SPIKE, an automatic theorem prover – revisited

Sorin Stratulat

SPIKE, an automatic theorem prover – revisited

Sorin Stratulat
Université de Lorraine, CNRS, LORIA
F-57000 Metz, France
Email: sorin.stratulat@loria.fr

***Abstract*—SPIKE, an induction-based theorem prover built to reason on conditional theories with equality, is one of the few formal tools able to perform automatically mutual and lazy induction. Designed at the beginning of 1990s, it has been successfully used in many non-trivial applications and served as a prototype for different proof experiments and extensions. The first paper introducing SPIKE is [14], published shortly after the tool was created. The goal of this paper is to highlight and bring together in one spot the major changes supported by SPIKE since then.**

I. Introduction

Historically, SPIKE was built during a period when several formula-based Noetherian induction methods issued from Musser's completion-based inductionless induction (or proof-by-consistency) technique [32] have been designed. Some of them have been implemented into theorem provers, for example, RRL [28] integrated the test-set induction method [27], and Focus [20] a generalization of the term-rewriting induction [21] for conditional theories. Inspired by the rewriting techniques previously tested with the ORME system [31], SPIKE [14] implemented a different induction method [15], [29] that combines features from explicit induction and inductive completion techniques.

As time went by, SPIKE was continuously considered among the 'active' automatic induction-based provers; it mainly served as a prototype for testing several extensions of conditional theories and induction-based reasoning techniques that led to many successful proof experiments on non-trivial applications. The goal of the paper is to highlight the major changes supported by SPIKE since the publication of [14], which gave rise to its current version. The source code, examples of specification files, and papers related to different applications and extensions are available online [41].

In the rest of the paper, we set the theoretical backgrounds of the reasoning by induction on equational clauses, then we introduce the inference system of SPIKE and the layout of a standard specification file. In the end, we show how to prove conjectures and interact with the tool.

II. The SPIKE prover

a) The 'reasoning by induction' setting: SPIKE implements an instance of the Noetherian induction principle applied on a Noetherian poset of equational clauses (or just clauses). Compared to other instances adapted for first-order reasoning, several clauses can be simultaneously tested to verify whether they are consequences of a given set of axioms Ax written as conditional equalities. SPIKE can reason on sorted and constructor-based specifications that should satisfy some properties, like the ground convergence and (strongly sufficient) completeness. These constraints guarantee the existence of the *initial model* for Ax. Formally, assuming that $\mathcal{M}$ is the initial model of Ax, we denote by $\Phi \models_{\mathcal{M}} \phi$ the fact that the clause ϕ is an initial $\mathcal{M}$-consequence (or just *consequence*) of a set of conditional equalities Φ. ϕ is initially $\mathcal{M}$-valid (or just valid) and is denoted by $\models_{\mathcal{M}} \phi$ iff it is a consequence of Ax. In general, given $(\mathcal{E}, \leq)$ a non-empty poset of formulas that is *wellfounded* (or Noetherian), i.e., any strictly decreasing sequence of formulas from $\mathcal{E}$ is finite, the formula-based instance [37] of the Noetherian induction principle states that if $\{\gamma \mid \gamma \in \mathcal{E}, \gamma < \delta\} \models_{\mathcal{M}} \delta$, for any formula $\delta \in \mathcal{E}$, then for all $\rho \in \mathcal{E}$, $\models_{\mathcal{M}} \rho$. Put it in a simpler way for our logical context, during the proof of a clause, smaller clauses can be used in terms of induction hypotheses (IHs). This makes it a natural choice for reasoning on proofs requiring *lazy induction*, when the IHs are generated by need, and *mutual induction*, when different clauses need each other to cross fertilize for making the induction reasoning successful. The induction proof method used by SPIKE, called *implicit induction*, is an application of this principle using reductive techniques as rewriting. It was firstly suggested in [29] and formally presented in [21]. The mutual induction feature helped SPIKE to prove the Gilbreath Card trick problem [26], firstly with 5 lemmas [16] then with only 2 lemmas [17], while other provers using different proof techniques succeeded with significantly more lemmas (see [17] for a comparison).

The ground convergence and completeness properties of a specification can be checked more easily, by using syntactic criteria, if the specification is *many sorted* and the set of function symbols is split into *constructor* and *defined function* symbols. SPIKE was initially designed to deal with *free* constructors such that there is no equality relation between any two different constructor symbols. Several extensions have been introduced in SPIKE since [14] in order to deal with: i) non-free constructors [13], ii) parameterized specifications [9], [10], iii) associative-commutative theories [6], iv) observational proofs [7], [18], and v) simultaneous check of the completeness and ground convergence properties of a specification [12]. Most of them led to distinct proof systems that are no longer maintained in spite of their theoretical and practical interests.

b) The inference system: In [14], the inference rules and the proof strategy implementing the implicit induction method were built-in. Each rule is a transition between pairs (E, H), where E are *conjectures* and H are *premises* consisting of previously processed conjectures that do not have minimal *counterexamples*, i.e., minimal ground clauses that are not valid. By applying a rule, a conjecture from the current proof state is replaced by a potentially empty set of new conjectures,

and may be added as a premise in order to participate to further inference steps. Proof *derivations* are built by successively applying inference rules starting from an initial state. They may finish with i) *success*, if they end with an empty set of conjectures, ii) *error*, if a counterexample is detected, and iii) *failure*, if none of the previous cases is encountered and no rule can be applied. A *proof* is a successful derivation that starts with an empty set of premises.

Later on, different proof needs led to hardcode into the system several variants of a same rule. More flexibility has been achieved with the addition of a strategy language [1] allowing to define new proof strategies by the user. It has been combined later with a methodology for building modular inference rules using *contextual cover sets* (CCSs) [35]. The core of the methodology is an abstract inference system made of two rules: ADDPREMISE and SIMPLIFY, defined as:

ADDPREMISE: $(E \cup \{\phi\},\ H) \vdash_A (E \cup \Phi,\ H \cup \{\phi\})$,
if, for any counterexample (ctx.) $\phi\tau$ of ϕ, it is a ctx. ψ in
i) $E \cup \Phi$ such that $\psi < \phi\tau$, or
ii) H such that $\psi \leq \phi\tau$.

SIMPLIFY: $(E \cup \{\phi\},\ H) \vdash_A (E \cup \Phi,\ H)$,
if, for any ctx. $\phi\tau$ of ϕ, there is a ctx. ψ in
$E \cup \Phi \cup H$ such that $\psi \leq \phi\tau$.

Each inference rule replaces a conjecture ϕ with a potentially empty set of new conjectures Φ. Φ is built in two steps as a CCS of ϕ, thanks to the compositional properties of CCSs. Firstly, an intermediate CCS of the replaced conjecture is built, then for each intermediate clause another CCS is built and stored as new conjectures. The set of IHs allowed by a rule to be used when building a CCS is referred to as *context*. ADDPREMISE adds ϕ as a premise and SIMPLIFY allows bigger contexts. It has been shown in [35] that i) the abstract inference system is *sound*, i.e., one can conclude from a proof that its initial conjectures are valid, and ii) the inference rules define the biggest contexts compared to similar abstract rules proposed in the literature. For practical reasons, the abstract system was extended with a third rule, DELETE, that is a particular case of SIMPLIFY when Φ is empty.

Any SPIKE inference rule instantiates one of the abstract rules by implementing its elementary CCSs, i.e. the CCSs that are not built using composition operations, by the means of *reasoning modules*. A reasoning module can produce a CCS with a particular reasoning technique using in terms of IHs clauses from the context defined by the instantiated abstract rule. The main reasoning techniques are based on rewriting, case analysis and variable instantiations.

We give as example the definition of a rewriting-based inference operation implemented as an instance of SIMPLIFY:

```
rewriting_rule =
simplify(id,[rewriting(rewrite,L|R|C,*)]);
```

In the first step, the identity reasoning module `id` builds $\{\phi\}$ as the intermediate CCS for ϕ. The application of the rule succeeds if, in the second step, the `rewriting` module succeeds to rewrite once, due to the `rewrite` argument and at any position ($\star$), the only clause ϕ of $\{\phi\}$ with conditional rewrite rules from the lemmas (`L`), axioms (`R`) and current context (`C`). The resulting clause is the unique clause of Φ.

Some of the reasoning techniques have been changed since [14]. For example, the technique for instantiating variables with elements of a test-set, based on the depth of the lhs of the axioms [17], was replaced by a narrowing technique involving only the axioms defining the head symbol of some (sub)term including the variables to be instantiated [5].

New reasoning techniques have been added to deal with non-trivial applications. The implementation of a combination between a decision procedure for linear arithmetic and a congruence closure algorithm [3], [34] allowed to validate the MJRTY algorithm [19] using a lemma proposed by N. Shankar (according to [25]); also, more than 60% of the conjectures required to certify the conformity algorithm for a telecommunication protocol [33] have been automatically proved. This implementation, as well as the 'black-box' integration schema of the Z3 [23] SMT solver as a reasoning module, are described in [38]. SPIKE also includes several decision procedures for proving inductive theorems without induction reasoning [2]. They help to decide the inductive validity of equations involving natural numbers and lists.

The validation of the JavaCard platform [5] was the most challenging case study ever experienced by SPIKE. The inference system has been adapted to manage variables of parameterized sorts as well as existential variables. New inference rules have been designed to better handle the information from the conditional part of (conditional) conjectures, like i) the *auto simplification rule* in order to rewrite with an equality condition other parts of the conjecture, and ii) the *augmentation rule* which adds new conditions issued from the conclusion of a conditional equality given as lemma if the conditions of the lemma are proved from the conditions of the conjecture. The efficiency of the implementation has been improved for dealing with specifications counting more than 400 defined function symbols and 2200 axioms, for example by recording the failure context at the conjecture level in order to avoid useless computation.

As shown in [21], the implicit induction method is based on a unique induction ordering, globally defined over the set of clauses derived during a proof. It is implemented by *reductive* inference systems such that, at the ground level, the new conjectures from any proof step are smaller than (and sometimes equal to) the replaced conjecture. The reductive techniques, as rewriting, introduce new ordering constraints to be satisfied by the specification as well as the conjectures from the proof derivation. The induction ordering is the multiset extension of the mpo ordering [4] over terms using a precedence over the function symbols provided by the user. The mpo ordering also serves to orient the axioms into rewrite rules. Since some reasoning techniques, like the instantiation of variables from the current conjecture, require the reduction of the instances by rewriting, SPIKE warns the user if the mpo ordering built from the input precedence cannot orient the axioms into rewrite rules. If no precedence is provided by the user, SPIKE analyses the axioms and tries to infer a successful precedence.

The inference system of SPIKE has been extended to implement for the first time reductive-free cyclic proofs [37], by keeping the best features of explicit and implicit induction

reasoning. A *cycle* consists of a circular linked list of proof derivation chunks, called *history chunks*. Each link symbolises the application of an instance of the head conjecture from a history chunk as IH in the proof of the conjecture ending the previous history chunk in the list. By following a particular proof strategy, referred to as the *DRaCuLa* strategy, the cycle can discharge its linking IHs by checking ordering constraints involving only instances of the conjectures starting the history chunks. Therefore, the cyclic induction reasoning allows to use non-reductive proof techniques along history chunks and axioms not orientable into rewrite rules as long as the ordering constraints are satisfied.

A useful property for an inference system is the *refutational soundness* which guarantees that, whenever a counterexample is detected at the current step, the initial conjectures are refuted. Very few inference rules implemented by SPIKE may add new counterexamples during the proof derivation, e.g., by the generalisation of existential into universal variables [5]. By attaching history information to each conjecture, the detected counterexamples can lead to particular ground instances of the initial conjectures that can be further checked for validity.

The *refutational completeness* is another useful property, satisfied by previous inference systems [11], [15]. Since the addition of a strategy language, this property is no longer guaranteed because one of the conditions to be satisfied is the fairness of the proof strategy. However, the user still can use some built-in strategies known to be refutationally complete. A classical proof strategy mainly privileges the inference rules that are instances of DELETE, then SIMPLIFY, and finally ADDPREMISE. We give as example the definition of the `fullind` strategy.

```
% instance of Delete
tautology_rule = delete(id, [tautology]);
% instance of Simplify
total_case_rewriting_rule =
   simplify(id, [total_case_rewriting (
      simplify_strat, R, *)]);
% instances of AddPremise
case_rewriting_rule =
  add_premise(total_case_rewriting(
     simplify_strat, R, *),[id]);
split_rule = add_premise(generate,[id]);
% proof strategies
stra = repeat (try (tautology_rule,
  rewriting_rule,total_case_rewriting_rule
  print_goals, case_rewriting_rule ));
fullind = (repeat(stra, split_rule),
  print_goals_with_history);
start_with: fullind
```

The reasoning module `total_case_rewriting`, implementing the conditional rewriting technique, was used to build instances of both SIMPLIFY and ADDPREMISE. `try` and `repeat` take a list of rules as arguments. `try` visits each rule in the list until the first succeeds. `repeat` executes them repeatedly until no new conjecture is produced or a counterexample is found. The `print_*` rules print the current state of the proof. `start_with` points to the used strategy.

c) Layout of a specification file: The structure of a standard SPIKE specification from the file *name*`.spike` is:

`specification:` *name*
`sorts:` *list of sorts*
`constructors:` *list of constructor symbols*
`defined functions:` *list of defined function symbols*
`axioms:` *list of axioms for each defined function symbol*
% the precedence used by the induction ordering
`greater:` *list of precedences over the function symbols*
`equiv:` *list of equivalent function symbols*
% the completeness and ground convergence properties
`properties:` *list of properties*
% the proof strategies and conjectures
`strategy:` *list of inference rules and proof strategies*
`conjectures:` *list of conjectures*

d) User interactions: The proofs, generated by the command `spike_bc` *name*`.spike`, are highly automatic, the user interactions mainly defining i) the precedence used by the mpo ordering, ii) the inference rules and the proof strategy, iii) the precedence over the head symbols of the (sub)terms to which the new instantiation technique can be applied, and iv) lemmas. Once a conjecture has been proved, it can participate as lemma in the proof of further conjectures listed in the `conjectures` section.

On the other hand, the generated proofs may involve many non-trivial inference steps for which the human checking process is tedious and error-prone. We have shown how to i) validate SPIKE proofs [36], [39] using the certification environment provided by the Coq system [40], and ii) define Coq tactics that call directly SPIKE and transform the generated proofs into Coq scripts [24], according to a methodology that automatically translates into Coq script the proof steps performed by most of its inference rules.

The user can also interact with SPIKE by the means of i) extra sections, for example `use: nats;`[1] for activating the combination of the decision procedure for linear arithmetic and the congruence closure procedure, and ii) command-line arguments given to `spike_bc`, as:

`-debug`: to identify syntactic errors in the specification file,
`-maximal`: to print the proof in detail,
`-coqc_spec` and `-coqc`: to generate the Coq specification and translate the generated proof into Coq script, respectively, and
`-dracula`: to generate cyclic proofs using the DRaCuLa strategy.

e) Coding languages: SPIKE was initially written in the 'light' version of the Caml [22] functional language by Adel Bouhoula during his PhD thesis [8]. SPIKE has been completely redesigned for adopting an object-oriented paradigm along the $\sim 18\,500$ lines of OCaml [30] code.

Some of the original features are still missing, like the graphical interface and the procedures for checking the completeness and ground convergence properties.

[1] To be added just after the `specification` section.

III. Conclusions

We have given an overview of the current version of SPIKE and highlighted the main changes and extensions since [14].

a) Acknowledgments: The author would like to thank Adel Bouhoula and Michaël Rusinowitch, as well as the people involved in supporting, developing and using SPIKE.

References

[1] I. Alouini and A. Bouhoula. Un langage de stratégie pour SPIKE. Working document, 1997.

[2] T. Aoto and S. Stratulat. Decision procedures for proving inductive theorems without induction. In *Proc. of the 16th International Symposium on Principles and Practice of Declarative Programming (PPDP 2014)*, pages 237–248. ACM Press, 2014.

[3] A. Armando, M. Rusinowitch, and S. Stratulat. Incorporating decision procedures in implicit induction. *Journal of Symbolic Computation*, 34(4):241–258, 2002.

[4] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[5] G. Barthe and S. Stratulat. Validation of the JavaCard platform with implicit induction techniques. In R. Nieuwenhuis, editor, *RTA (Rewriting Techniques and Applications)*, volume 2706 of *LNCS*, pages 337–351. Springer, 2003.

[6] N. Berregeb, A. Bouhoula, and M. Rusinowitch. SPIKE-AC: A system for proofs by induction in associative-commutative theories. In H. Ganzinger, editor, *Rewriting Techniques and Applications*, volume 1103 of *Lecture Notes in Computer Science*, pages 428–431. Springer Berlin Heidelberg, 1996.

[7] N. Berregeb, A. Bouhoula, and M. Rusinowitch. Observational proofs with critical contexts. In *Fundamental Approaches to Software Engineering*, pages 38–53, 1998.

[8] A. Bouhoula. *Preuves automatiques par récurrence dans les théories conditionnelles*. PhD thesis, Université Henri Poincaré - Nancy I, 1994.

[9] A. Bouhoula. SPIKE: A system for sufficient completeness and parameterized inductive proofs. In A. Bundy, editor, *Automated Deduction — CADE-12*, volume 814 of *Lecture Notes in Computer Science*, pages 836–840. Springer Berlin Heidelberg, 1994.

[10] A. Bouhoula. Using induction and rewriting to verify and complete parameterized specifications. *Theoretical Computer Science*, 170:170–1, 1996.

[11] A. Bouhoula. Automated theorem proving by test set induction. *Journal of Symbolic Computation*, 23(1):47–77, 1997.

[12] A. Bouhoula. Simultaneous checking of completeness and ground confluence for algebraic specifications. *ACM Transactions on Computational Logic (TOCL)*, 10(3):1–33, 2009.

[13] A. Bouhoula and J.P. Jouannaud. Automata-Driven Automated Induction. *Information and Computation*, 169(1):1–22, 2001.

[14] A. Bouhoula, E. Kounalis, and M. Rusinowitch. SPIKE, an automatic theorem prover. In *Logic Programming and Automated Reasoning (LPAR)*, pages 460–462, 1992.

[15] A. Bouhoula, E. Kounalis, and M. Rusinowitch. Automated mathematical induction. *Journal of Logic and Computation*, 5(5):631–668, 1995.

[16] A. Bouhoula and M. Rusinowitch. Automatic case analysis in proof by induction. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI)*, volume 1, pages 88–94, 1993.

[17] A. Bouhoula and M. Rusinowitch. Implicit induction in conditional theories. *Journal of Automated Reasoning*, 14(2):189–235, 1995.

[18] A. Bouhoula and M. Rusinowitch. Observational proofs by rewriting. *Theoretical Computer Science*, 275(1–2):675–698, 2002.

[19] R.S. Boyer and J S. Moore. MJRTY—A Fast Majority Vote Algorithm. *Automated Reasoning: Essays in Honor of Woody Bledsoe*, 1991.

[20] F. Bronsard and U. S. Reddy. Conditional rewriting in Focus. In *Conditional and Typed Rewriting Systems*, pages 1–13, 1991.

[21] F. Bronsard, U.S. Reddy, and R. Hasker. Induction using term orderings. In *CADE (Conf. on Automated Deduction)*, volume 814 of *LNCS*, pages 102–117. Springer, 1994.

[22] G. Cousineau and M. Mauny. *The Functional Approach to Programming with Caml*. Cambridge University Press, 1998.

[23] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[24] A. Henaien and S. Stratulat. Performing implicit induction reasoning with certifying proof environments. In A. Bouhoula, T. Ida, and F. Kamareddine, editors, *Proceedings Fourth International Symposium on Symbolic Computation in Software Science, Gammarth, Tunisia, 15-17 December 2012*, volume 122 of *Electronic Proceedings in Theoretical Computer Science*, pages 97–108. Open Publishing Association, 2013.

[25] D. J. Howe. Reasoning about functional programs in Nuprl. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *Lecture Notes in Computer Science*, pages 145–164. Springer Verlag, 1993.

[26] G. Huet. The Gilbreath trick: A case study in axiomatisation and proof development in the Coq proof assistant. Technical Report RR-1511, INRIA, 1991.

[27] D. Kapur, P. Narendran, and H. Zhang. Proof by induction using test sets. In *8th International Conference on Automated Deduction*, volume 230 of *Lecture Notes Computer Science*, pages 99–117. Springer, 1986.

[28] D. Kapur and H. Zhang. RRL: A rewrite rule laboratory. In Ewing Lusk and Ross Overbeek, editors, *9th International Conference on Automated Deduction*, volume 310 of *LNCS*, pages 768–769. Springer Berlin / Heidelberg, 1988.

[29] E. Kounalis and M. Rusinowitch. Mechanizing inductive reasoning. In *Proceedings of the eighth National conference on Artificial intelligence - Volume 1*, AAAI'90, pages 240–245. AAAI Press, 1990.

[30] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system - release 4.00. Documentation and user's manual*. INRIA.

[31] P. Lescanne. Implementation of completion by transition rules + control: ORME. In H. Kirchner and W. Wechler, editors, *Algebraic and Logic Programming*, volume 463 of *Lecture Notes in Computer Science*, pages 262–269. Springer Berlin Heidelberg, 1990.

[32] D. R. Musser. On proving inductive properties of abstract data types. In *POPL*, pages 154–162, 1980.

[33] M. Rusinowitch, S. Stratulat, and F. Klay. Mechanical verification of an ideal incremental ABR conformance algorithm. *Journal of Automated Reasoning*, 30(2):153–177, 2003.

[34] S. Stratulat. *Preuves par récurrence avec ensembles couvrants contextuels. Applications à la vérification de logiciels de télécommunications.* PhD thesis, Université Henri Poincaré, Nancy I, November 2000.

[35] S. Stratulat. A general framework to build contextual cover set induction provers. *Journal of Symbolic Computation*, 32(4):403–445, 2001.

[36] S. Stratulat. Integrating implicit induction proofs into certified proof environments. In *IFM'2010 (8th International Conference on Integrated Formal Methods)*, volume 6396 of *Lecture Notes in Computer Science*, pages 320–335, 2010.

[37] S. Stratulat. A unified view of induction reasoning for first-order logic. In A. Voronkov, editor, *Turing-100 (The Alan Turing Centenary Conference)*, volume 10 of *EPiC Series*, pages 326–352. EasyChair, 2012.

[38] S. Stratulat. Implementing reasoning modules in implicit induction theorem provers. In *SYNASC (International Symposium on Symbolic and Numeric Algorithms for Scientific Computing)*, pages 133–140. IEEE Computer Society, 2014.

[39] S. Stratulat and V. Demange. Automated certification of implicit induction proofs. In *CPP'2011 (First International Conference on Certified Programs and Proofs)*, volume 7086 of *Lecture Notes Computer Science*, pages 37–53. Springer Verlag, 2011.

[40] The Coq development team. *The Coq Reference Manual*. INRIA, 2020. http://coq.inria.fr/doc.

[41] The SPIKE development team. *The SPIKE prover*, 2020. https://github.com/sorinica/spike-prover.