2014-04-23

Automated Theorem Proving using the TPTP Process Instruction Language

Muhammad Nassar
University of Miami, muhammad.m.mansour@gmail.com

UNIVERSITY OF MIAMI

AUTOMATED THEOREM PROVING USING THE TPTP PROCESS INSTRUCTION LANGUAGE

By

Muhammad Nassar

A THESIS

Submitted to the Faculty
of the University of Miami
in partial fulfillment of the requirements for
the degree of Master of Science

Coral Gables, Florida

May 2014

UNIVERSITY OF MIAMI

A thesis submitted in partial fulfillment of
the requirements for the degree of
Master of Science

AUTOMATED THEOREM PROVING USING THE TPTP PROCESS INSTRUCTION LANGUAGE

Muhammad Nassar

Approved:

Geoff Sutcliffe, Ph.D. Associate Professor of Computer Science	Victor Milenkovic, Ph.D. Professor of Computer Science
Burton Rosenberg, Ph.D. Associate Professor of Computer Science	M. Brian Blake, Ph.D. Dean of the Graduate School

NASSAR, MUHAMMAD (M.S., Computer Science)
Automated Theorem Proving using the TPTP Process Instruction Language (May 2014)

Abstract of a thesis at the University of Miami.

Thesis supervised by Professor Geoff Sutcliffe.
No. of pages in text. (53)

The TPTP (Thousands of Problems for Theorem Provers) World is a well established infrastructure for Automated Theorem Proving (ATP). In the context of the TPTP World, a command language was needed to make possible the easy manipulation of logical formulae and provide better control over the use of ATP systems. The TPTP Process Instruction (TPI) language provides such capabilities. It is used to input, output and organize logical formulae, and control the execution of ATP systems. This thesis presents the work done for building and testing a shell interpreter for the TPI language. The thesis is divided into five parts. The first part provides a review of ATP, ATP systems, and the TPTP World. The second part provides an overview of the related work and presents the TPI language. The third part introduces the ATP process and discusses its benefits. The fourth part presents the shell interpreter that has been developed for the TPI language, describes its implementation and provides examples of how it can be used in theorem proving. The last part discusses generic control of ATP systems and demonstrates how such control is achieved through an extension to the TPI language.

Acknowledgements

I would like to sincerely thank my advisor Dr. Geoff Sutcliffe for his help, support and valuable guidance throughout my work on the thesis. I would also like to thank Dr. Burton Rosenberg and Dr. Victor Milenkovic for being part of my committee. I also thank the Department of Computer Science for providing me with the great opportunity of being here and being able to finish my studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 What is Automated Theorem Proving?

Automated Theorem Proving (ATP) is a form of automated reasoning that is concerned with proving that a statement, called the *conjecture*, is a *logical consequence* of a group of statements, called the *axioms* [1, 2, 3]. In the context of ATP, a *problem* is a group of axioms with a conjecture, expressed in some logical form and stored in a file. A variety of logics, e.g. first order logic, can be used to write the logical formulae. Using logic to express formulae allows for a precise and clear description of a problem, and doesn't allow for ambiguity, as is often the case with natural languages like English. ATP systems (discussed in Section 1.2) are computer programs capable of solving ATP problems, i.e., proving that the conjecture is a logical consequence - or a *Theorem* - of the axioms.

ATP is utilized in many disciplines, such as hardware verification, mathematics, and software verification [4]. ATP is used heavily in industry for the purpose of

hardware verification. Microprocessor manufacturers have been using ATP to verify the correctness of the floating point arithmetic on their chips. For example, the *ACL2* ATP system [5] has been used to verify the correctness of the floating point division for AMD's AMD5K86 microprocessor. One of the big successes in the field of mathematics was the ability of the *EQP* ATP system to solve the Robbins problem [6]. The Robbins problem states that a certain group of axioms is the basis of Boolean algebra, but nobody was able to provide a formal proof for the problem until EQP succeeded in producing a proof in 1996. One of the successful efforts in the field of software verification is *The KeY Project* [7]. It is a software development tool that integrates design, implementation, specification and verification of object-oriented software. At its core, the tool uses an innovative ATP system with an easy to use graphical user interface.

1.2 ATP Systems

An ATP system is a computer program capable of solving ATP problems. The input to an ATP system is an ATP problem, and the system attempts to find a proof of the conjecture from the axioms. If a proof is found, the system might provide a detailed output explaining how the conjecture was proved from the axioms. If the system cannot prove the conjecture, the user can try to prove an intermediate result, or go over the problem file to make sure the logical formulae correctly describe the problem. An ATP system is said to be complete for a given logic if it can solve any problem written in that logic, if a solution to that problem does exist. One of the

most powerful ATP systems that is complete for first order logic is `E` [8]. `iProver` [9] and `Vampire` [10] are two other examples of powerful ATP systems that are complete for first order logic. Example usages of `E` and `iProver` are presented in later chapters.

1.3 The TPTP World and the TPI Language

The TPTP (Thousands of Problems for Theorem Provers) World is a well-established infrastructure for Automated Theorem Proving (ATP) [11]. It hosts a comprehensive library of ATP test problems, supplies references and information about each problem, and provides utilities that facilitate problem manipulation and ATP systems' usage (see `http://www.tptp.org`).

The *TPTP language* provides a way of writing logical formulae. A TPTP language formula has the form:

```
language(name, role, formula, [source, [useful-info]])
```
[12]

An example first order formula is:

```
fof(pel55_3,axiom,
      ( !  [X] :
      ( lives(X)
      => ( X = agatha
      | X = butler
      | X = charles ) ) ),
      file('PUZ001+1.ax'),
      description('Who Killed Aunt Agatha')).
```

The TPTP language also provides a way for the output of different ATP systems to be standard. It does so through the use of the SZS ontologies [13]. The ontologies provide values that an ATP system can use to precisely report the status of its execution, e.g.

`SZS status Theorem`, `SZS status Unsatisfiable`, and so on. The phrase `"SZS status"` is used in later sections to refer to the output of an ATP system.

In the context of the TPTP World, a command language was needed to make possible the easy manipulation of logical formulae and provide better control over the use of ATP systems. The TPTP Process Instruction (TPI) language provides such capabilities [12]. It is used to input, output, and organize logical formulae, and to control the execution of ATP systems. This thesis presents the work done for building and testing a shell interpreter for the TPI language, demonstrates the use of the TPI language and the interpreter in theorem proving, discusses generic control of ATP systems, and shows how such control is achieved through an extension to the TPI language. Chapter 2 provides an overview of the related work and presents the TPI language as described in [12]. Chapter 3 introduces the ATP process and discusses its benefits. Chapter 4 presents the shell interpreter that has been developed for the TPI language, describes its implementation and provides examples of how the interpreter can be used in theorem proving. Chapter 5 discusses generic control of ATP systems and describes how the TPI language is extended to accommodate such control. Chapter 6 provides a conclusion of the thesis and discusses future work.

Chapter 2

Previous Work

2.1 ATP Execution and Control

There exist several ways in which an ATP system's execution can be controlled. Some aspects of an ATP system's execution, e.g., the type of reasoning used, the maximum allowed execution time, the output format and structure, are controllable. A direct approach to controlling an ATP system is to provide command line flags that users can specify, so that the system behaves in a way that reflects the values of the flags provided. Many ATP systems provide command line flags to control their execution. `E`, `iProver` and `Vampire` (mentioned in Chapter 1) provide such flags. The following example demonstrates a call to `E` with a command line flag that sets an execution time limit of 300 seconds:

```
Systems/E---1.8/eprover --cpu-limit=300 Problem1.p
```

Another approach to controlling an ATP system's execution is for the system to support specific commands that control its processing. Such commands are provided

in the problem file as part of the input to the ATP system. They are different from the command line flags mentioned earlier, in the sense that the latter are specified while executing the system from the command line. One well known ATP system that utilizes this approach is `Otter` [14]. `Otter`'s input language is used to write logical formulae, and provides commands to set and clear flags that control the system's behavior.

A third approach, which is used by the TPI language, is to provide a command language designed to be used in conjunction with an existing language for writing logical formulae. The Satisfiability Modulo Theories (SMT) command language [15] also utilizes this approach. Command languages are covered in more detail in Section 2.2.

2.2 Command Languages

Command languages are simple yet powerful tools that are used for many computing purposes today. They provide the ability to programmatically execute a series of commands in a controlled fashion, rather than requiring the user to run each command individually. One of most powerful and widely used command languages is the *Unix shell*. A shell is used in Unix-like operating systems, e.g., the various Linux distributions, to provide an interface for interacting with the operating system. A *shell script* is a script that contains commands and programming structures to be executed by the shell. A shell script can be used for a variety of purposes like manipulating files and folders, running programs, and writing text to the screen or files.

The following example is a simple `tcsh` shell script that initializes a counter, prints its value, increments it, and loops till a predefined value is reached.

```
1 echo "Simple Counter Example"
2 set Counter = 1
3 while ($Counter < 11)
4       echo "Current value of the counter is:  $Counter"
5       @ Counter = $Counter + 1
6       if("$Counter" == 6 ) then
7            set Counter = 7
8       endif
9 endif
```

Line 1 outputs the specified text to the screen. Line 2 initializes and a counter variable to 1. A loop is then started that keeps iterating as long as the counter value is less than 11. Line 4 outputs the value of the counter to the screen, after which line 5 increments the counter value by 1. Lines 6 through 8 skip the counter value of 6, and thus preventing the value of 6 from being printed on the screen.

In the context of ATP, one of the most recent and widely used command languages for interacting with ATP systems is the SMT command language. It interacts with SMT solvers using a textual interface that can be used for adding and deleting formulae, checking their satisfiability, and inspecting their models or their unsatisfiability proofs if no models exist [15]. The following simple example declares a Boolean value `q` and asks whether `q` and its inverse are satisfiable [16]:

```
> (set-logic QF_UF)
success
> (declare-fun q () Bool)
success
> (assert (and q (not q)))
success
> (check-sat)
unsat
```

```
> (exit)
success
```

Interactive Theorem Proving (ITP) systems provide rich, system-specific command languages that help in the proof building process. Matita [17] and Isabelle [18] are two examples of ITP systems. The TPI language introduced in Section 1.3 - and discussed in detail in Section 2.3 - provides a generic way of manipulating logical formulae and executing ATP systems. It is not specific to any ATP system, and can be used to control and co-ordinate the execution of multiple ATP systems simultaneously.

2.3 The TPI Language

In the context of ATP, most input languages, including the TPTP language, are used to write the logical formulae that are the input to ATP systems. Such languages do not provide support for manipulating the formulae, interacting with ATP systems, or handling non-logical data. The TPTP Process Instruction (TPI) language provides commands to control the input, output and organization of logical formulae, handle the execution of ATP systems, and manage non-logical data. The TPI language was first introduced in [12]. In its original form, the TPI language was designed to be used alongside the current TPTP language logical forms, e.g. FOF and CNF. The TPI commands maintain the same form as the TPTP language formulae introduced in Section 1.3. A TPI command is in the form:

```
tpi(name, command, command_details, [source, [useful-info]])
```

The TPI commands are discussed in detail in Section 2.4. The following simple

example demonstrates how the TPI language can be used with the existing TPTP language formulae:

```
1 tpi(1,input,'Axioms/SYN001+1.ax').
2 fof(another,axiom,pp => qq).
3 fof(a,conjecture,qq).
4 tpi(2,execute,'SZS_Status' = 'E---1.8/eprover --auto --cpu-limit=300
  --tstp-format %s')
```

Line 1 adds formulae from the specified problem file using the TPI `input` command. Lines 2 and 3 add an axiom and a conjecture. Line 4 uses the TPI `execute` command to run `E`, to try to prove that the conjecture is a *Theorem* of the axioms. A full interpreter for the TPI command language in the form described above was developed in C, and is available as part of the TPTP World.

This thesis presents the work done creating a TPI language interpreter that can be run within a Unix shell. This variant utilizes the same TPI language commands introduced in [12], but adapts the syntax of the language to be able to run the TPI commands in a Unix shell. The modified formula form is as follows:

```
tpi command_name command_details
```

In the modified form, the previous example becomes:

```
1 tpi input 'Axioms/SYN001+1.ax'
2 tpi input_formula 'fof(another,axiom,pp => qq)'
3 tpi input_formula 'fof(a,conjecture,qq)'
4 tpi execute 'SZS_Status' = 'E---1.8/eprover --auto --cpu-limit=300
  --tstp-format %s'
```

A detailed example of the usage of the TPI language commands in this adapted form is presented in Section 2.4.7. The full details of the interpreter's implementation are provided in Section 4.1.

2.4 The TPI Language Commands

This section describes the TPI language commands and their use. The commands are grouped according to the type of function they perform.

2.4.1 Input and Output

`input_formula, input, output, delete`

Command details:

```
tpi input_formula formula
tpi input [group_name = ] file_name
tpi output file_name [= group_name]
tpi delete formula_name
```

Action:

`input_formula` adds a single logical formula to the pool of available formulae. `input` adds all the logical formulae in a named file. `input` can also add the formulae to a formula group (formula groups are discussed in Section 2.4.2). `output` outputs all the logical formulae, or those in a specified group, to an output file. If `stdout` is specified as the output file name, the formulae are output to the screen. `delete` deletes a logical formula with a specified name from the available formula pool.

Examples:

```
tpi input_formula 'fof(f1,axiom,(tall(matt)))'
tpi input 'Problems/PUZ001-1.p'
tpi input 'NewGroup' = 'Problems/PUZ001-1.p'
tpi output 'Output_File.txt'
tpi output 'Output_File.txt' = 'NewGroup'
tpi delete 'f1'
```

2.4.2 Logical Formula Grouping

`start_group, end_group, delete_group`

Command details:

```
tpi start_group group_name
tpi end_group group_name
tpi delete_group group_name
```

Action:

Logical formulae can be organized into groups. `start_group` marks the start of a named group of logical formulae. All logical formulae that are subsequently added using the `input` or `input_formula` commands belong to this group, until a corresponding `end_group` command is encountered. Formula groups can be nested. In this case, the formulae belonging to a nested group would also belong to its outer group(s). `delete_group` deletes all the logical formulae that belong to a group. All formulae belong to the general group `tpi`, and the two groups `tpi_premises` and `tpi_conjectures` are provided as the default groups for the axioms and conjectures.

Examples:

```
tpi start_group 'Arithmatic_Axioms'
tpi end_group 'Arithmatic_Axioms'
tpi delete_group 'Arithmatic_Axioms'
```

2.4.3 Logical Formula Manipulation

`activate, deactivate, activate_group, deactivate_group, set_role`

Command details:

```
tpi activate formula_name
tpi deactivate formula_name
```

```
tpi activate_group group_name
tpi deactivate_group group_name
tpi set_role formula_name = value
```

Action:

Logical formulae can either be active or inactive. Active formulae are available for output and execution, while inactive formulae would still be available in the formula pool, but they cannot be used for output or execution. `deactivate` causes a named formula to become inactive. `activate` causes a named formula to become active. `deactivate_group` deactivates all the formulae in a named group and `activate_group` activates all the formulae in a named group. `set_role` sets the role [11] of a named formula to a given value, e.g. `axiom`, `conjecture`, `lemma`.

Examples:

```
tpi activate 'Formula1'
tpi deactivate 'Formula1'
tpi activate_group 'Arithmatic_Axioms'
tpi deactivate_group 'Arithmatic_Axioms'
tpi set_role 'Formula1' = 'conjecture'
```

2.4.4 Environment Variable Manipulation

`setenv, unsetenv, waitenv`

Command details:

```
tpi setenv variable_name = variable_value
tpi unsetenv variable_name
tpi waitenv variables
```

Action:

These three commands handle environment variables. The variables belong to the

interpreter's process. `setenv` creates a new environment variable - if it does not exist - with a given name and value. If the environment variable already exists, its value is set to the given value. `unsetenv` deletes an environment variable with a given name. `waitenv` waits for an environment variable(s) with a given name(s) to become set. The TPI language also supports the use of the term `$getenv()` with any of the TPI language commands. `$getenv()` is replaced by the value of a specified environment variable. For example, `$getenv('SZS_Status')` will be replaced by the value of the environment variable "SZS_Status".

Examples:

```
tpi setenv 'STATUS' = 'Satisfiable'
tpi unsetenv 'STATUS'
tpi waitenv 'STATUS'
tpi waitenv 'STATUS' | 'RESULT'
```

2.4.5 Execution

`execute, execute_async, filter, generate`

Command details:

```
tpi execute [EnvVar =] command
tpi execute_async [EnvVar =] command
tpi filter [EnvVar =] command
tpi generate [EnvVar =] command
```

Action:

These commands control the execution of ATP systems, or any other command the user would like to execute. `execute` runs a specified ATP system in the foreground. An environment variable can be provided to capture the SZS status of

the completed execution, e.g., *Satisfiable* or *Theorem*. The logical formulae to be processed by the ATP system are selected using the `$getgroups()` term(s). The arguments of the `$getgroups()` term are the names of formula groups. For example, a `$getgroups('Axioms')` term will cause all the formulae that belong to the group named `Axioms` to be selected for execution, and `$getgroups(Axioms,Conjecture)` will cause all the formulae that belong to the groups named `Axioms` and `Conjecture` to be selected for execution. `execute_async` runs like `execute` except that the ATP system is run in the background. This is useful in cases where the user wants to do other tasks while waiting for the ATP system(s) to finish its execution. `filter` runs like `execute`, then `delete`s the logical formulae specified by the `$getgroups()` term(s), and `input`s the logical formulae output by the ATP system. `generate` runs like `execute`, and `input`s the logical formulae output by the ATP system.

Examples:

```
tpi execute 'eprover --cpu-limit=30 $getgroups(tpi)'
tpi execute_async 'SZS_ASYNC' = 'eprover --cpu-limit=30
   $getgroups(Axioms,Conjecture)'
tpi filter 'SInESelect---1.8/sine --mode axiom_selection $getgroups(tpi)'
tpi generate 'STATUS' = 'FormulaeGenerator'
```

2.4.6 Utility Commands

`mktemp, write, assert, exit`

Command details:

```
tpi mktemp variable_name
tpi write output_data
tpi assert term = term | term != term
tpi exit (<integer_exit_code>)
```

Action:

`mktemp` creates a new temporary file and stores its name in the specified environment variable. `write` outputs data to `stdout`. `assert` checks the equality or inequality of two terms. If the assertion fails, a message is provided and the `exit` command is executed. `exit` deletes any files that were created using `mktemp` commands, and terminates any remaining processes that were started using `execute_async` commands.

Examples:

```
tpi mktemp 'New_File'
tpi write 'Status is' & '$getenv(SZS_Status)'
tpi assert '$getenv(SZS_Status)' = 'Satisfiable'
tpi exit
tpi exit(1)
```

2.4.7 An Example

The following example demonstrates a simple use of the interpreter:

```
1 tpi input_formula 'fof(ax,axiom,r).'
2 tpi input_formula 'fof(ax2,axiom,r => s).'
3 tpi execute 'SZS' = 'iprover 30 $getgroups(tpi_premises)'
4 tpi write 'Status:'  & $getenv('SZS')
5 tpi assert '$getenv('SZS')' = 'Satisfiable'
6 tpi input_formula 'fof(conj,conjecture,s).'
7 tpi execute 'SZS' = eprover -s --cpu-limit=30 --tstp-format $getgroups(tpi)'
8 tpi write 'Proof status:'  & $getenv('SZS')
9 tpi exit
```

Example 1: A simple use of the TPI Interpreter

Commands 1 and 2 add two axioms. The `iProver` ATP system is then invoked by command 3 to check whether the axioms are satisfiable or not. Command 4 writes the

SZS status returned by `iProver`. Command 5 asserts that the axioms are *Satisfiable* – if they are not an error code is returned. A conjecture is added by command 6, after which the `E` ATP system is invoked by command 7 to try to prove that the conjecture is a *Theorem* of the axioms. Command 8 writes the proof status and command 9 ends execution.

Chapter 3

The ATP Process

3.1 Overview

Automated Theorem Proving is primarily concerned with finding a proof that a certain conjecture is a *Theorem* of the axioms. The easiest way to do that is to pass the logical formulae representing the axioms and the conjecture directly to a theorem proving ATP system. While this process may be straightforward, it does have drawbacks that should be taken into account. Some of these drawbacks are:

- The logical formulae representing the axioms and the conjecture might contain syntax errors.
- The axioms might be *Unsatisfiable*, i.e., there is no model for the axioms. Establishing that the axioms are *Satisfiable* is a crucial step in the process, because a proof that the conjecture is a *Theorem* of an *Unsatisfiable* group of axioms can be found vacuously – this is typically not the user's intention.

- The user is not provided with further information if the status of system's execution is unknown, i.e, the given system couldn't decide whether or not the conjecture is a *Theorem* of the axioms.

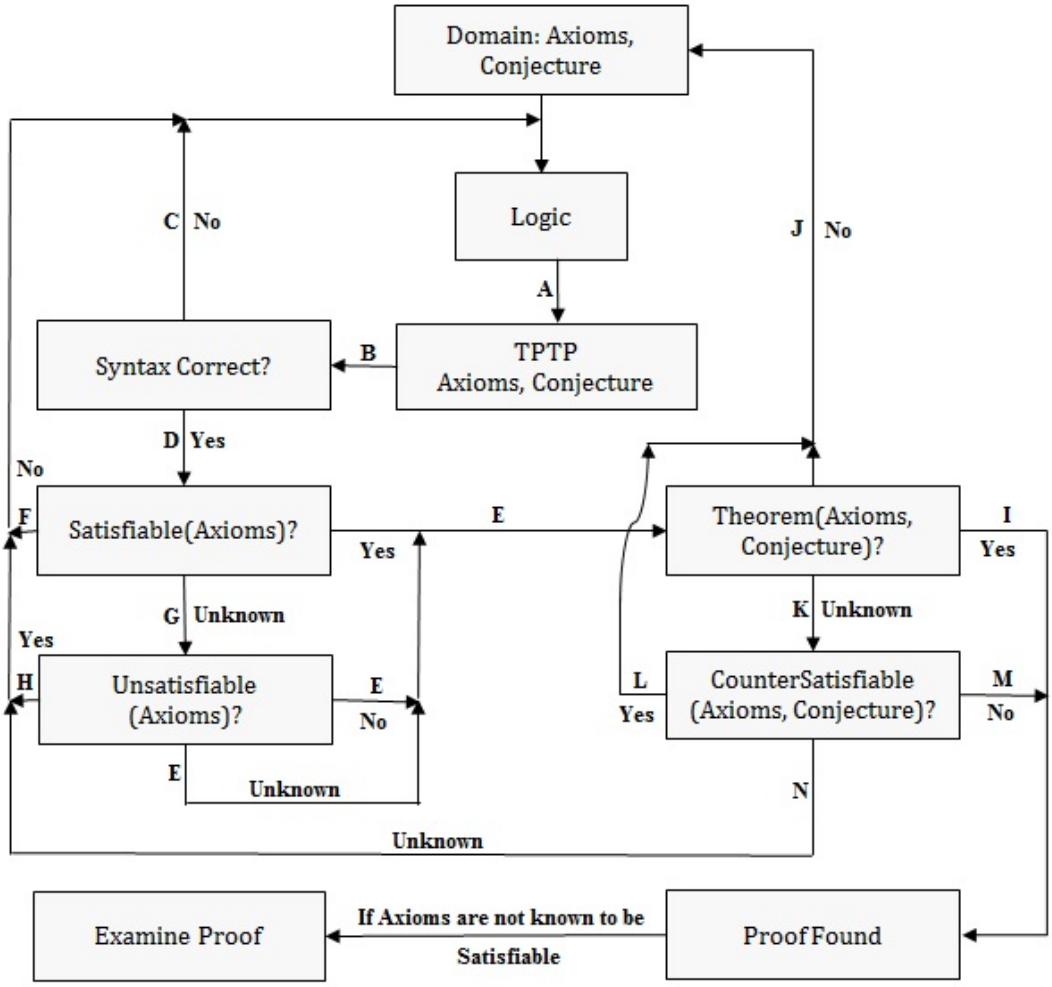

Figure 3.1: The ATP Process

The ATP process illustrated in Figure 3.1 avoids these drawbacks, provides more form and structure, and allows for earlier error detection compared to giving axioms and a conjecture directly to an ATP system. The initial step is to encode the world knowledge of the domain in question as axioms and a conjecture. The ATP process then

starts by checking the syntactic correctness of the logical formulae, and then checking the axioms' satisfiability. If the syntax is correct and the axioms are *Satisfiable*, then an attempt to find a proof can be performed. If a proof cannot be found, then an attempt to prove that the axioms and the conjecture are *Countersatisfiable* can be performed – if successful this provides useful feedback to the user. A more detailed explanation of the process is provided in Section 3.2.

3.2 The ATP Process Details

The ATP process goes through a series of steps while attempting to find a proof that the conjecture is a *Theorem* of the axioms. These steps are as follows:

1. The axioms and conjecture are added to the pool of available formulae (label A in Figure 3.1). A syntax checker system is executed to check the syntax of the formulae (label B). If the syntax checker returns a `SyntaxError` SZS status, the process is terminated (label C).

2. Once the syntax of the formulae has been verified, the process moves on to checking the satisfiability of the axioms. A model finding ATP system such as `iProver` is executed to check if the axioms are *Satisfiable* (label D). If the ATP system shows that the axioms are *Satisfiable*, the process moves on to trying to prove that the conjecture is a *Theorem* of the axioms (label E). Otherwise, if the system shows that the axioms are *Unsatisfiable*, the process is terminated (label F). If the model finder cannot reach a decision about the satisfiability

of the axioms, a refutation finding ATP system such as `E` is executed to try to prove that the axioms are *Unsatisfiable* (label G).

3. If the ATP system finds that the axioms are *Unsatisfiable*, the process is terminated (label H). If it shows that the axioms are *Satisfiable*, or if a decision cannot be reached, the process again moves on to trying to prove that the conjecture is a *Theorem* of the axioms (label E). This is an optimistic approach that assumes satisfiability of the axioms if nothing can be shown explicitly.

4. Once the axioms have been found to be *Satisfiable*, or if it couldn't be proved that they are *Unsatisfiable*, the process moves on to trying to prove that the conjecture is a *Theorem* of the axioms. A theorem proving ATP system such as `E` is executed on the logical formulae (label E). If a proof is found, the process is terminated with a *Theorem* status (label I). If the ATP system shows that the axioms and conjecture are *CounterSatisfiable* (i.e., the conjecture is not a *Theorem* of the axioms), the process is terminated with a *CSA* status (label J). If a decision cannot be reached, the system moves on to trying to prove that the axioms and conjecture are *CounterSatisfiable* (label K).

5. A countermodel finding ATP system such as `iProver` is executed to try to prove that the axioms and conjecture are *CounterSatisfiable* (label K). If the ATP system shows that the axioms and conjecture are *CounterSatisfiable*, the process is terminated with a *CSA* status (label L). If a decision cannot be reached, the process is terminated with a *Unknown* status (label N). Otherwise, if a proof is found, the process is terminated with a *Theorem* status (label M).

Section 4.2 provides a detailed explanation of how the TPI language, in its modified command line form discussed in Section 2.3, is used to implement the ATP process.

3.3 Benefits

Section 3.2 sheds the light on the benefits of applying the ATP process. Since there are multiple steps that the ATP process goes through, any problems with the logical formulae can be detected and pointed out at each step. The syntactic correctness of the logical formulae representing of the problem is checked and verified at early steps. Later steps check for the satisfiability of the problem axioms. Once the syntactic correctness of the formulae and the satisfiability of the axioms have been established, further steps in the process prove or disprove that the conjecture is a *Theorem* of the axioms. If the process terminates at any step, the user is provided with useful feedback and can take further actions accordingly.

Chapter 4

The TPI Interpreter

4.1 The TPI Language Interpreter

This section discusses the implementation details of the TPI language interpreter presented in Section 2.3. The interpreter has been developed in `perl`, and acts as a standalone program that executes one TPI command at a time. It can be used with, e.g., shell scripting, to provide a fully featured language for ATP processing.

The interpreter recognizes commands of the following format:

`tpi` *command_name command_details*

where `tpi` is the name of the interpreter.

The interpreter stores its logical formulae in TPTP format, in a text file called `DB_File`. The file provides persistent storage of the logical formulae between executions of TPI commands. The logical formulae can be organized into groups (as described in Section 2.4.2). Groups are implemented in the `DB_File` by adding the comment lines `%TPI start_group:` *group_name* and `%TPI end_group:` *group_name*

before and after the formulae in the group. All formulae belong to the general group `tpi`, i.e., the whole `DB_File` is used whenever a `$getgroups(tpi)` is encountered. The interpreter provides the default groups `tpi_premises` and `tpi_conjectures`. These two groups are created while the `input` and `input_formula` commands are being executed. All axiom-like formulae (axioms, definitions, etc.) are added to the `tpi_premises` group, and all conjectures are added to the `tpi_conjectures` group. The interpreter checks for each formula's role and adds it to the appropriate group accordingly. If the formulae belonging to a certain group are not contiguous while being input, the interpreter creates multiple entries representing the same group in the `DB_File`.

The TPI language has commands for setting and accessing "environment variables" (as described in Section 2.4.4). As the interpreter executes as a sub-process of the user's shell, the interpreter's environment variables are not passed back to the shell's environment. To provide persistence of its environment variables, the interpreter saves its environment variables in a file called `ENV_VARS` whenever the `setenv` command is used, and reloads the values at the start of each execution. If a conflict is detected, i.e., one of the shell's environment variables has the same name as a variable in the `ENV_VARS` file, the shell's variable overwrites the local one and the new value is written to the `ENV_VARS` file. This means that the overwritten variable is no longer persistent across multiple TPI command executions, and the user needs to be careful when creating shell environment variables to avoid this problem. The user should also not use environment variable names that are already in use by the shell, e.g. `PATH`, when creating environment variables using the TPI `setenv` command.

Example 2 illustrates some capabilities of the TPI language, and shows practical use of the TPI command form that is accepted by the interpreter.

```
 1 tpi start_group axioms
 2 tpi input_formula 'fof(ax,axiom,r).'
 3 tpi input_formula 'fof(ax2,axiom,r => s).'
 4 tpi end_group axioms
 5 tpi start_group conjecture
 6 tpi input_formula 'fof(conj,conjecture,s).'
 7 tpi end_group conjecture
 8 tpi deactivate_group conjecture
 9 tpi execute_async 'SZS_ASYNC' = 'iprover 30 $getgroups(axioms)'
10 tpi activate_group conjecture
11 tpi execute SZS = 'eprover -s --cpu-limit=30 --tstp-format
   $getgroups(axioms,conjecture)'
12 tpi waitenv 'SZS_ASYNC'
13 tpi write 'Axioms status:  '  & '$getenv('SZS_ASYNC')' &
   ' Proof status:  '  & '$getenv(SZS)'
14 tpi exit
```

Example 2: A typical use of the TPI commands accepted by the Interpreter

Commands 1, 4, 5 and 7 use `start_group` and `end_group` to create groups for the axioms and the conjecture. Command 8 deactivates the `conjecture` group. Command 9 invokes the `iProver` ATP system in the background to check if the axioms are *Satisfiable*. Command 10 activates the `conjecture` group. The `E` ATP system is invoked by command 11, to prove that the conjecture is a *Theorem* of the axioms. Command 12 waits until the `iProver` system finishes execution, after which command 13 outputs the execution results. Command 14 ends execution.

Example 3 shows how the TPI commands in their modified format can be used with shell scripting. It provides is a simple shell script that integrates TPI commands with shell commands.

```
 1 #!/bin/tcsh
 2 set MyProver = "Systems/E---1.8/eprover -s --auto --cpu-limit=300"
 3 tpi input Problems/PUZ/PUZ001+1.p
 4 tpi execute 'SZSProof' = "$MyProver"' $getgroups(tpi)'
 5 set TheResult = `tpi write '$getenv(SZSProof)'`
 6 if( "$TheResult" == "Theorem" ) then
 7    echo "FINAL STATUS: Theorem"
 8 else
 9    echo "$MyProver couldn't find a proof of the conjecture."
10    echo "FINAL STATUS: $TheResult"
11 endif
```

Example 3: Using the TPI commands with shell scripting

Line 2 sets the variable `MyProver` to `E` with the necessary command line flags. Line 3 uses the TPI `input` command to add the formulae from `PUZ001+1.p` problem file to the `DB_File`. Line 4 uses the TPI `execute` command to execute `E` against all the formulae in the `DB_File` - `$getgroups(tpi)` causes all the formulae in the `DB_File` to be selected for execution. Line 5 sets the variable `TheResult` to value of the environment variable `SZSProof`, which holds the `SZS Status` of `E`'s execution. The remaining lines demonstrate a conditional statement that outputs the proof result.

4.2 TPI Applications

This section provides three example applications where the TPI language, with its modified command format, is usefully utilized. The first example, in Section 4.2.1, is an implementation of the ATP process described in Section 3.1. The second example, in Section 4.2.2, also provides an implementation of the ATP process, with modifications that make some tasks run in parallel. The last example, in Section 4.2.3,

demonstrates how the TPI language is used to provide an estimate of the shortest running time a strategy scheduling ATP system can be given to solve a given ATP problem.

4.2.1 Serial Implementation of the ATP Process

Algorithm 1 illustrates how the TPI language can be used to implement the ATP process described in Section 3.1. It handles all the steps introduced in Section 3.2. Code Sample 1 provides the `tcsh` script code for Algorithm 1.

The axioms and conjecture are added to the `DB_File` using the `input` command (label A in Figure 3.1, line 1 in Algorithm 1, line 1 in Code Sample 1). Using the `execute` command, a syntax checker system is executed to check the syntax of the formulae (label B, line 2, line 2). If the syntax checker returns a `SyntaxError` status, the user is informed using the `write` command and the process ends with a call to the `exit` command (label C, lines 4-5, lines 5-6).

Once the syntax of the formulae has been verified, the process moves on to checking the satisfiability of the axioms. Using the `execute` command, a model finding ATP system like `iProver` is executed to check if the axioms are *Satisfiable* (label D, line 7, line 8). If the ATP system shows that the axioms are *Satisfiable*, the process moves on to trying to prove that the conjecture is a *Theorem* of the axioms (label E, lines 9-28, lines 11-32). Otherwise, if it shows that the axioms are *Unsatisfiable*, the user is informed using the `write` command and the process ends with a call to the `exit` command (label F, lines 30-31, lines 34-35).

```
1  input the logical formulae;
2  execute a syntax checker on the formulae;
3  if the returned status is SyntaxError then
4  |  write Wrong Syntax;
5  |  exit;
6  else
7  |  execute a model finder to check if the axioms are Satisfiable;
8  |  if the returned status is Satisfiable then
9  |  |  Repeated Section;
10 |  |  execute a prover to check if the conjecture is a Theorem of the axioms;
11 |  |  if the returned status is Theorem then
12 |  |  |  write  Proof Found;
13 |  |  |  exit;
14 |  |  else
15 |  |  |  if the returned status is CounterSatisfiable then
16 |  |  |  |  write  No Proof;
17 |  |  |  |  exit;
18 |  |  |  else
19 |  |  |  |  execute a countermodel finder to check if the axioms and
   |  |  |  |  conjecture are CounterSatisfiable;
20 |  |  |  |  if the returned status is CounterSatisfiable then
21 |  |  |  |  |  write  No proof;
22 |  |  |  |  |  exit;
23 |  |  |  |  else if the returned status is Theorem then
24 |  |  |  |  |  write  Proof Found;
25 |  |  |  |  |  exit;
26 |  |  |  |  else
27 |  |  |  |  |  write  No Proof;
28 |  |  |  |  |  exit;
29 |  |  else if the returned status is Unsatisfiable then
30 |  |  |  write  Axioms Unsatisfiable;
31 |  |  |  exit;
32 |  |  else
33 |  |  |  execute a prover to check if the axioms are Unsatisfiable;
34 |  |  |  if the returned status is Unsatisfiable then
35 |  |  |  |  write  Axioms Unsatisfiable;
36 |  |  |  |  exit;
37 |  |  |  else if the returned status is Satisfiable or Unknown then
38 |  |  |  |  Goto Repeated Section;
```

Algorithm 1: The ATP Process

```
 1 tpi input $1
 2 tpi execute 'SZS_Syntax' = 'tptp4X -q3 -z $getgroups(tpi)'
 3 set SyntaxResult = `tpi write '$getenv(SZS_Syntax)'`
 4 if ("$SyntaxResult" == "SyntaxError") then
 5    echo "% Syntax Error!"
 6    tpi exit
 7 else
 8    tpi execute 'SZS_Satisfiability' = 'iproveropt_run_sat.sh 300
         $getgroups(tpi_premises)'
 9    set SatResult = `tpi write '$getenv(SZS_Satisfiability)'`
10    if("$SatResult" == "Satisfiable") then
11      repeatedSection:
12      tpi execute 'SZS_Proof' = 'eprover --auto --cpu-limit=300
          $getgroups(tpi)'
13      set ProofResult = `tpi write '$getenv(SZS_Proof)'`
14      if("$ProofResult" == "Theorem") then
15        echo "% Proof Found!
16        tpi exit
17      else
18        if( "$ProofResult" == "CounterSatisfiable") then
19          echo "% No Proof!"
20          tpi exit
21        else
22          tpi execute 'SZS_CSA_AxConj'
             = 'iproveropt_run_sat.sh 300 $getgroups(tpi)'
23          set CSAResultAxConj = `tpi write '
             $getenv('SZS_CSA_AxConj')'`
24          if( "$CSAResultAxConj" == "CounterSatisfiable") then
25            echo "% No Proof!"
26            tpi exit
27          else if("$CSAResultAxConj" == "Theorem")
28            echo "% Proof Found!"
29            tpi exit
30          else
31            echo "% No Proof!"
32            tpi exit
33    else if("$SatResult" == "Unsatisfiable")
34      echo "% Axioms Unsatisfiable!"
35      tpi exit
36    else
37      tpi execute 'SZS_Unsatisfiability' = 'eprover --auto --cpu-limit=300
          $getgroups(tpi_premises)'
38      set UnsatResult = `tpi write '$getenv(SZS_Unsatisfiability)'`
39      if("$UnsatResult" == "Unsatisfiable")
40        echo "% Axioms Unsatisfiable!"
```

```
41          tpi exit
42        else
43          goto repeatedSection
```

Code Sample 1: The ATP Process Code

If the model finder cannot reach a decision about the satisfiability of the axioms, a refutation finding ATP system like `E` is executed using the `execute` command, to try to prove that the axioms are *Unsatisfiable* (label G, line 33, line 37).

If the ATP system finds that the axioms are *Unsatisfiable*, the user is informed using the `write` command and the process ends with a call to the `exit` command (label H, line 35-36, lines 40-41). If the system shows that the axioms are *Satisfiable*, or if a decision cannot be reached, the process again moves on to trying to prove that the conjecture is a *Theorem* of the axioms (label E, line 38, line 43). (This is an optimistic approach that assumes satisfiability of the axioms if nothing can be shown explicitly.)

Once the axioms have been found to be *Satisfiable*, or if it couldn't be proved that they are *Unsatisfiable*, the process moves on to trying to prove that the conjecture is a *Theorem* of the axioms. Using the `execute` command, a theorem proving ATP system like `E` is executed on the logical formulae (label E, line 10, line 12). If a proof is found, a *Theorem* status is output using the `write` command, and the process ends with a call to the `exit` command (label I, lines 12-13, lines 15-16). If the ATP system shows that the axioms and conjecture are *CounterSatisfiable* (i.e., the conjecture is not a *Theorem* of the axioms), the user is informed using the `write` command and the process ends with a call to the `exit` command (label J, lines 16-17, lines 19-20). If a decision cannot be reached, the process moves on to trying to prove that the

axioms and conjecture are *CounterSatisfiable* (label K, lines 19-28, lines 22-32).

Using the `execute` command, a countermodel finding ATP system like `iProver` is executed to try to prove that the axioms and conjecture are *CounterSatisfiable* (label K, line 19, line 22). If the ATP system shows that the axioms and conjecture are *CounterSatisfiable*, the user is informed using the `write` command and the process ends with a call to the `exit` command (label L, lines 21-22, lines 25-26). If a proof is found, a *Theorem* status is output using the `write` command, and the process ends with a call to the `exit` command (label M, lines 24-25, lines 28-29). Otherwise, if a decision cannot be reached, the user is informed using the `write` command and the process ends with a call to the `exit` command (label N, lines 27-28, lines 31-32).

The following is the output when the script is run on the `KRS233+1.p` problem (`http://www.tptp.org/cgi-bin/SeeTPTP?Category=Problems&Domain=KRS&File=KRS233+1.p`). It demonstrates the case where the axioms are assumed to be *Satisfiable* if they cannot to be proved otherwise. A proof of the conjecture is found by `E`.

```
%----------------------------------------------
% Now checking the syntax....
% Using:  Systems/TPTP4X---0.0/tptp4X -q3 -z
%----------------------------------------------
% ATP SYSTEM STATUS:
% SZS status Success
%----------------------------------------------
% Now checking if the axioms are Satisfiable....
% Using:  Systems/iProver---1.0-SAT/Source/iproveropt_run_sat.sh 15
%----------------------------------------------
% ATP SYSTEM STATUS:
% SZS status Unknown
%----------------------------------------------
% Couldn't reach a decision, checking if the axioms are Unsatisfiable....
% Using:  Systems/E---1.8/eprover --auto --cpu-limit=15 --tstp-format
%----------------------------------------------
% ATP SYSTEM STATUS:
```

```
# SZS status ResourceOut
%------------------------------------------------
% Now checking if the conjecture is a Theorem of the axioms....
% Using:  Systems/E---1.8/eprover --auto --cpu-limit=15 --tstp-format
%------------------------------------------------
% ATP SYSTEM STATUS:
# SZS status Theorem
%------------------------------------------------
% FINAL STATUS:
% SZS status Theorem
%------------------------------------------------
```

4.2.2 Parallelization of the ATP Process

Algorithm 2 provides another way of implementing the ATP process. In Algorithm 1, each ATP system is called when it is needed in the process according to the stage of processing. In Algorithm 2, all ATP systems used for the different execution steps presented in Algorithm 1 are run in the background using the `execute_async` command. The ATP systems are executed once the syntactic correctness of the formulae has been verified. This leads to the results of the ATP systems' executions being available sooner since all systems are executed early in the process, as opposed to having to wait for each system to finish execution as is the case in Algorithm 1. This approach would also make use of more of the available processor cores. Code Sample 2 provides the equivalent `tcsh` script code to Algorithm 2.

The axioms and conjecture are added to the `DB_File` using the `input` command (label A in Figure 3.1, line 1 in Algorithm 2, line 1 in Code Sample 2). Using the `execute` command, a syntax checker system is executed to check the syntax of the formulae (label B, line 2, line 2).

```
1  input the logical formulae;
2  execute a syntax checker on the formulae;
3  if the returned status is SyntaxError then
4  |  write Wrong Syntax;
5  |  exit;
6  else
7  |  execute_async a model finder to check axiom satisfiability;
8  |  execute_async a prover to check axiom unsatisfiability;
9  |  execute_async a prover to check if the conjecture is a Theorem of the
   |  axioms;
10 |  execute_async a model finder to check if the axioms and the conjecture
   |  are CounterSatisfiabile;
11 |  waitenv on the axiom satisfiability check;
12 |  if the returned status is Satisfiable then
13 |  |  Repeated Section;
14 |  |  waitenv on the proof that the conjecture is a Theorem of the axioms;
15 |  |  if the returned status is Theorem then
16 |  |  |  write  Proof Found;
17 |  |  |  exit;
18 |  |  else
19 |  |  |  if the returned status is CounterSatisfiable then
20 |  |  |  |  write  No Proof;
21 |  |  |  |  exit;
22 |  |  |  else
23 |  |  |  |  waitenv on the check if the axioms and the conjecture are
   |  |  |  |  CounterSatisfiabile;
24 |  |  |  |  if the returned status is CounterSatisfiable then
25 |  |  |  |  |  write  No proof;
26 |  |  |  |  |  exit;
27 |  |  |  |  else if the returned status is Theorem then
28 |  |  |  |  |  write  Proof Found; exit;
29 |  |  |  |  else
30 |  |  |  |  |  write  No Proof; exit;
31 |  else if the returned status is Unsatisfiable then
32 |  |  write  Axioms Unsatisfiable; exit;
33 |  else
34 |  |  waitenv on the axiom unsatisfiability check;
35 |  |  if the returned status is Unsatisfiable then
36 |  |  |  write  Axioms Unsatisfiable; exit;
37 |  |  else if the returned status is Satisfiable or Unknown then
38 |  |  |  Goto Repeated Section;
```

Algorithm 2: The Parallelized ATP Process

```
 1 tpi input $1
 2 tpi execute 'SZS_Syntax' = 'tptp4X -q3 -z $getgroups(tpi)'
 3 set SyntaxResult = `tpi write '$getenv(SZS_Syntax)'`
 4 if ("$SyntaxResult" == "SyntaxError") then
 5    echo "% Syntax Error!"
 6    tpi exit
 7 else
 8    tpi execute_async 'SZS_Sat' = 'iprover $getgroups(tpi_premises)'
 9    tpi execute_async 'SZS_AxUnsat' = 'eprover $getgroups(tpi_premises)'
10    tpi execute_async 'SZS_Proof'' = 'eprover $getgroups(tpi)'
11    tpi execute_async 'SZS_CSA_AxConj' = 'iprover $getgroups(tpi)'
12    tpi waitenv 'SZS_Sat'
13    set SatResult = `tpi write '$getenv(SZS_Sat)'`
14    if("$SatResult" == "Satisfiable" )
15      repeatedSection:
16      tpi waitenv 'SZS_Proof'
17      set ProofResult = `tpi write '$getenv(SZS_Proof)'`
18      if("$ProofResult" == "Theorem") then
19        echo "% Proof Found!
20        tpi exit
21      else
22        if( "$ProofResult" == "CounterSatisfiable") then
23          echo "% No Proof!"
24          tpi exit
25        else
26          tpi waitenv 'SZS_CSA_AxConj'
27          set CSAResultAxConj = `tpi write '$getenv('SZS_CSA_AxConj')'`
28          if( "$CSAResultAxConj" == "CounterSatisfiable") then
29            echo "% No Proof!"
30            tpi exit
31          else if("$CSAResultAxConj" == "Theorem")
32            echo "% Proof Found!"
33            tpi exit
34          else
35            echo "% No Proof!"
36            tpi exit
37    else if("$SatResult" == "Unsatisfiable")
38      echo "% Axioms Unsatisfiable!"
39      tpi exit
40    else
41      tpi waitenv 'SZS_AxUnsat'
42      set UnsatResult = `tpi write '$getenv(SZS_AxUnsat)'`
43      if"$UnsatResult" == "Unsatisfiable"
44        echo "% Axioms Unsatisfiable!"
45        tpi exit
```

```
46      else
47        goto repeatedSection
```

Code Sample 2: Parallelized ATP Process Code

If the syntax checker returns a `SyntaxError` status, the user is informed using the `write` command and the process ends with a call to the `exit` command (label C, lines 4-5, lines 5-6). If the syntax is found to be correct, four ATP systems are run in the background using the `execute_async` command.

The first system is a model finding ATP system like `iProver`, and is executed to check if the axioms are *Satisfiable*. The second is a refutation finding ATP system like `E`, and is executed to try to show that the axioms are *Unsatisfiable*. The third is a theorem proving ATP system like `E`, and is executed to try prove that the conjecture is a *Theorem* of the axioms. The last system is a countermodel finding ATP system like `iProver`, and is executed to try to prove that the axioms and conjecture are *CounterSatisfiable* (labels D,G,E,K, lines 7-10, lines 8-11).

Once the syntax of the formulae has been verified, the process moves on to checking the satisfiability of the axioms. Using the `waitenv` command, execution is paused until the environment variable is set by the first ATP system, indicating the status of the axiom satisfiability check (label D, line 11, line 12). If the ATP system has shown that the axioms are *Satisfiable*, the process moves on to the next step, which is to check if a proof if the conjecture is a *Theorem* of the axioms has been found (label E, lines 13-32, , lines 15-36). Otherwise, if the system has shown that the axioms are *Unsatisfiable* the user is informed using the `write` command and the process ends with a call to the `exit` command (label F, line 32, lines 38-39). If the model finder

could not reach a decision about the satisfiability of the axioms, execution is paused using the `waitenv` command until the environment variable is set by the second ATP system, indicating the status of the axiom unsatisfiability check (label G, line 34, line 41).

If the ATP system has shown that the axioms are *Unsatisfiable*, the user is informed using the `write` command and the process ends with a call to the `exit` command (label H, lines 36, lines 44-45). If the system has shown that the axioms are *Satisfiable*, or if a decision could not be reached, the process again moves on to check if a proof that the conjecture is a *Theorem* of the axioms has been found (label E, line 38, line 47). (Moving forward in the process if a decision could not be made is an optimistic approach that assumes satisfiability of the axioms if nothing can be shown explicitly.)

Once the axioms have been found to be *Satisfiable*, or if it couldn't be proved that they are *Unsatisfiable*, the process moves on to checking if a proof that the conjecture is a *Theorem* of the axioms has been found. Using the `waitenv` command, execution is paused until the environment variable is set by the third ATP system, indicating the proof status (label E, line 14, line 16). If a proof that the conjecture is a *Theorem* of the axioms has been found by the system, a *Theorem* status is output using the `write` command, and the process ends with a call to the `exit` command (label I, lines 16-17, lines 19-20). If the ATP system has shown that the axioms and conjecture are *CounterSatisfiable* (i.e., the conjecture is not a *Theorem* of the axioms), the user is informed using the `write` command and the process ends with a call to the `exit` command (label J, lines 20-21, lines 23-24). If a decision is not

reached, the process moves on to check if the axioms and conjecture have been found to be *CounterSatisfiable* (label K, lines 23-30, lines 26-36).

Using the `waitenv` command, execution is paused until the environment variable is set by the last ATP system, indicating whether the axioms and conjecture are *CounterSatisfiable* (label K, line 23, line 26). If the ATP system has shown that the axioms and conjecture are *CounterSatisfiable*, the user is informed using the `write` command and the process ends with a call to the `exit` command (label L, lines 25-26, lines 29-30). If a proof has been found, a *Theorem* status is output using the `write` command, and the process ends with a call to the `exit` command (label M, line 28, lines 32-33). Otherwise, if a decision could not be reached, the user is informed using the `write` command and the process ends with a call to the `exit` command (label N, line 30, lines 35-36).

The following is the output when the script is run on the `AGT001+1` problem (`http://www.tptp.org/cgi-bin/SeeTPTP?Category=Problems&Domain=AGT&File=AGT001+1.p`).

```
%------------------------------------------------
% Now checking the syntax....
% Using:  Systems/TPTP4X---0.0/tptp4X -q3 -z
%------------------------------------------------
% ATP SYSTEM STATUS:
% SZS status Success
%------------------------------------------------
% Now checking if the axioms are Satisfiable....
% Using:  Systems/iProver---1.0-SAT/Source/iproveropt_run_sat.sh 15
% Waiting for System:
%------------------------------------------------
% ATP SYSTEM STATUS:
% Satisfiable
%------------------------------------------------
% Now checking if the conjecture is a Theorem of the axioms....
```

```
% Using:  Systems/E---1.8/eprover -s --auto --cpu-limit=15 --tstp-format
% Waiting for System:
%-----------------------------------------------
% ATP SYSTEM STATUS:
% Theorem
%-----------------------------------------------
% FINAL STATUS:
% SZS status Theorem
%-----------------------------------------------
```

4.2.3 Shortest Time to Solve an ATP Problem

An ATP system usually accepts a command line flag that specifies the maximum time the system can use to try and solve a problem. Many ATP systems employ strategy scheduling techniques [19] in order to try multiple search strategies to solve a given problem in the amount of time they have available. Depending on the amount of time it has available, an ATP system will divide this time among different search strategies and try to solve the problem using each of those strategies. Providing an ATP system with less time may result in the system solving the problem in a shorter amount of time. This section shows how the TPI language is used with shell scripting to find out the least amount of time the system can be given to solve a problem successfully.

Figure 4.1 provides an example of how a strategy scheduling ATP system could adjust its scheduling strategies according to the amount of time available for execution. In this example, the ATP system uses four different search strategies to try to solve an ATP problem. Initially, the system is provided with a 120 second time limit. It allocates 15 seconds for strategies 1 and 2, 30 seconds for strategy 3 and 60 seconds for strategy 4. It solves the problem using strategy 4 in 25 seconds with a

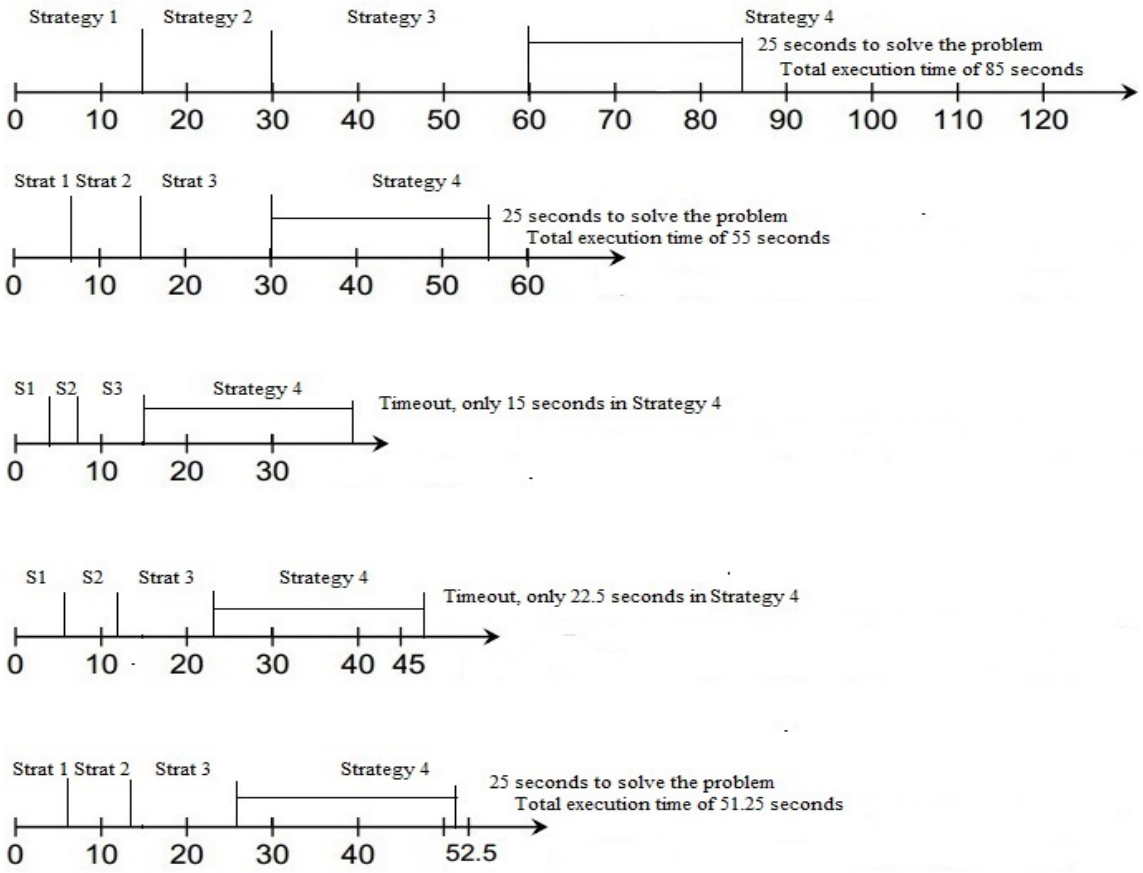

Figure 4.1: Strategy Scheduling Example

total execution time of 85 seconds (60 seconds for the first 3 strategies and 25 seconds for solving the problem using strategy 4). If the time limit is halved and the system is provided with 60 seconds instead, the total execution time drops to 55 seconds (30 seconds for the first 3 strategies and 25 seconds for solving the problem using strategy 4). When the execution time is further halved to become 30 seconds, the system can no longer solve the problem since the available time for strategy 4 drops to 15 seconds.

The last time limit value at which the system solved the problem is 60 seconds. The average of this time limit value and the current time limit value (30 seconds) is 45

seconds and can be used as the new time limit to try to solve the problem. Using 45 seconds as the new time limit value, the system allocates 5.625 seconds for strategies 1 and 2, 11.25 seconds for strategy 3 and 22.5 seconds for strategy 4. Again, the system cannot solve the problem since the available time for strategy 4 is less than 25 seconds. Using the same approach, the average of the last time limit value at which the system solved the problem and the current time limit value (45 seconds), is 52.5 seconds and is used as the new time limit value. The system allocates 6.5625 seconds for strategies 1 and 2, 13.125 seconds for strategy 3 and 26.25 seconds for strategy 4. The system solves the problem using strategy 4 in 25 seconds with a total execution time of 51.25 seconds. Therefore, the shortest time the system takes to solve the problem is 51.25 seconds. This is just an example of how an ATP system can split up the time it has available among different search strategies. Different ATP systems use different scheduling techniques to solve ATP problems.

A `tcsh` shell script has been developed to find the least amount of time a strategy scheduling ATP system can be given in order to solve a given problem. It makes use of the TPI `execute` command and implements a binary search technique to reach its goal. Algorithm 3 and Code Sample 3 provide the logic and implementation details. The algorithm starts with collecting the information needed from the user: the name of the problem file, the initial time limit, and a constant value - `Epsilon` - whose use will be explained shortly (lines 1-3 in Algorithm 3, lines 2-4 in Code Sample 3). Default values for the time limit and `Epsilon` in seconds are used if not provided by the user (lines 4-7, lines 5-8). Two variables are then initialized: the `LowerBound` variable that is the last reported time limit at which the given ATP system times

```
1  Read the Problem File name;
2  Read the initial TimeLimit value;
3  Read the Epsilon value;
4  if TimeLimit is not provided then
5  |  Set TimeLimit = 100;
6  if Epsilon is not provided then
7  |  Set Epsilon = 10;
8  Set LowerBound = 0;
9  Set UpperBound = TimeLimit;
10 execute the given ATP system against the given Problem with the current
   TimeLimit value;
11 if the returned status is not TimeOut then
12 |  while The difference between UpperBound and LowerBound is greater than
   |  Epsilon do
13 |  |  Set TimeLimit = average value of UpperBound and LowerBound;
14 |  |  execute the given ATP system against the give Problem with the
   |  |  current TimeLimit value;
15 |  |  if the returned status is Theorem then
16 |  |  |  Set UpperBound = current TimeLimit value;
17 |  |  if the returned status is TimeOut then
18 |  |  |  Set LowerBound = current TimeLimit value;
19 |  The shortest time to solve the problem is the current UpperBound value;
20 else
21 |  Problem could not be solved in the time provided;
```

Algorithm 3: Computing he shortest time to solve an ATP problem

out when trying to solve the problem, and the `Upperbound` variable that is the last reported time limit at which the given ATP system successfully solved the problem. The `LowerBound` is initialized to 0 and the `UpperBound` is initialized to the initial time limit value (lines 8-9, lines 9-10).

A strategy scheduling ATP system - `E` in this case - is then executed using the `execute` command, bounded by the initial time limit to try to solve the given problem (line 10, line 11). If it times out, the algorithm halts (line 11, line 13). Otherwise, the time limit is set to be midway between the `LowerBound` and `UpperBound` variables (line 13, line 15) and the system is executed again, bounded by the new time limit

```
 1 #!/bin/tcsh
 2 set Problem = $1
 3 set TimeLimit = $2
 4 set Epsilon = $3
 5 if(!$?TimeLimit)
 6    set TimeLimit = 100
 7 if(!$?Epsilon)
 8    set Epsilon = 10
 9 set LowerBound = 0
10 set UpperBound = $TimeLimit
11 tpi execute -q2 'SZS_Scheduling' = "/E---1.8/eprover --cpu-limit=
      $UpperBound $Problem"''
12 set ProofResult = `tpi write '$getenv(SZS_Scheduling)'`
13 if("$ProofResult" != "ResourceOut")
14    while(($UpperBound - $LowerBound) > $Epsilon)
15       @ TimeLimit = ($UpperBound + $LowerBound) / 2
16       tpi execute -q2 'SZS_Scheduling' = "/E---1.8/eprover
           --cpu-limit=$TimeLimit $Problem"''
17       set ProofResult = `tpi write '$getenv(SZS_Scheduling)'`
18       if("$ProofResult" == "Theorem")
19          set UpperBound = $TimeLimit
20       if("$ProofResult" == "ResourceOut")
21          set LowerBound = $TimeLimit
22    end
22    echo "Final time limit is:  $UpperBound"
24 else
25    echo "Problem could not be solved in the time provided!"
```

Code Sample 3: TCSH code to compute the shortest time to solve an ATP problem

value (line 14, line 16). If it successfully solves the problem, the `UpperBound` value is set to the time limit value (lines 15-16, lines 18-19). If it times out, the `LowerBound` value is set to the time limit value (lines 17-18, lines 20-21). This process is repeated in a loop until the difference between `UpperBound` and `LowerBound` becomes smaller than `Epsilon` (lines 12-18, lines 14-21). Upon exiting from the loop, the value held by the `UpperBound` variable is the time limit value at which the system last solved the problem successfully, if it did not time out at the first attempt.

The following is a sample output when the script is run against the ALG103+1 problem (http://www.tptp.org/cgi-bin/SeeTPTP?Category=Problems&Domain=ALG&File=ALG103+1.p).

```
Problem is /home/tptp/TPTP/Problems/ALG/ALG103+1.p
Current time limit:  300
E---1.8/eprover --auto-schedule --cpu-limit=300 /Problems/ALG/ALG103+1.p
Current proof status SZS: Theorem
Current time limit:  150
E---1.8/eprover --auto-schedule --cpu-limit=150 /Problems/ALG/ALG103+1.p
Current proof status SZS: Theorem
Current time limit:  75
E---1.8/eprover --auto-schedule --cpu-limit=75 /Problems/ALG/ALG103+1.p
Current proof status SZS: Theorem
Current time limit:  37
E---1.8/eprover --auto-schedule --cpu-limit=37 /Problems/ALG/ALG103+1.p
Current proof status SZS: ResourceOut
Current time limit:  56
E---1.8/eprover --auto-schedule --cpu-limit=56 /Problems/ALG/ALG103+1.p
Current proof status SZS: ResourceOut
Current time limit:  65
E---1.8/eprover --auto-schedule --cpu-limit=65 /Problems/ALG/ALG103+1.p
Current proof status SZS: Theorem
Final time limit is:  65
```

Chapter 5

Generic Control

5.1 Introduction

The previous chapters explain how the TPI language provides a practical way of executing ATP systems. The TPI language `execute` command can be used to execute any ATP system, but it is the user's responsibility to provide the correct command line flags that are needed for the system's execution. Different ATP systems have different command line flags that control different aspects of their execution. This chapter is concerned with flags that control how an ATP system goes about its proof finding process, and how it produces its output. For example, `E` provides flags that determine its reasoning process, decide which axioms get selected in the proof building process, and control how its output is produced. Similar flags are provided by many ATP systems. It would be desirable to abstract them to generic options. Instead of the user having to provide these command line flags individually for each ATP system, the `execute` command has been modified to accept them as generic parameters. This

would provide a standard interface for the execution of different ATP systems. Three aspects of an ATP system's proof finding process have been chosen:

- **Forward vs. Backward Reasoning:** Forward reasoning is a reasoning process in which an ATP system starts with the axioms, and then derives new formulae in the hope of reaching the conjecture. Backward reasoning involves an ATP system starting from the conjecture, then repeatedly generating new sub-goals until the axioms are reached. Forward and backward reasoning yield different execution times and possibly different execution results when used on a given problem with the same ATP system.

- **Axiom Selection:** Some ATP systems employ techniques to select axioms that are more likely to be necessary for generating a proof. The system will try to select the axioms that are most relevant to the conjecture. Using axiom selection leads to a shorter execution time if the selected axioms are sufficient to generate a proof. This might have a big impact on efficiency, especially for problems that have a large number of axioms. Axiom selection can either be turned on or off when running the `execute` command on systems that support it.

- **Proof Output:** Most ATP systems provide a meaningful output of the steps that led to a proof. Generating this output could be resource and time consuming for big problems, and has an impact on the system's efficiency. Proof output can either be turned on or off. When it is turned off, less details are provided about the system's execution, depending on the system.

5.2 Implementation

In order for the `execute` command to provide a standard way of using the generic options discussed in Section 5.1, it has been modified to accept such options as generic parameters. The modified form of the `execute` command is as follows:

`tpi execute[(List_of_GenericParameters)]` *`[EnvVar =] command`*

where the list of generic parameters can contain any of the following values:

- `Forward:` The ATP system should use forward reasoning in the proof building process.
- `Backward:` The ATP system should use backward reasoning in the proof building process.
- `AxSelect:` The ATP system should use axiom selection in the proof building process if applicable.
- `NoAxSelect:` The ATP system should not use axiom selection in the proof building process.
- `Proof:` The ATP system should provide the full details of the proof.
- `NoProof:` The ATP system should provide less details about the proof, depending on the system.

When `execute` is called in this form, the TPI language interpreter detects the name of the ATP system in the provided command, determines the generic parameters used, and add their values - according to the system used - to the execution command.

The process of determining whether or not a given ATP system uses these control flags and determining their correct values was sometimes challenging. In some cases, the values of the flags could be retrieved directly from the system's user manual. In other cases, they could not be determined and the system's developer had to be contacted. Once these values had been determined, they could be used directly through the generic parameters, saving the system's user a lot of time and effort.

5.2.1 Example 1

The following example demonstrates a call to `execute` in its modified form. The system used is `E` and the problem used is `PUZ001+1`:

```
tpi execute(AxSelect,NoProof) 'SZS_STATUS' = 'eprover --cpu-limit=30
Problems/PUZ001+1.p'
```

In this example, the interpreter detects the word `eprover` and determines that the system used is `E`. It detects that the generic control flags used are `AxSelect` and `NoProof`. It then determines that the values of the flags are `--sine=Auto` and `-s`, and adds them to the command. The final call to `execute` becomes:

```
tpi execute 'SZS_STATUS' = 'eprover --sine=Auto -s --cpu-limit=30
Problems/PUZ001+1.p'
```

5.2.2 Example 2

The following example demonstrates another call to `execute` in its modified form. The system used is `iProver` and the problem used is `KRS233+1`:

```
tpi execute(Backward) 'SZS_STATUS' = 'iproveropt --time_out_real 240
--out_options control Problems/KRS233+1.p'
```

In this example, the interpreter detects the word `iproveropt` and determines that the system used is `iProver`. It detects that the generic control flag used is `Backward`. It then determines that the value of the flag is `--inst_lit_sel "[+sign;+ground;-num_var;-num_symb]" --res_lit_sel kbo_max` and adds it to the command. The final call to `execute` becomes as follows:

```
tpi execute 'SZS_STATUS' = 'iproveropt --inst_lit_sel "[+sign;+ground;
-num_var;-num_symb]" --res_lit_sel kbo_max --time_out_real 240 --out_op
tions control Problems/KRS233+1'
```

5.2.3 Further Improvements

The current implementation has the values of the generic parameters hard coded. This has been convenient for the sake of demonstration and since only `E` and `iProver` were used for testing. When the generic parameters for more systems are determined, they will be added to an external file that holds the names of the ATP systems and the values of their corresponding generic parameters. When `execute` is called, the TPI language interpreter will detect the name of the ATP system in the provided command, determine the generic parameters used, fetch their values from the external file according to the system used, and add the values to the execution command.

5.3 Results

Tables 5.1, 5.2 and 5.3 provide the execution times, in seconds, of `E` and `iProver` when they are executed with different combinations of the generic control parameters described in Section 5.1.

E	`Default`	`Backward AxSelect NoProof`	`Forward NoAxSelect NoProof`
`KRS233+1`	4.99	Timeout	4.81
`CSR071+3`	2.06	1.13	2.09
`HAL003+3`	0.07	36.52	0.07
`SWW271+1`	38.82	Timeout	38.65
`SWW314+1`	4.49	Timeout	4.59
`NUM860+1`	2.89	14.53	3.01
`SWW297+1`	4.4	Timeout	4.8

Table 5.1: `E` generic control

E	`Back. NoAxSel NoProof`	`For. AxSel Proof`	`For. NoAxSel Proof`
`KRS233+1`	Timeout	6.06	5.5
`CSR071+3`	1.47	2.52	2.24
`HAL003+3`	37.44	0.08	0.08
`SWW271+1`	Timeout	43.32	42.29
`SWW314+1`	Timeout	6.4	4.91
`NUM860+1`	16.57	3.94	3.92
`SWW297+1`	Timeout	5.17	5.53

Table 5.2: `E` generic control - Continued

Six different combinations were used to execute `E` on the ATP problems. These combinations are the default configuration (forward reasoning with axiom selection and no proof), backward reasoning with axiom selection and no proof, forward reasoning with no axiom selection and no proof, backward reasoning with no axiom selection and no proof, forward reasoning with axiom selection and proof, and forward reasoning with no axiom selection and proof. The ATP problems have been selected from

a pool of first order problems that are complex enough to provide good comparison results. The results show how using different reasoning strategies can yield different execution times and in some cases a different execution status, like the cases where the system times out while solving a problem when Backward Reasoning is used but successfully solves it using Forward Reasoning.

iProver	Default	Forward	Backward
KRS233+1	0.36	0.33	0.35
CSR071+3	66.95	27.14	30.63
HAL003+3	3.33	3.89	3.1
SWW271+1	51.23	116.01	145.67
SWW314+1	14.21	14.51	13.71
NUM860+1	10.8	17.29	10.78
SWW297+1	11.99	12.09	4.4

Table 5.3: `iProver` generic control

Three different combinations were used to execute `iProver` on the ATP problems. These combinations are the default configuration (which does not apply either forward or backward reasoning solely), forward reasoning, and backward reasoning. The results also show how using different reasoning strategies can yield different execution times, like the case with problem `CSR071+3` where execution time is doubled when the default configuration is used, and the case with problem `SWW271+1` where execution time is halved when the default configuration is used.

Chapter 6

Conclusion

This thesis presents the work done for the purpose of building and testing a command line interpreter for the TPI language. The example TPI applications presented in Section 4.2 and the generic control of ATP systems presented in Chapter 5 show how the interpreter adds more flexibility and versatility to the use of ATP systems. The implementation of the ATP process, for example, wouldn't have been possible without the ability to capture the results of ATP systems' executions and use them within a program in order to take further execution decisions. The parallel implementation of the ATP process wouldn't have been possible either without the ability to run multiple systems simultaneously in the background.

The key advantage of a command line interpreter is that it allows the users of ATP systems to use the TPI language commands within the context of a shell. This allows the users to take advantage of the full power of shell programming in order to manipulate logical formulae and execute ATP systems. For example, the algorithm for calculating the shortest time to solve an ATP problem, presented in Section 4.2.3,

relies on conditional and loop statements to repeatedly run an ATP system until the desired goal is reached. The example applications presented are only a sample of what can be achieved when using the interpreter for ATP purposes. It is up to the user to utilize the interpreter within the context of a shell program to achieve his needs.

Future work on the interpreter will focus on making it available to more ATP system developers and users, as they could benefit from the standard interface the TPI language provides for dealing with ATP systems. The generic control of ATP systems discussed in Chapter 5 will be extended to include more ATP systems and more generic parameters. The values of the generic parameters will be stored in a separate file rather than being hard coded as discussed in Section 5.2.3. The interpreter will need to incorporate any new TPI commands, if any are introduced.

References

[1] D. W. Loveland, *Automated Theorem Proving: A Logical Basis (Fundamental Studies in Computer Science)*. Elsevier North-Holland, 1978.

[2] J. Harrison, *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[3] C.-L. Chang and R. C. T. Lee, *Symbolic Logic and Mechanical Theorem Proving*. Computer science classics, Academic Press, 1973.

[4] G. Sutcliffe, "An Overview of Automated Theorem Proving." `http://www.cs.miami.edu/~tptp/OverviewOfATP.html`.

[5] M. Kaufmann and J. S. Moore, "The ACL2 Home Page," Dept. of Computer Sciences, University of Texas at Austin, 2004. `http://www.cs.utexas.edu/users/moore/acl2/`.

[6] W. McCune, "Well-Behaved Search and the Robbins Problem," in *Proceedings of the 8th International Conference on Rewriting Techniques and Applications*, RTA '97, (London, UK, UK), pp. 1–7, Springer-Verlag, 1997.

[7] B. Beckert, R. Hähnle, and P. H. Schmitt, eds., *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334, Springer-Verlag, 2007.

[8] S. Schulz, "System Description: E 1.8," in *Proceedings of the 19th LPAR, Stellenbosch* (K. McMillan, A. Middeldorp, and A. Voronkov, eds.), vol. 8312 of *LNCS*, Springer, 2013.

[9] K. Korovin, "iProver — An Instantiation-Based Theorem Prover for First-Order Logic (System Description)," in *Proceedings of the 4th International Joint Conference on Automated Reasoning*, IJCAR '08, (Berlin, Heidelberg), pp. 292–298, Springer-Verlag, 2008.

[10] L. Kovács and A. Voronkov, "First-Order Theorem Proving and Vampire," in *Proceedings of the 25th International Conference on Computer Aided Verification*, CAV'13, (Berlin, Heidelberg), pp. 1–35, Springer-Verlag, 2013.

[11] G. Sutcliffe, "The TPTP World - Infrastructure for Automated Reasoning," in *Proceedings of the 16th International Conference on Logic for Programming Artificial Intelligence and Reasoning* (E. Clarke and A. Voronkov, eds.), no. 6355 in Lecture Notes in Artificial Intelligence, pp. 1–12, Springer-Verlag, 2010.

[12] G. Sutcliffe, "The TPTP Process Instruction Language," in *Proceedings of the 19th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, 2013.

[13] G. Sutcliffe, "The SZS Ontologies for Automated Reasoning Software," in *Proceedings of the LPAR Workshops: Knowledge Exchange: Automated Provers and Proof Assistants, and The 7th International Workshop on the Implementation of Logics* (G. Sutcliffe, P. Rudnicki, R. Schmidt, B. Konev, and S. Schulz, eds.), no. 418 in CEUR Workshop Proceedings, pp. 38–49, 2008.

[14] W. McCune, "Otter 3.3 Reference Manual," Tech. Rep. ANL/MSC-TM-263, Argonne National Laboratory, Argonne, USA, 2003.

[15] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB Standard: Version 2.0," in *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories* (A. Gupta and D. Kroening, eds.), 2010.

[16] D. R., "The SMT-LIBv2 Language and Tools: A Tutorial," 2013. `http://www.grammatech.com/resource/smt/SMTLIBTutorial.pdf`.

[17] A. Asperti, C. Sacerdoti Coen, E. Tassi, and S. Zacchiroli, "User Interaction with the Matita Proof Assistant," *Journal of Automated Reasoning*, vol. 39, no. 2, pp. 109–139, 2007.

[18] T. Nipkow, L. Paulson, and M. Wenzel, "Isabelle/HOL: A Proof Assistant for Higher-Order Logic." `http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle/doc/tutorial.pdf`.

[19] A. Wolf, "Strategy Selection for Automated Theorem Proving," in *Artificial Intelligence: Methodology, Systems, and Applications* (F. Giunchiglia, ed.), vol. 1480 of *Lecture Notes in Computer Science*, pp. 452–465, Springer Berlin Heidelberg, 1998.