Automated Theorem Proving using the TPTP Process Instruction Language

Muhammad Nassar and Geoff Sutcliffe

University of Miami, Coral Gables, USA
m.mansour1@umiami.edu, geoff@cs.miami.edu

Abstract

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, the TPTP Process Instruction (TPI) language provides capabilities to input, output, and organize logical formulae, and to control the execution of ATP systems. This paper reviews the TPI language, describes a shell interpreter for the language, and demonstrates how useful they can be in theorem proving.

1 Introduction

The TPTP World [5] is a well established infrastructure that supports research, development, and deployment of Automated Theorem Proving (ATP) systems. The TPTP World is based on the Thousands of Problems for Theorem Provers (TPTP) problem library, and includes the TPTP language, the SZS ontologies, the Thousands of Solutions from Theorem Provers (TSTP) solution library, various tools associated with the libraries, and the CADE ATP System Competition (CASC). Originally the TPTP supported only first-order problems in clause normal form (CNF). Over the years support for full first-order form (FOF), typed first-order form (TFF), and typed higher-order form (THF), have been added.

Most input languages for ATP systems, including the existing TPTP language, are designed primarily to write the logical formulae that are input to and output from ATP systems' reasoning processes. There is minimal or no support for controlling the processing of the logical formulae: tasks such as incrementally adding and deleting formulae, controlling ATP systems' reasoning processes, and interacting with non-logical data. Various approaches have been used to overcome this weakness. A direct approach is for ATP systems to support their own system specific commands that control their processing. One well known first-order ATP system that uses this approach is Otter [3]. Another approach is to provide programmatic APIs to ATP systems. MiniSAT [2] is a well known ATP system for propositional logic that offers an API. A third approach is to provide a command language designed to be used in conjunction with an existing language for writing logical formulae. The TPTP Process Instruction (TPI) language [6] and the SMT command language [1] are examples.

The TPI language provides command and control instructions for manipulating logical formulae in the context of the TPTP World. The TPI language makes it easy(er) for ATP users to specify their problems, process the logical formulae, and report results, in a fully automatic way. This paper reviews the TPI language, describes an interpreter for the language, and demonstrates how useful they can be in theorem proving.

2 The TPI Language

The TPI language provides commands to input, output, and organize logical formulae, and to control the execution of ATP systems. TPI commands can be input to an ATP system, or can be input to an interpreter that has no reasoning capabilities and instead invokes external ATP systems. In either case a system that can process TPI commands is referred to as a *TPI system*. A TPI system is capable of inputing and storing logical formulae, and manipulating those formulae according to subsequent TPI commands.

The logical formulae that have been input and stored by a TPI system belong to one or more named *group*s. All formulae belong to the group `tpi`, all premises (formulae with the TPTP role `axiom`, `hypothesis`, `definition`, `lemma`, etc.) belong to the group `tpi_premises`, all `conjecture`s and `negated_conjecture`s belong to the group `tpi_conjectures`, and all formulae can belong to multiple other user-defined groups. The TPI language provides commands for manipulating groups. Logical formulae that have been input and stored are initially in an *active* state. Formulae can be deactivated, placing them into an *inactive* state, so that they do not appear to exist. Inactive formulae can be activated so that they are again used in processing.

Two new TPTP interpreted functors have been defined for use with TPI commands:

- `$getenv(`*Name*`)` - Evaluates to the value of the environment variable *Name*. If *Name* is not set it evaluates to an empty string.
- `$getgroups(`*Group name*`,`*Group name*`,`*Group name ...*`)` - Evaluates to the name of a temporary file containing all the active formulae in the named groups. If a formulae is in multiple of the groups, it appears in the file multiple times.

The following (slightly contrived) example illustrates some capabilities of the TPI language described in Section 2.1, and illustrates the TPI command form that is accepted by the TPI command interpreter described in Section 3. Commands 1 and 4 create a group for the axioms input by commands 2 and 3. The conjecture is input by command 5. Command 6 invokes the `iProver` ATP system in the background, to check that the axioms are *Satisfiable*. While command 6 is running, command 7 deactivates the axioms so that command 8 can use the `E` ATP system to try prove that the conjecture is a tautology. The result of command 8 is written out by command 9. Command 10 reactivates the axioms so that command 11 can try prove that the conjecture is a *Theorem* the axioms. Command 12 waits until the `iProver` system finishes execution, after which command 13 outputs the results. Command 14 ends execution.

```
 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 input_formula 'fof(conj,conjecture,s).'
 6 tpi execute_async SZS_ASYNC = 'iprover 30 $getgroups(axioms)'
 7 tpi deactivate_group axioms
 8 tpi execute SZS = 'eprover --cpu-limit=30 --tstp-format $getgroups(tpi)'
 9 tpi write 'Conjecture alone:  ' & '$getenv(SZS)'
10 tpi activate_group axioms
11 tpi execute SZS = 'eprover --cpu-limit=30 --tstp-format $getgroups(tpi)'
12 tpi waitenv SZS_ASYNC
13 tpi write 'Axioms:' & '$getenv(SZS_ASYNC)' & ' Proof:'  & '$getenv(SZS)'
14 tpi exit
```

Script 1: Use of TPI Commands

2.1 The TPI Commands

The TPI language provides command and control instructions for manipulating logical formulae. In the following, the commands are grouped according to the type of function they perform.

Input and Output: `input_formula, input, output, delete`
`input_formula` inputs and stores a single logical formula. `input` inputs and stores all the formulae in a named file. `input` can also put the formulae into a group. `output` outputs all the stored formulae, or the formulae in a named group, to a file. If `stdout` is the output file name, the formulae are output to the screen. `delete` removes a named formula from the store.

Formula Grouping: `start_group, end_group, delete_group`
`start_group` marks the start of a named group of logical formulae. All formulae that are subsequently stored using the `input_formula` or `input` commands belong to the group, until a corresponding `end_group` command is encountered. `delete_group` removes all the formulae that belong to a named group from the store.

Formula Manipulation: `deactivate, activate, deactivate_group, activate_group, set_role`
`deactivate` causes a named formula to become inactive, and `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 [5] of a named formula to a given value, e.g. `axiom`, `conjecture`, `lemma`

Environment Variable Manipulation `setenv, unsetenv, waitenv`
`setenv` sets an environment variable to a given value. `unsetenv` deletes an environment variable with a given name. `waitenv` waits for environment variables with given names to be set.

Execution `execute, execute_async, filter, generate`
`execute` runs an ATP system (or anything!) in the foreground. An environment variable can be provided to capture the SZS status [4] of the completed execution, e.g., *Satisfiable* or *Theorem*. The logical formulae to be processed by the ATP system are selected using `$getgroups()` terms. `execute_async` runs like `execute` except that the ATP system is run in the background. `filter` runs like `execute`, then `delete`s the formulae specified by the `$getgroups()` terms, and `input`s the formulae output by the ATP system. `generate` runs like `execute`, and `input`s the formulae output by the ATP system.

Utility Commands `mktemp, write, assert, exit`
`mktemp` creates a new temporary file and sets an environment variable to the file name. `write` outputs data to `stdout`. `assert` checks the syntactic 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.

3 The TPI Language Interpreter

A TPI language interpreter for executing TPI commands has been developed in `perl`. The interpreter is a standalone program that recognizes commands of the following form:

```
tpi command_name command_details
```

where `tpi` is the name of the interpreter. The interpreter can be executed from the user's command shell or from within a shell script, to execute one TPI command at a time. Shell scripting with TPI commands provides a fully featured language for ATP processing.

Logical formulae that are input to the interpreter are stored in a text file called `DB_File`. The file provides persistent storage of the formulae between executions of the interpreter. The input commands add formulae to the `DB_File`, and the delete commands remove formulae from the `DB_File`. Groups are implemented by inserting comment lines `%TPI start_group: group_name` and `%TPI end_group: group_name` before and after the formulae in the `DB_File`. If the formulae belonging to a group are not contiguous when being input, multiple `%TPI start_group` and `%TPI end_group` comments are used. The role of a formulae is changed by an edit in the `DB_File`. Formulae are deactivated by commenting them out in the `DB_File`, and reactivated by uncommenting them.

The TPI language expects its environment variables to have the same properties as regular command shell environment variables, i.e., to be global and persistent. However, as the TPI 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, i.e., they do not naturally persist. To provide persistence of its environment variables, the interpreter saves/deletes its environment variables in a file called `ENV_VARS` whenever a `setenv`/`unsetenv` command is used. At the start of each execution the interpreter augments the environment variables that it has inherited from its parent process with the values stored in the file. If a conflict is detected, i.e., one of the parent's environment variables has the same name as a variable in the `ENV_VARS` file, the parent's variable takes precedence and the new value is written to the `ENV_VARS` file.

The execution commands are implemented by starting a new process for the ATP system, and the resultant SZS value is extracted by parsing the `stdout` from the ATP system, expecting the result to be output according to SZS conventions.

4 TPI Applications

4.1 The ATP Process

The ATP process illustrated in Figure 1 shows a comprehensive flow of ATP processing that should be followed when trying to prove a theorem. It provides more structure and allows for earlier error detection compared to giving the axioms and conjecture directly to a theorem proving ATP system. The initial step is to convert the world knowledge of the domain in question to axioms and a conjecture in a chosen logic, and writing them in TPTP syntax. The ATP process then starts by checking the syntactic correctness of the formulae, and then checking the axioms' satisfiability. 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* set of axioms can be found vacuously – this is typically not the user's intention. 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 conjecture is not a *Theorem* of the axioms can be performed – if successful this provides useful feedback to the user.

Script 2 provides a `tcsh` script that uses TPI commands to implement this ATP process. The axioms and conjecture are added using the `input` command (label A in Figure 1, line 1 in Script 2). Using the `execute` command, a syntax checker system is executed to check the syntax of the formulae (label B, line 2). If the syntax checker returns a `SyntaxError` status (label C, line 4), the user is informed using the `write` command and the process is terminated

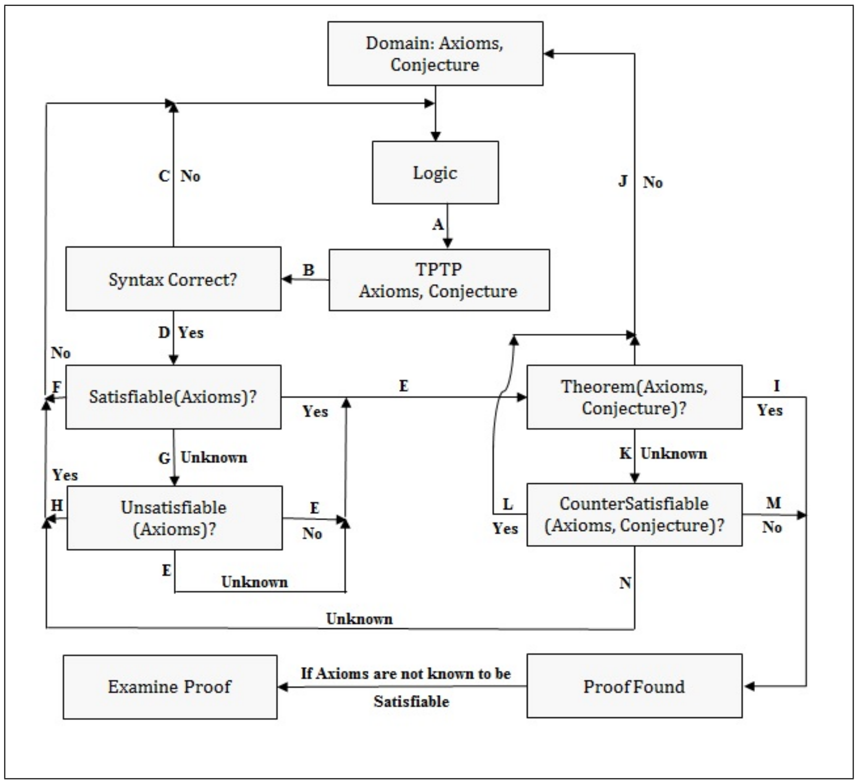

Figure 1: The ATP Process

using the `exit` command (lines 5-6).

Once the syntax of the formulae has been checked, 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 8). If the ATP system shows that the axioms are *Satisfiable* (label E, line 10), the process moves on to trying to prove that the conjecture is a *Theorem* of the axioms (lines 11-32). Otherwise, if it shows that the axioms are *Unsatisfiable* (label F, line 33), the user is informed using the `write` command and the process is terminated using the `exit` command (lines 34-35). 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 37). If the ATP system shows that the axioms are *Unsatisfiable* (label H, line 39), the user is informed using the `write` command and the process is terminated using the `exit` command (lines 40-41). If the system shows that the axioms are not *Unsatisfiable*, or if a decision cannot be reached (line 42), the process again moves on to trying to prove that the conjecture is a *Theorem* of the axioms (label E, line 43). (This is an optimistic approach that assumes satisfiability of the axioms if nothing can be shown explicitly.)

```
tpi input $1
tpi execute ’SZS_Syntax’ = ’tptp4X -q3 -z $getgroups(tpi)’
set SyntaxResult = ‘tpi write ’$getenv(SZS_Syntax)’‘
if ("$SyntaxResult" == "SyntaxError") then
    echo "% Syntax Error!"
    tpi exit
else
    tpi execute ’SZS_Satisfiability’ = ’iproveropt_run_sat.sh 300
       $getgroups(tpi_premises)’
    set SatResult = ‘tpi write ’$getenv(SZS_Satisfiability)’‘
    if("$SatResult" == "Satisfiable") then
       repeatedSection:
       tpi execute ’SZS_Proof’ = ’eprover --auto --cpu-limit=300
$getgroups(tpi)’
       set ProofResult = ‘tpi write ’$getenv(SZS_Proof)’‘
       if("$ProofResult" == "Theorem") then
          echo "% Proof Found!
          tpi exit
       else
          if( "$ProofResult" == "CounterSatisfiable") then
             echo "% No Proof!"
             tpi exit
          else
             tpi execute ’SZS_CSA_AxConj’
               = ’iproveropt_run_sat.sh 300 $getgroups(tpi)’
             set CSAResultAxConj = ‘tpi write ’ $getenv(’SZS_CSA_AxConj’)’‘
             if( "$CSAResultAxConj" == "CounterSatisfiable") then
                echo "% No Proof!"
                tpi exit
             else if("$CSAResultAxConj" == "Theorem")
                echo "% Proof Found!"
                tpi exit
             else
                echo "% No Proof!"
                tpi exit
    else if("$SatResult" == "Unsatisfiable")
       echo "% Axioms Unsatisfiable!"
       tpi exit
    else
       tpi execute ’SZS_Unsatisfiability’ = ’eprover --auto --cpu-limit=300
          $getgroups(tpi_premises)’
       set UnsatResult = ‘tpi write ’$getenv(SZS_Unsatisfiability)’‘
       if("$UnsatResult" == "Unsatisfiable")
          echo "% Axioms Unsatisfiable!"
          tpi exit
       else
          goto repeatedSection
```

Script 2: `tcsh` Script Implementing the ATP Process

Once the axioms have been found to be *Satisfiable*, or if it could not be shown 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 formulae (label E, line 12). If a proof is found (label I, line 14), a *Theorem* status is output using the `write` command, and the process ends using the `exit` command (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 (label J, line 18), the user is informed using the `write` command and the process is terminated using the `exit` command (lines 19-20). If a decision cannot be reached (label K, line 21), the process moves on to trying to prove that the axioms and conjecture are *CounterSatisfiable* (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* (line 22). If the ATP system shows that the axioms and conjecture are *CounterSatisfiable* (label L, line 24), the user is informed using the `write` command and the process is terminated using the `exit` command (lines 25-26). If a proof of the conjecture is found (label M, line 27), a *Theorem* status is output using the `write` command, and the process ends using the `exit` command (lines 28-29). Otherwise, if a decision cannot be reached (label N, line 30), the user is informed using the `write` command and the process is terminated using the `exit` command (lines 31-32).

4.2 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 on a problem. Many ATP systems employ strategy scheduling [7], which slices this time among different search strategies, and tries to solve the problem by trying each of the strategies ith its time slice, until one solves the problem or all have failed. Providing a strategy scheduling ATP system with less time may result in the system solving the problem in less time. Figure 2 illustrates how this can occur, using the strategy scheduling approach introduced in version 1.8 of the E ATP system. E's strategy scheduling chooses four strategies. One eighth of the overall time limit is given to each of the first two (weakest) strategies, one quarter to the third strategy, and one half to the last (strongest) strategy. In the example the overall time limit is 120s. The first three strategies fail to solve the problem in their allocated time slices, but the fourth strategy succeeds using 25s of its 60s slice. A total of 85s is used by the four strategies. If the time limit is then halved to 60s, a solution is found using a total of 55s is used. If the time limit is halved again, no solution is found because the fourth strategy is allocated only 15s. The time limit can then be increased again, here to 45s – half way between the successful 60s limit and the unsuccessful 30s limit, but again no solution is found because the fourth strategy is allocated only 22.5s. Increasing the limit again to half way between the successful 60s limit and the unsuccessful 45s limit produces a solution.

Script 3 provides a `tcsh` script that uses TPI commands to implement this "binary search" for a minimal time limit. The script starts by collecting information needed from the user: the name of the problem file, the overall time limit, and a constant value `Epsilon` whose use will be explained shortly (lines 1-3). The E ATP system is then executed using the `execute` command, bounded by the overall time limit to try to solve the given problem (line 4). If it times out (line 6), the algorithm halts (line 7). Otherwise, two variables are initialized (lines 9-10): the `LowerBound` variable that holds the last reported time limit at which the given ATP system timed out when trying to solve the problem, and the `Upperbound` variable that holds the last reported time limit at which the given ATP system solved the problem. Then, while the difference between the `UpperBound` and `LowerBound` is greater than `Epsilon` (line 11), the

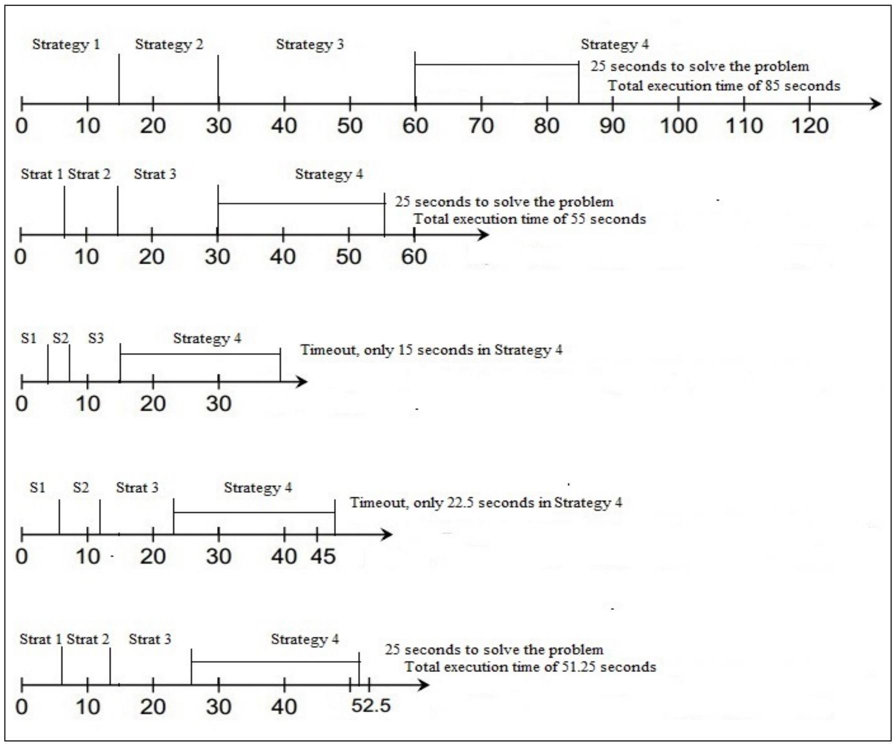
Strategy 1
Strategy 2
Strategy 3
Strategy 4
25 seconds to solve the problem
Total execution time of 85 seconds
0 10 20 30 40 50 60 70 80 90 100 110 120
Strat 1 Strat 2 Strat 3
Strategy 4
25 seconds to solve the problem
Total execution time of 55 seconds
0 10 20 30 40 50 60
S1 S2 S3
Strategy 4
Timeout, only 15 seconds in Strategy 4
0 10 20 30
S1 S2 Strat 3
Strategy 4
Timeout, only 22.5 seconds in Strategy 4
0 10 20 30 40 45
Strat 1 Strat 2 Strat 3
Strategy 4
25 seconds to solve the problem
Total execution time of 51.25 seconds
0 10 20 30 40 52.5

Figure 2: Strategy Scheduling Example

time limit is set to be midway between the `LowerBound` and `UpperBound` (line 12), and the ATP system is executed again, bounded by the new time limit (line 13). If the ATP system solves the problem (line 15), the `UpperBound` is set to the time limit (line 16). If the ATP system times out (line 17), the `LowerBound` is set to the time limit (line 18). This process is repeated until the difference between `UpperBound` and `LowerBound` becomes smaller than `Epsilon` (line 11). Upon exiting the loop the `UpperBound` is the time limit at which the system last solved the problem.

5 Conclusion

This paper has described the TPI language, introduced a shell interpreter for it, and demonstrated how useful they can be in theorem proving. The TPI language provides command and control instructions for manipulating logical formulae, and makes it possible to use ATP systems in a fully automated manner. 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.

```
set Problem = $1
set TimeLimit = $2
set Epsilon = $3
tpi execute -q2 ’SZSSchedule’ = "eprover --cpu-limit=$TimeLimit $Problem"’’
set ProofResult = ‘tpi write ’$getenv(SZSSchedule)’‘
if ("$ProofResult" == "Timeout")
    echo "Problem could not be solved in the time provided!"
else
    set LowerBound = 0
    set UpperBound = $TimeLimit
    while (($UpperBound - $LowerBound) > $Epsilon)
       @ TimeLimit = ($UpperBound + $LowerBound) / 2
       tpi execute -q2 ’SZSSchedule’ = "eprover --cpu-limit=$TimeLimit
$Problem"
       set ProofResult = ‘tpi write ’$getenv(SZSSchedule)’‘
       if ("$ProofResult" == "Theorem")
          set UpperBound = $TimeLimit
       if ("$ProofResult" == "Timeout")
          set LowerBound = $TimeLimit
    end
    echo "Final time limit is:  $UpperBound"
```

Script 3: `tcsh` Script to find the Minimal Time to Solve an ATP Problem

References

[1] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories*, 2010.

[2] N. Eén and N. Sörensson. MiniSat - A SAT Solver with Conflict-Clause Minimization. In F. Bacchus and T. Walsh, editors, *Posters of the 8th International Conference on Theory and Applications of Satisfiability Testing*, 2005.

[3] W.W. McCune. Otter 3.3 Reference Manual. Technical Report ANL/MSC-TM-263, Argonne National Laboratory, Argonne, USA, 2003.

[4] G. Sutcliffe. The SZS Ontologies for Automated Reasoning Software. In G. Sutcliffe, P. Rudnicki, R. Schmidt, B. Konev, and S. Schulz, editors, *Proceedings of the LPAR Workshops: Knowledge Exchange: Automated Provers and Proof Assistants, and The 7th International Workshop on the Implementation of Logics*, number 418 in CEUR Workshop Proceedings, pages 38–49, 2008.

[5] G. Sutcliffe. The TPTP World - Infrastructure for Automated Reasoning. In E. Clarke and A. Voronkov, editors, *Proceedings of the 16th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, number 6355 in Lecture Notes in Artificial Intelligence, pages 1–12. Springer-Verlag, 2010.

[6] G. Sutcliffe. The TPTP Process Instruction Language. In B. Konev, S. Schulz, and G. Sutcliffe, editors, *Proceedings of the 10th International Workshop on the Implementation of Logics*, 2013.

[7] A. Wolf. Strategy Selection for Automated Theorem Proving. In F. Giunchiglia, editor, *Proceedings of the 8th International Conference on Artificial Intelligence: Methodology, Systems, Applications (AIMSA)*, number 1480 in Lecture Notes in Artificial Intelligence, pages 452–465. Springer-Verlag, 1998.