The CADE-20 ATP System Competition (CASC-20)

Geoff Sutcliffe
Department of Computer Science
University of Miami
geoff@cs.miami.edu

June 28, 2005

Abstract

The CADE ATP System Computer (CASC) evaluates the performance of sound, fully automatic, classical first-order logic, ATP systems. The evaluation is in terms of the number of problems solved, the number of acceptable proofs and models produced, and the average runtime for problems solved, in the context of a bounded number of eligible problems chosen from the TPTP problem library, and a specified time limit for each solution attempt. The CADE-20 ATP System Competition (CASC-20) will be held on 26th July 2005. The design of the competition and it's rules, and information regarding the competing systems, are provided in this report.

1 Introduction

The CADE conferences are the major forum for the presentation of new research in all aspects of automated deduction. In order to stimulate ATP research and system development, and to expose ATP systems within and beyond the ATP community, the CADE ATP System Competition (CASC) is held at each CADE conference. CASC-20 will be held at the 20th International Joint Conference on Automated Deduction, on 26th July 2005. CASC-20 is the tenth such ATP system competition [49, 57, 53, 38, 40, 48, 46, 47, 44].

CASC evaluates the performance of sound, fully automatic, classical first-order logic, ATP systems. The evaluation is in terms of:

- the number of problems solved,
- the number of acceptable proofs and models produced, and
- the average runtime for problems solved;

in the context of:

- a bounded number of eligible problems, chosen from the TPTP problem library [52], and
- a specified time limit for each solution attempt.

Twenty-one ATP systems and variants, listed in Table 1, are entered into the various competition and demonstration divisions. The winners of the CASC-J2 (the previous CASC) divisions have been automatically entered into those divisions, to provide benchmarks against which progress can be judged (the competition archive provides access to the systems' executables and source code).

The design and procedures of this CASC evolved from those of previous CASCs [56, 50, 45, 51, 36, 37, 39, 41, 42, 43]. Important changes for this CASC are:

- For systems running on entrant supplied hardware in the demonstration division, entrants must deliver a source code package to the competition organizers by the start of the competition. The entrant specifies whether or not the source code is placed on the CASC WWW site after the competition.

The competition organizers are Geoff Sutcliffe and Christian Suttner. The competition is overseen by a panel of knowledgeable researchers who are not participating in the event; the panel members are Uli Furbach, Roberto Nieuwenhuis, and John Slaney. The rules, specifications, and deadlines given here are absolute. Only the panel has the right to make exceptions. The competition will be run on computers provided by the Department of Computer Science at the University of Manchester. The CASC-20 WWW site provides access to resources used before, during, and after the event:

`http://www.tptp.org/CASC/20`

2 Divisions

CASC is divided into divisions according to problem and system characteristics. There are competition divisions in which systems are explicitly ranked, and a demonstration division in which systems demonstrate their abilities without being formally ranked. Some divisions are further divided into problem categories. The problem categories are defined in order to make it possible to analyze, at a more fine grained level, which systems work well for what types of problems. The problem categories have no effect on the competition rankings, which are made at only the division level.

2.1 The Competition Divisions

Each competition division uses problems that have certain logical, language, and syntactic characteristics, so that the ATP systems that compete in the division are, in principle, able to attempt all the problems in the division.

The **MIX** division: Mixed CNF really non-propositional theorems (unsatisfiable clause sets). *Mixed* means Horn and non-Horn problems, with or without equality, but not unit equality problems (see the UEQ division below). *Really non-propositional* means with an infinite Herbrand universe. The MIX Division has five problem categories:

- The **HNE** category: Horn with No Equality
- The **HEQ** category: Horn with some (but not pure) Equality
- The **NNE** category: Non-Horn with No Equality
- The **NEQ** category: Non-Horn with some (but not pure) Equality
- The **PEQ** category: Pure Equality

Table 1: The ATP systems and entrants

ATP System	Divisions	Entrants	Affiliation
Darwin 1.2	MIX SAT* EPR	Alexander Fuchs, Peter Baumgartner, Cesare Tinelli	University of Iowa, Max-Planck-Institut für Informatik, University of Iowa
DCTP 10.21p	EPR	CASC	*CASC-J2 EPR winner*
E 0.9pre	MIX FOF SAT EPR UEQ	Stephan Schulz	Technische Universität München
EP 0.9pre	MIX* FOF*		*E 0.9pre variant*
Equinox 1.0	MIX FOF EPR	Koen Claessen	Chalmers University of Technology
Gandalf c-2.6-SAT	SAT	CASC	*CASC-J2 SAT winner*
Mace2 2.2	SAT*	William McCune	Argonne National Laboratory
Mace4 2004-D	SAT*	William McCune	Argonne National Laboratory
MathServ 0.62	MIX FOF SAT EPR UEQ (demo)	Jürgen Zimmer Serge Autexier	Universität des Saarlandes
MUSCADET 2.5	FOF	Dominque Pastre	Université René Descartes - Paris
Octopus JN05	MIX FOF (demo)	Monty Newborn, Zongyan Wang	McGill University
Otter 3.3	MIX* FOF UEQ	CASC, William McCune	Argonne National Laboratory
Paradox 1.0	SAT*	CASC	*CASC-J2 SAT winner*
Paradox 1.3	SAT*	Koen Claessen, Niklas Sörensson	Chalmers University of Technology
Paradox 1.3-P	EPR		*Paradox 1.3 variant*
THEO JN05	MIX FOF	Monty Newborn	McGill University
Vampire 7.0	MIX* FOF*	CASC	*CASC-J2 MIX*, FOF* winner*
Vampire 8.0	MIX* FOF* EPR UEQ	Andrei Voronkov	University of Manchester
Waldmeister 704	UEQ	CASC	*CASC-J2 UEQ winner*
Wgandalf 0.1	MIX FOF EPR UEQ	Tanel Tammet	Tallinn University of Technology

MIX* indicates participation in the MIX division proof class,
FOF* indicates participation in the FOF division proof class, and
SAT* indicates participation in the SAT division model class - see Section 2.

The **FOF** division: Mixed FOF non-propositional theorems (axioms with a provable conjecture). The FOF Division has two problem categories:

- The FNE category: FOF with No Equality
- The FEQ category: FOF with Equality

The **SAT** division: Mixed CNF really-non-propositional non-theorems (satisfiable clause sets). The SAT Division has two problem categories:

- The SNE category: SAT with No Equality
- The SEQ category: SAT with Equality

The **EPR** division: CNF effectively propositional theorems and non-theorems. *Effectively propositional* means non-propositional with a finite Herbrand Universe. The EPR Division has two problem categories:

- The EPT category: Effectively Propositional Theorems (unsatisfiable clauses)
- The EPS category: Effectively Propositional non-theorems (Satisfiable clauses)

The **UEQ** division: Unit equality CNF really non-propositional theorems.

Section 3.2 explains what problems are eligible for use in each division and category. Section 4 explains how the systems are ranked in each division.

2.2 The Demonstration Division

ATP systems that cannot run on the general hardware, or cannot be entered into the competition divisions for any other reason, can be entered into the demonstration division. Demonstration division systems can run on the general hardware, or the hardware can be supplied by the entrant. Hardware supplied by the entrant may be brought to CASC, or may be accessed via the internet.

The entry specifies which competition divisions' problems are to be used. The results are presented along with the competition divisions' results, but may not be comparable with those results.

3 Infrastructure

3.1 Computers

The general hardware is TO BE CHECKED Dell computers, each having:

- AMD Athlon XP 2200+, 1797MHz CPU
- 512MB memory
- Linux 2.4.20-30.9 operating system

3.2 Problems

3.2.1 Problem Selection

The problems are from the TPTP problem library, version v3.1.0. The TPTP version used for the competition is not released until after the system installation deadline, so that new problems have not seen by the entrants.

The problems have to meet certain criteria to be eligible for selection:

- The TPTP uses system performance data to compute problem difficulty ratings, and from the ratings classifies problems as one of [54]:
  - Easy: Solvable by all state-of-the-art ATP systems
  - Difficult: Solvable by some state-of-the-art ATP systems
  - Unsolved: Solvable by no ATP systems
  - Open: Theoremhood unknown

  Difficult problems with a rating in the range 0.21 to 0.99 are eligible. Performance data from systems submitted by the system submission deadline is used for computing the problem ratings for the TPTP version used for the competition.
- The TPTP distinguishes versions of problems as one of standard, incomplete, augmented, especial, or biased. All except biased problems are eligible.

The problems used are randomly selected from the eligible problems at the start of the competition, based on a seed supplied by the competition panel.

- The selection is constrained so that no division or category contains an excessive number of very similar problems.
- The selection mechanism is biased to select problems that are new in the TPTP version used, until 50% of the problems in each category have been selected, after which random selection (from old and new problems) continues. The actual percentage of new problems used depends on how many new problems are eligible and the limitation on very similar problems.

3.2.2 Number of Problems

The minimal numbers of problems that have to be used in each division and category, to ensure sufficient confidence in the competition results, are determined from the numbers of eligible problems in each division and category [8] (the competition organizers have to ensure that there is sufficient CPU time available to run the ATP systems on this minimal number of problems). The minimal numbers of problems is used in determining the CPU time limit imposed on each solution attempt - see Section 3.3.

A lower bound on the total number of problems to be used is determined from the number of computers available, the time allocated to the competition, the number of ATP systems to be run on the general hardware over all the divisions, and the CPU time limit, according to the following relationship:

$$NumberOfProblems = \frac{NumberOfComputers * TimeAllocated}{NumberOfATPSystems * CPUTimeLimit}$$

It is a lower bound on the total number of problems because it assumes that every system uses all of the CPU time limit for each problem. Since some solution attempts succeed before the CPU time limit is reached, more problems can be used.

The numbers of problems used in the categories in the various divisions is (roughly) proportional to the numbers of eligible problems than can be used in the categories, after taking into account the limitation on very similar problems.

The numbers of problems used in each division and category are determined according to the judgement of the competition organizers.

3.2.3 Problem Preparation

In order to ensure that no system receives an advantage or disadvantage due to the specific presentation of the problems in the TPTP, the `tptp2X` utility (distributed with the TPTP) is used to:

- rename all predicate and function symbols to meaningless symbols
- randomly reorder the clauses and literals in CNF problems
- randomly reorder the formulae in FOF problems
- randomly reverse the unit equalities in UEQ problems
- remove equality axioms that are not needed by the ATP systems
- add equality axioms that are needed by the ATP systems
- output the problems in the formats required by the ATP systems. (The clause type information, one of `axiom`, `hypothesis`, or `conjecture`, may be included in the final output of each formula.)

Further, to prevent systems from recognizing problems from their file names, symbolic links are made to the selected problems, using names of the form `CCCNNN-1.p` for the symbolic links, with `NNN` running from `001` to the number of problems in the respective division or category. The problems are specified to the ATP systems using the symbolic link names.

In the demonstration division the same problems are used as for the competition divisions, with the same `tptp2X` transformations applied. However, the original file names are retained.

3.3 Resource Limits

In the competition divisions, CPU and wall clock time limits are imposed on each solution attempt. A minimal CPU time limit of 240 seconds is used. The maximal CPU time limit is determined using the relationship used for determining the number of problems, with the minimal number of problems as the $NumberOfRoblems$. The CPU time limit is chosen as a reasonable value within the range allowed, and is announced at the competition. The wall clock time limit is imposed in addition to the CPU time limit, to prevent very high memory usage that causes swapping. The wall clock time limit is double the CPU time limit.

In the demonstration division, each entrant can choose to use either a CPU or a wall clock time limit, whose value is the CPU time limit of the competition divisions.

4 System Evaluation

All the divisions have an assurance ranking class, ranked according to the number of problems solved (a "yes" output, giving an assurance of the existence of a proof). The MIX, FOF, and SAT divisions additionally have and a proof/model ranking class, ranked according to the number of problems solved with an acceptable proof/model output on `stdout`. Ties are broken according to the average CPU times over problems solved. All systems are automatically ranked in the assurance classes, and are ranked in the proof/model classes if they output acceptable proofs/models.

During the competition, for each ATP system, for each problem attempted, three items of data are recorded: whether or not a solution was found, the CPU time taken, and whether or not a solution (proof or model) was output on `stdout`. The systems are ranked from this performance data. Division and class winners are announced and prizes are awarded.

The competition panel decides whether or not the systems' proofs and models are acceptable. The criteria include:

- Derivations must be complete, starting at formulae from the problem, and ending at the conjecture (for axiomatic proofs) or a *false* formula (for proofs by contradiction, including CNF refutations).
- For proofs of FOF problems by CNF refutation, the conversion from FOF to CNF must be adequately documented.
- Derivations must show only relevant inference steps.
- Inference steps must document the parent formulae, the inference rule used, and the inferred formula.
- Inference steps must be reasonably fine-grained.
- An unsatisfiable set of ground instances of clauses is acceptable for establishing the unsatisfiability of a set of clauses.
- Models must be complete, documenting the domain, function maps, and predicate maps. The domain, function maps, and predicate maps may be specified by explicit ground lists (of mappings), or by any clear, terminating algorithm.

In the assurance classes, and the SAT and EPR divisions, the ATP systems are not required to output solutions (proofs or models). However, systems that do output solutions to `stdout` are highlighted in the presentation of results.

If a system is found to be unsound during or after the competition, but before the competition report is published, and it cannot be shown that the unsoundness did not manifest itself in the competition, then the system is retrospectively disqualified. At some time after the competition, all high ranking systems in each division are tested over the entire TPTP. This provides a final check for soundness (see Section 6 Properties regarding soundness checking before the competition). At some time after the competition, the proofs from the winners of the MIX and FOF division proof classes, and the models from the winner of the SAT division model class, are checked by the panel. If any of the proofs or models are unacceptable, i.e., they are significantly worse than the samples provided, then that system is retrospectively disqualified. All disqualifications are explained in the competition report.

5 System Entry

To be entered into CASC, systems have to be registered using the CASC system registration form. No registrations are accepted after the registration deadline. For each system entered, an entrant has to be nominated to handle all issues (including execution difficulties) arising before and during the competition. The nominated entrant must formally register for CASC. However, it is not necessary for entrants to physically attend the competition.

Systems can be entered at only the division level, and can be entered into more than one division (a system that is not entered into a competition division is assumed to perform worse than the entered systems, for that type of problem - wimping out is not an option). Entering many similar versions of the same system is deprecated, and entrants may be required to limit the number of system versions that they enter. The division winners from the previous CASC are automatically entered into their divisions, to provide benchmarks against which progress can be judged.

It is assumed that each entrant has read the WWW pages related to the competition, and has complied with the competition rules. Non-compliance with the rules could lead to disqualification. A "catch-all" rule is used to deal with any unforseen circumstances: *No cheating is allowed.* The panel is allowed to disqualify entrants due to unfairness, and to adjust the competition rules in case of misuse.

5.1 System Description

A system description has to be provided for each ATP system entered, using the HTML schema supplied on the CASC WWW site. The system description must fit onto two pages, using 12pt times font. The schema has the following sections:

- Architecture. This section introduces the ATP system, and describes the calculus and inference rules used.
- Implementation. This section describes the implementation of the ATP system, including the programming language used, important internal data structures, and any special code libraries used.
- Strategies. This section describes the search strategies used, why they are effective, and how they are selected for given problems. Any strategy tuning that is based on specific problems' characteristics must be clearly described (and justified in light of the tuning restrictions).
- Expected competition performance. This section makes some predictions about the performance of the ATP system in each of the divisions and categories in which the system is competing.
- References.

The system description has to be emailed to the competition organizers before the system description deadline. The system descriptions, along with information regarding the competition design and procedures, form the proceedings for the competition.

5.2 Sample Solutions

For systems in the MIX and FOF division proof classes, and the SAT division model class, representative sample solutions must be emailed to the competition organizers before the sample solutions deadline. Proof samples for the MIX division must include a proof for `SYN075-1`. Proof samples for the FOF division must include a proof for `SYN075+1`. Model samples for the SAT division must include a model for `MGT031-1`. The sample solutions must illustrate the use of all inference rules. A key must be provided if any non-obvious abbreviations for inference rules or other information are used.

6 System Properties

Systems are required to have the following properties:

- The ATP systems have to run on a single locally provided standard UNIX computer (the *general hardware* - see Section 3.1). ATP systems that cannot run on the general hardware can be entered into the demonstration division.
- Systems have to be fully automatic, i.e., any command line switches have to be the same for all problems.
- Systems have to be sound. At some time before the competition all the systems in the competition divisions are tested for soundness. Non-theorems are submitted to the systems in the MIX, FOF, EPR, and UEQ divisions, and theorems are submitted to the systems in the SAT and EPR divisions. Finding a proof of a non-theorem or a disproof of a theorem indicates unsoundness. If an ATP system fails the soundness testing it must be repaired by the unsoundness repair deadline or be withdrawn. The soundness testing eliminates the possibility of an ATP system simply delaying for some amount of time and then claiming to have found a solution. At some time after the competition, all high ranking systems in the competition divisions are tested over the entire TPTP. This provides a final check for soundness. For systems running on entrant supplied hardware in the demonstration division, the entrant must perform the soundness testing and report the results to the competition organizers.
- Systems do not have to be complete in any sense, including calculus, search control, implementation, or resource requirements.
- The ATP systems have to be executable by a single command line, using an absolute path name for the executable, which might not be in the current directory. The command line arguments are the absolute path name of a symbolic link as the problem file name, the time limit (if required by the entrant), and entrant specified system switches (the same for all problems). No shell features, such as input or output redirection, may be used in the command line. No assumptions may be made about the format of the problem file name.
- The ATP systems that run on the general hardware have to be interruptable by a `SIGXCPU` signal, so that the CPU time limit can be imposed on each solution attempt, and interruptable by a `SIGALRM` signal, so that the wall clock time limit can be imposed on each solution attempt. For systems that create multiple processes, the signal is sent first to the process at the top of the hierarchy, then one second later to all processes in the hierarchy. Any orphan processes are killed after that, using `SIGKILL`. The default action on receiving these signals is to exit (thus complying with the time limit, as required), but systems may catch the signals and exit of their own accord. If a system runs past a time limit this is noticed in the timing data, and the system is considered to have not solved that problem.
- When terminating of their own accord, the ATP systems have to output a distinguished string (specified by the entrant) to `stdout` indicating the result, one of:

  - A solution exists (for CNF problems, the clause set is unsatisfiable, for FOF problems, the conjecture is a theorem)

  - No solution exists (for CNF problems, the clause set is satisfiable, for FOF problems, the conjecture is a non-theorem)
  - No conclusion reached

  Only the first such string is recognized, and accepted as the system's claimed result.
- When outputing proofs for MIX and FOF divisions' proof classes, and models for the SAT division's model class, the start and end of the solution must be identified by distinguished strings (specified by the entrant). These pairs of strings must be different for proofs and models. The string specifying the result must be output before the start of a proof or model.
- If an ATP system terminates of its own accord, it may not leave any temporary or other output files. If an ATP system is terminated by a `SIGXCPU` or `SIGALRM`, it may not leave any temporary or other output files anywhere other than in `/tmp`. Multiple copies of the ATP systems have to be executable concurrently on different machines but in the same (NFS cross mounted) directory. It is therefore necessary to avoid producing temporary files that do not have unique names, with respect to the machines and other processes. An adequate solution is a file name including the host machine name and the process id.
- For practical reasons excessive output from the ATP systems is not allowed. A limit, dependent on the disk space available, is imposed on the amount of `stdout` and `stderr` output that can be produced. The limit is at least 10KB per problem (averaged over all problems so that it is possible to produce *some* long proofs).
- The precomputation and storage of any information specifically about TPTP problems is not allowed. Strategies and strategy selection based on the characteristics of a few specific TPTP problems are not allowed, i.e., strategies and strategy selection must be general purpose and expected to extend usefully to new unseen problems. If automatic strategy learning procedures are used, the learning must ensure that sufficient generalization is obtained, and that no learning at the individual problem level is performed.
- For every problem solved, the system's solution process must be reproducible by running the system again.

Access to the general hardware (or equivalent) is available from the general hardware access deadline. Entrants must install their systems on the general hardware, and ensure that their systems execute in the competition environment, according to the system checks described in Section 6.3. Entrants are advised to perform these checks well in advance of the system installation deadline. This gives the competition organizers time to help resolve any difficulties encountered.

6.1 System Delivery

For systems running on the general hardware, entrants must deliver an installation package to the competition organizers by the installation deadline. The installation package must be a `.tar.gz` file containing the system source code, any other files required for installation, and a `ReadMe` file. The `ReadMe` file must contain:

- Instructions for installation

- Instructions for executing the system
- Format of problem files, in the form of `tptp2X` format and transformation parameters.
- Command line, using `%s` and `%d` to indicate where the problem file name and CPU time limit must appear.
- The distinguished strings output.

The installation procedure may require changing path variables, invoking `make` or something similar, etc, but nothing unreasonably complicated. All system binaries must be created in the installation process; they cannot be delivered as part of the installation package. The system is reinstalled onto the general hardware by the competition organizers, following the instructions in the `ReadMe` file. Installation failures before the installation deadline are passed back to the entrant. After the installation deadline access to the general hardware is denied, and no further changes or late systems are accepted (i.e., deliver your installation package before the installation deadline so if the installation fails you have a chance to fix it!). If you are in doubt about your installation package or procedure, please email the competition organizers.

For systems running on entrant supplied hardware in the demonstration division, entrants must deliver a source code package to the competition organizers by the start of the competition. The source code package must be a `.tar.gz` file containing the system source code.

After the competition all competition division systems' source code, is made publically available on the CASC WWW site. In the demonstration division, the entrant specifies whether or not the source code is placed on the CASC WWW site.

6.2 System Execution

Execution of the ATP systems on the general hardware is controlled by a `perl` script, provided by the competition organizers. The jobs are queued onto the computers so that each computer is running one job at a time. All attempts at the Nth problems in all the divisions and categories are started before any attempts at the (N+1)th problems.

During the competition a `perl` script parses the systems' outputs. If any of an ATP system's distinguished strings are found then the CPU time used to that point is noted. A system has solved a problem iff it outputs its "success" string within the CPU time limit, and a system has produced a proof iff it outputs its "end of solution" string within the CPU time limit. The result and timing data is used to generate an HTML file, and a WWW browser is used to display the results.

The execution of the demonstration division systems is supervised by their entrants.

6.3 System Checks

- Check: The ATP system can be run by an absolute path name for the executable. For example:

```
prompt> pwd
/home/tptp
prompt> which MyATPSystem
```

```
/home/tptp/bin/MyATPSystem
prompt> /home/tptp/bin/MyATPSystem /home/tptp/TPTP/Problems/GRP/GRP001-1.p
Proof found in 147 seconds.
```

- Check: The ATP system accepts an absolute path name of a symbolic link as the problem file name. For example:

```
prompt> cd /home/tptp/tmp
prompt> ln -s /home/tptp/TPTP/Problems/GRP/GRP001-1.p CCC001-1.p
prompt> cd /home/tptp
prompt> /home/tptp/bin/MyATPSystem /home/tptp/tmp/CCC001-1.p
Proof found in 147 seconds.
```

- Check: The ATP system makes no assumptions about the format of the problem file name. For example:

```
prompt> ln -s /home/tptp/TPTP/Problems/GRP/GRP001-1.p \_foo-Blah
prompt> /home/tptp/bin/MyATPSystem \_foo-Blah
Proof found in 147 seconds.
```

- Check: The ATP system can run under the `TreeLimitedRun` program (sources are available from the CASC-J2 WWW site). For example:

```
prompt> which TreeLimitedRun
/home/tptp/bin/TreeLimitedRun
prompt> /home/tptp/bin/TreeLimitedRun -q0 200 400 /home/tptp/bin/MyATPSystem
        /home/tptp/tmp/CCC001-1.p
TreeLimitedRun: ------------------------------------------------
TreeLimitedRun: /home/tptp/bin/MyATPSystem
TreeLimitedRun: CPU time limit is 200s
TreeLimitedRun: WC  time limit is 400s
TreeLimitedRun: PID is 4867
TreeLimitedRun: ------------------------------------------------
Proof found in 147 seconds.
FINAL WATCH: 147.8 CPU 150.0 WC
```

- Check: The ATP system's CPU time can be limited using the `TreeLimitedRun` program. For example:

```
prompt> /home/tptp/bin/TreeLimitedRun -q0 10 20 /home/tptp/bin/MyATPSystem
        /home/tptp/tmp/CCC001-1.p
TreeLimitedRun: ------------------------------------------------
TreeLimitedRun: /home/tptp/bin/MyATPSystem
TreeLimitedRun: CPU time limit is 10s
TreeLimitedRun: WC  time limit is 20s
TreeLimitedRun: PID is 5827
TreeLimitedRun: ------------------------------------------------
CPU time limit exceeded
FINAL WATCH: 10.7 CPU 13.1 WC
```

- Check: The ATP system's wall clock time can be limited using the `TreeLimitedRun` program. For example:

```
prompt> /home/tptp/bin/TreeLimitedRun -q0 20 10 /home/tptp/bin/MyATPSystem
        /home/tptp/tmp/CCC001-1.p
TreeLimitedRun: -----------------------------------------------
TreeLimitedRun: /home/tptp/bin/MyATPSystem
TreeLimitedRun: CPU time limit is 20s
TreeLimitedRun: WC  time limit is 10s
TreeLimitedRun: PID is 5827
TreeLimitedRun: -----------------------------------------------
Alarm clock
FINAL WATCH: 9.7 CPU 10.1 WC
```

- Check: The system outputs a distinguished string when terminating of its own accord. For example, here the entrant has specified that the distinguished string `Proof found` indicates that a solution exists. If appropriate, similar checks should be made for the cases where no solution exists and where no conclusion is reached.

```
prompt> /home/tptp/bin/TreeLimitedRun -q0 200 400 /home/tptp/bin/MyATPSystem
        /home/tptp/tmp/CCC001-1.p
TreeLimitedRun: -----------------------------------------------
TreeLimitedRun: /home/tptp/bin/MyATPSystem
TreeLimitedRun: CPU time limit is 200s
TreeLimitedRun: WC  time limit is 400s
TreeLimitedRun: PID is 5827
TreeLimitedRun: -----------------------------------------------
Proof found in 147 seconds.
FINAL WATCH: 147.8 CPU 150.0 WC
```

- Check: The system outputs distinguished strings at the start and end of its solution. For example, here the entrant has specified that the distinguished strings `START OF PROOF` and `END OF PROOF` identify the start and end of the solution.

```
prompt> /home/tptp/bin/TreeLimitedRun -q0 200 400 /home/tptp/bin/MyATPSystem
        -output_proof /home/tptp/tmp/CCC001-1.p
TreeLimitedRun: -----------------------------------------------
TreeLimitedRun: /home/tptp/bin/MyATPSystem
TreeLimitedRun: CPU time limit is 200s
TreeLimitedRun: WC  time limit is 400s
TreeLimitedRun: PID is 5827
TreeLimitedRun: -----------------------------------------------
Proof found in 147 seconds.
START OF PROOF
     ... acceptable proof here ...
END OF PROOF
FINAL WATCH: 147.8 CPU 150.0 WC
```

- Check: No temporary or other files are left if the system terminates of its own accord, and no temporary or other files are left anywhere other than in `/tmp` if the system is terminated by a `SIGXCPU` or `SIGALRM`. Check in the current directory, the ATP system's directory, the directory where the problem's symbolic link is located, and the directory where the actual problem file is located.

```
prompt> pwd
/home/tptp
prompt> /home/tptp/bin/TreeLimitedRun -q0 200 400 /home/tptp/bin/MyATPSystem
        /home/tptp/tmp/CCC001-1.p
TreeLimitedRun: ------------------------------------------------
TreeLimitedRun: /home/tptp/bin/MyATPSystem
TreeLimitedRun: CPU time limit is 200s
TreeLimitedRun: WC  time limit is 400s
TreeLimitedRun: PID is 13526
TreeLimitedRun: ------------------------------------------------
Proof found in 147 seconds.
FINAL WATCH: 147.8 CPU 150.0 WC
prompt> ls /home/tptp
     ... no temporary or other files left here ...
prompt> ls /home/tptp/bin
     ... no temporary or other files left here ...
prompt> ls /home/tptp/tmp
     ... no temporary or other files left here ...
prompt> ls /home/tptp/TPTP/Problems/GRP
     ... no temporary or other files left here ...
prompt> ls /tmp
     ... no temporary or other files left here by decent systems ...
```

- Check: Multiple concurrent executions do not clash. For example:

```
prompt> (/bin/time /home/tptp/bin/TreeLimitedRun -q0 200 400
        /home/tptp/bin/MyATPSystem /home/tptp/tmp/CCC001-1.p) &
        (/bin/time /home/tptp/bin/TreeLimitedRun -q0 200 400
        /home/tptp/bin/MyATPSystem /home/tptp/tmp/CCC001-1.p)
TreeLimitedRun: ------------------------------------------------
TreeLimitedRun: /home/tptp/bin/MyATPSystem
TreeLimitedRun: CPU time limit is 200s
TreeLimitedRun: WC  time limit is 400s
TreeLimitedRun: PID is 5827
TreeLimitedRun: ------------------------------------------------
TreeLimitedRun: ------------------------------------------------
TreeLimitedRun: /home/tptp/bin/MyATPSystem
TreeLimitedRun: CPU time limit is 200s
TreeLimitedRun: WC  time limit is 400s
TreeLimitedRun: PID is 5829
TreeLimitedRun: ------------------------------------------------
Proof found in 147 seconds.
FINAL WATCH: 147.8 CPU 150.0 WC

Proof found in 147 seconds.
FINAL WATCH: 147.8 CPU 150.0 WC
```

7 The ATP Systems

These system descriptions were written by the entrants.

7.1 Darwin 1.2

Peter Baumgartner[1], Alexander Fuchs[2], Cesare Tinelli[2]
[1]Max-Planck-Institut für Informatik Saarbrücken, Germany,
`baumgart@mpi-sb.mpg.de`
[2]University of Iowa, USA
{`fuchs,tinelli`}`@cs.uiowa.edu`

Architecture

Darwin [3] is an automated theorem prover for first order clausal logic. It is the first implementation of the Model Evolution Calculus [4]. The Model Evolution Calculus lifts the propositional DPLL procedure to first-order logic. One of the main motivations for this approach was the possibility of migrating to the first-order level some of those very effective search techniques developed by the SAT community for the DPLL procedure.

The current version of Darwin implements first-order versions of unit propagation inference rules analogously to a restricted form of unit resolution and subsumption by unit clauses. To retain completeness, it includes a first-order version of the (binary) propositional splitting inference rule.

Proof search in Darwin starts with a default interpretation for a given clause set, which is evolved towards a model or until a refutation is found.

Implementation

The central data structure is the *context*. A context represents an interpretation as a set of first-order literals. The context is grown by using instances of literals from the input clauses. The implementation of Darwin is intended to support basic operations on contexts in an efficient way. This involves the handling of large sets of candidate literals for modifying the current context. The candidate literals are computed via simultaneous unification between given clauses and context literals. This process is sped up by storing partial unifiers for each given clause and merging them for different combinations of context literals, instead of redoing the whole unifier computations. For efficient filtering of unneeded candidates against context literals, discrimination tree or substitution tree indexing is employed. The splitting rule generates choice points in the derivation which are backtracked using a form of backjumping similar to the one used in DPLL-based SAT solvers.

Darwin is implemented in OCaml and has been tested under various Linux distributions (compiled but untested on FreeBSD, MacOS X, Windows). It is available from:

`http://goedel.cs.uiowa.edu/Darwin/`

Strategies

Darwin traverses the search space by iterative deepening over the term depth of candidate literals. Darwin employs a uniform search strategy for all problem classes.

Expected Competition Performance

Darwin is a first implementation for the Model Evolution calculus. We expect its performance to be strong in the EPR divisions; we anticipate performance below average in the MIX division, and weak performance in the SAT division.

7.2 DCTP 10.21p

Gernot Stenz
Technische Universität München, Germany
`stenzg@informatik.tu-muenchen.de`

Architecture

DCTP 1.31 [33] is an automated theorem prover for first order clause logic. It is an implementation of the disconnection calculus described in [5, 11, 34]. The disconnection calculus is a proof confluent and inherently cut-free tableau calculus with a weak connectedness condition. The inherently depth-first proof search is guided by a literal selection based on literal instantiatedness or literal complexity and a heavily parameterised link selection. The pruning mechanisms mostly rely on different forms of *variant deletion* and *unit based strategies*. Additionally the calculus has been augmented by full tableau pruning.

DCTP 10.21p is a strategy parallel version using the technology of E-SETHEO [35] to combine several different strategies based on DCTP 1.31.

Implementation

DCTP 1.31 has been implemented as a monolithic system in the Bigloo dialect of the Scheme language. The most important data structures are perfect discrimination trees, which are used in many variations. DCTP 10.21p has been derived of the Perl implementation of E-SETHEO and includes DCTP 1.31 as well as the E prover as its CNF converter. Both versions run under Solaris and Linux.

We are currently integrating a range of new techniques into DCTP which are mostly based on the results described in [12], as well as a certain form of unit propagation. We are hopeful that these improvements will be ready in time for CASC-J2.

Strategies

DCTP 1.31 is a single strategy prover. Individual strategies are started by DCTP 10.21p using the schedule based resource allocation scheme known from the E-SETHEO system. Of course, different schedules have been precomputed for the syntactic problem classes. The problem classes are more or less identical with the sub-classes of the competition organisers. Again, we have no idea whether or not this conflicts with the organisers' tuning restrictions.

Expected Competition Performance

DCTP 10.21p is the CASC-J2 EPR division winner.

7.3 E and EP 0.9

Stephan Schulz
Technische Universität München, Germany, and ITC/irst, Italy
schulz@eprover.de

Architecture

E 0.9 [30, 32] is a purely equational theorem prover. The core proof procedure operates on formulas in clause normal form, using a calculus that combines superposition (with selection of negative literals) and rewriting. No special rules for non-equational literals have been implemented, i.e., resolution is simulated via paramodulation and equality resolution. The basic calculus is extended with rules for AC redundancy elemination, some contextual simplification, and pseudo-splitting. The latest version of E also supports simultaneous paramodulation, either for all inferences or for selected inferences.

E is based on the DISCOUNT-loop variant of the *given-clause* algorithm, i.e. a strict separation of active and passive facts. Proof search in E is primarily controlled by a literal selection strategy, a clause evaluation heuristic, and a simplification ordering. The prover supports a large number of preprogrammed literal selection strategies, many of which are only experimental. Clause evaluation heuristics can be constructed on the fly by combining various parameterized primitive evaluation functions, or can be selected from a set of predefined heuristics. Supported term orderings are several parameterized instances of Knuth-Bendix-Ordering (KBO) and Lexicographic Path Ordering (LPO).

The prover uses a preprocessing step to convert formulas in full first order format to clause normal form. Preprocessing also unfolds equational definitions and performs some simplifications on the clause level. The automatic mode can select the literal selection strategy, term ordering, and search heuristic based on simple problem characteristics of the preprocessed clausal problem.

EP 0.9 is just a combination of E 0.9 in verbose mode and a proof analysis tool extracting the used inference steps.

Implementation

E is implemented in ANSI C, using the GNU C compiler. The most outstanding feature is the global sharing of rewrite steps. Current versions of E add *rewrite links* from rewritten to new terms. In effect, E is caching rewrite operations as long as sufficient memory is available. Other important features are the use of *perfect discrimination trees* with age and size constraints for rewriting and unit-subsumption, *feature vector indexing* [31] for forward and backward subsumption and contextual literal cutting, and a new polynomial implementation of LPO [13].

The program has been successfully installed under SunOS 4.3.x, Solaris 2.x, HP-UX B 10.20, MacOS-X, and various versions of Linux. Sources of the latest released version are available freely from:

`http://www.eprover.org`

EP 0.9 is a simple Bourne shell script calling E and the postprocessor in a pipeline.

Strategies

E's automatic mode is optimized for performance on TPTP v3.0.1. The optimization is based on about 90 test runs over the library (and previous experience) and consists of the selection of one of about 40 different strategies for each problem. All test runs have been performed on SUN Ultra 60/300 machines with a time limit of 300 seconds (or roughly equivalent configurations). All individual strategies are general purpose, the worst one solves about 49% of TPTP v3.0.1, the best one about 60%.

E distinguishes problem classes based on a number of features, all of which have between two and four possible values. The most important ones are:

- Is the most general non-negative clause unit, Horn, or Non-Horn?
- Is the most general negative clauce unit or non-unit?
- Are all negative clauses unit clauses?
- Are all literals equality literals, are some literas equality literals, or is the problem non-equational?
- Are there a few, some, or many clauses in the problem?
- Is the maximum arity of any function symbol 0, 1, 2, or greater?
- Is the sum of function symbol arities in the signature small, medium, or large?

Wherever there is a three-way split on a numerical feature value, the limits are selected automatically with the aim of splitting the set of problems into approximately equal sized parts based on this one feature.

For classes above a threshold size, we assign the absolute best heuristic to the class. For smaller, non-empty classes, we assign the globally best heuristic that solves the same number of problems on this class as the best heuristic on this class does. Empty classes are assigned the globally best heuristic. Typically, most selected heuristics are assigned to more than one class.

Expected Competition Performance

In the last years, E performed well in most proof categories. We believe that E will again be among the strongest provers in the MIX division, in particular due to its good performance for Horn problems. Since we are currently working on a new CNF translator, we cannot predict performance on FOF problems yet, but hope that E will be competitive. Similarly, we have no experience with satisfiable problems, but hope that E will at least be a useful complement to dedicated systems.

EP0.9 will be hampered by the fact that it has to analyse the inference step listing, an operation that typically is about as expensive as the proof search itself. Nevertheless, it should be competitive among the MIX and FOF proof class systems.

7.4 Equinox 1.0

Koen Claessen
Chalmers University of Technology, Sweden
`koen@cs.chalmers.se`

Architecture

Equinox is a new theorem prover for pure first-order logic with equality. It finds ground proofs of the input theory, by solving successive ground instantiations of the theory using an incremental SAT-solver. Equality is dealt with using a Nelson-Oppen framework.

Implementation

The main part of Equinox is implemented in Haskell using the GHC compiler. Equinox also has a built-in incremental SAT solver which is written in C++. The two parts are linked together on the object level using Haskell's Foreign Function Interface.

Strategies

There is only one strategy in Equinox:

1. Give all ground clauses in the problem to a SAT solver.
2. Run the SAT-solver.
3. If a contradiction is found, we have a proof and we terminate.
4. If a model is found, we use the model to indicate which new ground instantiations should be added to the SAT-solver.
5. Goto 2.

Expected Competition Performance

Equinox is still in its infancy. There should however be problems that it can solve that few other provers can handle.

7.5 Gandalf c-2.6-SAT

Tanel Tammet
Tallinn Technical University, Estonia
`tammet@cc.ttu.ee`

Architecture

Gandalf [58, 59] is a family of automated theorem provers, including classical, type theory, intuitionistic and linear logic provers, plus finite a model builder. The version c-2.6 contains the classical logic prover for clause form input and the finite model builder. One distinguishing feature of Gandalf is that it contains a large number of different search strategies and is capable of automatically selecting suitable strategies and experimenting with these strategies.

The finite model building component of Gandalf uses the Zchaff propositional logic solver by L.Zhang [22] as an external program called by Gandalf. Zchaff is not free, although it can be used freely for research purposes. Gandalf is not optimised for Zchaff or linked together with it: Zchaff can be freely replaced by other satisfiability checkers.

Implementation

Gandalf is implemented in Scheme and compiled to C using the Hobbit Scheme-to-C compiler. Version scm5d6 of the Scheme interpreter scm by A.Jaffer is used as the underlying Scheme system. Zchaff is implemented in C++.

Gandalf has been tested on Linux, Solaris, and MS Windows under Cygwin.

Gandalf is available under GPL from:

`http://www.ttu.ee/it/gandalf`

Strategies

One of the basic ideas used in Gandalf is time-slicing: Gandalf typically runs a number of searches with different strategies one after another, until either the proof is found or time runs out. Also, during each specific run Gandalf typically modifies its strategy as the time limit for this run starts coming closer. Selected clauses from unsuccessful runs are sometimes used in later runs.

In the normal mode Gandalf attempts to find only unsatisfiability. It has to be called with a `-sat` flag to find satisfiability. The following strategies are run:

- Finite model building by incremental search through function symbol interpretations.
- Ordered binary resolution (term depth): only for problems not containing equality.
- Finite model building using MACE-style flattening and the external propositional prover.

Expected Competition Performance

Gandalf c-2.6-SAT is the CASC-J2 SAT division, assurance class, winner.

7.6 MathServ 0.62

Jürgen Zimmer
Universitä des Saarlandes, Germany
`jzimmer@ags.uni-sb.de`

Architecture

MathServ is a framework for integrating reasoning systems as Web Services into a networked environment. The functionality of these Reasoning Web Services is captured in Semantic Web Service descriptions using the OWL-S [14] upper ontology for semantic web services. The MathServ Broker is a middle-agent which knows all available reasoning services and answers queries for client applications. Given a problem description, the MathServ Broker can automatically retrieve services and to combine services in case one single service is not sufficient to tackle a problem. Our Broker performs a "semantic" best match by analysing incoming problems and choosing the best service available for that problem.

MathServ currently integrates the ATP systems Otter 3.3, EP 0.82, SPASS 2.1 and Vampire 7.0. All ATP services get a problem description in the new TPTP format

(TPTP Library v3.0.1) and a time limit in seconds as an input. The problem is transformed into the provers' input format using the tptp2X utility. The result specifies the status of the given problem with one of the statuses defined in the SZS Status Ontology [55]. If the system delivered a refutation proof then the proof is translated into TPTP format (optionally in XTSTP) using tools developed by Geoff Sutcliffe.

Since MathServ does not form an ATP system on its own it only enters the Demonstration Division of the CASC.

Implementation

MathServ is implemented in Java and uses an Apache Tomcat web server and the AXIS package to offer reasoning systems as Web Services. All services are accessible programmatically using WSDL descriptions of their interface. ATP systems are integrated via Java wrappers that manage the translation of the input problems into the prover's input format and the translation of resulting resolution proofs into TPTP format.

Strategies

We collected data on the performance of the underlying ATP systems on all the Specialists Problem Classes (SPCs) [54] of the TPTP Library. The OWL-S descriptions of all ATP services have been annotated with this data. The MathServ Broker determines the SPC of an incoming TPTP problem and uses the performance data to choose a suitable prover for that problem.

Expected Competition Performance

The system is too new to make any predictions.

7.7 MUSCADET 2.5

Dominique Pastre
Université René Descartes - Paris, France
pastre@math-info.univ-paris5.fr

Architecture

The MUSCADET theorem prover is a knowledge-based system. It is based on Natural Deduction, following the terminology of [6] and [25], and uses methods which resembles those used by humans. It is composed of an inference engine, which interprets and executes rules, and of one or several bases of facts, which are the internal representation of "theorems to be proved". Rules are either universal and put into the system, or built by the system itself by metarules from data (definitions and lemmas). Rules may add new hypotheses, modify the conclusion, create objects, split theorems into two or more subtheorems or build new rules which are local for a (sub-)theorem.

Implementation

MUSCADET 2 [28] is implemented in SWI-Prolog. Rules are written as declarative Prolog clauses. Metarules are written as sets of Prolog clauses, more or less declarative. The inference engine includes the Prolog interpreter and some procedural Prolog clauses. MUSCADET 2.5 is available from:

`http://www.math-info.univ-paris5.fr/ pastre/muscadet/muscadet.html`

Strategies

There are specific strategies for existential, universal, conjunctive or disjunctive hypotheses and conclusions. Functional symbols may be used, but an automatic creation of intermediate objects allows deep subformulae to be flattened and treated as if the concepts were defined by predicate symbols. The successive steps of a proof may be forward deduction (deduce new hypotheses from old ones), backward deduction (replace the conclusion by a new one) or refutation (only if the conclusion is a negation).

The system is also able to work with second order statements. It may also receive knowledge and know-how for a specific domain from a human user; see [26] and [27]. These two possibilities are not used while working with the TPTP Library.

Expected Competition Performance

The best performances of MUSCADET will be for problems manipulating many concepts in which all statements (conjectures, definitions, axioms) are expressed in a manner similar to the practice of humans, especially of mathematicians. It will have poor performances for problems using few concepts but large and deep formulas leading to many splittings.

7.8 Mace2 2.2

William McCune
Argonne National Laboratory, USA
`mccune@mcs.anl.gov`

Architecture

Mace2 [17] searches for finite models of first-order (including equality) statements. Mace2 iterates through domain sizes, starting with 2. For a given domain size, a propositional satisfiability problem is constucted from the ground instances of the statements, and a DPLL procedure is applied.

Mace2 is an entirely different program from Mace4 [18], in which the ground problem for a given domain size contains equality and is decided by rewriting.

Implementation

Mace2 is coded in ANSI C. It uses the same code as Otter [19] for parsing input, and (for the most part) accepts the same intput files as Otter. Mace2 is packaged and distributed

with Otter, which is available from:
`http://www.mcs.anl.gov/AR/otter`

Strategies

Mace2 has been evolving slowly for about ten years. Two important strategies have been added recently. In 2001, a method to reduce the number of isomorphic models was added; this method is similar in spirit to the least number optimization used in rewrite-based methods, but it applies only to the first five constants. In 2003, a clause-parting method (based on the variable occurrences in literals of flattened clauses) was added to improve performace on inputs with large clauses. Although Mace2 has several experimental features, it uses one fixed stragegy for CASC.

Expected Competition Performance

Mace2 is not expected to win any prizes, because it uses one fixed strategy, and no tuning has been done with respect to the TPTP problem library. Also, Mace2 does not accept function symbols with arity greater than 3 or predicate symbols with arity greater than 4. Overall performace, however, should be respectable.

7.9 Mace4 2004-D

William McCune
Argonne National Laboratory, USA
`mccune@mcs.anl.gov`

Architecture

Mace4 [18] searches for finite models of first-order (unsorted, with equality) statements. Given input clauses, it generates ground instances over a finite domain, then it uses a decision procedure based on rewriting try to determine satisfiability. If there is no model of that size, it increments the domain size and tries again. Input clauses are not "flattened" as they are in procedures that reduce the problem to propositional satisfiability without equality, for example, Mace2 [17].

Mace4 is an entirely different program from Mace2, in which the problem for a given domain size is reduced to a purely propositional SAT problem that is decided by DPLL.

Implementation

Mace4 is coded in ANSI C and is available from:
`http://www.mcs.anl.gov/AR/mace4`

Strategies

The two main parts of the Mace4 method are (1) selecting the next empty cell in the tables of functions being constructed and deciding which values need to be considered for that cell, and (2) propagating assignments. Mace4 uses the basic least number heuristic (LNH) to reduce isomorphism. The LNH was introduced in Falcon [60] and is also used in SEM. Effective use of the LNH requires careful cell selection. Propagation is by ground rewriting and inference rules to derive negated equalities.

Expected Competition Performance

Mace4 is not expected to win any prizes, because it uses one fixed strategy, and no tuning has been done with respect to the TPTP problem library. Overall performace, however, should be respectable. An early version of Mace4 competed in CASC-2002 under then name ICGNS.

7.10 Octopus JN05

Monty Newborn, Zongyan Wang
McGill University
newborn@cs.mcgill.ca

Architecture

Octopus [24] is a parallel version of THEO (see Section 7.15), and is designed to run on as many computers as are available. For this competition, we expect to run on between 100 and 200 computers. In the 2004 Cork competition, it ran on 120 PCs. Octopus's single processor version, THEO [23], is a resolution-refutation theorem prover for first order logic. It accepts theorems in either clausal form or FOL form. THEO's inference procedures include binary resolution, binary factoring, instantiation, demodulation, and hash table resolutions.

Implementation

Octopus is written in C and runs under both LINUX and FREEBSD. It contains about 70000 lines of source code. It normally runs on a PC with at least 512Mb of memory.

Strategies

Each processor of Octopus is a copy of THEO. When the master receives a theorem to prove, it delivers it to all the slaves with instructions on which weakened version of the theorem to try. Each slave then tries a different weakened version. If a slave finds a proof of a weakened version, it goes on to attempt to prove the given theorem. Strategies similar to those in THEO are used if the weakened version is not proved by some slave. THEO's procedure is described in Section 7.15. Once the master has sent the theorem to the slave, there is no communication among the computers except when one finds a proof.

Expected Competition Performance

Octopus is only marginally better than it was last year.

7.11 Otter 3.3

William McCune
Argonne National Laboratory, USA
`mccune@mcs.anl.gov`

Architecture

Otter 3.3 [19] is an ATP system for statements in first-order (unsorted) logic with equality. Otter is based on resolution and paramodulation applied to clauses. An Otter search uses the "given clause algorithm", and typically involves a large database of clauses; subsumption and demodulation play an important role.

Implementation

Otter is written in C. Otter uses shared data structures for clauses and terms, and it uses indexing for resolution, paramodulation, forward and backward subsumption, forward and backward demodulation, and unit conflict. Otter is available from:
`http://www-unix.mcs.anl.gov/AR/otter`

Strategies

Otter's original automatic mode, which reflects no tuning to the TPTP problems, will be used.

Expected Competition Performance

Otter has been entered into CASC-J2 as a stable benchmark against which progress can be judged (there have been only minor changes to Otter since 1996 [21], nothing that really affects its performace in CASC). This is not an ordinary entry, and we do not hope for Otter to do well in the competition.

Acknowledgments: Ross Overbeek, Larry Wos, Bob Veroff, and Rusty Lusk contributed to the development of Otter.

7.12 Paradox 1.0

Koen Claessen, Niklas Sörensson
Chalmers University of Technology and Gothenburg University, Sweden
{`koen,nik`}`@cs.chalmers.se`

Architecture Paradox 1.0 [7] is a finite-domain model generator. It is based on a MACE-style [15] flattening and instantiating of the FO clauses into propositional clauses, and then the use of a SAT solver to solve the resulting problem.

Paradox incorporates the following novel features: New polynomial-time *clause splitting heuristics*, the use of *incremental SAT*, *static symmetry reduction* techniques, and the use of *sort inference*.

Implementation

The main part of Paradox is implemented in Haskell using the GHC compiler. Paradox also has a built-in incremental SAT solver which is written in C++. The two parts are linked together on the object level using Haskell's Foreign Function Interface. Paradox uses the following non-standard Haskell extensions: local universal type quantification and hash-consing.

Strategies

There is only one strategy in Paradox:

1. Analyze the problem, finding an upper bound N on the domain size of models, where N is possibly infinite. A finite such upper bound can for example be found for EPR problems.
2. Flatten the problem, and split clauses and simplify as much as possible.
3. Instantiate the problem for domain sizes 1 up to N, applying the SAT solver incrementally for each size. Report "SATISFIABLE" when a model is found.
4. When no model of sizes smaller or equal to N is found, report "CONTRADICTION".

In this way, Paradox can be used both as a model finder and as an EPR solver.

Expected Competition Performance

Paradox 1.0 is the CASC-J2 SAT division, model class, winner.

7.13 Paradox 1.3

Koen Claessen, Niklas Sörensson
Chalmers University of Technology and Gothenburg University, Sweden
`{koen,nik}@cs.chalmers.se`

Architecture

Paradox 1.3 [7] is a finite-domain model generator. It is based on a MACE-style [15] flattening and instantiating of the first-order clauses into propositional clauses, and then the use of a SAT solver to solve the resulting problem.

Paradox incorporates the following features: Polynomial-time *clause splitting heuristics*, the use of *incremental SAT*, *static symmetry reduction* techniques, and the use of *sort inference*.

The main differences with Paradox 1.0 are: a better SAT-solver, better memory behaviour, and a faster clause instantiation algorithm.

Implementation

The main part of Paradox is implemented in Haskell using the GHC compiler. Paradox also has a built-in incremental SAT solver which is written in C++. The two parts are linked together on the object level using Haskell's Foreign Function Interface.

Strategies

There is only one strategy in Paradox:

1. Analyze the problem, finding an upper bound N on the domain size of models, where N is possibly infinite. A finite such upper bound can for example be found for EPR problems.
2. Flatten the problem, and split clauses and simplify as much as possible.
3. Instantiate the problem for domain sizes 1 up to N, applying the SAT solver incrementally for each size. Report "SATISFIABLE" when a model is found.
4. When no model of sizes smaller or equal to N is found, report "CONTRADICTION".

In this way, Paradox can be used both as a model finder and as an EPR solver.

Expected Competition Performance

Paradox 1.3 should perform slightly better than Paradox 1.0 and 1.1.'

7.14 Prover9 July-2005

William McCune
Argonne National Laboratory, USA
`mccune@mcs.anl.gov`

Architecture

Prover9, version July-2005, is a resolution/paramodulation prover for first-order logic with equality. Its overall architecture is very similar to that of Otter-3.3 [19]. It uses the "given clause algorithm", in which not-yet-given clauses are available for rewriting and for other inference operations (sometimes called the "Otter loop").

Prover9 has available positive ordered (and nonordered) resolution and paramodulation, negative ordered (and nonordered) resolution, factoring, positive and negative hyperresolution, UR-resolution, and demodulation (term rewriting). Terms can be ordered with LPO, RPO, or KBO. Selection of the "given clause" is by an age-weight ratio.

Proofs can be given at two levels of detail: (1) standard, in which each line of the proof is a stored clause with detailed justification, and (2) expanded, with a separate line for each operation. When FOF problems are input, proof of transformation to clauses is not given.

Completeness is not guaranteed, so termination does not indicate satisfiability.

Implementation

Prover9 is coded in C, and it uses the LADR libraries [20]. Some of the code descended from EQP [16]. (LADR has some AC functions, but Prover9 does not use them). Term data structures are not shared (as they are in Otter). Term indexing is used extensively, with discrimination tree indexing for finding rewrite rules and subsuming units, FPA/Path indexing for finding subsumed units, rewritable terms, and resolvable literals. Feature vector indexing [31] is used for forward and backward nonunit subsumption. At the time of CASC, Prover9 will be available at:

`http://www.mcs.anl.gov/ mccune/prover9/`

Strategies

Like Otter, Prover9 has available many strategies; the following statements apply to CASC-2005.

Given a problem, Prover9 adjusts its inference rules and strategy according to the category of the problem (HNE, HEQ, NNE, NEQ, PEQ, FNE, FEQ, UEQ) and according several other syntactic properties of the input clauses.

Terms are ordered by LPO for demodulation and for the inference rules, with a simple rule for determining symbol precedence.

For the FOF problems, a preprocessing step attempts to reduce the problem to independent subproblems by a miniscope transformation; if the problem reduction succeeds, each subproblem is clausified and given to the ordinary search procedure; if the problem reduction fails, the original problem is clausified and given to the search procedure.

As this description is being written, the specific rules for deciding the strategy have not been finalized. They will be available from the URL given above by the time CASC occurs.

Expected Competition Performance

Some of the strategy development for CASC was done by experimentation with the CASC-2004 competition selected problems. (Prover9 has not yet been run on other TPTP problems.) Prover9 is unlikely to challenge the CASC leaders, because (1) extensive testing and tuning over TPTP problems has not been done, (2) theories (e.g., ring, combinatory logic, set theory) are not recognized, (3) term orderings and symbol precedences are not fine-tuned, and (4) multiple searches with differing strategies are not run.

Finishes in the middle of the pack are anticipated in all categories in which Prover9 competes (MIX, FOF, and UEQ).

7.15 THEO JN05

Monty Newborn
McGill University
newborn@cs.mcgill.ca

Architecture

THEO [23] is a resolution-refutation theorem prover for first order clause logic. It accepts theorems in either clausal form or FOL form. THEO's inference procedures include binary resolution, binary factoring, instantiation, demodulation, and hash table resolutions.

Implementation

THEO is written in C and runs under both LINUX and FREEBSD. It contains about 45000 lines of source code. Originally it was called The Great Theorem Prover.

Strategies

THEO uses a large hash table (16 million entries) to store clauses. This permits complex proofs to be found, some as long as 500 inferences. It uses what might be called a brute-force iteratively deepening depth-first search looking for a contradiction while storing information about clauses - unit clauses in particular - in its hash table.

When THEO participated in the 2004 CASC Competition, it used a learning strategy described in this paragraph. When given a theorem, it first created a list of potential ways to "weaken" the theorem by weakening one of the clauses. It then randomly selected one of the weakenings, tried to prove the weakened version of the theorem, and then used the results from this effort to help prove the given theorem. A weakened version was created by modifying one clause by replacing a constant or function by a variable or by deleting a literal. Certain clauses from the proof of the weakened version were added to the base clauses when THEO next attempted to prove the given theorem. In addition, base clauses that participated in the proof of the weakened version were placed in the set-of-support. THEO then attempted to prove the given theorem with the revised set of base clauses.

Over the last year, this learning strategy has been further developed as described here. When THEO is given a theorem, it first tries to prove a weakened version. It does this for 50% of the allotted time. If successful, it then attempts to prove the given theorem as described in the previous paragraph. If unsuccessful, however, THEO then weakens the given theorem further by weakening an additional clause. With two clauses now weakened, THEO then attempts to prove the given theorem for 50% of the remaining time (25% of the originally allotted time). If successful, THEO uses the results to attempt to prove the given theorem. If unsuccessful, THEO picks two other weakenings and ties again for 50% of the remaining time.

Now certain weakened proofs are thrown out, as they generally seem to be useless. Weakened proofs that are shorter than three inferences are considered useless, as are those that are less than five but do not involve the negated conclusion. Attempts to improve the learning strategy are of major interest in the ongoing development of THEO. Octopus, the parallel version of THEO (Section 7.10, uses variations of the described learning strategy.

Expected Competition Performance

While the learning strategy has been quite successful, THEO is only marginally better than it was last year.

7.16 Vampire 7.0

Alexandre Riazanov, Andrei Voronkov
University of Manchester, England
{riazanoa,voronkov}@cs.man.ac.uk

Architecture

Vampire [29] 7.0 is an automatic theorem prover for first-order classical logic. Its kernel implements the calculi of ordered binary resolution and superposition for handling equality. The splitting rule and negative equality splitting are simulated by the introduction of new predicate definitions and dynamic folding of such definitions. A number of standard redundancy criteria and simplification techniques are used for pruning the search space: subsumption, tautology deletion (optionally modulo commutativity), subsumption resolution, rewriting by ordered unit equalities, basicness restrictions and irreducibility of substitution terms. The reduction orderings used are the standard Knuth-Bendix ordering and a special non-recursive version of the Knuth-Bendix ordering. A number of efficient indexing techniques is used to implement all major operations on sets of terms and clauses. Run-time algorithm specialisation is used to accelerate some costly operations, e.g., checks of ordering constraints. Although the kernel of the system works only with clausal normal forms, the preprocessor component accepts a problem in the full first-order logic syntax, clausifies it and performs a number of useful transformations before passing the result to the kernel. When a theorem is proven, the system produces a verifiable proof, which validates both the clausification phase and the refutation of the CNF. The current release features a built-in proof checker for the clausifying phase, which will be extended to check complete proofs.

Implementation

Vampire 7.0 is implemented in C++. The supported compilers are gcc 3.2.x, gcc 3.3.x, and Microsoft Visual C++. This version has been successfully compiled for Linux, but has not been fully tested on Solaris and Win32. It is available (conditions apply) from:

`http://www.cs.man.ac.uk/ riazanoa/Vampire`

Strategies

The Vampire kernel provides a fairly large number of features for strategy selection. The most important ones are:

- Choice of the main saturation procedure : (i) OTTER loop, with or without the Limited Resource Strategy, (ii) DISCOUNT loop.
- A variety of optional simplifications.
- Parameterised reduction orderings.

- A number of built-in literal selection functions and different modes of comparing literals.
- Age-weight ratio that specifies how strongly lighter clauses are preferred for inference selection.
- Set-of-support strategy.

The automatic mode of Vampire 7.0 is derived from extensive experimental data obtained on problems from TPTP v2.6.0. Input problems are classified taking into account simple syntactic properties, such as being Horn or non-Horn, presence of equality, etc. Additionally, we take into account the presence of some important kinds of axioms, such as set theory axioms, associativity and commutativity. Every class of problems is assigned a fixed schedule consisting of a number of kernel strategies called one by one with different time limits.

Expected Competition Performance

Vampire 7.0 is the CASC-J2 MIX and FOF divisions, assurance and proof classes, winner.

7.17 Vampire 8.0

Andrei Voronkov
University of Manchester, England
`voronkov@cs.man.ac.uk`

No system description provided.

7.18 Waldmeister 704

Jean-Marie Gaillourdet[1], Thomas Hillenbrand[2], Bernd Löchner[1]
[1]Technische Universität Kaiserslautern, Germany
[2]Max-Planck-Institut für Informatik Saarbrücken, Germany,
`waldmeister@informatik.uni-kl.de`

Architecture

Waldmeister 704 is a system for unit equational deduction. Its theoretical basis is unfailing completion in the sense of [2] with refinements towards ordered completion (cf. [1]). The system saturates the input axiomatization, distinguishing active facts, which induce a rewrite relation, and passive facts, which are the one-step conclusions of the active ones up to redundancy. The saturation process is parameterized by a reduction ordering and a heuristic assessment of passive facts [10]. For an in-depth description of the system, see [9].

Waldmeister 704 improves over last year's version in several respects. Firstly, the detection of redundancies in the presence of associative-commutative operators has been strenghtened (cf. [13]). In a class of AC-equivalent equations, an element is redundant if each of its ground instances can be rewritten, with the ground convergent rewrite system for AC, into an instance of another element. Instead of elaborately checking this kind of reducability explicitly, it can be rephrased in terms of ordering constraints and efficiently be approximated with a polynomial test. Secondly, the last teething troubles of the implementation of the Waldmeister loop have been overcome. Thirdly, a number of strategies have slightly been revised.

Implementation

The prover is coded in ANSI-C. It runs on Solaris, Linux, and newly also on MacOS X. In addition, it is now available for Windows users via the Cygwin platform. The central data strucures are: perfect discrimination trees for the active facts; group-wise compressions for the passive ones; and sets of rewrite successors for the conjectures. Visit the Waldmeister Web pages at:
`http://www.waldmeister.org`

Strategies

The approach taken to control the proof search is to choose the search parameters according to the algebraic structure given in the problem specification [10]. This is based on the observation that proof tasks sharing major parts of their axiomatization often behave similar. Hence, for a number of domains, the influence of different reduction orderings and heuristic assessments has been analyzed experimentally; and in most cases it has been possible to distinguish a strategy uniformly superior on the whole domain. In essence, every such strategy consists of an instantiation of the first parameter to a Knuth-Bendix ordering or to a lexicographic path ordering, and an instantiation of the second parameter to one of the weighting functions *addweight*, *gtweight*, or *mixweight*, which, if called on an equation $s = t$, return $|s| + |t|$, $|max_>(s,t)|$, or $|max_>(s,t)| \cdot (|s| + |t| + 1) + |s| + |t|$, respectively, where $|s|$ denotes the number of symbols in s.

Expected Competition Performance

Waldmeister 704 is the CASC-J2 UEQ division winner.

7.19 Wgandalf 0.1

Tanel Tammet
Tallinn Technical University, Estonia
`tammet@cc.ttu.ee`

Architecture

Wgandalf 0.1 is an early, pre-alpha version of the full 1st order prover intended to be useful in rule-based databases, semantic web applications, and similar data-centric applications, while being a powerful conventional prover at the same time. The current pre-alpha version achieves none of the intentions. It is usable only as a conventional prover, while being a fairly weak conventional prover. It is not able to output a proof and it does not contain any components for model building. However, it does contain several different search strategies and it uses time slicing to call these strategies one after another.

Wgandalf uses some ideas, but no code from the earlier Gandalf system of the same author: essentially, it is a completely new system.

Implementation

Wgandalf is available under GPL, and at some point in the future it will be available from the site:
`http://www.waldmeister.org`
which will contain further details about the architecture, strategies and implementation.

Strategies

Wgandalf uses a number of conventional strategies on top of ordinary binary resolution: literal orderings of various kinds, set of support, clause simplification methods, ordinary subsumption, demodulation and paramodulation. The basic resolution loop is a DISCOUNT loop, not the OTTER loop. Only the key data about the derivation history is kept, based on which the actual clauses are re-created when picked as given clauses. Wgandalf does contain a few strategies of a more exotic kind, which will be described in the wgandalf documentation.

Expected Competition Performance

Wgandalf 0.1 is expected to perform significantly worse than the current state-of-the-art provers. However, it is likely to perform better than the classic provers and the older Gandalf system of the same author.

8 Conclusion

The CADE-20 ATP System Competition is the tenth large scale competition for classical first-order logic ATP systems. The organizers believe that CASC fulfills its main motivations: stimulation of research, motivation for improving implementations, evaluation of relative capabilities of ATP systems, and providing an exciting event. For the entrants, their research groups, and their systems, there is substantial publicity both within and outside the ATP community. The significant efforts that have gone into developing the ATP systems receive public recognition; publications, which adequately present theoretical work, have not been able to expose such practical efforts appropriately. The competition provides an overview of which researchers and research groups have decent, running, fully automatic ATP systems.

References

[1] J. Avenhaus, T. Hillenbrand, and B. Löchner. On Using Ground Joinable Equations in Equational Theorem Proving. *Journal of Symbolic Computation*, 36(1-2):217–233, 2003.

[2] L. Bachmair, N. Dershowitz, and D.A. Plaisted. Completion Without Failure. In H. Ait-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, pages 1–30. Academic Press, 1989.

[3] P. Baumgartner, A. Fuchs, and C. Tinelli. Darwin - A Theorem Prover for the Model Evolution Calculus. In G. Sutcliffe, S. Schulz, and T. Tammet, editors, *Proceedings of the Workshop on Empirically Successful First Order Reasoning, 2nd International Joint Conference on Automated Reasoning*, 2004.

[4] P. Baumgartner and C. Tinelli. The Model Evolution Calculus. In F. Baader, editor, *Proceedings of the 19th International Conference on Automated Deduction*, number 2741 in Lecture Notes in Artificial Intelligence, pages 350–364. Springer-Verlag, 2003.

[5] J-P. Billon. The Disconnection Method: A Confluent Integration of Unification in the Analytic Framework. In P. Miglioli, U. Moscato, D. Mundici, and M. Ornaghi, editors, *Proceedings of TABLEAUX'96: the 5th Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, number 1071 in Lecture Notes in Artificial Intelligence, pages 110–126. Springer-Verlag, 1996.

[6] W.W. Bledsoe. Splitting and Reduction Heuristics in Automatic Theorem Proving. *Artificial Intelligence*, 2:55–77, 1971.

[7] K. Claessen and N. Sorensson. New Techniques that Improve MACE-style Finite Model Finding. In P. Baumgartner and C. Fermueller, editors, *Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*, 2003.

[8] M. Greiner and M. Schramm. A Probablistic Stopping Criterion for the Evaluation of Benchmarks. Technical Report I9638, Institut für Informatik, Technische Universität München, München, Germany, 1996.

[9] T. Hillenbrand. Citius altius fortius: Lessons Learned from the Theorem Prover Waldmeister. In I. Dahn and L. Vigneron, editors, *Proceedings of the 4th International Workshop on First-Order Theorem Proving*, number 86.1 in Electronic Notes in Theoretical Computer Science. Elsevier Science, 2003.

[10] T. Hillenbrand, A. Jaeger, and B. Löchner. Waldmeister - Improvements in Performance and Ease of Use. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, number 1632 in Lecture Notes in Artificial Intelligence, pages 232–236. Springer-Verlag, 1999.

[11] R. Letz and G. Stenz. System Description: DCTP - A Disconnection Calculus Theorem Prover. In R. Gore, A. Leitsch, and T. Nipkow, editors, *Proceedings of the International Joint Conference on Automated Reasoning*, number 2083 in Lecture Notes in Artificial Intelligence, pages 381–385. Springer-Verlag, 2001.

[12] R. Letz and G. Stenz. Generalised Handling of Variables in Disconnection Tableaux. In M. Rusinowitch and D. Basin, editors, *Proceedings of the 2nd International Joint*

Conference on Automated Reasoning, number 3097 in Lecture Notes in Artificial Intelligence, pages 289–306, 2004.

[13] B. Loechner. What to know when implementing LPO. In G. Sutcliffe, S. Schulz, and T. Tammet, editors, *Proceedings of the Workshop on Empirically Successful First Order Reasoning, 2nd International Joint Conference on Automated Reasoning*, 2004.

[14] et al. Martin, D. OWL-S 1.1 Release. http://www.daml.org/services/owl-s/1.1/, URL.

[15] W.W. McCune. A Davis-Putnam Program and its Application to Finite First-Order Model Search: Quasigroup Existence Problems. Technical Report ANL/MCS-TM-194, Argonne National Laboratory, Argonne, USA, 1994.

[16] W.W. McCune. Solution of the Robbins Problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.

[17] W.W. McCune. MACE 2.0 Reference Manual and Guide. Technical Report ANL/MCS-TM-249, Argonne National Laboratory, Argonne, USA, 2001.

[18] W.W. McCune. Mace4 Reference Manual and Guide. Technical Report ANL/MCS-TM-264, Argonne National Laboratory, Argonne, USA, 2003.

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

[20] W.W. McCune. LADR: Library of Automated Deduction Routines. http://www.mcs.anl.gov/AR/ladr, URL.

[21] W.W. McCune and L. Wos. Otter: The CADE-13 Competition Incarnations. *Journal of Automated Reasoning*, 18(2):211–220, 1997.

[22] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In Blaauw D. and L. Lavagno, editors, *Proceedings of the 39th Design Automation Conference*, pages 530–535, 2001.

[23] M. Newborn. *Automated Theorem Proving: Theory and Practice.* Springer, 2001.

[24] M. Newborn and Z. Wang. Octopus: Combining learning and parallel search. *Journal of Automated Reasoning*, 33(2):171–218, 2004.

[25] D. Pastre. Automatic Theorem Proving in Set Theory. *Artificial Intelligence*, 10:1–27, 1978.

[26] D. Pastre. MUSCADET : An Automatic Theorem Proving System using Knowledge and Metaknowledge in Mathematics. *Artificial Intelligence*, 38:257–318, 1989.

[27] D. Pastre. Automated Theorem Proving in Mathematics. *Annals of Mathematics and Artificial Intelligence*, 8:425–447, 1993.

[28] D. Pastre. Strong and Weak Points of the MUSCADET Theorem Prover. *AI Communications*, 15(2-3):147–160, 2002.

[29] A. Riazanov and A. Voronkov. The Design and Implementation of Vampire. *AI Communications*, 15(2-3):91–110, 2002.

[30] S. Schulz. A Comparison of Different Techniques for Grounding Near-Propositional CNF Formulae. In S. Haller and G. Simmons, editors, *Proceedings of the 15th Florida Artificial Intelligence Research Symposium*, pages 72–76. AAAI Press, 2002.

[31] S. Schulz. Simple and Efficient Clause Subsumption with Feature Vector Indexing. In G. Sutcliffe, S. Schulz, and T. Tammet, editors, *Proceedings of the Workshop on Empirically Successful First Order Reasoning, 2nd International Joint Conference on Automated Reasoning*, 2004.

[32] S. Schulz. System Abstract: E 0.81. In M. Rusinowitch and D. Basin, editors, *Proceedings of the 2nd International Joint Conference on Automated Reasoning*, number 3097 in Lecture Notes in Artificial Intelligence, pages 223–228, 2004.

[33] G. Stenz. DCTP 1.2 - System Abstract. In C. Fermüller and U. Egly, editors, *Proceedings of TABLEAUX 2002: Automated Reasoning with Analytic Tableaux and Related Methods*, number 2381 in Lecture Notes in Artificial Intelligence, pages 335–340. Springer-Verlag, 2002.

[34] G. Stenz. *The Disconnection Calculus*. PhD thesis, Institut für Informatik, Technische Universität München, Munich, Germany, 2002.

[35] G. Stenz and A. Wolf. E-SETHEO: Design, Configuration and Use of a Parallel Automated Theorem Prover. In N. Foo, editor, *Proceedings of AI'99: The 12th Australian Joint Conference on Artificial Intelligence*, number 1747 in Lecture Notes in Artificial Intelligence, pages 231–243. Springer-Verlag, 1999.

[36] G. Sutcliffe. Proceedings of the CADE-16 ATP System Competition. Trento, Italy, 1999.

[37] G. Sutcliffe. Proceedings of the CADE-17 ATP System Competition. Pittsburgh, USA, 2000.

[38] G. Sutcliffe. The CADE-16 ATP System Competition. *Journal of Automated Reasoning*, 24(3):371–396, 2000.

[39] G. Sutcliffe. Proceedings of the IJCAR ATP System Competition. Siena, Italy, 2001.

[40] G. Sutcliffe. The CADE-17 ATP System Competition. *Journal of Automated Reasoning*, 27(3):227–250, 2001.

[41] G. Sutcliffe. Proceedings of the CADE-18 ATP System Competition. Copenhagen, Denmark, 2002.

[42] G. Sutcliffe. Proceedings of the CADE-19 ATP System Competition. Miami, USA, 2003.

[43] G. Sutcliffe. Proceedings of the 2nd IJCAR's CADE ATP System Competition. Cork, Ireland, 2004.

[44] G. Sutcliffe. The IJCAR-2004 Automated Theorem Proving Competition. *AI Communications*, 18(1), 2005.

[45] G. Sutcliffe and C. Suttner. The CADE-14 ATP System Competition. Technical Report 98/01, Department of Computer Science, James Cook University, Townsville, Australia, 1998.

[46] G. Sutcliffe and C. Suttner. The CADE-18 ATP System Competition. *Journal of Automated Reasoning*, 31(1):23–32, 2003.

[47] G. Sutcliffe and C. Suttner. The CADE-19 ATP System Competition. *AI Communications*, 17(3):103–182, 2004.

[48] G. Sutcliffe, C. Suttner, and F.J. Pelletier. The IJCAR ATP System Competition. *Journal of Automated Reasoning*, 28(3):307–320, 2002.

[49] G. Sutcliffe and C.B. Suttner. Special Issue: The CADE-13 ATP System Competition. *Journal of Automated Reasoning*, 18(2), 1997.

[50] G. Sutcliffe and C.B. Suttner. The Procedures of the CADE-13 ATP System Competition. *Journal of Automated Reasoning*, 18(2):163–169, 1997.

[51] G. Sutcliffe and C.B. Suttner. Proceedings of the CADE-15 ATP System Competition. Lindau, Germany, 1998.

[52] G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[53] G. Sutcliffe and C.B. Suttner. The CADE-15 ATP System Competition. *Journal of Automated Reasoning*, 23(1):1–23, 1999.

[54] G. Sutcliffe and C.B. Suttner. Evaluating General Purpose Automated Theorem Proving Systems. *Artificial Intelligence*, 131(1-2):39–54, 2001.

[55] G. Sutcliffe, J. Zimmer, and S. Schulz. TSTP Data-Exchange Formats for Automated Theorem Proving Tools. In W. Zhang and V. Sorge, editors, *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, number 112 in Frontiers in Artificial Intelligence and Applications, pages 201–215. IOS Press, 2004.

[56] C.B. Suttner and G. Sutcliffe. The Design of the CADE-13 ATP System Competition. *Journal of Automated Reasoning*, 18(2):139–162, 1997.

[57] C.B. Suttner and G. Sutcliffe. The CADE-14 ATP System Competition. *Journal of Automated Reasoning*, 21(1):99–134, 1998.

[58] T. Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, 1997.

[59] T. Tammet. Towards Efficient Subsumption. In C. Kirchner and H. Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction*, number 1421 in Lecture Notes in Artificial Intelligence, pages 427–440. Springer-Verlag, 1998.

[60] J. Zhang. Constructing Finite Algebras with FALCON. *Journal of Automated Reasoning*, 17(1):1–22, 1996.