Contradiction Detection and Repair in a Large Theory

Adam Pease and **Stephan Schulz**
Articulate Software, San Jose, CA USA DHBW Stuttgart, Stuttgart, Germany
apease@articulatesoftware.com schulz@eprover.org

Abstract

As with any software, the challenges of developing large and manually-created axiomatizations in an expressive logic such as first order logic with equality can be very different from those found in comparatively small theories. We present some of the tools and practices that have supported development of a logical theories with tens of thousands of statements, and ensured that they are free of logical contradiction, and suitable for automated theorem reasoning.

Introduction

Large theories in an expressive logic have the potential for powerful reasoning, but that can come at the cost of complexity. Large theories that use less expressive logics, such as taxonomies or description logics, can have relatively simple methods for determining, for example, that there are no type conflicts in a class hierarchy or property signatures.

First-order logic with equality is a lot more expressive, and supports complex mathematical and practical reasoning. However, once first order logic with equality is used, reasoning tools have more difficult problems to solve. Since first-order logic is only semi-decidable, consistency can be impossible to establish in general, and even inconsistency can only be proven in the limit, i.e. with unrestricted resources in time and memory. Still, automated methods can help maintaining the consistency of such a theory.

Our use case is the continued development of the Suggested Upper Merged Ontology (SUMO) (Niles and Pease 2001; Pease 2011)[1], a comprehensive ontology of around 20,000 concepts and 80,000 hand-authored logical statements in a (mildly) higher-order logic. SUMO has an associated integrated development environment called Sigma (Pease and Schulz 2014)[2] that interfaces to leading theorem provers such as E (Schulz 2002; Schulz, Cruanes, and Vukmirović 2019), Vampire (Kovács and Voronkov 2013) and LEO-II (Benzmüller et al. 2008). As the SUMO language has no native implementation in a theorem prover for its language, we rely on provers that support different subsets of its language. We have automatic translations (Pease and Schulz 2014) to the strictly first order language of TPTP (Trac, Sutcliffe, and Pease 2008), as well as the TF0 language (first order logic with typed arithmetic) (Pease 2019) and THF (Benzmüller and Pease 2010). The recent development of our TF0 translator means that we can reason with numerical measures in a truth-preserving representation (Read 2003).

Using an expressive logic allows us to state knowledge about the world that could not be expressed in weaker logics. In order to state general rules, we must have variables, ruling out use of a propositional logic. For example

```
(=>
  (and
    (instance ?PROC IntentionalProcess)
    (agent ?PROC ?AGENT))
  (exists (?PURP)
    (hasPurposeForAgent ?PROC ?PURP ?AGENT)))
```

we need to be able to state that when an `Agent` conducts an `IntentionalProcess`, that agent has some intended purpose that process fulfills or contributes to. This is an essential part of the definition of an intentional process. In order to apply this to a particular instance, such as that of a carpenter "John" building a house, we need variables to allow a theorem prover to substitute the instance `John` for variable `?AGENT` and then conclude that there exists some purpose `?PURP` held by John for the process. Use of true negation, predicates of arity greater than two, and full first order quantification all require at least first order logic. An even more expressive logic however is desirable in order to reason with a larger subset of SUMO.

Numerical reasoning is very common in practical reasoning problems over world knowledge. For example, a robot may need to know its capacity to carry heavy objects in order to respond effectively to human commands. It may have to calculate weight or dimensions, and possibly do unit conversions.

The alternative to having arithmetic as part of the logical language has been procedural attachment. Provers such as SNARK (Stickel, Waldinger, and Chaudhri 2000) have supported procedural attachment as a way to handle arithmetic. This means that during the course of theorem proving, an arbitrary procedure can be called, implemented in a programming language such as Lisp or C, that takes the arguments to a logical function and uses them as parameters to a procedural function call, after which the logical function can be

[1] `https://www.ontologyportal.org`

[2] `https://github.com/ontologyportal/sigmakee`

replaced with the returned value. This is simple and expedient, but it removes the logical definition of the function call from the domain of theorem proving. If the semantics of the function and its values during proving are inconsistent with the rest of the logical theory, that fact cannot be detected unless the semantics of the function are part of the logical language and theory. Procedural attachment takes theorem proving back partly into the realm of procedural programming and testing through debugging, rather than mathematical proof.

A different issue is that of efficiency. One would not want to conduct mathematical operations in logic, such as with Peano arithmetic, as that would be computationally inefficient. But we can have the best of both worlds by supporting arithmetic calculations in the theorem prover, implemented efficiently, while also having the logical specification of such operations in the logical theory, so that they entire system can be subject to verification. The TFF family of languages (Sutcliffe et al. 2012) addresses these issues.

In an ideal world, all specifications would be correct, all software would be bug-free, and all logical theories would be consistent. However, in practice, these are large and complicated systems, and they are built by imperfect human beings. Thus, some errors are probably unavoidable. However, we can use automated reasoning tools to minimize both the likelihood of errors, as well as their impact.

Unfortunately, the size of SUMO is well beyond what model finders (such as Paradox (Claessen and Sörensson 2003)) are capable of handling. Thus, we do not have a way to prove that all of SUMO is consistent. With respect to inconsistency, nearly all current automated theorem provers are built on a refutational paradigm. At the core of the prover is an algorithm that tries to show that a set of formulas (or clauses) is inconsistent. In classical theorem proving, this is used to derive an explicit contradiction from the axioms and the negated conjecture, thus proving the original goal. However, it also gives us a tool to possibly detect (and then repair) inconsistencies in the knowledge base.

A challenge is that proofs of any given contradiction can be large, easily running to 100 steps for recent ones that have been found. Finding a contradiction is just part of the problem. Repairing the contradiction while not unduly reducing the inferential power of the theory is also important.

Contradiction Detection

In principle, it would be possible to run a refutational theorem prover on all of SUMO with the aim of finding possible contradictions. Most theorem provers are able to output a proof object, the analysis of which would allow us to identify an inconsistent subset. However, in practice, most theorem provers are not able to find such a contradiction under reasonable resource constraints.

In 2009, the CADE ATP System Competition (CASC) (Sutcliffe 2007; 2019), one of the major drivers of ATP development, introduced the *Large Theory Batch* (LTB) division, in which large theories like SUMO are used as the base for reasoning problems. One of the interesting observations of LTB was that provers that were unable to find inconsistencies in the full theories nevertheless managed find inconsistent subsets of the axioms when focused by the presence of a conjecture.

The explanation for this is that the successful systems either before or during proof search identify parts of the theory likely relevant to the conjecture, and focus on this coherent subset of axioms. Thus, they are able to find contradictions in these much smaller sets that would be lost in the full theory.

We can utilize this observation and the mechanisms behind it in several ways.

Local Contradictions

Without infinite resources, it is in general impossible to show if an FOL theory is contradictory. This has historically been thought to be a major obstacle, and still is if the goal is, for example, proving the consistency of a mathematical theory. However, for common sense reasoning, the goal is closer to "utility".

As stated above, nearly all current ATP systems perform proof by contradiction. The system negates the conjecture, and then tries to find an inconsistency from axioms and negated conjecture. If such a contradiction is found, we can be in one of two situations. First, the derivation of the inconsistency does involve the negated conjecture. In this case, the assumption is that the subset of axioms used is consistent, and the conjecture is logically implied by the axioms. In the second case, the negated conjecture is not involved in the inconsistency. In that case, we have uncovered a *local contradiction* in the theory. We can easily distinguish the two cases if the prover provides a suitable proof object (Schulz and Sutcliffe 2015), as done by most state-of-the-art systems.

A local contradiction has two implications: First, the status of the conjecture with respect to the user domain is open - while it has been formally proved (by *ex falso, quodlibet*), that proof is the result of a buggy formalisation, and hence vacuous. Secondly, it should trigger the process of theory repair.

A proof that does include the (negated) conjecture is valid for the portion of the theory that is used in the proof. There may be an undiscovered problem in the theory as a whole, but the proof is still valid for the formulas that it contains. This has been the case with SUMO. As SUMO has grown over the last two decades, theorem proving has gotten more powerful, especially with respect to large theories. We have regularly found small numbers of contradictions, typically just a few each year, which have been hidden in very deep chains of reasoning that have only been encountered with the continuous improvements in the provers used. SUMO has been used productively and accurately even while some of these deep contradictions remained undiscovered.

It should also not be surprising to find a problem occasionally in a theory of this size. Just as with any large software program, one would not expect it to be written from the beginning completely free of bugs.

Sigma Diagnostics

Theorem proving can be time consuming, even though great improvements have been made (Hoder and Voronkov 2011; Pease et al. 2010). It's efficient to implement the sort of fast

```
% SZS output start Refutation
fof(189,negated_conjecture,
  (~(?[X31]: s__member__02(X31,s__Org1_1__00))),
  inference('input',[])).
fof(135,axiom,
  (![X14]: (s__instance__02(X14,
                    s__Collection__00) =>
  (?[X15]: (s__instance__02(X15,
                s__SelfConnectedObject__00) &
    s__member__02(X15,X14))))),
  inference('input',[])).
fof(103,axiom, s__subclass__02(
   s__Organization__00, s__Collection__00),
  inference('input',[])).
fof(159,axiom,
  (![X24,X25]: (s__subclass__02(X24,X25) =>
  (s__instance__02(X25,s__SetOrClass__00) &
  s__instance__02(X24,s__SetOrClass__00)))),
  inference('input',[])).
fof(114,axiom, s__instance__02(
    s__Organization__00, s__SetOrClass__00),
  inference('input',[])).
fof(240,plain,
  (![X0]: ((~s__instance__02(X0,
            s__Collection__00))
    | (?[X1]: (s__member__02(X1,X0) &
      s__instance__02(X1,
              s__SelfConnectedObject__00))))),
  inference('ENNF transformation',[206])).
fof(299,plain,
  ((~s__instance__02(X0,s__Collection__00)) |
    (s__member__02(sk_0(X0),X0) &
     s__instance__02(sk_0(X0),
                s__SelfConnectedObject__00))),
  inference('skolemization',[240])).
cnf(525,plain,
   s__instance__02(X2,X1)|~s__subclass__02(X0,X1)
  |~s__instance__02(X2,X0)
  |~s__instance__02(X0,s__SetOrClass__00)
  | ~s__instance__02(X1,s__SetOrClass__00),
  inference('cnf transformation',[330])).
fof(190,axiom,
  s__instance__02(s__Org1_1__00,
  s__Organization__00), inference('input',[])).
cnf(491,plain,
  s__member__02(sk_0(X0),X0)
  | ~s__instance__02(X0,s__Collection__00),
  inference('cnf transformation',[299])).
cnf(487,plain,
  ~s__member__02(X0,s__Org1_1__00),
  inference('cnf transformation',[295])).
cnf(861,plain,
  ~s__subclass__02(s__Organization__00,X1)
  | s__instance__02(s__Org1_1__00,X1),
  inference('resolution',
  [506,359,525,333])).
cnf(897,plain,
  s__instance__02(s__Org1_1__00,
  s__Collection__00),
  inference('resolution',[414,861])).
cnf(1029,plain,
  $false, inference('resolution',[487,491,897])).
% SZS output end Refutation
```

Figure 1: Selected steps of a simple TPTP proof - Vampire's output for CSR075+3 from CASC-22 (2009)

type-checking that is done in description logic systems on the portion of the theory that is expressible in that language. We implemented checking types (classes) for disjoint parents. SUMO is a strongly typed system, in which all relations (including functions) have a required type signature, so we can check whether arguments are in conflict with the required types for given relations.

There are also many checks that can be done that are indicative of problems even if they do not cause a logical contradiction. This includes a term that does not have a documentation string. Another common problem is having a term that does not ultimately have a root at `Entity`, which is the parent class of all terms in SUMO. We can also find formulas that have a quantified variable that is not used in the formula.

These diagnostics have been integrated into a programmer's text editor that can catch the most common mistakes from SUMO developers (Pease 2020).

Theorem Proving

The most straightforward check one can do with a logical theory and a theorem prover is to check for a contradiction by asking the prover to prove falsehood. However, that provides no direction to the theorem prover for where to look in a large theory for possible contradictions. Providing guidance to a prover has been a recent fruitful approach.

Axiom Filtering

While issues can often be found simply by asking the provers to prove "false" (i.e. to find a contradiction among the axioms only), we can also employ an automated means of focusing on different parts of the ontology. The E distribution contains a tool (Schulz et al. 2017) (called EAXFilter) for generating thousands of test problems that focus on theories connected with certain function and predicate symbols. These can be either picked at random by the system, or provided by the user (e.g. symbols newly introduced when extending the theory). The system uses different variants of the SInE algorithm (Hoder and Voronkov 2011) to extract relatively small and coherent subsets of the axiomatization. These can then be handed to theorem provers to find possible contradictions. We employ the StarExec (Stump, Sutcliffe, and Tinelli 2014) server cluster to make such large scale testing practical.

Contradiction Repair

Graphical Proofs

A traditional proof that one might find in a logic textbook is a strictly linear presentation of deductions. However, a deductive proof in FOL is only partially ordered - different threads of deduction can lead to a single conclusion. While a linear proof can be created, it can often be confusing, and a graphical presentation that shows parallel threads of deduction is often much clearer.

The Interactive Derivation Viewer (IDV) (Trac, Puzis, and Sutcliffe 2007) is an innovative proof viewer that allows the user to select a varying degree of detail in the proof.

The Sigma Graph tool[3] employs the GraphViz system to generate aesthetically appealing graph of proofs. It has a simpler version of IDV's detail control. It can either display the entire proof, or just the human-authored formulas used in the proof, or an intermediate alternative that only shows deductive steps that follow from two or more premises.

Proof Ablation

A recent addition to Sigma[4] is what we are calling *Proof Ablation*. It iterates through a proof of a contradiction, removing human-authored formulas one by one, and then attempting to re-derive the contradiction. This allows the system to identify a minimal inconsistent core of axioms, or, in other words, to expose (one of) the smallest subsets that is contradictory. It is often the case that several different instances of local inconsistency can thus be traced to the same problematic axioms - and can be fixed by repairing this small set.

Example

We now examine a contradiction that was recently found by Vampire, using the test files created by EAXFilter. The theory used was the full SUMO, including the top, mid-level and all domain ontologies. SUMO was first converted to TPTP format by Sigma. The result was 2,094,675 TPTP formulas. EAXfilter generated 45,387 test theories, each with anywhere from a few hundred to tens of thousands of fomulas, and each comprising a different subset of the formulas in the full TPTP SUMO theory. Generating these problems took a few hours on a modern laptop. We then ran Vampire on each of those problems. That step took over a day. The most recent test found a contradiction with a 120-step proof in one of the test theories.

Proof by refutation can be challenging to explain in common-sense language, which has motivated work on *natural deduction* (Prawitz 1965). However, there is still no general and automatic method for doing natural deduction on theories in FOL that is sound, complete and of comparable performance. So, we will give an approximation of the line of reasoning here and the interested reader can refer to both the full linear proof [5] and graphical proof[6] available online (see Figure 2 for a small section of the graphical proof).

We can group the proof into several sections, which eventually come together to produce a contradiction. The domain of the proof is primarily weather, with formulas primarily from the Weather.kif domain ontology file of SUMO[7]. Constant names below shown in `typewriter font` are terms with definitions in SUMO and their associated formulas can be viewed online[8].

1. • `naturalHazardTypeInArea` means there exists an instance of the hazard in that area (at some point)
   • `CyclonicStorm` is `WeatherSystem` and is a natural hazard in the `SouthernOcean`
   • A `WeatherSystem` has an `AirStream`
   • So, there is or has been a `CyclonicStorm` with an `AirStream`
2. • anything (everything) located in an `AirStream` is a `Gas` (has the attribute of being a `Gas`)
   • an `AirStream` is a `Region`
   • `WindFlow` is an `AirStream`
   • Every `Region` has something `Physical` in it (at least another `Region`)
3. • Substances have `PhysicalState`(s) and anything with a `PhysicalState` is a `Substance`
   • `UnitedNations` is a `Collection` and therefore not a `SelfConnectedObject` (or an `Object`) since those classes are `disjoint`
   • `UnitedNations` is a `Physical` thing
   • `UnitedNations` can't have a `PhysicalState` as an `Attribute` so it's not a `Gas`
4. • So from list 2 above, the thing in an `AirStream` could be the `UnitedNations` because that's a `Physical` thing
   • But from list 3 `UnitedNations` can't be a `Gas` and anything located in an `AirStream` is a `Gas`
   • **contradiction**

Distilling this explanation from such a long and complicated proof is quite challenging, but proof ablation quickly narrows the search for a fix by determining that `UnitedNations` is not central to the proof, as any other `Collection` such as `UnitedStatesMinorOutlyingIslands` would contribute to the contradiction. The fact that cyclones are a hazard in the Southern ocean is also incidental, but proof ablation can't determine that, since the only chain of reasoning that creates an instance of an airflow results from a statement in the Geography.kif domain ontology that `(naturalHazardTypeInArea SouthernOcean CyclonicStorm)`. In contrast, the following formula is essential:

```
(=>
    (and
        (instance ?AS AirStream)
        (located ?AIR ?AS))
    (attribute ?AR Gas))
```

The error in this formula is using `located` when the relation `part` should have been used. A `Bird` could be `located` in the `AirStream` and it wouldn't be a `Gas`. With that change to the formula, E and Vampire no longer find this contradiction.

[3] https://github.com/ontologyportal/sigmakee/blob/master/src/java/com/articulate/sigma/Graph.java

[4] See https://github.com/ontologyportal/sigmakee/blob/master/src/java/com/articulate/sigma/KB.java#L3660.

[5] https://www.ontologyportal.org/FLAIRScontra.txt

[6] https://www.ontologyportal.org/FLAIRSproof.gif

[7] řlhttps://github.com/ontologyportal/sumo/blob/master/Weather.kif

[8] For example, https://sigma.ontologyportal.org:8443/sigma/Browse.jsp?kb=SUMO&term=CyclonicStorm

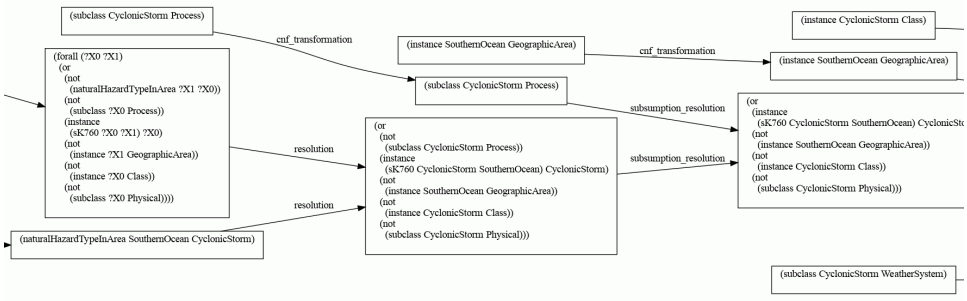

Figure 2: A section of a graphical view of a proof by contradiction

Workflow

A typical workflow for SUMO-based theory development makes use of all the tools described above. The ontologist authors a theory using the SUMOjEdit text editor, which provides immediate feedback to the author with syntax checking and spell checking of theory symbols. A menu option executes a set of static tests that looks for missing variables, incorrect arity for existing relations, type checking and disjointness violations. After completing a first draft of a substantial new theory, the user can then test it by asking a theorem prover to prove falsehood, without the presence of a negated conjecture. To perform this step, the Sigma system is first called to translate a subset of the theory that is either FOL (the TPTP language), or FOL with typed arithmatic (the TF0 language). If an undirected search for a contradiction does not succeed, the user can apply EAXFilter to conduct a more extensive search, asking a prover to look for a contradiction in each of tens of thousands of generated subsets of a theory.

Hand-written tests can serve a function similar to EAXFilter in terms of focusing the attention of a prover on a portion of the theory. Since 2007, SUMO has had such a set of roughly 50 tests (Pease et al. 2008). Many more would be desirable, but even this small set has helped to find contradictions over the years. These tests also serve to alert the user when a change to SUMO may have inadvertently removed axiomatic support for a test that should succeed. These tests are available online[9].

If a contradiction is found, the user then employs Sigma to translate the TPTP3 proof language back into SUMO's native SUO-KIF language. Sigma will generate a list of the source axioms used in the proof, as well as a graphical proof. The user can also run the proof ablation utility to get hints on which axioms from the proof are always present when a contradiction is found, and which are not found in all proofs, thereby allowing the user to focus on critical axioms that appear to be causing the problem.

Conclusion

Working with large theories in an expressive logic supports rich and comprehensive modeling of the world. With that power comes significant challenges. Tools and methods that address those challenges are needed and we have presented some of them, and their implementation and use in the context of the development of SUMO. Although currently usable, and released as open source, considerable effort remains to make their implementation faster, and create modern user interfaces that make their employment simpler. Currently, the axiom filtering is only supported for TPTP and not the TFF family of languages that support arithmetic, or the THF higher-order logic language implemented in provers such as Satalax (Brown 2013) and LEO-III (Steen and Benzmüller 2020), but also increasingly supported by E (Vukmirović et al. 2021). Use of THF will ultimately be required to realize the full potential of SUMO in automated theorem proving.

[9] https://github.com/ontologyportal/sumo/tree/master/tests

References

Benzmüller, C., and Pease, A. 2010. Progress in automating higher-order ontology reasoning. In Konev, B.; Schmidt, R.; and Schulz, S., eds., *Workshop on Practical Aspects of Automated Reasoning (PAAR-2010)*. Edinburgh, UK: CEUR Workshop Proceedings.

Benzmüller, C.; Paulson, L.; Theiss, F.; and Fietzke, A. 2008. (2008). *LEO-II - A Cooperative Automatic Theorem Prover for Higher-Order Logic. In Proceedings of the Fourth International Joint Conference on Automated Reasoning (IJCAR'08), LNAI volume* 5195:162–170.

Brown, C. E. 2013. Reducing higher-order theorem proving to a sequence of sat problems. *J. Autom. Reason.* 51(1):57–77.

Claessen, K., and Sörensson, N. 2003. New techniques that improve mace-style finite model finding. In *Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*.

Hoder, K., and Voronkov, A. 2011. Sine qua non for large theory reasoning. In *Proceedings of the 23rd International Conference on Automated Deduction*, CADE'11, 299–314. Berlin, Heidelberg: Springer-Verlag.

Kovács, L., and Voronkov, A. 2013. First-order theorem proving and vampire. In *Proceedings of the 25th International Conference on Computer Aided Verification*, volume 8044 of *CAV 2013*, 1–35. New York, NY, USA: Springer-Verlag New York, Inc.

Niles, I., and Pease, A. 2001. Toward a Standard Upper

Ontology. In Welty, C., and Smith, B., eds., *Proceedings of the 2nd International Conference on Formal Ontology in Information Systems (FOIS-2001)*, 2–9.

Pease, A., and Schulz, S. 2014. Knowledge Engineering for Large Ontologies with Sigma KEE 3.0. In *The International Joint Conference on Automated Reasoning*.

Pease, A.; Sutcliffe, G.; Siegel, N.; and Trac, S. 2008. The Annual SUMO Reasoning Prizes at CASC. In *Proceedings of IJCAR '08 Workshop on Practical Aspects of Automated Reasoning (PAAR-2008)*. CEUR Workshop Proceedings.

Pease, A.; Sutcliffe, G.; Siegel, N.; and Trac, S. 2010. Large Theory Reasoning with SUMO at CASC. *AI Commun., Special issue on Practical Aspects of Automated Reasoning* 23(2-3):137–144.

Pease, A. 2011. *Ontology: A Practical Guide*. Angwin, CA: Articulate Software Press.

Pease, A. 2019. Arithmetic and inference in a large theory. In *AI in Theorem Proving*.

Pease, A. 2020. A programmer's text editor for a logical theory: The sumojedit editor (system description). In Peltier, N., and Sofronie-Stokkermans, V., eds., *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, 472–479. Springer.

Prawitz, D. 1965. *Natural deduction: a proof-theoretical study*. Ph.D. Dissertation, Almqvist & Wiksell.

Read, S. 2003. Logical consequence as truth-preservation. *Logique and Analyse* 183(4):479–493.

Schulz, S., and Sutcliffe, G. 2015. Proof generation for saturating first-order theorem provers. In Delahaye, D., and Woltzenlogel Paleo, B., eds., *All about Proofs, Proofs for All*, volume 55 of *Mathematical Logic and Foundations*. London, UK: College Publications. 45–61.

Schulz, S.; Sutcliffe, G.; Urban, J.; and Pease, A. 2017. Detecting inconsistencies in large first-order knowledge bases. In *Proceedings of CADE 26*, 310–325. Springer.

Schulz, S.; Cruanes, S.; and Vukmirović, P. 2019. Faster, higher, stronger: E 2.3. In Fontaine, P., ed., *Proc. of the 27th CADE, Natal, Brasil*, number 11716 in LNAI, 495–507. Springer.

Schulz, S. 2002. E – A Brainiac Theorem Prover. *Journal of AI Commun.* 15(2/3):111–126.

Steen, A., and Benzmüller, C. 2020. The higher-order prover leo-iii. In *ECAI*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, 2937–2938. IOS Press.

Stickel, M. E.; Waldinger, R. J.; and Chaudhri, V. K. 2000. A guide to SNARK. Technical report, SRI International, Menlo Park, United States.

Stump, A.; Sutcliffe, G.; and Tinelli, C. 2014. Starexec: A cross-community infrastructure for logic solving. In Demri, S.; Kapur, D.; and Weidenbach, C., eds., *Automated Reasoning*, 367–373. Cham: Springer International Publishing.

Sutcliffe, G.; Schulz, S.; Claessen, K.; and Baumgartner, P. 2012. The TPTP Typed First-order Form with Arithmetic. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2012)*, 406–419.

Sutcliffe, G. 2007. TPTP, TSTP, CASC, etc. In *Proceedings of the Second International Conference on Computer Science: Theory and Applications*, CSR'07, 6–22. Berlin, Heidelberg: Springer-Verlag.

Sutcliffe, G. 2019. The CADE-27 Automated theorem proving System Competition - CASC-27. *AI Communications* 32(5–6):373–389.

Trac, S.; Puzis, Y.; and Sutcliffe, G. 2007. An interactive derivation viewer. *Electron. Notes Theor. Comput. Sci.* 174:109–123.

Trac, S.; Sutcliffe, G.; and Pease, A. 2008. Integration of the TPTPWorld into SigmaKEE. In *Proceedings of IJCAR '08 Workshop on Practical Aspects of Automated Reasoning (PAAR-2008)*. CEUR Workshop Proceedings.

Vukmirović, P.; Blanchette, J. C.; Cruanes, S.; and Schulz, S. 2021. Extending a Brainiac Prover to Lambda-free Higher-Order Logic. *International Journal on Software Tools for Technology Transfer*.