \geq 0)$, is *inductive*, so we should in principle be able to be verify it using rule B-INV. We can do this in the following steps:

1. Load the program MUX-SEM by loading the file `mux-sem.spl`. This will bring up the Program Text window shown in Figure 9.2. Apart from the program, it contains a list of how the program's control locations correspond to STeP's internal variables. Control locations are explained in detail in Appendix C.2. When an SPL program is loaded, a fair transition system is automatically generated, which you can view by selecting View Transitions from the File menu.

2. Load the specification file `mux-sem/yge0.spec`. The property appears in the current goal window, as shown in Figure 9.3.

3. Enable automatic simplification by selecting the Automatic Simplification ON/OFF option under the Settings menu.

4. Enable the use of decision procedures for linear arithmetic, by selecting the Simplification Flags option under the Settings menu. A window with the title "`Main Flags`" will appear, as shown in Figure 6.1. Click on the second switch to include decision procedures for linear arithmetic in the standard simplification, and press the OK button.

   (Throughout this tutorial we will assume that the decision procedures for equality are also enabled; this is the default for STeP).

5. Click on the B-INV button in the action region. One by one, the verification conditions for all transitions are displayed in the current goal window, and the result of their simplification is displayed in the output window (see Figure 9.4). Since all verification conditions are simplified to *true*, the proof is complete, as indicated.

The single original goal has now been proved, so there is no current (unproven) goal, as indicated in the current goal window. The proved property is saved as a background property and can be used in the proof of subsequent properties. The background properties can be viewed by selecting the Activate/Deactivate option under the Properties menu. This brings up a Property Checklist window, shown in Figure 9.5. By default, properties are active and will be used in the proof of subsequent properties. However, you may deactivate a property by clicking on the switch next to it.

Inactive properties can be reactivated later on; however, you may also entirely delete a property by clicking on the skull.

To save the proof to a file (in ASCII format), select the Save last proof option of the File menu. The File Browser brings up a window with the title "`Save completed proof`".

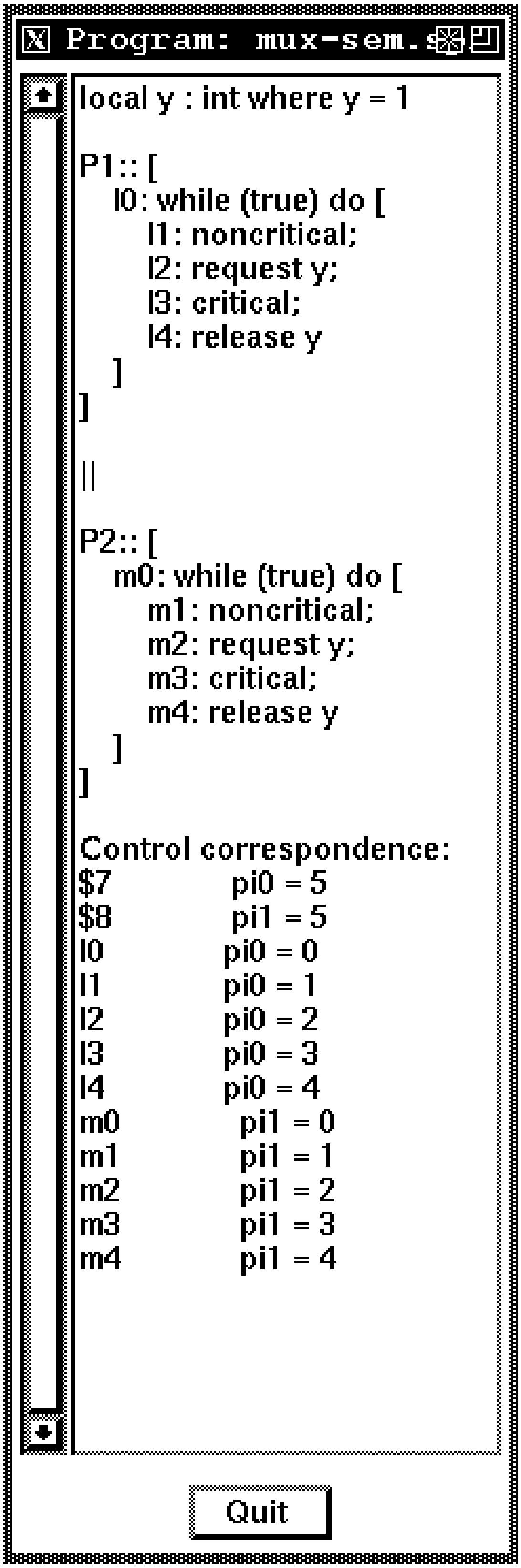


Figure 9.2: MUX-SEM program text window

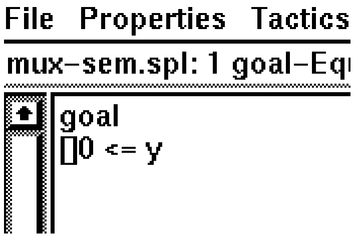


Figure 9.3: Loaded specification for MUX-SEM

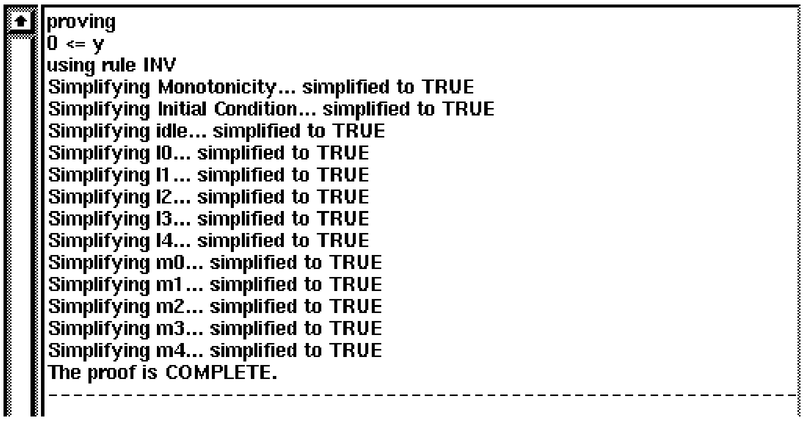


Figure 9.4: Verification of `[]y >= 0` for MUX-SEM

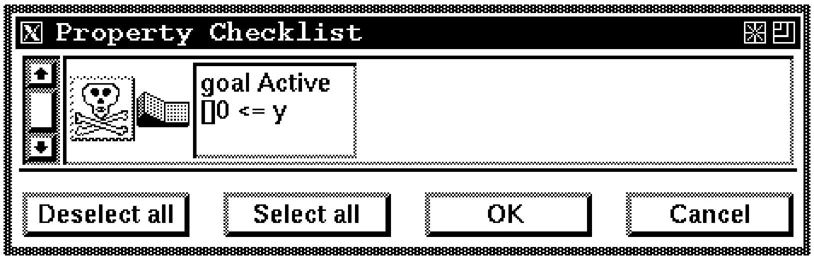


Figure 9.5: Background properties

As when loading a file, you can select an existing filename or enter a new one. The default extension for saved proofs is `.search`. The window also allows you to enter three lines of text that will be added to the top of the file as a comment. Finally, click on $\boxed{\text{OK}}$ to save your proof. Figure 9.6 shows the format of the saved proof.

```
% Proof of [](y>=0) for program mux-sem, with decision
% procedures for linear arithmetic turned on, using rule B-INV.
Formula abbreviations:
phi0: 0 <= y
Reduced via INV:
[]0 <= y
given the strengthening formula:
  0 <= y
1 Established via Simplify: true
2 Established via Simplify: y = 1 /\ pi0 = 0 /\ pi1 = 0 --> 0 <= y
3 Established via Simplify: {phi0} idle {phi0}
4 Established via Simplify: {phi0} l0 {phi0}
5 Established via Simplify: {phi0} l1 {phi0}
6 Established via Simplify: {phi0} l2 {phi0}
7 Established via Simplify: {phi0} l3 {phi0}
8 Established via Simplify: {phi0} l4 {phi0}
9 Established via Simplify: {phi0} m0 {phi0}
10 Established via Simplify: {phi0} m1 {phi0}
11 Established via Simplify: {phi0} m2 {phi0}
12 Established via Simplify: {phi0} m3 {phi0}
13 Established via Simplify: {phi0} m4 {phi0}
```

Figure 9.6: Transcript for proof of `[](y>=0)` over MUX-SEM

9.2.2 Using G-INV

The next property we will prove, φ_2, expresses mutual exclusion. This property is not inductive, even relative to the simple property φ_1 proved before. However, we can prove it by *strengthening* the invariant being proved, as follows:

1. Load the specification file `mux-sem/mux.spec`. The property φ_2 will appear in the current goal window.

2. Click on $\boxed{\text{G-INV}}$. This will bring up a window with the title "`Input auxiliary assertions`". By default, the auxiliary assertion is the goal to be proven, so when the default is used G-INV behaves exactly as B-INV. Now enter the following formula after deleting the default one:

   `(l3+l4+m3+m4+y) = 1`

and click on $\boxed{\text{OK}}$. If there is a syntax error in the entered formula, a message will be displayed at the bottom of the entry window, and you will be asked to try again.

Using boolean terms such as `l3` and `m3` as integers is a convenient feature for writing succinct specifications, and is called *arithmetization*. The boolean value *true* is also interpreted as 1, and *false* as 0 (see Chapter 2).

After the strengthened assertion is entered, all verification conditions are simplified to *true* and the proof is complete.

Note: The strengthened property only implies mutual exclusion based on the invariant $\varphi_1 : \Box(y \geq 0)$ that we proved before. If you do not prove this property before attempting the proof of mutual exclusion, the Monotonicity verification condition will fail to simplify to *true*.

The basic but useful property $y \geq 0$ can also be generated automatically using STeP's invariant generation tools. To see how this can be used, reset the searches with the Reset searches option under the File menu, and repeat the two steps using $\boxed{\text{G-INV}}$ above. The Monotonicity property will not be simplified to *true*. Now we can do the following:

3. Select the Get local invariants option from the Properties menu. Five invariants appear in the output window. The first invariant, in Figure 9.7, is φ_1, the property we need.

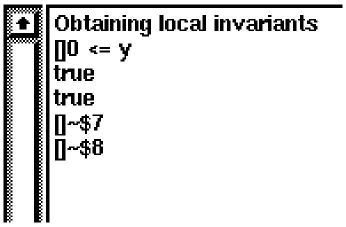


Figure 9.7: Local invariants for MUX-SEM

4. Select the $\boxed{\text{Simplify}}$ button. Now the Monotonicity condition should simplify to *true*, and the proof is complete.

Note also that these simplification steps succeed only if the linear arithmetic decision procedures are enabled. (This is done with the Simplification Flags option under the Settings menu). If they are not enabled, the Monotonicity premise will still fail to be simplified to *true*.

9.2.3 Using the Model Checker

Since program MUX-SEM is a finite-state program we may use the Model Checker to prove mutual exclusion, as follows:

1. Load again the mutual-exclusion property from the `mux-sem/mux.spec` file. (The model checker does not use background properties, so you don't have to reset the current background properties at this point.)

2. Click on the Modelcheck button. This will bring up a window titled "`Input auxiliary assertions`". This window allows the user to enter additional assumptions under which model checking is to be done. Since we don't need any, we can just click the OK button. A File Browser window titled "`Modelcheck log file`" will appear. Here, you have to specify a file in which you want to have model checker results recorded. The default extension of the model checker log file is `.mclog`.

3. After the log file is specified, the model checker will run. We will then see in the output window:

   ```
   Modelcheck succeeded. The proof is COMPLETE.
   ```

   The model checker log file provides more detailed information, such as the number of states explored. In general, however, the log file is interesting only when the model checker fails. In this case, the file will include a counterexample computation, that is, a computation of the system which violates the original property.

9.2.4 Using Verification Diagrams

In the discussion of MUX-SEM, [Manna and Pnueli, 1995] presents a state-space partition graph. This graph can easily be converted into the invariance verification diagram shown in Figure 9.8. This diagram may be used to prove mutual exclusion for MUX-SEM as follows:

1. Reset the background properties and current searches with the Reset searches option under the File menu.

2. Load again the mutual-exclusion property from the `mux-sem/mux.spec` file.

3. Select the Edit diagram option under the Diagrams menu. This brings up the Verification Diagram Editor.

4. Select the Load option under the File menu in the Verification Diagram Editor window. This brings up the File Browser.

5. Select the file `mux-sem/state-partition.diagram`. This loads the verification diagram, which is displayed in the editor window, as shown in Figure 9.8.

6. Select the Current-to-Verify option of the File menu of the Verification Diagram Editor. This will make the verification diagram accessible to the Top-level Prover.

7. Select the Verification diagram rule option of the Diagrams option. This generates all the verification conditions corresponding to the given verification diagram as subgoals. If automatic simplification is enabled, the simplifier will proceed to simplify each subgoal.

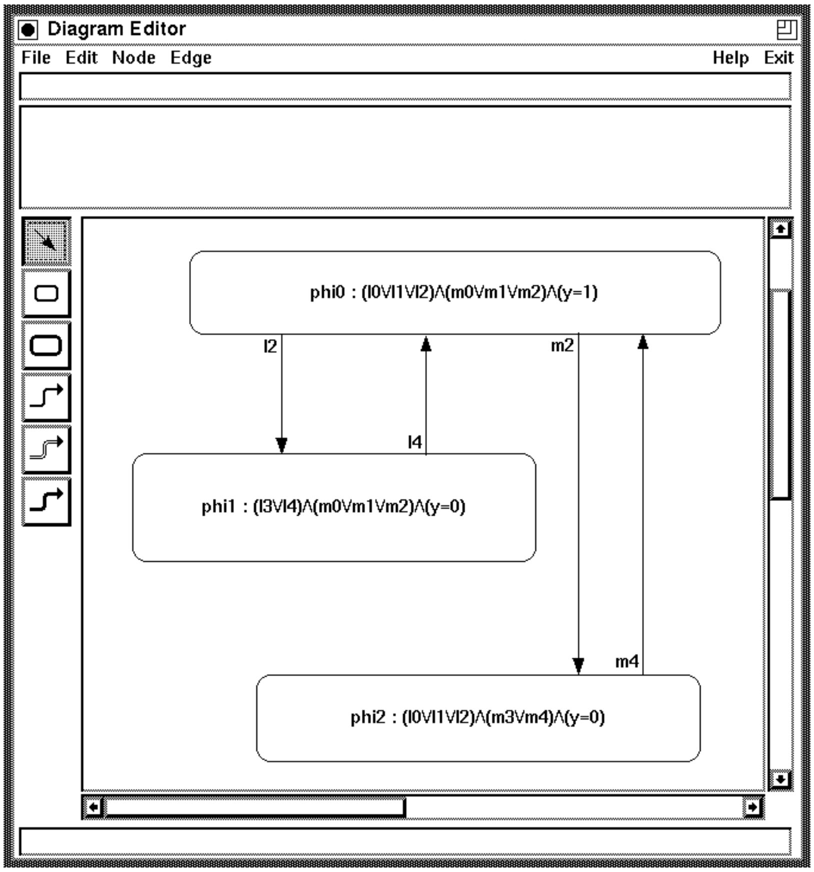


Figure 9.8: Invariance verification diagram for MUX-SEM

The first two subgoals are *side verification conditions*. They state that the label on each of the nodes implies the invariant to be proven, and that the initial condition implies the label of one of the nodes. Then the verification conditions corresponding to each of the edges in the diagram are simplified (including those for the implicit self-loops). In this case, all verification conditions simplify to *true* (even without the linear arithmetic decision procedures), and the proof is complete.

9.2.5 Using MON-I and Linear Invariants

Finally, we show how we can prove φ_2 based on φ_1 and automatically generated *linear invariants*:

1. Load the `examples/mux-sem.spl` program and prove $\varphi_1 : \Box(y \geq 0)$ using B-INV, as described in Section 9.2.1.

2. Load the goal φ_2 from `examples/mux.spec`.

3. Select the Get global linear invariants option under the Properties menu. This will automatically generate a number of linear invariants which are added to the set of background properties, including

$$\Box(l_0 + l_1 + l_2 = m_3 + m_4 + y).$$

   Your set of background properties should now contain this formula, as well as φ_1.

4. Make sure that linear decision procedures are enabled (with the Simplification Flags option under the Settings menu).

5. Select the MON-I rule under the Logical rules menu. This will generate a single verification condition as the subgoal, stating that the conjunction of all the background invariants implies the original goal.

6. Click on the $\boxed{\text{Simplify}}$ button; the verification condition will be simplified to *true*, and the proof is complete. (If automatic simplification is ON, this simplification will be done automatically after MON-I is invoked.)

9.3 MUX-PET1: Invariance Strengthening

In this section we illustrate the construction of the proof tree with the program MUX-PET1, shown in Figure 9.9. The program and specification used in this section can be found in the files `mux-pet1/mux-pet1.spl` and `mux-pet1/mux.spec` respectively.

9.3.1 Using WPC

Mutual exclusion for MUX-PET1 is not an inductive property. Instead of finding a strengthened assertion and using G-INV, as we did for program MUX-SEM, we use STeP to do the strengthening for us:

$$
\begin{array}{lll}
\textbf{local} & y_1, y_2 & : \textbf{boolean where } y_1 = y_2 = \mathrm{F} \\
 & s & : [1..2]
\end{array}
$$

$$
P_1 :: \left[\begin{array}{l}
\ell_0\text{: } \textbf{while } \mathrm{T} \textbf{ do} \\
\quad \left[\begin{array}{ll}
\ell_1\text{:} & \textbf{noncritical} \\
\ell_2\text{:} & (y_1, s) := (\mathrm{T}, 1) \\
\ell_3\text{:} & \textbf{await } \neg y_2 \textbf{ or } s = 2 \\
\ell_4\text{:} & \textbf{critical} \\
\ell_5\text{:} & y_1 := \mathrm{F}
\end{array}\right]
\end{array}\right]
\quad || \quad
P_2 :: \left[\begin{array}{l}
m_0\text{: } \textbf{while } \mathrm{T} \textbf{ do} \\
\quad \left[\begin{array}{ll}
m_1\text{:} & \textbf{noncritical} \\
m_2\text{:} & (y_2, s) := (\mathrm{T}, 2) \\
m_3\text{:} & \textbf{await } \neg y_1 \textbf{ or } s = 1 \\
m_4\text{:} & \textbf{critical} \\
m_5\text{:} & y_2 := \mathrm{F}
\end{array}\right]
\end{array}\right]
$$

Figure 9.9: MUX-PET1 Peterson's algorithm for mutual exclusion

1. Load the program `mux-pet1.spl` and the mutual exclusion property `mux.spec` from the `mux-pet1` directory.
2. Turn automatic simplification on by selecting the Automatic Simplification ON/OFF option under the Settings menu.
3. Generate local invariants by selecting the Get local invariants from the Properties menu.
4. Invoke rule B-INV. As may be expected, the verification conditions for ℓ_3 and m_3 do not simplify to *true*, since they are not valid. The first unproven verification condition is now displayed in the current goal window (in unsimplified form). The other open subgoal, the verification condition for ℓ_3, can be viewed by selecting the [Next] button. Again selecting [Next] goes back to the previous goal.

The Proof Tree

We interrupt our proof for a moment to look at the construction of the proof tree. We started with the property we wanted to prove as the single top-level goal. This goal is the root of the proof tree. When rule B-INV was applied, 15 subgoals were generated in this case: these are the verification conditions for monotonicity (always trivial for B-INV), initial condition, the idle transition, and the 12 program transitions. Since automatic simplification was on, all but two of these subgoals were immediately simplified to *true* and closed, which left us with two open goals.

The buttons [Previous] and [Next] allow you to cycle through the entire list of open goals in the tree. Two other buttons that manipulate the proof tree are [Undo] and [Redo]. Applying [Undo] to a subgoal returns to its parent in the proof tree, and the subgoal and all its siblings are removed from the tree. However, they may be recovered by clicking on [Redo], which restores the proof tree in the reverse order that steps were undone.

When multiple properties are being proved at the same time, multiple proof trees are maintained, each with its own set of open subgoals. The Next search and Select search options under the Properties menu are used to move from one proof search to another.

Finishing the Proof

We now continue with the proof:

5. Click on the $\boxed{\text{WPC}}$ button, which stands for Weakest Pre-Condition. This generates the subgoal $\square\, wpc(\neg(\ell_4 \wedge m_4), m_3)$, in this case

$$(\neg y_1 \vee s = 1) \wedge pi_1 = 3 \Rightarrow \neg(pi_0 = 4),$$

where $pi_1 = 3$ stands for at_m_3 and $pi_0 = 4$ stands for at_ℓ_4 (you may check this by looking at the control correspondence information provided in the Program Text window).

6. The subgoal generated by WPC is again an invariance property. Thus, we can try to prove it using B-INV again. Click on $\boxed{\text{B-INV}}$ to prove this subgoal: all the verification conditions generated by B-INV are simplified to *true*, closing this branch of the proof tree.

7. We can do the same for the unproved verification condition for ℓ_3, which is now the only open goal left. Click again on the $\boxed{\text{WPC}}$ button, and then on the $\boxed{\text{B-INV}}$. Again, all verification conditions are simplified to *true*, thus closing this second branch. This leaves us without any open goals in the tree, so the proof of our original top-level goal is complete.

9.3.2 Using Tactics ⋆

In the previous section we manually applied $\boxed{\text{WPC}}$ to the two verification conditions that were not valid. Applying weakest precondition to invalid verification condition is a common technique that often succeeds, so we may want the application to be automatic. This can be done with the help of *tactics*. STeP provides a simple tactic language in which sequences of rules can be specified. The tactic language is described in detail in Chapter 6.3, and more examples of tactics are presented in Section 6.3.5.

To prove mutual exclusion for MUX-PET1 using tactics we do the following:

1. Load program `mux-pet1.spl` and the specification file `mux.spec`.

2. Select Get local invariants from the Properties menu.

3. Select Enter batchmode tactic from the Tactics menu. This will bring up a data entry window with the title "`Enter tactic`".

4. Enter the following in this window:

```
repeat(B-INV;Simplify;Undo;WPC)
```

and click on $\boxed{\text{OK}}$. This will apply B-INV to the goal. All resulting subgoals will be simplified. For those subgoals that do not simplify to *true*, the simplification will be undone, going back to the unsimplified verification condition, to which WPC is then applied. Then the process is repeated for all open subgoals. In this case, they are invariance formulas, since WPC always generates an invariance formula. Application of the tactic is repeated in this way until there are no open subgoals left. This involves two iterations in this case, and the proof is complete.

Thus, mutual exclusion for `mux-pet1` can be proved automatically by this tactic. Note that the *batchmode* application of a tactic such as this one is risky, since it may not terminate. However, tactics can be interrupted by clicking on the interrupt button (the STeP logo) in the bottom-right corner of the main interface.

When applying a tactic for the first time, it is recommended to apply it in *interactive mode*. In this way, the behavior of the tactic can be followed at each step, and the tactic can be easily interrupted at any time.

9.3.3 Using verification diagrams

The chain diagram given in Figure 5.1 is a high-level proof-outline for process P_1's accessibility to its critical section. It establishes that whenever P_1 requests access to its critical section it will eventually get there. When specified in STeP this reads

```
l2 ==> <>l4.
```

To use the chain-diagram in proving this response property, assume we have already loaded the program, selected automatic simplication, and generated local invariants as described in Section 9.3.1. We now apply the following steps:

1. Select Enter new goal under the Properties menu. Enter the goal `l2 ==> <>l4`.

2. Invoke the diagram editor by selecting Edit verification diagram in the Diagrams menu.

3. From the verification diagram editor select Load in the File menu to load the verification diagram `mux-pet1/response.diagram`. The diagram in Figure 5.1 should now appear in the verification diagram window.

4. Enable STeP's top-level interface to use the verification diagram by selecting Current-to-verify under the File menu in the verification diagram editor.

5. Back in STeP's top-level interface use the verification diagram as a rule by selecting Verification diagram rule from the Diagrams menu. The 88 verification conditions generated from the verification diagram will now all simplify to true automatically.

Appendix A

Computational Model

A.1 Fair Transition Systems

An SPL program is compiled into a fair transition system. A fair transition system $\langle V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C}\rangle$ is given by the following components:

- V: *system variables*, including both *data variables* and *control variables*.

- Θ: *initial condition*, which is a satisfiable assertion characterizing all the initial states of a computation.

- $\mathcal{T}$: set of *transitions*. Each transition $\tau \in \mathcal{T}$ is a function

  $$\tau : \; \Sigma \mapsto 2^{\Sigma},$$

  where Σ is the set of all states, and a state is a type-consistent interpretation of V. Each state in $\tau(s)$ is called a τ-successor of s. A transition τ is said to be *enabled* on S if $\tau(s) \neq \emptyset$. Otherwise it is said to be *disabled*.

- $\mathcal{J} \subseteq \mathcal{T}$: the set of *just* transitions (also called weakly fair transitions).

- $\mathcal{C} \subseteq \mathcal{T}$: the set of *compassionate* transitions (also called strongly fair transitions).

Each transition $\tau \in \mathcal{T}$ is represented by a first-order formula $\rho_\tau(V, V')$, called the *transition relation*, which may refer to both unprimed and primed versions of the system variables. The transition relation expresses the relation holding between a state s and any of its τ-successors $s' \in \tau(s)$. The unprimed version of a variable refers to its value in s, and the primed version refers to its value in s'.

Thus, the state s' is a τ-successor of the state s if the formula $\rho_\tau(V, V')$ evaluates to `true`, when we interpret each $x \in V$ as $s[x]$, and its primed version x' as $s'[x]$.

If not explicitly specified, STeP adds to each fair transition system the *idling* transition τ_I (also called the stuttering transition), whose transition relation is $\rho_I : (V = V')$.

A.2 Computations

An infinite sequence of states

$$\sigma : s_0, s_1, s_2, \ldots$$

is defined to be a *computation* of a fair transition system P if it satisfies the following requirements:

- *Initiality:* s_0 is initial, that is, it satisfies the initial condition.
- *Consecution:* For each $j = 0, 1, \ldots$ the state s_{j+1} is a τ-successor of the state s_j.
- *Justice:* For each transition $\tau \in \mathcal{J}$, it is not the case that τ is continually enabled beyond some position j in σ but not taken.
- *Compassion:* For each transition $\tau \in \mathcal{C}$, it is not the case that τ is enabled at infinitely many positions in σ but taken at only finitely many positions.

A.3 SPL semantics

System Variables

The system variables V consist of the program variables declared in the program, and a set of control variables (approximately one for each spawned subprocess), and a set of location variables (one for each label in the program). A control variable ranges over a subset of the integers, where each value corresponds to a location in a process (see Section C.2). A location variable is a boolean variable with the same name as a program label. Its value is true iff control currently resides at the location of the label, i.e., iff one of the control variables equals the value of the corresponding location.

Initial Condition

The initial condition Θ for program P, with n top-level processes, is defined as

$$\Theta : \ \pi_0 = 0 \ \wedge \ \pi_1 = 0 \ \wedge \ \ldots \ \wedge \ \pi_n = 0 \ \wedge \ \varphi,$$

where φ is the conjunction of all the assertions that appear in the *where* clause of the declaration of P. For a parameterized program, with top-level processes $P[i]$, $1 \leq i \leq N$, the initial condition is given by

$$\Theta : \ \forall i : [1 \ldots N].\pi_0[i] = 0 \ \wedge \ \varphi.$$

Transitions

A transition relation is defined for each of the statements in the language.

To make the definitions more succinct, we define the abbreviation

$$move_S(\ell, \widehat{\ell}) : \ \pi_S = val(\ell) \ \wedge \ \pi'_S = val(\widehat{\ell}) \ \wedge \ loc(\ell) \ \wedge \ loc(\widehat{\ell}'),$$

where π_S refers to the control variable of the innermost subprocess in which the statement S appears, and $val(\ell)$ refers to the integer value associated with the location of ℓ, and $loc(\ell)$ refers to the location variable associated with the location of ℓ.

Since every transition usually modifies only a few variables, we define

$$pres(U) : \bigwedge_{u \in U} (u' = u),$$

where $U \subseteq V$. It states that all variables in U preserve their value at the current step.

In the following we represent a statement S with label ℓ and post-label $\widehat{\ell}$ in the form $[\ell : S; \widehat{\ell}]$. For a statement $[\ell : S; \widehat{\ell}]$ we denote by Y_S, $V - \{\pi_S, loc(\ell), loc(\widehat{\ell})\}$. Unless otherwise specified, all transitions are `Just`.

Basic Statements

- *Skip*
  Statement: $\ell :$ **skip**$; \widehat{\ell} :$
  Transition relation:

  $$\rho_\ell : \; moves_S(\ell, \widehat{\ell}) \;\wedge\; pres(Y_S)$$

- *Assignment*
  Statement: $\ell : \; \bar{u} := \bar{e}; \; \widehat{\ell} :$
  Transition relation:

  $$\rho_\ell : \; moves_S(\ell, \widehat{\ell}) \;\wedge\; \bar{u}' = \bar{e} \;\wedge\; pres(Y_S - \{\bar{u}\}).$$

  The assignment is always enabled (if control is at the assignment statement), even if $\bar{e}$ is not in the range of $\bar{u}$, which may happen with e.g., range types. A proof obligation to check for this type of runtime error is generated by the **Check Runtime System** menu option of the Top-level Prover.

- *Await*
  Statement: $\ell :$ **await** $c; \; \widehat{\ell} :$
  Transition relation:

  $$\rho_\ell : \; moves_S(\ell, \widehat{\ell}) \;\wedge\; c \;\wedge\; pres(Y_S)$$

- *Asynchronous Send*
  Statement: $\ell : \; \alpha \Leftarrow e; \; \widehat{\ell} :$, where α is an asynchronous channel.

  Unbounded channel Transition relation:

  $$\rho_\ell : \; moves_S(\ell, \widehat{\ell}) \;\wedge\; \alpha' = append(\alpha, e) \;\wedge\; pres(Y_S - \{\alpha\}).$$

  Bounded channel For a channel α with bound k, the transition relation is:

  $$\rho_\ell : \; moves_S(\ell, \widehat{\ell}) \;\wedge\; length(\alpha) < k \;\wedge\; \alpha' = append(\alpha, e) \;\wedge\; pres(Y_S - \{\alpha\}).$$

  To ensure that a process cannot be excluded from sending to a bounded channel if it is infinitely often non-full, we associate the `Compassionate` justice requirement with this transition.

- *Asynchronous Receive*
  Statement: $\ell: \ \alpha \Rightarrow u; \widehat{\ell}:$
  Transition relation:

  $$\rho_\ell: \ move(\ell, \widehat{\ell}) \wedge length(\alpha) > 0 \wedge u' = head(\alpha) \wedge \alpha' = tail(\alpha) \wedge pres(Y_S - \{u, \alpha\}).$$

  The asynchronous receive statement is enabled only if the channel is currently nonempty.

  To ensure that a process must receive something if the channel is infinitely often nonempty, the `Compassionate` fairness requirement is associated with this transition.

- *Synchronous Send-Receive*
  With each pair of matching *send* and *receive* statements:

  $$\ell: \ \alpha \Leftarrow e; \widehat{\ell}: \qquad m: \ \alpha \Rightarrow u; \ \widehat{m}:$$

  where two parallel statements are considered matching if they form an $\alpha \Leftarrow e$, $\alpha \Rightarrow u$ pair for some e and u for the same synchronous channel α.
  Transition relation:

  $$\begin{array}{lrcl} \rho_{\langle \ell, m \rangle}: & \pi_{send} & = & val(\ell) \wedge \pi'_{send} = val(\widehat{\ell}) \wedge \\ & \pi_{receive} & = & val(m) \wedge \pi'_{receive} = val(\widehat{m}) \wedge \neg loc(\ell) \wedge \\ & & & loc(\widehat{\ell}) \wedge \neg loc(m) \wedge loc(\widehat{m}) \wedge \\ & & & u' = e \wedge pres(Y_{send,receive}) \end{array}$$

  Thus, transition $\tau_{\langle \ell, m \rangle}$ is enabled if control is at ℓ and m simultaneously. When executed, the transition causes joint progress in the two processes containing the send and receive statement.

  The fairness requirement associated with the generated transitions is `Compassionate`, to ensure that a process cannot wait to transmit or receive a message indefinitely if infinitely many messages are sent on the channel it is waiting on.

- *Request*
  With the statement ℓ: **request**(r); $\widehat{\ell}$:, we associate a transition τ_ℓ, with the transition relation

  $$\rho_\ell: \quad move(\ell, \widehat{\ell}) \ \wedge \ r > 0 \ \wedge \ r' = r - 1 \ \wedge \ pres(Y_S - \{r\}).$$

  Thus, this statement is enabled when control is at ℓ and r is positive. When executed it decrements r by 1.

  To model fair scheduling of semaphores (in the form of for instance single or multi-level feedback queues), the transition associated with **request** is `Compassionate`.

- *Release*
  With the statement ℓ: **release**(r); $\widehat{\ell}$:, we associate a transition τ_ℓ, with the transition relation

  $$\rho_\ell: \quad move(\ell, \widehat{\ell}) \ \wedge \ r' = r + 1 \ \wedge \ pres(Y_S - \{r\}).$$

  This statement increments r by 1.

- *Noncritical*
  With the statement ℓ: **noncritical**; $\widehat{\ell}$:, we associate a transition τ_ℓ, with transition relation
  $$\rho_\ell: \quad move(\ell, \widehat{\ell}) \ \wedge \ pres(Y_S).$$
  Thus, the only observable action of this statement is to terminate. The situation that execution of the noncritical section does not terminate is modeled by a computation that does not take transition τ_ℓ. This is allowed by excluding τ_ℓ from the justice set.

- *Critical*
  With the statement ℓ: **critical**; $\widehat{\ell}$:, we associate a transition τ_ℓ, with transition relation
  $$\rho_\ell: \quad move(\ell, \widehat{\ell}) \ \wedge \ pres(Y_S).$$
  The only observable action of the critical statement is to terminate.

- *Produce*
  With the statement ℓ: **produce** x; $\widehat{\ell}$:, we associate a transition τ_ℓ, with transition relation
  $$\rho_\ell: \quad move(\ell, \widehat{\ell}) \ \wedge \ x' \neq 0 \ \wedge \ pres(Y_S - \{x\}).$$
  The observable action of the produce statement is to assign a nonzero value to variable x. Note that this transition is nondeterministic; that is, a state s may have more than one τ_ℓ-successors. In fact it may have infinitely many successors, one for each possible value of x'.

- *Consume*
  With the statement ℓ: **consume** y; $\widehat{\ell}$:, we associate a transition τ_ℓ, with transition relation
  $$\rho_\ell: \quad move(\ell, \widehat{\ell}) \ \wedge \ y' = 0 \ \wedge \ pres(Y_S - \{y\}).$$
  The observable action of the consume statement is to set variable y to zero.

- *Guarded Assignment*
  Statement: ℓ : **guard** c **do** $\bar{u} := \bar{e}$; $\widehat{\ell}$:
  Transition relation:
  $$\rho_\ell : \ c \ \wedge \ moves_S(\ell, \widehat{\ell}) \ \wedge \ \bar{u}' = \bar{e} \ \wedge \ pres(Y_S - \{\bar{u}\}).$$
  The assignment is enabled if control is at the assignment statement and c evaluates to *true*. Run-time type errors are not considered in the enabling condition of the transition. As with standard assignment, the possibility of run-time errors can be checked with the Check Runtime System option under the Properties menu.

Composite Statements

Standard control constructs and primitives for concurrency compose simpler program statements into more elaborate statements. A composite statement is in general of the form $\mathcal{C}[S_1, ...S_n]$, where $S_1, \ldots, S_n$ are the immediate substatements. The transitions associated with a composite statement comprise of the union of the transitions associated with

$S_1, \ldots, S_n$ and the transitions associated with the construct $\mathcal{C}$. Below we describe the transitions associated with each construct. Here Y_S refers to all system variables except the control and location variables changed by each construct.

- *Conditional*

  One-way With the statement ℓ: [**if** c **then** $\widetilde{\ell}$: S] ; $\widehat{\ell}$:, we associate a transition τ_ℓ, with the transition relation

  $$\rho_\ell\colon \quad \begin{pmatrix} c \;\wedge\; move(\ell, \widetilde{\ell}) \\ \vee \\ \neg c \;\wedge\; move(\ell, \widehat{\ell}) \end{pmatrix} \;\wedge\; pres(Y_S).$$

  According to ρ_ℓ, when c evaluates to *true* control moves from ℓ to $\widetilde{\ell}$, and when c evaluates to *false* control moves from ℓ to $\widehat{\ell}$.

  Two-way With the statement ℓ: [**if** c **then** $\widetilde{\ell}_1$: S_1 **else** $\widetilde{\ell}_2$: S_2] ; $\widehat{\ell}$:, we associate a transition τ_ℓ, with the transition relation

  $$\rho_\ell\colon \quad \begin{pmatrix} c \;\wedge\; move(\ell, \widetilde{\ell}_1) \\ \vee \\ \neg c \;\wedge\; move(\ell, \widetilde{\ell}_2) \end{pmatrix} \;\wedge\; pres(Y_S).$$

  According to ρ_ℓ, when c evaluates to *true* control moves from ℓ to $\widetilde{\ell}_1$, and when c evaluates to *false* control moves from ℓ to $\widetilde{\ell}_2$.

- *While*

  With the statement ℓ: [**while** c **do** [$\widetilde{\ell}$: $\widetilde{S}$]]; $\widehat{\ell}$: we associate a transition τ_ℓ, with the transition relation

  $$\rho_\ell\colon \quad \begin{pmatrix} c \;\wedge\; move(\ell, \widetilde{\ell}) \\ \vee \\ \neg c \;\wedge\; move(\ell, \widehat{\ell}) \end{pmatrix} \;\wedge\; pres(Y_S).$$

  According to ρ_ℓ, when c evaluates to *true* control moves from ℓ to $\widetilde{\ell}$, and when c evaluates to *false* control moves from ℓ to $\widehat{\ell}$. Note that, in this context, the post-location of $\widetilde{S}$ is ℓ. Note also that the enabling condition of τ_ℓ is at_ℓ, which does not depend on the value of c.

- *Loop forever*

  With the statement ℓ: [**loop forever do** [$\widetilde{\ell}$: $\widetilde{S}$]]; $\widehat{\ell}$: we do not associate any transitions. However the locations ℓ, $\widetilde{\ell}$ and post-location of $\widetilde{S}$ are equivalent.

- *Repeat until*

  With the statement ℓ: [**repeat** [$\widetilde{\ell}$: $\widetilde{S}$]$\widehat{\ell}_1$: **until** c];$\widehat{\ell}_2$: we associate a transition $\tau_{\widehat{\ell}_1}$, with the transition relation

  $$\rho_{\widehat{\ell}_1}\colon \quad \begin{pmatrix} c \;\wedge\; move(\widehat{\ell}_1, \widehat{\ell}_2) \\ \vee \\ \neg c \;\wedge\; move(\widehat{\ell}_1, \ell) \end{pmatrix} \;\wedge\; pres(Y_S).$$

According to $\rho_{\widehat{\ell}_1}$, when c evaluates to *true* control moves from $\widehat{\ell}_1$ to $\widehat{\ell}_2$, and when c evaluates to *false* control moves from $\widehat{\ell}_1$ to ℓ. The enabling condition of $\tau_{\widehat{\ell}_1}$ is $at\_\widehat{\ell}_1$, which does not depend on the value of c. Note that the locations ℓ and $\widetilde{\ell}$ are equivalent.

- *When*
  With the statement ℓ: [**when** c **do** $[\widetilde{\ell}{:}\,\widetilde{S}\,]$]; $\widehat{\ell}$: we associate a transition τ_ℓ, with the transition relation
  $$\rho_\ell: \quad c \ \wedge \ move(\ell, \widetilde{\ell}) \ \wedge \ pres(Y_S).$$
  According to ρ_ℓ, when c evaluates to *true* control moves from ℓ to $\widetilde{\ell}$.

- *Or*
  No special transition is associated with the statement S_1 **or** S_2. When control resides at this statement, it is interpreted to reside simultaneously at the starting points of S_1 and S_2.

- *Cooperation*
  With the statement ℓ: $[\ell_1\colon\ S_1{:}\,\widehat{\ell}_1]||[\ell_2\colon\ S_2{:}\,\widehat{\ell}_2]||\ldots||[\ell_n\colon\ S_n{:}\,\widehat{\ell}_n]$ $\widehat{\ell}$: we associate two transitions, $\tau_{\ell_{Entry}}$ and $\tau_{\ell_{Exit}}$, with transition relations:
  $$\rho_{\ell_{Entry}}: \quad move(\ell, \{\ell_1, \ell_2, \ldots, \ell_n\}) \ \wedge \ pres(Y_S).$$
  $$\rho_{\ell_{Exit}}: \quad move(\{\widehat{\ell}_1, \widehat{\ell}_2, \ldots, \widehat{\ell}_n\}, \widehat{\ell}) \ \wedge \ pres(Y_S).$$
  All they do is to move control from the guarding location ℓ into the prelocation of each parallel statement, and from the post-location of each parallel statement to the post-location $\widehat{\ell}$ of the entire statement.

  STeP makes a special case for parallel composition at the top-level of the program. They do not get a designated pre- and post-location, but the initial state is instead what corresponds to $\ell_1, \ell_2, \ldots, \ell_n$ in the schema.

- *Concatenation*
  No special transition is associated with sequential composition of statements:
  $$S_1 \ ; \ S_2$$
  Since the post-location of S_1 is the pre-location of S_2 the transitions generated from each substatement S_1 and S_2 will move control appropriately from S_1 to S_2.

- *Parameterized Cooperation*
  With the statement ℓ: $i : [1..N].\ ||\ \ [\ \ell[i]\colon\ S[i]\ \widehat{\ell}[i]\colon\ ]\ \widehat{\ell}$: we associate two transitions , $\tau_{\ell_{Entry}}$ and $\tau_{\ell_{Exit}}$, with transition relations:
  $$\rho_{\ell_{Entry}}: \quad move(\ell, \{\ell[i] \mid i = 1, \ldots, N\}) \ \wedge \ pres(Y_S).$$
  $$\rho_{\ell_{Exit}}: \quad move(\{\widehat{\ell}[i] \mid i = 1, \ldots, N\}, \widehat{\ell}) \ \wedge \ pres(Y_S).$$
  They are the parameterized versions of the transitions associated with non-parameterized cooperation.

- *Parameterized Or*
  No special transition is associated with parameterized selection. This reflects that the construct is the parameterized version of standard *or*.

- *Blocks*
  Blocks are used to encapsulate variable declarations, and don't generate transitions on their own.

Grouped Statements

To compose several simple statements into one atomic entity, SPL provides the grouping construct **<< >>**. For instance the transition associated with a grouped statement of the form $<< S_1; S_2 >>$ is the relational composition of their transition relations.

Appendix B

Linear-Time Temporal Logic

As a requirement specification language for reactive systems, we take *linear-time temporal logic*. There is an underlying first-order assertion language $\mathcal{L}$ over interpreted symbols for expressing functions and relations over concrete domains such as the integers, arrays, and lists of integers (see Section 2.1).

We refer to a formula in the assertion language $\mathcal{L}$ as a *state formula*, or simply as an *assertion*. A *temporal formula* is constructed out of state formulas by applying the boolean operators $\neg$ and $\vee$ (the other boolean operators can be defined from these), and temporal operators. There are two classes of temporal operators, *future* and *past*. The future and past temporal operators used in STeP are presented in Tables B.1 and B.2, respectively.

Operator	Name	STeP representation
$\Box\, p$	Henceforth p	`[]`
$\Diamond\, p$	Eventually p	`<>`
$p\,\mathcal{U}\,q$	p Until q	`Until`
$p\,\mathcal{W}\,q$	p Waiting-for (Unless) q	`Awaits`
$\bigcirc\, p$	Next p	`()`

Table B.1: Future Temporal Operators

Operator	Name	STeP representation
$\boxminus\, p$	So-far p	`[-]`
$\diamondminus\, p$	Once p	`<->`
$p\,\mathcal{S}\,q$	p Since q	`Since`
$p\,\mathcal{B}\,q$	p Back-to q	`Backto`
$\ominus\, q$	Previously p	`(-)`

Table B.2: Past Temporal Operators

A *model* for a temporal formula p is an infinite sequence of states $\sigma : s_0, s_1, \ldots,$ where each state s_j provides an interpretation for the vocabulary of p, i.e., the variables occurring in p. For a given state s_j and state formula q, we write $s_j \vDash q$ when q is true when evaluated over s_j. Given a model σ, we now present an inductive definition for the notion of a temporal formula p holding at a position $j \geq 0$ in σ, written as $(\sigma, j) \vDash p$:

- For a state formula p,
  $(\sigma, j) \vDash p \iff s_j \vDash p$
  That is, we evaluate p locally, using the interpretation provided by s_j.
- $(\sigma, j) \vDash \neg p \iff (\sigma, j) \nvDash p$
- $(\sigma, j) \vDash p \vee q \iff (\sigma, j) \vDash p$ or $(\sigma, j) \vDash q$
- $(\sigma, j) \vDash \Box p \iff$ for all $k \geq j$, $(\sigma, k) \vDash p$
- $(\sigma, j) \vDash \Diamond p \iff$ for some $k \geq j$, $(\sigma, k) \vDash p$
- $(\sigma, j) \vDash p\, \mathcal{U}\, q \iff$ for some $k \geq j$, $(\sigma, k) \vDash q$,
  and $(\sigma, i) \vDash p$ for every i such that $j \leq i < k$
- $(\sigma, j) \vDash p\, \mathcal{W}\, q \iff (\sigma, j) \vDash p\, \mathcal{U}\, q$ or $(\sigma, j) \vDash \Box p$
- $(\sigma, j) \vDash \bigcirc p \iff (\sigma, j+1) \vDash p$

For past operators, we have:

- $(\sigma, j) \vDash \boxminus p \iff$ for all k, $0 \leq k \leq j$, $(\sigma, k) \vDash p$
- $(\sigma, j) \vDash \diamondminus p \iff$ for some k, $0 \leq k \leq j$, $(\sigma, k) \vDash p$
- $(\sigma, j) \vDash p\, \mathcal{S}\, q \iff$ for some k, $0 \leq k \leq j$, $(\sigma, k) \vDash p$
  and for every i such that $k < i \leq j$, $(\sigma, i) \vDash p$
- $(\sigma, j) \vDash p\, \mathcal{B}\, q \iff (\sigma, j) \vDash p\, \mathcal{S}\, q$ or $(\sigma, j) \vDash \boxminus p$
- $(\sigma, j) \vDash \ominus p \iff j > 0$ and $(\sigma, j-1) \vDash p$

Another useful derived operator is the *entailment* operator, defined by: $p \Rightarrow q \iff \Box(p \rightarrow q)$. For a state formula p and a state s such that p holds on s, we say that s is a *p-state*. A state formula that holds on all states is called *state-valid*.

For a temporal formula p and a position $j \geq 0$ such that $(\sigma, j) \vDash p$, we say that j is a *p-position* (in σ). If $(\sigma, 0) \vDash p$, we say that p *holds* on σ, and write it as $\sigma \vDash p$. A formula p is called *satisfiable* if it holds on some model. A formula is called (*temporally*) *valid* if it holds on all models.

Two formulas p and q are defined to be *equivalent*, written as $p \sim q$, if the formula $p \leftrightarrow q$ is valid, i.e., $\sigma \vDash p$ iff $\sigma \vDash q$, for all models σ. We adopt the convention by which a formula p that is claimed to be valid is state-valid if p is an assertion, and is temporally valid if p contains at least one temporal operator.

The formulas p and q are defined to be *congruent*, written as $p \approx q$, if the formula $\Box(p \leftrightarrow q)$ is valid, i.e., $(\sigma, j) \vDash p$ iff $(\sigma, j) \vDash q$, for all models σ and all positions $j \geq 0$. If $p \approx q$ then p can be replaced by q in any context, i.e., $\varphi(p) \approx \varphi(q)$ for any formula $\varphi(p)$ containing occurrences of p.

The notion of (temporal) validity requires that the formula holds over *all* models. Given a program P, we can restrict our attention to the set of models which correspond to computations of P, i.e., $\mathcal{C}omp(P)$. This leads to the notion of P-validity, by which a temporal formula p is *P-valid* (valid over program P) if it holds over all the computations of P. Obviously, any formula that is temporally valid is also P-valid for any program P. In a similar way, we obtain the notions of P-satisfiability and P-equivalence.

A state s that appears in some computation of P is called a *P-accessible* state. A state formula is called *P-state valid* if it holds over all P-accessible states. Obviously, any state formula that is state-valid is also P-state valid for any program P.

Again, we refer to a P-state valid formula simply as P-valid.

Appendix C

Differences with the Book

STeP was implemented based on [Manna and Pnueli, 1995]. However, due to implementation considerations and limitations, there are some differences between STeP and [Manna and Pnueli, 1995], hereafter referred to as MP95. For the convenience of those using STeP together with MP95, this appendix highlights most of the differences. These are deviations from the syntax of programs as well as differences in specification style.

C.1 SPL Programs

STeP closely follows the SPL syntax described in MP95. However, some constructs had to be translated into an ASCII equivalent. To ease transcribing programs from the book, Table C.1 presents each statement as it appears in MP95 and its representation in STeP. In the table, u refers to an arbitrary variable, $\bar{u}$ refers to a list of variables (i.e., $u_1, \ldots, u_n$), e refers to an arbitrary expression, $\bar{e}$ refers to a list of expressions, z refers to an arbitrary channel, and S refers to an arbitrary statement.

Table C.2 presents the various types of expressions as they appear in MP95 and their representation in STeP. Again, e is an arbitrary expression, u is an arbitrary variable, and $\bar{e}$ refers to a list of expressions.

The various types allowed in STeP are presented in Table C.3. In this table, e is an arbitrary expression, u is an arbitrary variable, b is an arbitrary base type, t is an arbitrary base or simple type, and T is an arbitrary type (i.e., base, simple or complex). A *range* in STeP is written $e_1..e_2$.

C.2 Control Locations in STeP

As in MP95, STeP uses control variables to indicate where the current program control resides. However, MP95 uses a single control variable π, whose value is the set of locations where control currently resides. STeP, on the other hand, uses a set of *control counters* where each control counter has its own subscript. We illustrate the differences between MP95 and STeP with an example.

Consider the program schema presented in Figure C.1, which lists the value of the control variable π (in italics) as it is used in MP95. In the figure, S_i refers to any simple

Class	In MP95	In STeP	Description
Simple	**skip**	$\texttt{skip}$	*skip*
	$u := e$	$u\ \texttt{:=}\ e$	*single assignment*
	$(\bar{u}) := (\bar{e})$	$\texttt{(}\bar{u}\texttt{)}\ \texttt{:=}\ \texttt{(}\bar{e}\texttt{)}$	*multiple assignment*
	await e	$\texttt{await}\ e$	*await*
	$z \Leftarrow e$	$z\ \texttt{<==}\ e$	*send*
	$z \Rightarrow u$	$z\ \texttt{==>}\ u$	*receive*
	request r	$\texttt{request}\ r$	*request*
	release r	$\texttt{release}\ r$	*release*
Schematic	**noncritical**	$\texttt{noncritical}$	*noncritical*
	critical	$\texttt{critical}$	*critical*
	produce u	$\texttt{produce}\ u$	*produce*
	consume u	$\texttt{consume}\ u$	*consume*
	choose u	$\texttt{choose}\ u$	*choose*
Compound	**if** e **then** S_1	$\texttt{if}\ e\ \texttt{then}\ S_1$	*1-branch cond.*
	if e **then** S_1 **else** S_2	$\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2$	*2-branch cond.*
	$S_1; S_2$	$S_1\texttt{;}\ S_2$	*concatenation*
	when e **do** S	$\texttt{when}\ e\ \texttt{do}\ S$	*when*
	S_1 **or** S_2	$S_1\ \texttt{or}\ S_2$	*selection*
	while e **do** S	$\texttt{while}\ e\ \texttt{do}\ S$	*while loop*
	repeat S **until** e	$\texttt{repeat}\ S\ \texttt{until}\ e$	*repeat*
	loop forever do S	$\texttt{loop forever do}\ S$	*loop forever*
	$S_1 \Vert S_2$	$S_1\ \Vert\ S_2$	*cooperation*
	$[\mathit{local\ decl};\ S]$	$\texttt{[}\mathit{local\ decl};\ S\texttt{]}$	*block*
	$\mathop{\mathbf{OR}}_{j=1}^{M} P[j] :: S[j]$	$\texttt{or}\ P[j:[1..M]]\texttt{::}S[j]$	*param. selection*
	$\mathop{\mathbf{OR}}_{j=1}^{M} S[j]$	$\texttt{or}\ j:[1..M]\,\texttt{.}\,S[j]$	*param. selection*
	$\mathop{\Vert}_{j=1}^{M} P[j] :: S[j]$	$\Vert\ P[j:[1..M]]\texttt{::}S[j]$	*param. cooperation*
	$\mathop{\Vert}_{j=1}^{M} S[j]$	$\Vert\ j:[1..M]\,\texttt{.}\,S[j]$	*param. cooperation*
Grouped	$\langle S \rangle$	$\texttt{<<}\ S\ \texttt{>>}$	*group*

Table C.1: The statements of SPL

In MP95	In STeP	Description
T	`true`	*true*
F	`false`	*false*
integer	*integer*	*integer value*
u	u	*variable*
$u[e]$	u`[`e`]`	*array access*
(e)	`(`e`)`	*parenthesized expression*
$(\bar{e})$	`(`$\bar{e}$`)`	*tuple of expressions*
$e_1 = e_2$	e_1 `=` e_2	*equal*
$e_1 \neq e_2$	e_1 `!=` e_2	*not equal*
$e_1 < e_2$	e_1 `<` e_2	*less than*
$e_1 > e_2$	e_1 `>` e_2	*greater than*
$e_1 \leq e_2$	e_1 `<=` e_2	*less than or equal to*
$e_1 \geq e_2$	e_1 `>=` e_2	*greater than or equal to*
$-e$	`-`e	*unary minus*
e_1 **mod** e_2	e_1 `mod` e_2	*modulo*
e_1 **div** e_2	e_1 `div` e_2	*integer division*
$e_1 + e_2$	e_1 `+` e_2	*addition*
$e_1 - e_2$	e_1 `-` e_2	*subtraction*
$e_1 \cdot e_2$	e_1 `*` e_2	*multiplication*
e_1 / e_2	e_1 `/` e_2	*rational division*
$\prod_{i=e_1}^{e_2} e$	`Prod` i`:[`e_1`..`e_2`].`e	*product*
$\sum_{i=e_1}^{e_2} e$	`Sum` i`:[`e_1`..`e_2`].`e	*summation*
$\neg e$	`~`e	*negation*
$\neg e$	`!`e	*negation*
$e_1 \wedge e_2$	e_1 `/\` e_2	*conjunction*
$e_1 \vee e_2$	e_1 `\/` e_2	*disjunction*
$e_1 \rightarrow e_2$	e_1 `-->` e_2	*implication*
$e_1 \leftrightarrow e_2$	e_1 `<-->` e_2	*equivalence*
$\forall\ u.\ e$	`Forall` $u.\ e$	*universal quantification*
$\exists\ u.\ e$	`Exists` $u.\ e$	*existential quantification*
$\exists!\ u.\ e$	`Exists!` $u.\ e$	*unique existential quant.*

Table C.2: SPL expressions

Class	In MP95	In STeP	Description

Base type	**integer**
	boolean
	rational
		`int`
		`bool`
		`rat`
			integer
			boolean
			rational

Simple type	[*range*]
	$t_1 \times \cdots \times t_k$
	{ *values* }
		`[`*range*`]`
		t_1 `*` $\cdots$ `*` t_k
		`{` *values* `}`
			range type
			tuple type
			enumeration type

Complex type	**array** [*ranges*] **of** T **channel of** b **channel** [1..] **of** b **channel** [*range*] **of** b (T)	`array [`*ranges*`] of` T `channel of` b `channel [1..] of` b `channel [`*range*`] of` b `(`T`)`	*array type* *synchronous* *unbounded asynch.* *bounded asynch.* *parenthesized type*

Table C.3: SPL types in STeP

or schematic statement (i.e., any statement that has no substatements). The STeP version of the same program schema is presented in Figure C.2, which lists all the control counters used (in italics) and their value.

The following highlights the similarities and differences between STeP's interpretation of control counters and the MP95 use of the control variable π:

1. STeP: The ordering of the labels does not affect the value of the control counters. For example, if we renamed label `l2` to be `l25`, the control counters would still have the same values. In general, the program counter *pi-j* associated with the process P_j starts at 0 and increases by 1 as control proceeds sequentially through the process.

   MP95: The ordering of the labels does affect the value of the control variable π. So, in the above example, if we rename ℓ_2 to ℓ_{25} then $\pi = \{\ell_2\}$ would change to $\pi = \{\ell_{25}\}$.

2. STeP: A cooperation statement among N processes employs N control counters. If the cooperation is a top-level cooperation statement (i.e., no entry and exit transitions) then the program has only N control counters. This is not the case in Figure C.2, where the cooperation statement has entry and exit transitions. In this case, we have two control counters, `pi1` and `pi2`, in addition to `pi0`, whose value within the cooperation statement is constant.

   MP95: There is always only one control variable, called π.

$$
P_1 :: \left[\begin{array}{ll}
\ell_0\colon\ (\pi = \{\ell_0\})\ S_0 & \\
\ell_2\colon\ (\pi = \{\ell_2\})\ S_1 & \\
\ell_3\colon\ (\pi = \{\ell_3\}) & \left[\begin{array}{l}
P2 :: \left[\begin{array}{l} m_0\colon\ (m_0 \in \pi)\ S_2 \\ m_1\colon\ (m_1 \in \pi)\ S_3 \\ m_2\colon\ (m_2 \in \pi) \end{array}\right] \\
\| \\
P3 :: \left[\begin{array}{l} k_0\colon\ (k_0 \in \pi)\ S_4 \\ k_1\colon\ (k_1 \in \pi) \end{array}\right]
\end{array}\right] \\
\ell_4\colon\ (\pi = \{\ell_4\}) & \left[\begin{array}{c}
\left[\begin{array}{l} m_4\colon\ (\pi = \{m_4\})\ S_5 \\ m_5\colon\ (\pi = \{m_5\})\ S_6 \\ m_6\colon\ (\pi = \{m_6\})\ S_7 \\ m_7\colon\ (\pi = \{m_7\})\ S_8 \\ m_8\colon\ (\pi = \{m_8\}) \end{array}\right] \\
\textbf{or} \\
\left[\begin{array}{l} k_4\colon\ (\pi = \{k_4\})\ S_9 \\ k_5\colon\ (\pi = \{k_5\})\ S_{10} \\ k_6\colon\ (\pi = \{k_6\})\ S_{11} \\ k_7\colon\ (\pi = \{k_7\}) \end{array}\right] \\
\textbf{or} \\
\left[\begin{array}{l} j_4\colon\ (\pi = \{j_4\})\ S_{12} \\ j_5\colon\ (\pi = \{j_5\}) \end{array}\right]
\end{array}\right] \\
\ell_5\colon\ (\pi = \{\ell_5\})\ S_{13} & \\
\ell_6\colon\ (\pi = \{\ell_6\}) &
\end{array}\right]
$$

Figure C.1: Program schema illustrating MP95's use of the control variable π

```
P1::  [ l0:  (pi0=0) S0;
        l2:  (pi0=1) S1;
        l3:  [(pi0=2)
               P2::  [ m0:  (pi0=3, pi1=0) S2;
                       m1:  (pi0=3, pi1=1) S3;
                       m2:  (pi0=3, pi1=2)
                      ]
               ||
               P3::  [ k0:  (pi0=3, pi2=0) S4;
                       k1:  (pi0=3, pi2=1)
                      ]
         ];
        l4:  [(pi0=4)
               [ m4:  (pi0=4) S5;
                 m5:  (pi0=5) S6;
                 m6:  (pi0=6) S7;
                 m7:  (pi0=7) S8;
                 m8:  (pi0=10)
                 ]
               or
               [ k4:  (pi0=4) S9;
                 k5:  (pi0=8) S10;
                 k6:  (pi0=9) S11;
                 k7:  (pi0=10)
                 ]
               or
               [ j4:  (pi0=4) S12;
                 j5:  (pi0=10)
                 ]
         ];
        l5:  (pi0=10) S13;
        l6:  (pi0=11)
```

Figure C.2: Program schema illustrating STeP's use of control counters

3. In both STeP and MP95, the same rules govern the notion of label equivalence.

   STeP: This is achieved by assuming label-equivalent locations the same control-counter values.

   MP95: This is achieved by using the notational convenience that a label ℓ_i really refers to its (label) equivalence class $[\ell_i]$. Consequently, $\ell_4 \sim m_4 \sim k_4 \sim j_4$ and $\ell_5 \sim m_8 \sim k_7 \sim j_5$ (see page 13-14 in MP95).

Location equivalences

In STeP, a **loop forever do** statement does not generate a transition, so the pre-location and post-location of this statement are equivalent. However, both locations are entered as location variables in the symbol table and can thus be referred to in the specification.

C.3 Type Declarations

In several programs MP95 uses multiple lines of declarations as follows:

in	x, y, z:	**integer**
	a, b, c:	**bool**
local	d, e, f:	**integer**
	g, h, i:	**bool**

In STeP each line of declarations must be preceded by the mode of the variable, so the above has to be entered as:

```
in     x,y,z: integer
in     a,b,c: bool
local d,e,f: integer
local g,h,i: bool
```

C.4 Initialization

Array Initialization

In several programs MP95 uses the following construct to initialize an array:

in	M:	**integer**	**where** $M \geq 1$
	y:	**array**$[1..M]$ **of integer**	**where** $y = \mathrm{T}$

In STeP this causes an error due to the incompatible types of `y` and `T`. In STeP this has to be written as:

```
in M: int where M >= 1
in x: array[1..M] of int where Forall i:[1..M] . y[i] = true
```

Channel Initialization

In MP95 a channel may be initialized as the empty channel with

$$send = \Lambda$$

In STeP this is accomplished with

$$\texttt{length(send)} \;=\; \texttt{0}$$

In MP95 a channel may be initialized with values as follows:

$$ack = \underbrace{[1 \ldots 1]}_{N}$$

STeP does not provide a construct to initialize a channel in the **where** clause. If the initial values matter, the channel has to be initialized explicitly within the program with, for example, a `while` statement. If the initial values do not matter, which is the case for program PROD-CONS-C of Figure 1.18 in MP95, the channel may be initialized with the following `where` clause:

```
in      N :int where N > 0
local   send, ack: channel [1..] of int where length(ack) = N /\
                                              length(send) = 0
```

C.5 Parameterized Programs

In MP95 (page 174), terms such as N_3 are often used to denote the number of processes residing at a particular location. This abbreviation is not available in STeP, so there are two ways to express such formulas: In a formula of the form

$$N_3 \leq 1$$

(i.e., the number of processes at location 3 is less than 1), the recommended STeP representation is the following:

```
  Forall i:[1..N] . Forall j:[1..N] . (l3[i] /\ l3[j] --> i=j)
```

This may be generalized to small numbers greater than 1 using disjunction in the consequent. When this representation is not possible, e.g., in a formula of the form

$$N_3 + y = z$$

you can use `Sum` as follows:

$$(\texttt{Sum i}:[\texttt{1..N}]\ .\ \texttt{l3[i]}) \;+\; \texttt{y} \;=\; \texttt{z}$$

We discourage the use of `Sum` for the simpler cases because reasoning with `Sum` is more difficult than reasoning with quantifiers only.

Appendix D

Distribution and Installation

Installation

STeP is available for platforms that are supported by Standard ML of New Jersey version 1.08, including:

- Sun SPARC workstations (SunOS and Solaris)
- DEC Alpha workstations (OSF1)

You should have copied the distribution file specific to your machine. The only difference is in the binaries included with each.

Files associated with running STeP are located in the `bin` directory. The executable files are found in the `.bin` and `.heap` directories, and accessed through the `STeP`, `polyserv`, and `model_checker` scripts. These scripts determine the machine type and operating system, using the `.arch-n-opsys` command, and then invoke the appropriate binary file (from the `.run` or `.bin` directory) on the given arguments. Thus, none of these files should be moved or renamed.

You can edit the `STeP` script to set the `STEP_DIR` environment variable once and for all.

Note: These scripts use the `/bin/sh` shell; if this shell is not found, you should edit them and change the first line to indicate the appropriate location (actually, `ksh` is preferred).

STeP Distribution

When the distribution has been untarred it will have created three directories:

- The `bin` directory containing binaries `step`, `model_checker`, and `polyserv` for executing STeP, and from within it the model checker and utilities for generating polyhedral invariants.
- The `doc` directory. It contains:
    1. The STeP manual (postscript).

2. The STeP Technical Report (postscript), STAN-CS-TR-94-1518, Stanford Computer Science Department, June 1994. This report complements the STeP manual by giving a high-level description of the system and its goals and some extra technical details.
3. The list of known problems and bugs `known-bugs.txt`.
4. A quick installation guide `installation.txt`.
5. The text of the licence agreement `license-agreement.txt`.
6. The `WWW` subdirectory, which contains the help pages.

- The `examples` directory containing a selection of simple example programs and specifications that can be verified using STeP.

New releases will include updated help pages; please let us know how those pages, and this manual, can be improved.

D.1 Example Programs

The examples directory adheres to the default filename extensions used by the File Browser. These conventions are shown in Table D.1.

Filename extension	contents of file
`.spl`	SPL program
`.trans`	transition system
`.cts`	clocked transition system
`.spec`	specification
`.search`	proof search
`.diagram`	verification diagram
`.tactic`	tactic
`.mclog`	modelchecker logfile

Table D.1: Filename conventions

The example programs and specification files listed in Tables D.2 and D.3 are included in the distribution. They correspond to simple examples and exercises from [Manna and Pnueli, 1995]. In most cases the user will have to complete the specifications for exercises in order to succeed deductive verification. Thus, possible answers to the exercises are not included in the distribution. The directory mux-fisher contains a simple real-time protocol implemented with Clocked Transition Systems.

	directory	program file	specification files	other files
ADD-TWO	add-two	add-two.spl	yge0.spec l1x=2.spec	
ANY-NAT	any-nat	any-nat.spl		
BINOM BINOM-C	binom binom	binom.spl binom-c.spl	part-corr.spec str-part-corr.spec part-corr.spec str-part-corr.spec	
CUBE	cube	cube.spl	part-corr.spec	
DIFF-INC	diff-inc	diff-inc.spl	psi.spec	
DINE DINE-CONTR DINE-EXCL	dine dine-contr dine-excl	dine.spl dine-contr.spl dine-excl.spl		
DOUBLE	double	double.spl		
EUCLID	euclid	euclid.trans		
FACT	fact	fact.spl	part-corr.spec	
GCD GCDM	gcd gcdm	gcd.spl gcdm.spl	part-corr.spec part-corr.spec	
INC	inc	inc.spl	psi.spec	psi.search
LOOP LOOP+	loop loop	loop.trans loop+.trans	phi3.spec psi3.spec phi3.spec psi3.spec	

Table D.2: Example programs and specifications

	directory	program file	specification files	other files
MAX-ARRAY	max-array	max-array.spl	part-corr.spec	phi3.search phi4.search phi5.search
MPX-SEM	mpx-sem	mpx-sem.spl		
MUX-BAK-A MUX-BAK-C MUX-BAK-D	mux-bak-a mux-bak-c mux-bak-d	mux-bak-a.spl mux-bak-c.spl mux-bak-d.spl	mux.spec	
MUX-DEK MUX-DEK-A MUX-DEK-B	mux-dek mux-dek-a mux-dek-b	mux-dek.spl mux-dek-a.spl mux-dek-b.spl	mux.spec mux.spec mux.spec	
MUX-FISHER	mux-fisher	mux-fisher.cts	mux-fisher.spec	
MUX-PET1	mux-pet1	mux-pet1.spl	mux.spec mux-vd.spec simple-prec.spec 1bo.spec 1bo-vd.spec	mux.tactic mux.diagram 1bo.diagram
MUX-PET2	mux-pet2	mux-pet2.spl	mux.spec mux-vd.spec 1bo-back.spec 1bo.spec 1bo-vd.spec	mux.tactic mux.diagram 1bo.diagram
MUX-PET3	mux-pet3	mux-pet3.spl	mux.spec accessibility.spec 1bo.spec	mux.tactic accessibility.diagram
MUX-SEM	mux-sem	mux-sem.spl	mux.spec yge0.spec	mux.search state-partition.diagram
MUX-VAL3 PROD-CONS PROD-CONS-C	mux-val3 prod-cons prod-cons	mux-val3.spl prod-cons.spl prod-cons-c.spl	mux.spec no-overflow.spec no-overflow-c.spec	mux.diagram mux-compound.diagram
PROD-CONS-SV	prod-cons-sv	prod-cons-sv.spl	part-corr.spec	
RES-ALLOC	res-alloc	res-alloc.spl		
RES-ND	res-nd	res-nd.spl	mux.spec	
RES3	res3	res3.spl	mux.spec	
SEM3	sem3	sem3.spl	mux.spec	
SQRT SQRT-C	sqrt sqrt	sqrt.spl sqrt-c.spl	phi-hat.spec str-part-corr.spec	

Table D.3: More example programs and specifications.

Appendix E

STeP Environment Variables

A number of UNIX environment variables can be set before running STeP. The only mandatory variables are `STEP_DIR`, which indicates the directory where STeP resides, and the `DISPLAY` environment variable, which should be set to the IP-address of your screen. If these variables are not set, you will not be able to run STeP.

To set a UNIX environment variable, enter

`setenv` *variable-name new-value*

from the UNIX prompt. You can also add these commands to your `.cshrc` file, to make them the defaults when you log in.

General variables

- `STEP_DIR`: Local STeP installation directory.
  Default: `/local/step`.
- `STEP_BROWSER`: The location of the `WWW/http` browser executable used for on-line help.
  Default `/usr/local/bin/mosaic`.
- `STEP_AUTO_SIMPLIFY`: If this flag is set to anything other than "OFF," this will cause STeP to start with the automatic simplification flag (see Section 4.1) ON.
- `STEP_SHOW_LOADED`: Set this to "OFF," if you do not want loaded programs and transition systems to be automatically displayed.

Verification Diagram settings

The following values can be increased when very large diagrams need to be drawn.

- `STEP_DIAGRAM_WIDTH`: Maximum width, in pixels, of the verification diagram canvas.
  Default: 1200.
- `STEP_DIAGRAM_HEIGHT`: Maximum height, in pixels, of the verification diagram canvas.
  Default: 1200.

BDD-package settings

These variables control the amount of memory used by the BDD's.

- `STEP_BDD_NODES`: Maximum number of BDD nodes that are kept around by the BDD package. After this maximum is reached, the BDD packages has to be reset for BDD-based operations to succeed.
  Default: 100,000.
- `STEP_BDD_SPLIT`: Maximum number of subgoals that are generated by the BDD-split rule in the Top-level Prover and Interactive Prover. If this maximum is reached, the BDD-split rule fails. (BDD split can generate exponentially many subgoals in the worst case.)
  Default: 1,000.
- `STEP_BDD_CACHE`: Size of the BDD result cache, used for efficient BDD operations.
  Default: 32,000.

Modelchecker and polyhedral invariant resources

The following variables limit the space and time used by the external Model Checker and polyhedral invariant packages. The `_MC_` variables control the Model Checker, and the `_POLY_` variables control the polyhedral invariant generator. Both of them will fail if any of the limits is exceeded.

- `STEP_MC_SPACE`, `STEP_POLY_SPACE`: Maximum amount of memory, in Megabytes.
  Default: unlimited.
- `STEP_MC_TIME`, `STEP_POLY_TIME`: Maximum amount of user time, in minutes.
  Default: unlimited.
- `STEP_MC_CPU`, `STEP_POLY_CPU`: Maximum amount of CPU time, in minutes.
  Default: unlimited.

Finally, remember that the *interrupt button* can be used to interrupt lengthy computations in STeP.

Bibliography

[Bjørner *et al.*, 1995] N. Bjørner, I.A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. In *First Intl. Conference on Principles and Practice of Constraint Programming*, volume 976 of *LNCS*, pages 589–623, Cassis, France, September 1995. Springer-Verlag.

[Bledsoe, 1975] W. W. Bledsoe. A new method for proving certain Presburger formulas. In *Proc. of the* 4^{th} *International Joint Conference on Artificial Intelligence*, pages 15–21, Tbilisi, Georgia, USSR, September 1975.

[Boyer and Moore, 1988] R.S. Boyer and J S. Moore. Integrating decision procedures into heuristic theorem provers: A case study with linear arithmetic. *Machine Intelligence*, 11:83–124, 1988.

[Bryant, 1986] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[Bryant, 1992] R.E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[Dershowitz and Jouannaud, 1990] N. Dershowitz and J-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 6, pages 244–320. Elsevier Science Publishers, 1990.

[Gallier, 1987] J.H. Gallier. *Logic for Computer Science—Foundations for Automatic Theorem Proving.* Wiley, New York, 1987.

[Harel, 1987] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comp. Prog.*, 8:231–274, 1987.

[Hojati *et al.*, 1993] R. Hojati, V. Singhal, and R.K. Brayton. Edge-Street/Edge-Rabin automata environment for formal verification using language containment. SRC report, University of California, Berkeley, 1993.

[Kesten *et al.*, 1993] Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli. A decision algorithm for full propositional temporal logic. In C. Courcoubetis, editor, *Proc.* 5^{th} *Intl. Conference on Computer Aided Verification*, volume 697 of *LNCS*, pages 97–109. Springer-Verlag, 1993.

[Kesten *et al.*, 1996] Y. Kesten, Z. Manna, and A. Pnueli. Verifying clocked transition systems. In *Hybrid Systems III*, LNCS. Springer-Verlag, 1996. To appear.

[Manna and Pnueli, 1991] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1991.

[Manna and Pnueli, 1994] Z. Manna and A. Pnueli. Temporal verification diagrams. In *Proc. Int. Symp. on Theoretical Aspects of Computer Software*, volume 789 of *LNCS*, pages 726–765. Springer-Verlag, 1994.

[Manna and Pnueli, 1995] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[Manna and Waldinger, 1993] Z. Manna and R. Waldinger. *The Deductive Foundations of Computer Programming.* Addison-Wesley, Reading, MA, 1993.

[McCune, 1992] W.W. McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *J. Automated Reasoning*, 9(2):147–67, October 1992.

[Nelson and Oppen, 1980] G. Nelson and D.C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, April 1980.

[Shostak, 1979] R.E. Shostak. A practical decision procedure for arithmetic with function symbols. *J. ACM*, 26(2):351–360, April 1979.

[Stickel, 1989] M.E. Stickel. The path indexing method for indexing terms. Technical Note 473, SRI International, Menlo Park, CA, October 1989.

[Zhang, 1993] H. Zhang. Contextual rewriting in automated reasoning. Technical Report Technical Report 93-07, Department of Computer Science, University of Iowa, August 1993.

Index