
The 2006 Federated Logic Conference

The Seattle Sheraton Hotel and Towers

Seattle, Washington

August 10 - 22, 2006



IJCAR'06 Workshop

DISPROVING'06: Non-Theorems, Non-Validity, Non-Provability

August 16th, 2006

Proceedings

Editors:

W. Ahrendt, P. Baumgartner, H. de Nivelle

Preface

Our field is called *automated theorem proving* because traditionally it has been concerned with the art of finding proofs automatically. In the beginning, researchers were motivated by the wish to build computer systems that can automatically solve difficult mathematical problems. When searching for a difficult proof, it is acceptable for a system to consume all resources and not to recognise false theorems.

However, in the last years one has become aware of the fact that for applications, one also needs to be able to efficiently identify non-theorems. For example, automated theorem proving systems are now being used as assistants which must automatically solve easy subtasks in large, interactive projects. For such problems, the requirements to the automated theorem prover are different: The input problems are not terribly hard, usually contain additional irrelevant information, and often they are not provable. In case the subgoal is incorrect, it is not acceptable to simply remain silent and consume all resources in an interactive system.

In this year's DISPROVING workshop, we have again collected an interesting range of papers covering both theory and practice of disproving. Most of the papers do not only discuss theoretical contributions, but also working implementations. This demonstrates that the area of disproving is both theoretically interesting and practically relevant.

This year's workshop consists of seven contributed talks and two invited talks by Jian Zhang and Silvio Ranise. We thank the PC for their reviewing efforts:

Johan Bos	Simon Colton
Christian Fermüller	Bernhard Gramlich
Bill McCune	Michael Norrish
Renate Schmidt	Carsten Schürmann
John Slaney	Graham Steel
Cesare Tinelli	Calogero Zarba


We also thank the following additional reviewer:

John Charnley

July 2006,

Wolfgang Ahrendt, Peter Baumgartner, Hans de Nivelle

Contents

Model and Counterexample Search: Successes and Challenges <i>Jian Zhang</i>	1
Sudokus as Logical Puzzles <i>Thomas Hillenbrand, Dalibor Topic, Christoph Weidenbach</i>	2
Automated relative consistency proving <i>André Rognes</i>	13
Diagnosing a Failed Proof in Fault-Tolerance: A Disproving Challenge Problem <i>Lee Pike, Paul Miner, Wilfredo Torres-Pomales</i>	24
Program Slicing and Middle-Out Reasoning for Error Location and Repair <i>Louise A. Dennis</i>	34
Satisfiability Solving for Program Verification: towards the efficient combination of Automated Theorem Provers and Satisfiability Modulo Theory Tools <i>Silvio Ranise</i>	49
A Fast Disprover for  VeriFun <i>Markus Aderhold, Christoph Walther, Daniel Szallies, Andreas Schlosser</i>	59
Predicting Failures of Inductive Proof Attempts <i>Mahadevan Subramaniam, Deepak Kapur, Stephan Falke</i>	70
Computing Finite Models by Reduction to Function-Free Clause Logic <i>Peter Baumgartner, Alexander Fuchs, Hans de Nivelle, Cesare Tinelli</i>	82

Model and Counterexample Search: Successes and Challenges

Jian Zhang

Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences

To disprove a conjecture, we can try to find a counterexample (i.e., a model of the axioms and premises, in which the conjecture does not hold). For the first-order predicate logic, model finding is an undecidable problem in general. But in many cases, a satisfiable formula has a finite model, and we can find the model by exhaustive search. A few decades ago, this seemed to be infeasible in practice. But thanks to many people's efforts, the approach became quite successful. Efficient tools have been developed, and open problems have been solved. In this talk, I will give a brief overview of the basic techniques of finite model searching and describe how such a tool can be used. Then I will discuss some research issues and challenging benchmark problems.

Sudokus as Logical Puzzles

Thomas Hillenbrand, Dalibor Topic, Christoph Weidenbach
 Max-Planck-Institut für Informatik, D-66123 Saarbrücken
 {hillen,topic,weidenbach}@mpi-inf.mpg.de

1 Introduction

Currently Sudoku puzzles are *en vogue*. They show up in many newspapers and magazines; and puzzle collections are printed as books. The goal of a Sudoku game is to place the digits 1 through 9 onto a 9×9 -grid with some given numbers, see Figure 1 for an example. The grid is subdivided into nine 3×3 -blocks. Numbers may only be placed onto the grid such that each occurs exactly once in each block, row and column.

The question how many possible final configurations exist, has been answered for 9×9 -grids by Felgenhauer and Jarvis [RJ06]: 5,472,730,538 ones, modulo elementary symmetries.

Most of the published puzzles have a unique solution. The question how many givens are necessary for uniqueness is still open. Royle [Roy05] has collected over 35,000 different puzzles with 17-givens placements, the current known minimum. We used these puzzles for our exhaustive experiments.

Naturally, the standard 9×9 -puzzle can easily be solved mechanically by various methods, amongst which are constraint-based approaches [Sim05]. More recently, several authors [LO06, Web05] have shown how to crack these problems in under one second via propositional encodings. This does not come as a big surprise, because SAT solvers have proven their impact on finite-domain problems in the past. Bill McCune has presented a formulation in first-order logic with equality that can be efficiently solved by Mace4 [McC03]. To this end Mace4 generates an explicit Herbrand model assumption of the clauses and explores equations in an intelligent way. Typically, first-order theorem proving is not assumed to be efficient on such problems. One contribution of this paper is to present a more compact encoding that our first-order theorem prover SPASS [WBH⁺02] can cope with in the blink of an eye. A puzzle is solvable iff its encoding is satisfiable iff SPASS disproves that it entails the empty clause, by exhibiting a clausal representation of a Herbrand model for it. Indeed disproving is effective here because SPASS always terminates on the formula class at hand.

4		1
2		
	5	4 7
8		3
1	9	
3	4	2
5	1	
	8	6

6 9 3	7 8 4	5 1 2
4 8 7	5 1 2	9 3 6
1 2 5	9 6 3	8 7 4
9 3 2	6 5 1	4 8 7
5 6 8	2 4 7	3 9 1
7 4 1	3 9 8	6 2 5
3 1 9	4 7 5	2 6 8
8 5 6	1 2 9	7 4 3
2 7 4	8 3 6	1 5 9

Figure 1: A 17-givens Sudoku puzzle and its solution

In general, problem solving via some logical formalization requires the decidability of the logic by some adequate calculus. In particular, solving a Sudoku problem, if possible, means to compute the unique Herbrand model for the given partially filled grid. What makes Sudokus decidable is that the problem is inherently finite. So solving it in first-order logic requires the possibility to code and reason over formulae over finite sorts efficiently.

Dealing with finite sorts in first-order resolution-based calculi is very often painful. Despite the principal decidability, superposition implementations typically will not terminate. Per se, superposition is not a decision procedure for finite sort problems.

On the other hand, finite sorts occur frequently and naturally as part of many relevant theories and applications. So it is worthwhile to turn the superposition calculus into a decision procedure for finite sorts. Then we get for free the combination of this finite sort refinement with the overall superposition-based reasoning technology and the established results. This is our main motivation to study Sudokus as an example challenge to devise an efficient calculus refinement for finite sort problems. In this paper we show how this can be done for the Sudoku puzzle. The puzzle can be completely formalized over a finite sort containing exactly nine elements.

Let us first understand why superposition is not a decision procedure. For example, let us assume we need to model a sort S with nine distinct elements $1, \dots, 9$ and then reason on formulae over S . The crucial axiom is the following, stating the finiteness of S :

$$\forall x (S(x) \rightarrow (x = 1 \vee \dots \vee x = 9))$$

Now there are basically two possibilities in the standard superposition calculus to deal with this clause: (i) we select $S(x)$, (ii) we reason on the maximal literal.

Following (i) will lead to a behaviour where we eventually produce all (ground) instances of the formula. This can be a successful strategy. In general, it generates exponentially many clauses in the number of universally quantified variables. For a formula with k universally quantified variables over S , it will generate 9^k ground clauses. This is something we would like to prevent.

Following (ii) is also not a good solution, since even a sequence of superposition right inferences with $x = 9$ on a second clause, where we assume wlog. that $x = 9$ is maximal, can easily produce an infinite sequence of clauses with a growing number of literals. Although these clauses eventually become redundant in the theory, there are no practically working redundancy criteria known that actually explore this fact.

To sum it up, there are currently no refinements to the superposition calculus known that can effectively deal with finite sorts. In this paper we suggest such a refinement for Sudoku problems. After a general mathematical framework (Section 2) we start with well-known formulations of Sudoku in propositional logic (Section 3), then present a more compact, but also efficiently solvable formulation by first-order equational ground clauses (Section 4), show that this encoding can be efficiently solved by SPASS (Section 5) and finally lift this formulation to an even more compact encoding in first-order logic with variables (Section 6), devising appropriate inference and reduction rules that lift the results from the ground encoding (Section 4,5)

Our five main contributions are:

1. We devise a framework that allows to explain and study the known propositional codings.
2. We embed the known propositional encodings into this framework.
3. We present more compact first-order ground encodings.
4. We show by exhaustive experiments that this encoding can be efficiently solved.
5. We lift the ground encoding to a more compact one with variables, and define inference and reduction rules that can simulate the efficient behaviour of the ground encoding.

			j	1	2	3	4	5	6	7	8	9
			j_1	1			2			3		
			j_2	1	2	3	1	2	3	1	2	3
i	i_1	i_2	A_1^{box}				A_2^{box}			A_3^{box}		
1		1	a_{11}	a_{12}	a_{13}		a_{14}	a_{15}	a_{16}	a_{17}	a_{18}	a_{19}
2	1	2	a_{21}	a_{22}	a_{23}		a_{24}	a_{25}	a_{26}	a_{27}	a_{28}	a_{29}
3		3	a_{31}	a_{32}	a_{33}		a_{34}	a_{35}	a_{36}	a_{37}	a_{38}	a_{39}
4		1	A_4^{box}				A_5^{box}			A_6^{box}		
5	2	2	a_{41}	a_{42}	a_{43}		a_{44}	a_{45}	a_{46}	a_{47}	a_{48}	a_{49}
6		3	a_{51}	a_{52}	a_{53}		a_{54}	a_{55}	a_{56}	a_{57}	a_{58}	a_{59}
			a_{61}	a_{62}	a_{63}		a_{64}	a_{65}	a_{66}	a_{67}	a_{68}	a_{69}
7		1	A_7^{box}				A_8^{box}			A_9^{box}		
8	3	2	a_{71}	a_{72}	a_{73}		a_{74}	a_{75}	a_{76}	a_{77}	a_{78}	a_{79}
9		3	a_{81}	a_{82}	a_{83}		a_{84}	a_{85}	a_{86}	a_{87}	a_{88}	a_{89}
			a_{91}	a_{92}	a_{93}		a_{94}	a_{95}	a_{96}	a_{97}	a_{98}	a_{99}

Figure 2: Indexing the Sudoku grid

2 Formalizing Sudokus of Arbitrary Size

The common case of Sudoku is a 9×9 grid, with 3×3 boxes. For the description of the general case, let m denote the edge length of the boxes, and $n = m^2$ the edge length of the whole grid. The n^2 grid cells have to be filled with numbers from $[1; n]$. Viewing the grid as a square matrix, the individual cells are indexed by their row and column positions, see Figure 2 for the $n = 9$ case. The index range is $[1; n]$ again.

An alternative indexing scheme is by box number and position within box. In order not to introduce another indexing range, we number boxes from 1 to n , counting them from left to right and row by row, cf. Figure 2. Within each box, the individual cell positions are numbered in the same order, which is not displayed in the figure. We will define a function φ that maps coordinates given as box number and position onto row and column format. In doing so, we exploit that for every $k \in [1; n]$, there exists one and only one decomposition $k = (k_1 - 1)m + k_2$ with $k_1, k_2 \in [1; m]$.

2.1 Definition The function φ transforms pairs over $[1; n]$. In terms of decomposed numbers, the mapping is:

$$((c_1 - 1)m + c_2, (d_1 - 1)m + d_2) \xrightarrow{\varphi} ((c_1 - 1)m + d_1, (c_2 - 1)m + d_2)$$

Let us try an example with Figure 2. The fifth cell in A_7^{box} is a_{82} . Decomposing the box and position coordinate $(7, 5)$, we obtain $((3 - 1) \cdot 3 + 1, (2 - 1) \cdot 3 + 2)$. Hence its φ -image is $((3 - 1) \cdot 3 + 2, (1 - 1) \cdot 3 + 2)$ or $(8, 2)$, as expected.

As can be seen in the definition, essentially φ swaps the decomposed index parts c_2 and d_1 , such that φ applied twice is the identity. In other words, φ is its own inverse. That is, it interestingly also transforms row-column coordinates into box-position format.

Some more notions and notations are needed for the formalization. We write $\bigwedge_{i=1}^n \phi$ as shorthand notation for the finite conjunction $\phi\{i \mapsto 1\} \wedge \dots \wedge \phi\{i \mapsto n\}$, where i is a meta-level variable; and $\bigvee_{i=1}^n \phi$ is defined correspondingly. Even shorter, in \bigwedge_i and \bigvee_j the index variables i and j run through the integer range $[1; n]$ as before. Additionally $\bigwedge_{i \neq j}$ abbreviates $\bigwedge_{i=1}^n \bigwedge_{j=1; j \neq i}^n$; and in \bigwedge_δ the index δ is meant to range over the symbolic *Sudoku dimensions* $\{\text{row}, \text{col}, \text{box}\}$.

An n -tuple $y = (y_1, \dots, y_n)$ where $y_i \in [1; n]$ is called a *vector*. We may view y as a function that maps i to y_i . Accordingly we say that y is *surjective* if $\bigwedge_d \bigvee_i y_i = d$ holds, *injective* if $\bigwedge_{i \neq j} y_i \neq y_j$ is true, and *bijective* if it is both surjective and injective. For the sake of brevity

we write $\text{surj}(y)$, $\text{inj}(y)$ and $\text{bij}(y)$. Let us note down that these notions coincide because $\text{dom } y = \text{codom } y = [1; n]$ is finite.

2.2 Proposition For every vector y , the conditions $\text{bij}(y)$, $\text{inj}(y)$ and $\text{surj}(y)$ are equivalent.

In the following A denotes an $n \times n$ -matrix over $[1; n]$, i.e. a map from $[1; n] \times [1; n]$ to $[1; n]$. We abbreviate $A(i, j)$ as a_{ij} . For such a matrix A , let $A_i^{\text{row}} = (a_{i1}, \dots, a_{in})$ denote the i -th row vector, $A_j^{\text{col}} = (a_{1j}, \dots, a_{nj})$ the j -th column vector, and $A_k^{\text{box}} = (a_{\varphi(k,1)}, \dots, a_{\varphi(k,n)})$ the elements of the k -th box.

2.3 Definition Sudoku puzzling is covered by the following notions:

- (i) A *Sudoku puzzle* is a partial matrix over $[1; n]$, that is, a function A with $\text{dom } A \subseteq [1; n] \times [1; n]$ and $\text{codom } A = [1; n]$.
- (ii) A total matrix B is a *solution* of a Sudoku puzzle A if $A \subseteq B$ and $\bigwedge_{\delta i} \text{bij}(B_i^\delta)$ hold.
- (iii) A Sudoku puzzle is *solvable* if it has a solution, and *uniquely solvable* if it has one and only one solution.

Note that if in the expression $\bigwedge_{\delta i} \text{bij}(A_i^\delta)$ the index δ ranged over the dimensions $\{\text{row}, \text{col}\}$ only, then we had characterized a Latin square. Putting it the other way around, the additional box constraints sharpens a Latin square into a Sudoku.

3 Propositional Encodings

Turning to logic, we now consider formulae as syntactic objects. The overall idea of the propositional approaches is to replace every atom $a_{ij} = d$ by a propositional variable P_{ij}^d . Since atoms $a_{ij} = a_{i'j'}$ and $d = d'$ cannot be transformed this way, formulae containing such expressions have to be rephrased appropriately. In particular this is the case for the vector injectivity axiom. An alternative is given by $\bigwedge_{i \neq j} \bigwedge_d (y_i = d \rightarrow y_j \neq d)$, or $\widehat{\text{inj}}(y)$ for short.

3.1 Definition The building blocks of the encodings, propositional or non-propositional, are the following formulae:

- (i) $\text{surj} := \bigwedge_{\delta i} \text{surj}(A_i^\delta)$
- (ii) $\text{inj} := \bigwedge_{\delta i} \text{inj}(A_i^\delta)$
- (iii) $\widehat{\text{inj}} := \bigwedge_{\delta i} \widehat{\text{inj}}(A_i^\delta)$
- (iv) $\text{given}(B) := \bigwedge_{(i,j) \in \text{dom } B} a_{ij} = b_{ij}$, where B is any Sudoku puzzle
- (v) $\text{codom} := \bigwedge_{ij} \bigvee_d a_{ij} = d$
- (vi) $\text{dist} := \bigwedge_{i \neq j} i \neq j$
- (vii) $\text{fct} := \bigwedge_{ij} \bigwedge_{d \neq d'} (a_{ij} = d \rightarrow a_{ij} \neq d')$

Note that there is some repetition within $\widehat{\text{inj}}$ and inj . For example $a_{11} \neq a_{12}$ is contributed both from $\text{inj}(A_1^{\text{row}})$ and $\text{inj}(A_1^{\text{box}})$. But this repetition is usually left as such.

For the purpose of this section let us now identify each atom $a_{ij} = d$ with the propositional variable P_{ij}^d . Consider an arbitrary Sudoku puzzle B . Then we can find the encodings

- $\text{given}(B) \wedge \text{codom} \wedge \text{fct} \wedge \widehat{\text{inj}} \wedge \text{surj}$ and
- $\text{given}(B) \wedge \text{codom} \wedge \widehat{\text{inj}}$ in the article [LO06],
- $\text{given}(B) \wedge \text{codom} \wedge \text{fct} \wedge \text{surj}$ and
- $\text{given}(B) \wedge \text{codom} \wedge \text{fct} \wedge \widehat{\text{inj}}$ in the paper [Web05].

The abbreviated formulae are ordered by the subformula relation. Dropping the conjunct $\text{given}(B)$, we obtain the lattice depicted in Figure 3. Each encoding uses n^3 propositional variables. The number of atom occurrences in the building blocks is n^3 for codom , $n^4 - n^3$ for fct , $3n^4 - 3n^3$ for $\widehat{\text{inj}}$ and $3n^3$ for surj . Therefore every encoding has $O(n^4)$ atom occurrences.

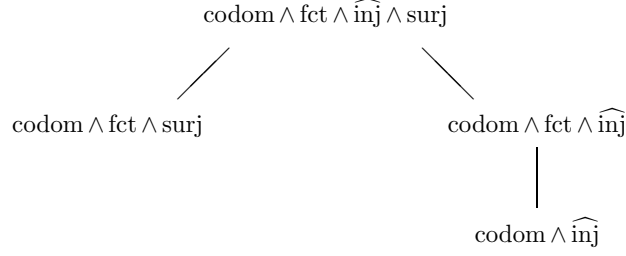


Figure 3: Lattice of propositional Sudoku encodings

4 Ground-level Encodings

In this section we describe how to encode Sudoku puzzles in first-order logic with equality, but without using quantifiers or variables. In particular, we refrain from any formula like $\forall x. x = 1 \vee \dots \vee x = n$ that would fix an upper bound of the model cardinality. Instead we will have to get along with appropriate ground instances thereof. Note also that the formulae inj and surj are not equivalent in this setting.

Our signature Σ consists of the digits $1, \dots, n$ plus a binary function symbol A . That is, from now on A denotes not a parameter, but a syntactic object. We continue to write a_{ij} for $A(i, j)$. As said, we do not use any of the first-order quantifiers \forall and \exists , whereas the expressions $\bigwedge_\delta, \bigvee_i$ etc. are abbreviations on the meta-level.

4.1 Lemma For every Sudoku puzzle B the following are equivalent:

- (i) B is solvable.
- (ii) $\text{surj} \wedge \text{dist} \wedge \text{given}(B)$ is satisfiable.
- (iii) $\text{surj} \wedge \text{inj} \wedge \text{given}(B)$ is satisfiable.
- (iv) $\text{codom} \wedge \text{inj} \wedge \text{given}(B)$ is satisfiable.

Proof: First we show that (i) implies each of the remaining conditions. If B is solvable, then there exists a total matrix $C \supseteq B$ over $[1; n]$ such that $\bigwedge_{\delta i} \text{bij}(C_i^\delta)$ holds, which implies $\text{surj}(C_i^\delta)$ and $\text{inj}(C_i^\delta)$ for all δ and i . Let \mathcal{I} denote the Σ -algebra where every digit is interpreted as itself, and where $\mathcal{I}(A(i, j)) = \mathcal{I}(A)(i, j) = c_{ij}$. The algebra \mathcal{I} satisfies dist , surj and inj by construction. Furthermore codom is satisfied because C is a matrix over $[1; n]$, and $\text{given}(B)$ because C is a solution for B .

(ii) implies (i): Assume that $\text{surj} \wedge \text{dist} \wedge \text{given}(B)$ has a model \mathcal{I} . Because of $\mathcal{I} \models \text{dist}$, the digits are all interpreted distinctly. Our first goal is to show that any $\mathcal{I}(a_{ij})$ equals the interpretation of some digit. Consider any row i . Because of $\mathcal{I} \models \text{surj}(A_i^{\text{row}})$, for every digit d there exists a column index $j(d)$ such that $\mathcal{I}(a_{ij(d)})$ equals $\mathcal{I}(d)$. Now j is a function from $[1; n]$ to $[1; n]$. If $j(d) = j(d')$, then we also have $\mathcal{I}(d) = \mathcal{I}(a_{ij(d)}) = \mathcal{I}(a_{ij(d')}) = \mathcal{I}(d')$, which because of $\mathcal{I} \models \text{dist}$ entails that d and d' are the same. So j is injective. By Prop. 2.2, the function j is surjective as well, such that for every k there exists some d with $j(d) = k$. Hence $\mathcal{I}(a_{ik})$ is $\mathcal{I}(a_{ij(d)})$, which is the same as $\mathcal{I}(d)$. So our first goal is reached. Putting it in other words, if \mathcal{I}' is the restriction of \mathcal{I} to the carrier $\{\mathcal{I}(1), \dots, \mathcal{I}(n)\}$, then this Σ -algebra is a model of $\text{surj} \wedge \text{dist} \wedge \text{given}(B)$ as well, since all the terms within that formula are either digits, or some a_{ij} 's. Let now ρ denote the bijection from $\{\mathcal{I}(1), \dots, \mathcal{I}(n)\}$ to $[1; n]$ such that $\rho(\mathcal{I}(i)) = i$. Then $\mathcal{J} = \rho \circ \mathcal{I}'$ is a Σ -algebra and satisfies by construction the same formulae as \mathcal{I}' . Consider the total $n \times n$ -matrix C with $c_{ij} = \mathcal{J}(a_{ij})$. For every $(i, j) \in \text{dom } B$ the property $\mathcal{J} \models \text{given}(B)$ implies $c_{ij} = \mathcal{J}(a_{ij}) = \mathcal{J}(b_{ij}) = b_{ij}$, such that $B \subseteq C$ holds. Because of $\mathcal{J} \models \text{surj}$ we have $\bigwedge_{\delta i} \text{surj}(C_i^\delta)$, which by Prop. 2.2 is equivalent to $\bigwedge_{\delta i} \text{bij}(C_i^\delta)$. Therefore C is a solution of B , and B is solvable.

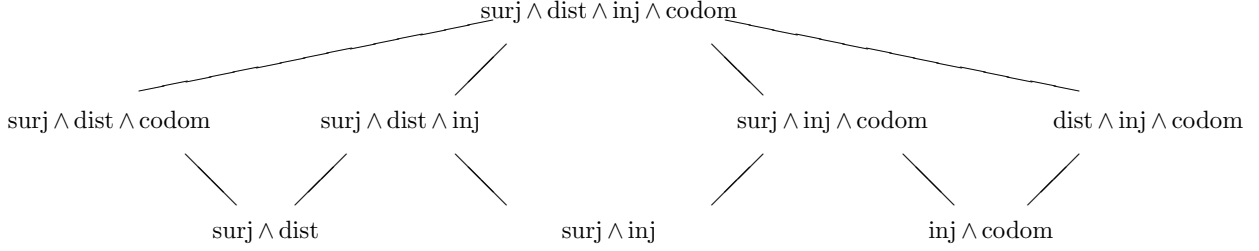


Figure 4: Lattice of ground-level Sudoku encodings

(iii) implies (ii): Let $\mathcal{I} \models \text{surj} \wedge \text{inj} \wedge \text{given}(B)$, and consider any distinct digits i and j . By $\mathcal{I} \models \text{inj}$ we have for example $\mathcal{I} \models a_{1i} \neq a_{1j}$, which spells out as $\mathcal{I} \models A(1, i) \neq A(1, j)$ and implies $\mathcal{I} \models i \neq j$. Consequently \mathcal{I} satisfies dist .

(iv) implies (iii): Assume $\mathcal{I} \models \text{codom} \wedge \text{inj} \wedge \text{given}(B)$. From $\mathcal{I} \models \text{inj}$ we obtain $\mathcal{I} \models \text{dist}$ like in the preceding paragraph. Because of $\mathcal{I} \models \text{codom}$ it is clear that any $\mathcal{I}(a_{ij})$ equals the interpretation of some digit. By the same arguments as in the last but one paragraph we obtain another model \mathcal{J} that satisfies $\text{codom} \wedge \text{inj} \wedge \text{dist} \wedge \text{given}(B)$, but has the additional property that $\mathcal{J}(i) = i$ for every digit i . We next show that $\mathcal{J} \models \text{surj}(A_1^{\text{row}})$. Let j denote the function that maps every digit k to $\mathcal{J}(a_{1k})$. Because of $\mathcal{J} \models \text{codom}$ the function f maps from $[1; n]$ to $[1; n]$. If $j(k) = j(k')$, then we also have $\mathcal{J} \models a_{1k} = a_{1k'}$, which by $\mathcal{J} \models \text{inj}$ implies injectivity of j . By Prop. 2.2, the function j is surjective as well, such that for every digit d there exists a column index such that $\mathcal{J} \models a_{1j} = d$. That is, \mathcal{J} satisfies $\text{surj}(A_1^{\text{row}})$. The remaining $\text{surj}(A_i^{\delta})$ are proved valid within \mathcal{J} the same way. \square

Note that encoding (ii) would not work if we had employed the a_{ij} 's not as abbreviations for $A(i, j)$, but directly as constants within the signature Σ , since then from $\mathcal{I} \models \text{inj}$ we could no longer conclude that the digits were interpreted distinctly.

4.2 Corollary Any conjunction of the encodings of Lemma 4.1 with conjunctions over surj , inj , codom and dist correctly describes Sudoku solvability as well.

Proof: Let ϕ denote such a conjunction for some puzzle B . If B is solvable, then the algebra \mathcal{I} constructed in the first part of the proof of Lemma 4.1 satisfies all the properties at hand, and therefore ϕ as well. If now in turn ϕ is satisfiable, then also one of the encodings of Lemma 4.1 (ii)-(iv) is, and the lemma ensures solvability of B . \square

4.3 Proposition None of the formulae (i) $\text{codom} \wedge \text{dist} \wedge \text{given}(B)$, (ii) $\text{surj} \wedge \text{codom} \wedge \text{given}(B)$ and (iii) $\text{inj} \wedge \text{dist} \wedge \text{given}(B)$ characterizes Sudoku solvability. Since no subformula thereof does either, the encodings of Lemma 4.1 are minimal.

Proof: Formula (i) is satisfiable regardless of B , for example over $[1; n]$ with $\mathcal{I}(i) = i$ and $\mathcal{I}(a_{ij}) = b_{ij}$ if $(i, j) \in \text{dom } B$, and say $\mathcal{I}(a_{ij}) = 1$ otherwise. Formula (ii) is trivially satisfiable over every single-element domain because it is positive. Formula (iii) is satisfiable for such B that do not violate injectivity directly. For example the 4×4 -puzzle to the right is unsolvable, but the corresponding formula $\text{inj} \wedge \text{dist} \wedge \text{given}(B)$ has a model \mathcal{I} where $\mathcal{I}(i) = i$, $\mathcal{I}(a_{ij}) = b_{ij}$ whenever applicable, and $\mathcal{I}(a_{ij})$ is a distinct element of $[3; 16]$ for the remaining a_{ij} . \square

1	1
2	1

Summing it up, the correct Sudoku encodings that can be composed from the building blocks of Def. 3.1 naturally form a lattice which is ordered by the subformula relation. It is

encoding	solved	within 1 s	max. time	avg. time
surj \wedge inj	100 %	3.4 %	40.49 s	2.41 s
surj \wedge inj \wedge codom	100 %	24.1 %	35.75 s	1.72 s
surj \wedge dist \wedge inj	100 %	99.5 %	5.60 s	0.25 s
surj \wedge dist \wedge inj \wedge codom	100 %	99.4 %	6.32 s	0.28 s

Table 1: SPASS 2.2 on Royle’s collection

depicted in Figure 4; the conjunct $\text{given}(B)$ has always been omitted. The number of atom occurrences in the building blocks is $3n^3$ for surj, $\frac{1}{2}n^2 - \frac{1}{2}n$ for dist, $\frac{3}{2}n^3 - \frac{3}{2}n^2$ for inj and n^3 for codom. Therefore every encoding has $O(n^3)$ atom occurrences. This is one order less than in the propositional case.

5 Experimental Results

We took a snapshot of Gordon Royle’s collection [Roy05] of 17-givens Sudoku puzzles on January 11th 2006, when it contained 35396 entries. From these puzzles, a simple Perl script generates formulae in SPASS input format according to the encodings of the previous section. For convenience we introduced an n^2 -place predicate P and added a clause $P(a_{11}, a_{12}, \dots, a_{nn-1}, a_{nn})$. In case the puzzle is solved, the reduced form of this clause is printed and allows to read off the solution in plain text.

After a number of samples it became clear that the encodings in the sublattice from surj \wedge inj up to surj \wedge dist \wedge inj \wedge codom were promising, whereas for each of the remaining ones we quickly came across some puzzle where waiting for SPASS we lost our patience. So we decided to run SPASS version 2.2 on the whole collection in the selected encodings. SunFire V20z servers equipped with 8GB memory and two AMD Opteron processors running at 2.4 MHz with 1M cache were used for the experiment. The results can be found in Table 1.

To our surprise SPASS solved all the puzzles, and most of them in the blink of an eye with two of the encodings. Why does SPASS find a model so quickly for these formulae? Notably all positive literals occur within positive clauses. SPASS performs splitting explicitly, and with higher priority than any inferences. The formula class at hand is already decided by eager splitting, unit rewriting and trivial literal elimination alone: The first breaks the positive clauses into unit equations. Then the second replaces superposition inferences and the third equality resolution steps. Additionally, disequations are propagated into non-unit clauses by matching replacement resolution, speeding up the solution process. For an in-depth description both of splitting and of these reduction rules, see [Wei01, Section 4.4f]. It seems that such a performance were not achievable with implicit splitting, since the resulting clauses could not be used for reductions. Note also that the decision procedure can be extended to clauses with predicative atoms via the reduction rule matching replacement resolution.

SPASS 2.2, as well as the mentioned Sudoku collection and the transformation script for the topmost encoding, is available via the download area of the SPASS Web pages, see: <http://www.spass-prover.org/download>

6 Encodings with Quantifiers

We now develop a more compact representation by introducing finite sets (sorts) for the necessary indices, and then using quantifiers on the object level. We show that in principle the procedure of Section 4 can be lifted to non-ground formulae, using new inference and reduction

rules that exploit the formulae such that all reductions on the ground level are preserved. For simplicity we only consider 9×9-boards in this section.

Our goal here is threefold: (i) we want to keep the experimental results from the ground level when we lift to the general level with variables, (ii) we want to have a more compact representation and (iii) eventually we want to establish the new inference/reduction rules such that they fit into the superposition framework and can then be explored in the general first-order logic with equality context. To this end we add to the encoding of Section 4 a sort (unary predicate) S that represents the finite set $\{1, \dots, 9\}$. For otherwise any model of a formula like

$$\forall x (x = 1 \vee x = 2 \vee \dots \vee x = 9)$$

already has an overall domain with at most nine elements. So, e.g., a combination with a theory requiring an infinite domain is already impossible.

6.1 Formulae

First of all we define the index and value set for the different numbers 1 to 9. This is done by introduction of a unary sort S :

- (i) $S(1) \wedge S(2) \wedge \dots \wedge S(9)$
- (ii) $\forall x (S(x) \rightarrow (x = 1 \vee x = 2 \vee \dots \vee x = 9))$
- (iii) $1 \neq 2 \wedge 1 \neq 3 \wedge \dots \wedge 8 \neq 9$

In order to describe the box indexes we define three subsets, S_{123} , S_{456} and S_{789} :

- (iv) $S_{123}(1) \wedge S_{123}(2) \wedge S_{123}(3)$
- (v) $S_{456}(4) \wedge S_{456}(5) \wedge S_{456}(6)$
- (vi) $S_{789}(7) \wedge S_{789}(8) \wedge S_{789}(9)$
- (vii) $\forall x (S_{123}(x) \rightarrow (x = 1 \vee x = 2 \vee x = 3))$
- (viii) $\forall x (S_{456}(x) \rightarrow (x = 4 \vee x = 5 \vee x = 6))$
- (ix) $\forall x (S_{789}(x) \rightarrow (x = 7 \vee x = 8 \vee x = 9))$

In the minimal term model for the above clauses, the interpretation of S is finite and contains exactly the constants $1, \dots, 9$. Accordingly S_{123} , S_{456} , S_{789} are interpreted as expected. Hence when generating the ground instances of some clause in a superposition model construction, we only need to consider finitely many thereof.

The lifted surjectivity and injectivity formulae for the columns are:

- (surj-c) $\forall x, y ((S(x) \wedge S(y)) \rightarrow (A(1, x) = y \vee \dots \vee A(9, x) = y))$
- (inj-c) $\forall x, y, z ((S(x) \wedge S(y) \wedge S(z) \wedge y \neq z) \rightarrow A(y, x) \neq A(z, x))$

For the rows we get:

- (surj-r) $\forall x, y ((S(x) \wedge S(y)) \rightarrow (A(x, 1) = y \vee \dots \vee A(x, 9) = y))$
- (inj-r) $\forall x, y, z ((S(x) \wedge S(y) \wedge S(z) \wedge y \neq z) \rightarrow A(x, y) \neq A(x, z))$

The boxes are subject to the following constraints:

$$\begin{aligned}
(\text{surj-b1}) \quad & \forall z (S(z) \rightarrow (A(1, 1) = z \vee A(2, 1) = z \vee \dots \vee A(3, 3) = z)) \\
& \vdots \\
(\text{surj-b9}) \quad & \forall z (S(z) \rightarrow (A(7, 7) = z \vee A(8, 7) = z \vee \dots \vee A(9, 9) = z)) \\
(\text{inj-b1}) \quad & \forall x, y, z, u ((S_{123}(x) \wedge S_{123}(y) \wedge S_{123}(z) \wedge S_{123}(u) \wedge (y \neq z \vee x \neq u)) \\
& \rightarrow A(y, x) \neq A(z, u)) \\
(\text{inj-b2}) \quad & \forall x, y, z, u ((S_{123}(x) \wedge S_{456}(y) \wedge S_{456}(z) \wedge S_{123}(u) \wedge (y \neq z \vee x \neq u)) \\
& \rightarrow A(y, x) \neq A(z, u)) \\
(\text{inj-b3}) \quad & \forall x, y, z, u ((S_{123}(x) \wedge S_{789}(y) \wedge S_{789}(z) \wedge S_{123}(u) \wedge (y \neq z \vee x \neq u)) \\
& \rightarrow A(y, x) \neq A(z, u)) \\
& \vdots \\
(\text{inj-b9}) \quad & \forall x, y, z, u ((S_{789}(x) \wedge S_{789}(y) \wedge S_{789}(z) \wedge S_{789}(u) \wedge (y \neq z \vee x \neq u)) \\
& \rightarrow A(y, x) \neq A(z, u))
\end{aligned}$$

Finally, the domain constraint for the binary operator A is:

$$(\text{dom-c}) \quad \forall x, y ((S(x) \wedge S(y)) \rightarrow (A(x, y) = 1 \vee \dots \vee A(x, y) = 9))$$

6.2 Size of the first-order encoding

The formulae (i)-(ix) contain $\frac{1}{2}n^2 + \sqrt{n} + \frac{7}{2}n + 1$ atom occurrences. The constraints on columns and rows contribute $2n + 14$ atoms and those on the boxes $n^2 + 8n$, whereas the domain constraint consists of $n + 2$ atoms. Summing it up, we obtain $O(n^2)$ atom occurrences, which is one order less than for the ground-level encodings.

6.3 Reduction and inference rules

The type of reasoning on the ground level we need to lift to the non-ground level was introduced in Section 6.1. Here is a typical reduction on the ground level that is driven by unit propagation from the column, row and box given values, and eventually leads to the conclusion that $A(7, 5) = 7$ in the example from the introduction (see Figure 1). Here we show the propagation of the row values:

$$\begin{array}{ll}
A(7, 5) \neq A(7, 1) & \text{reduced instance of (inj-r)} \\
A(7, 1) = 3 & \text{given value in the board} \\
A(7, 5) \neq A(7, 3) & \text{reduced instance of (inj-r)} \\
A(7, 4) = 4 & \text{given value in the board} \\
A(7, 5) \neq A(7, 2) & \text{reduced instance of (inj-r)} \\
A(7, 7) = 2 & \text{given value in the board} \\
A(7, 5) = 1 \vee \dots \vee A(7, 5) = 9 & \text{reduced instance of (dom-c)} \\
\hline
A(7, 5) = 1 \vee A(7, 5) = 5 \vee A(7, 5) = 6 \vee A(7, 5) = 7 \vee A(7, 5) = 8 \vee A(7, 5) = 9 &
\end{array}$$

Continuing this reasoning for the respective column and box values eventually yields the identity $A(7, 5) = 7$, which then can be propagated further.

This reasoning can be described by the following parameterized inference rule *Propagate1-k*, where $0 \leq k \leq 8$:

$$\begin{array}{ll}
A(d_1, c_1) \neq A(b_1, a_1) & \text{reduced instance of (inj-*)} \\
A(b_1, a_1) = e_1 & \text{given value or deduced unit} \\
& \vdots \\
A(d_k, c_k) \neq A(b_k, a_k) & \text{reduced instance of (inj-*)} \\
A(b_k, a_k) = e_k & \text{given value or deduced unit} \\
A(d_1, c_1) = e_1 \vee \dots \vee A(d_9, c_9) = e_9 & \text{reduced instance of (dom-c) or (surj-*)} \\
\hline
A(d_{k+1}, c_{k+1}) = e_{k+1} \vee \dots \vee A(d_9, c_9) = e_9 &
\end{array}$$

where all the a_i, b_i, c_i, d_i are constants and we consider \vee to be associative and commutative for the application of the reduced instance of (dom-c) or (surj-*). Obviously, the instance $A(d_1, c_1) = e_1 \vee \dots \vee A(d_9, c_9) = e_9$ becomes redundant after the introduction of $A(d_{k+1}, c_{k+1}) = e_{k+1} \vee \dots \vee A(d_9, c_9) = e_9$ in the standard superposition theory sense. For $n = 0$ the rule derives a reduced ground instance of (dom-c) or (surj-*).

We call this rule Propagate1 because it explores the constraints applicable to one specific square or available value. This can be further generalized to rules that consider the constraints for two or more different squares or values simultaneously. However, in order to solve the 9×9-grid puzzles, these more complicated propagation steps are not needed to solve the puzzle efficiently. They are also not available when the standard superposition calculus is applied to the ground encoding (Section 4). Therefore we need not consider more sophisticated propagation rules here.

Finally we can use the following reduction criterion to remove clauses with variables. We call a clause $C \in N$ *ground redundant* in N if for all productive ground instances $C\sigma$ of C , there is a clause $D \in N$ such that $D \models C\sigma$ and $D \prec C\sigma$.

Note that with respect to the above theory, there are only finitely many productive ground instances for the clauses (surj-*) and (dom-c).

Now the lifting of the ground reasoning to the general level with variables is to (i) exhaustively apply the rules Propagate1- i , starting with $i = 8$, and then (ii) to test for ground redundancy. If this does not saturate the problem, we apply (iii) splitting to a ground clause $A(d_{k+1}, c_{k+1}) = e_{k+1} \vee \dots \vee A(d_9, c_9) = e_9$. If such a clause does not exist, we apply once (iv) Propagate1-0 and restart from (i). This procedure lifts exactly the reasoning from the ground level (Sections 4, 5) to the general level.

7 Conclusion

There are two surprising results in this paper. Firstly, Sudoku problems can be coded in first-order logic such that the encoding is more compact than the well-known propositional encodings, but still can be efficiently solved by first-order theorem proving methods. The encodings for both logics have been embedded in a framework that allows for a uniform representation.

Secondly, we can lift the ground-level encoding to an even more compact one with variables, but keep the efficiency of the calculus by introducing new inference and reduction rules. These rules lift the possible reductions from the ground level to the general one whilst preventing the generation of redundant ground instances. We have not implemented the rules introduced in Section 6 but shown that they exactly simulate the behaviour of the standard superposition calculus on the ground clauses without the need to generate these.

The principles behind the finite sort specification in formulas (i)-(ix) are applicable to any finite-sort problem. What remains to be done is to generalize in particular the new inference rule such that it also fits other problems in a generic way.

Acknowledgements

We thank the reviewers for their valuable comments that helped to improve the paper.

References

- [LO06] Inês Lynce and Joël Ouaknine. Sudoku as a SAT problem. In *Electronic Proceedings of the 9th International Symposium on Artificial Intelligence and Mathematics*, 2006. Available from <http://anytime.cs.umass.edu/aimath06/proceedings.html>.
- [McC03] William McCune. Mace4 reference manual and guide. Technical Report ANL/MCS-TM-264, Argonne National Laboratory, August 2003.
- [RJ06] Ed Russel and Frazer Jarvis. Mathematics of Sudoku II. Preprint. Available from <http://www.shef.ac.uk/~pm1afj/sudoku>, 2006.
- [Roy05] Gordon F. Royle. Minimum sudoku. See <http://www.csse.uwa.edu.au/~gordon/sudokumin.php>, 2005.
- [Sim05] Helmut Simonis. Sudoku as a constraint problem. In Brahim Hnich, Patrick Prosser, and Barbara Smith, editors, *Electronic Proceedings of the 4th International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pages 13–27, 2005. Available from <http://4c.ucc.ie/~brahim/mod-proc.pdf>.
- [WBH⁺02] Christoph Weidenbach, Uwe Brahm, Thomas Hillenbrand, Enno Keen, Christian Theobald, and Dalibor Topic. SPASS version 2.0. In Andrei Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction*, volume 2392 of *LNAI*, pages 275–279. Springer-Verlag, 2002.
- [Web05] Tjark Weber. A SAT-based sudoku solver. In Geoff Sutcliffe and Andrei Voronkov, editors, *Short Electronic Proceedings of the 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 11–15, December 2005. Available from <http://www.cs.miami.edu/~geoff/Conferences/LPAR-12/ShortPapers.pdf>.
- [Wei01] Christoph Weidenbach. Combining superposition, sorts and splitting. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 27, pages 1965–2012. Elsevier, 2001.

Automated relative consistency proving

André Rognes

`rogn@users.berlios.de`

Abstract. A way of presenting essential information about first order theories is introduced. The presentation allows for the automation of relative consistency proving for sentences from a known conservative reduction class for first order logic. The theories are presented by algebras of partial polyadic signature, some of which correspond to consistent theories. A downward Löwenheim-Skolem type of result is shown in that first order sentences are satisfiable iff there is a satisfying interpretation of their reduced form in a finite algebra corresponding to a consistent theory. This is also the case for infinity axioms, i.e. sentences that have infinite models only. An algorithm for extracting algebras from theories, presented by decision procedures, has been implemented. Moreover a disprover has been implemented utilising extracted algebras, a relative consistency prover and an introduced construction on algebras. In principle the disprover recognises any finitely satisfiable sentence. Moreover examples of infinity axioms, have been found that the disprover recognises in reasonable time, while various state-of-the-art semi-decision procedures do not.

1 Introduction

The work presented herein is motivated by the possibility of using search for satisfying interpretations in theories, as part of a disprover that exceeds procedures based on search for satisfying interpretations in finite structures. That is; a disprover that in principle recognises any first order sentence that has a finite model as well as some that have infinite models only, namely infinity axioms. Finding a satisfying interpretation for a sentence in a theory, is proof of consistency of the sentence relative to the theory.

The use of interpretations in theories, to reason about sentences or theories is known from various parts of formal logic. They are an object of study in algebraic logic. A result of the present paper is that search for such interpretations can be automated, provided that the sentences and the theories are of a suitable form. A relative consistency prover has been implemented and put to the test, as part of a disprover. That is, a procedure that terminates successfully only if the input is a satisfiable sentence. The relative consistency prover works like this if one fixates the input theory and the input theory is consistent. It turns out that in principle the disprover terminates successfully for any finitely satisfiable sentence. Moreover examples of both finitely satisfiable sentences and infinity axioms have been found for which successful termination occurs in reasonable time.

The form of the input sentences is the following; they are relational, use only three variables and are a finite conjunction of sentences in prenex normal form. In many a sense this is rather a small fragment of first order logic, it is however substantial by the following. The existence of algorithms are known that transform any first order sentence to one of the required form, in such a way that inconsistency, satisfiability and finite satisfiability are preserved and reflected [Büc62],[Gur66], [Kos66], [Aan71],[Bör71]. One may see these restricted sentences as a way of presenting first order sentences in general. The transformations, as specified in the proofs of their existence, are rather involved and not expected to preserve the property of tractability very well. Therefore no such algorithm is part of any of the procedures described in the present paper. Rather such sentences are treated as an alternative to first order language, as this enables the automation of reasoning techniques not known to be possible otherwise.

The theories have to be given in the form of finite partial polyadic algebras. It is well known that polyadic algebras are a form of first order theory, but a brief introduction to them is given in section 2. In section 3 a particular breed of partial polyadic algebras, that is believed to be new, is introduced. A downward Löwenheim-Skolem type of result is shown, in that any satisfiable sentence of the afore mentioned form is satisfiable in a finite such algebra, even if the sentence has infinite models only. In section 4 a construction of polyadic algebras is defined for building richer algebras from a given one. In particular one can build algebras rich enough to allow a satisfying interpretation for any finitely satisfiable sentence. Section 5 reports on some experimental results with an implementation of a relative consistency prover employing partial polyadic algebras and the enriching construction, for disproving.

1.1 Related work

As regards establishing the satisfiability of first order sentences there are two prominent ways of implementation. Firstly there is finite model generation, one of the earliest non-naïve implementations of which must be [Sla94]. Secondly there are approaches based on the fact that complete refutation procedures sometimes terminate without having found a refutation. According to [Tam92] implementations of this approach date as far back as to the 1960's by S.Maslov. As opposed to finite model generation, the refutation approach sometimes tackles infinity axioms, as does the approach described in the current paper. For some history of implementing finite model generators, refutation procedures, and combinations thereof, see [Tam], [FLS] and descriptions of the systems that have taken part in recent theorem proving- and model generation competitions [Sut05].

Note that the term satisfiability modulo theories as defined in [ARR03], amounts to deciding the satisfiability of sets on the form $\Gamma \cup \{\phi\}$ where Γ is a theory and ϕ is a formula. This is not the same as satisfiability relative to a theory as defined in the current paper. A difference is that renaming of predicate symbols in ϕ does not preserve satisfiability modulo Γ , whereas it does in the case of relative consistency.

1.2 First order formulae

Two first order formulae ϕ and ψ are said to be provably equivalent by some theory Γ if $\phi \leftrightarrow \psi$ can be proven from Γ , by a proof procedure sound and complete for first order logic. Apart from formulae implicit in theories or proofs of equivalence, first order formulae in the present paper are taken from the pure predicate calculus of three variables, L_3 . Pure here means being without equality or function symbols. Relation symbols are assumed to be of arity 3. One may replace each occurrence of Rx with $Rxxx$ and Rxy with $Rxyy$ in a sentence while preserving satisfiability, as well as inconsistency.

Let 3 denote the set $\{0, 1, 2\}$ and ω the natural numbers. The set of mappings from 3 to 3 is written both as 3^3 and as $3 \rightarrow 3$. Mappings are written as triples of elements of 3, where the value of 0 is leftmost. So the identity mapping for example is written 012 and the mapping that subtracts one modulo 3 is written 201. Assume a countable supply of relation symbols. For each relation symbol R and each mapping $\sigma \in 3 \rightarrow 3$, $R\sigma$ is an atomic formula. An example of an atomic formula is $R012$ which is short for $R(x_0, x_1, x_2)$. A formula is a literal if it is atomic or the negation of an atomic one. Moreover open formulae and L_3 are minimal such that; An atomic formula is an open formula. If ϕ and ψ are open formulae then $\neg\phi$, and $\phi \vee \psi$ are open formulae. An open formula is in L_3 . If ϕ is in L_3 then $\exists_0\phi$, $\exists_1\phi$ and $\exists_2\phi$ are in L_3 . Again $\exists_i\phi$ is short for $\exists_{x_i}\phi$. The following are used as abbreviations; \perp for some contradiction, $\phi \wedge \psi$ for $\neg(\neg\phi \vee \neg\psi)$ and $\forall_i\phi$ for $\neg\exists_i(\neg\phi)$. A formula is said to be on prenex normal form if it is of the form $Q_0Q_1Q_2A$ where A is open and each Q_i is one of \exists_i or \forall_i .

2 Polyadic algebras

Polyadic Algebras were introduced and studied by Halmos in a series of papers collected in [Hal62]. They can be seen as abstract first order theories. A concrete theory Γ is turned into a polyadic algebra by identifying formulae that are provably equivalent by Γ . Being provably equivalent by a consistent Γ means defining the same relation in each model of Γ . On the equivalence classes obtained like this, operations that correspond to quantifiers, variable substitutions and propositional connectives, are defined. In the case of L_3 we denote this algebra L_3/Γ . If we forget about the operations for quantifications and variable-substitutions this is the well known Lindenbaum-Tarski algebra of Γ . A particular class of polyadic algebras are polyadic set algebras of dimension 3, denoted \mathbf{Ps}_3 . These are exactly the algebras isomorphic to L_3/Γ , for complete and consistent Γ .

One may view the elements of a \mathbf{Ps}_3 , isomorphic to L_3/Γ , as names for the L_3 -definable relations in a given model for Γ . The relative consistency prover presented in the current paper is in this view doing model search amongst some of the L_3 -definable relations.

Observe that the number of equivalence classes of L_3/Γ may be finite even if Γ does not have finite models. The theory of a dense order without endpoints

is an example of such a Γ . To see this, note that this Γ has finite relational signature and quantifier elimination. Any definable relation in a model for a theory with quantifier elimination is provably equivalent to an open formula. An open formula can be brought to a provably equivalent one in conjunctive normal form, whose length is limited by a constant depending on the number of relation-symbols. So in the example, L_3/Γ has a finite number of equivalence classes each having an infinite number of formulae defining the same relation over a densely ordered set.

\mathbf{Ps}_3 's are called **0-valued functional algebras** in [Hal62]. Some known results about \mathbf{Ps}_3 's follow. Propositions are given without proofs. The notation mostly follows [Ném91]. One exception is that there a larger set of operations for variable substitutions is used in the definition of polyadic algebras, here only 3 are used.

2.1 Operations for variable substitutions

If f and g are mappings $f \circ g$ denotes their composition, applying g first then f . The three mappings p' , s' and r' in $3 \rightarrow 3$ are $p' = 102$ called permutation, $s' = 112$ called 01-substitution and $r' = 201$ called rotation.

Proposition 2.1. *p', s' and r' under composition generate all mappings in $3 \rightarrow 3$.*

2.2 Algebras of polyadic signature

An algebra is a set together with a family of operations on that set. The given set is called the carrier set. Information on the arities of the operations is called the signature of that algebra. Homomorphisms are mappings between algebras that preserve the operations as given by the signature. From the concept of homomorphism concepts such as isomorphism, embedding, closure, sub-algebra etc. are derived.

Definition 2.1. *An algebra of polyadic signature is an $\mathcal{A} = (\mathcal{B}, r, p, s, c_0, c_1, c_2)$ such that \mathcal{B} has the signature of a boolean algebra and r, p, s, c_0, c_1, c_2 are unary operations on \mathcal{B} . c_0, c_1, c_2 are called cylindrifications, and correspond to existential quantifiers.*

2.3 L_3 as an algebra of polyadic signature

Here the language L_3 seen as an algebra with, operations for boolean connectives, is expanded to make an algebra of polyadic signature. Interpretations of L_3 are by definition the polyadic homomorphisms from this algebra to other algebras of polyadic signature. An interpretation is satisfying for a sentence if the sentence is sent to something other than zero. The fact that this coincides with interpretation in the usual tarskian sense when the target is derived from a consistent theory is well known.

Definition 2.2. *for $i \in 3$ and $\sigma \in \{r, p, s\}$ define $\sigma^* \in L_3 \rightarrow L_3$ by*

$$\begin{aligned}
\sigma^*(R\tau) &= R(\sigma' \circ \tau) \\
\sigma^*(\neg\phi) &= \neg(\sigma^*\phi) \\
\sigma^*(\phi \vee \psi) &= \sigma^*(\phi) \vee \sigma^*(\psi) \\
r^*(\exists_i\phi) &= \exists_{r'(i)}r^*(\phi) \\
p^*(\exists_i\phi) &= \exists_{p'(i)}p^*(\phi) \\
s^*(\exists_0\phi) &= \exists_0\phi \\
s^*(\exists_1\phi) &= \exists_0p^*(\phi) \\
s^*(\exists_2\phi) &= \exists_2s^*(\phi)
\end{aligned}$$

Note that r^* , p^* and s^* may be “moved inwards” relative to each of the connectives of L_3 . So L_3 the open and the atomic formulae are each closed under these operations. Compare this definition to the axiomatisation of so called QPA_3 ’s in [Ném91], at the end of appendix 1.

Definition 2.3. $\mathbf{L}_3 = (L_3, \vee, \neg, \perp, r^*, p^*, s^*, \exists_0, \exists_1, \exists_2)$

3 Extracting algebras from a decision procedure

A decision procedure, seen as a string of symbols, is a way of finitely presenting a complete theory, say Γ . Usually such procedures are of high complexity, and therefore unsuitable as input to an algorithm for relative consistency proving. Rather one can use a finite part of L_3/Γ in the form of pre-computed tables of operations on a finite set of numbers. Enough information from the decision procedure can be extracted to interpret any finite conjunction of sentences on prenex normal form, while keeping the information finite. Recall that such sentences constitute a conservative reduction class for first order logic.

3.1 Partial polyadic set algebras

The particular partial algebras introduced in the present paper, suffice for interpreting formulae that are conjunctions of prenex formulae in L_3 . These algebras are shown to be finite if they are finitely generated. The algebras are defined in the form of a 4-sorted algebra, which imposes some requirements on where operations are defined.

Definition 3.1. *A partial polyadic set algebra, \mathbf{pPs}_3 , is a structure $\mathcal{A} = (\mathcal{B}_3, \mathcal{B}_2, \mathcal{B}_1, \mathcal{B}_0, r, p, s, c_0, c_1, c_2)$ where $\mathcal{B}_3, \mathcal{B}_2, \mathcal{B}_1, \mathcal{B}_0$ are boolean algebras each with their own sup/join, negations and zero. r, p, s are elements of $\mathcal{B}_3 \rightarrow \mathcal{B}_3$. c_2 is an element of $\mathcal{B}_3 \rightarrow \mathcal{B}_2$, c_1 is an element of $\mathcal{B}_2 \rightarrow \mathcal{B}_1$ and c_0 is an element of $\mathcal{B}_1 \rightarrow \mathcal{B}_0$. Moreover \mathcal{A} must be a sub-algebra of some $\mathcal{C} \in \mathbf{Ps}_3$. Sub-algebra here means that there are boolean embeddings from each of $\mathcal{B}_3, \mathcal{B}_2, \mathcal{B}_1, \mathcal{B}_0$ to \mathcal{C} that preserve each of r, p, s, c_0, c_1, c_2 when ever they are defined in \mathcal{A} .*

Conjunctions of L_3 -sentences in prenex normal form, and their sub-formulae, may be interpreted in \mathbf{pPs}_3 ’s as they are in \mathbf{Ps}_3 ’s, where open formulae are interpreted in \mathcal{B}_3 , formulae in the form $\exists_2\phi$ where ϕ is open are interpreted in

\mathcal{B}_2 and so forth. A sentence is *satisfiable* if it is sent to something other than zero in \mathcal{B}_0 , by some interpretation. This coincides with satisfyability in the usual sense as can be seen by the following. As with \mathbf{Ps}_3 's, which are isomorphic to some L_3/Γ , the elements of a \mathbf{pPs}_3 can be viewed as names for some of the L_3 -definable relations in a model for Γ . These relations include the relations definable by sub-formulae of conjunctions of L_3 sentences in prefix normal form.

3.2 The partial polyadic closure

Below it is shown that any satisfiable finite conjunction of prenex formulae is satisfied in a finite \mathbf{pPs}_3 , even if such a conjunction has infinite models only (i.e it is an infinity axiom).

Let O_3 denote the open formulae of L_3 , and O_3/Γ the appropriate sub-boolean algebra of L_3/Γ . Recall that O_3 is closed under the operations $\{r^*, p^*, s^*\}$.

Definition 3.2. *The following defines the partial polyadic closure of the atomic formulae of L_3 in L_3/Γ . Elements of L_3 are used as names for elements of L_3/Γ . The closure defines sets $B_3, B_2, B_1, B_0, r, p, s, c_0, c_1, c_2$, which constitute a \mathbf{pPs}_3 .*

$B_3 =$ the boolean algebra O_3/Γ
 $B_2 =$ the sub-boolean-algebra of L_3/Γ generated by $c_2(B_3)$
 $B_1 =$ the sub-boolean-algebra of L_3/Γ generated by $c_1(B_2)$
 $B_0 =$ the sub-boolean-algebra of L_3/Γ generated by $c_0(B_1)$
for each $\sigma \in \{r, p, s\}$ let $\sigma = \{(\phi, \psi) \in B_3 \times B_3 : \forall_0 \forall_1 \forall_2 (\sigma^*(\phi) \leftrightarrow \psi) \in \Gamma\}$
 $c_2 = \{(\phi, \psi) \in B_3 \times B_2 : \forall_0 \forall_1 (\exists_2(\phi) \leftrightarrow \psi) \in \Gamma\}$
 $c_1 = \{(\phi, \psi) \in B_2 \times B_1 : \forall_0 (\exists_1(\phi) \leftrightarrow \psi) \in \Gamma\}$
 $c_0 = \{(\phi, \psi) \in B_1 \times B_0 : \exists_0(\phi) \leftrightarrow \psi \in \Gamma\}$

Proposition 3.1. *If the number of atomic formulae in Γ is finite then their partial polyadic closure in L_3/Γ is finite.*

Proof. The initial boolean algebra B_3 is O_3/Γ , the boolean algebra generated by the equivalence classes containing atomic formulae. It is finite since it is a finitely generated boolean algebra. The subsequent boolean algebras are generated by images of finite ones. \square

Corollary 3.1. *A finite conjunction of L_3 sentences in prenex normal form, is satisfiable iff it is satisfiable in a finite \mathbf{pPs}_3 .*

Proof. View such a conjunction as a first order theory by it self, and let Γ denote one of its consistent completions. The sentence is then satisfied in L_3/Γ by the homomorphism that sends each sentence to it's equivalence class. The sentence is also satisfied in the \mathbf{pPs}_3 that is the closure of its atoms in L_3/Γ . For the other direction; sentences satisfied in a finite \mathbf{pPs}_3 are also satisfied in the algebras L_3/Γ of which the \mathbf{pPs}_3 is a sub-algebras, which coincides with satisfiability in the usual sense. \square

Corollary 3.2. *The class of finite \mathbf{pPs}_3 's is not recursively enumerable.*

Proof. Since the class of finite conjunctions of L_3 sentences on prenex normal form is a reduction class, the satisfiable such are not recursively enumerable. They would be if finite \mathbf{pPs}_3 were recursively enumerable, since checking satisfaction is recursive. \square

3.3 The closure as an algorithm

Assume that Γ is a complete and consistent theory in a language with a finite number of atomic formulae. Also assume that there is a decision procedure for Γ . The partial polyadic closure of Γ in L_3/Γ can then be seen as an algorithm. First of all, the definition of the closure has the overall structure of an algorithm. More over B_3 for instance, can be computed by starting with atomic formulae and generating new formula by the boolean operations. As new formulae ϕ are generated these are kept or discarded, depending on whether they are provably equivalent to some already generated formula ψ . That is; depending on what the decision procedure has to say about the sentence $\forall_0 \forall_1 \forall_2 (\phi \leftrightarrow \psi)$. The kept formulae serve as names for distinct elements of O_3/Γ , and the process terminates because the number of elements of O_3/Γ is finite. The rest of the partial polyadic closure is seen to be algorithmic in a similar fashion.

Note that the procedure doesn't really have to be a decision procedure. It doesn't have to represent a complete nor a consistent theory. A sufficient condition for termination is, that the procedure works as a characteristic function for sentences with an interpretation in a \mathbf{pPs}_3 and that closure under boolean operations is finite.

The outlined, straight forward approach, which is presenting the sub-boolean algebras of L_3/Γ with one formula for each element and tables for the operations, is less than optimal. The following well known property of polyadic algebras allows one to make do with presenting the sub-boolean algebras by one formula for each of its atoms. This reduces the number of calls to the decision procedure, and the size of the tables for the corresponding algebra.

Definition 3.3. *An operation σ on a boolean algebra is called additive if $\sigma(0) = 0$ and $\sigma(x \vee y) = \sigma(x) \vee \sigma(y)$.*

This coincides with being a join-semi-lattice homomorphism on the appropriate reduct of the boolean algebra.

Proposition 3.2. *In a \mathbf{Ps}_3 , r, p, s, c_0, c_1, c_2 are all additive.*

4 A construction on \mathbf{Ps}_3 's

In the previous section effort was put into keeping only finite parts of polyadic algebras, as this makes them suitable for input to a relative consistency prover. Finite partial algebras are however somewhat depleted. For example; to each

such algebra there exists a finitely satisfiable sentence that is not satisfiable in that algebra. Such a sentence may be constructed in a language containing more relation-symbols than there are elements in a given algebra, by stating that each of the named relations are pairwise different.

This section describes a way of constructing a new and bigger finite \mathbf{pPs}_3 from a given one. The construction is such that any finitely satisfiable sentence is satisfiable in an iterate of the construction. This can be used in combination with a relative consistency prover to ensure termination in case of finite satisfiability. Here it is defined for total \mathbf{Ps}_3 's. The construction is analogous for partial ones.

Definition 4.1. *Let \mathcal{A} be a \mathbf{Ps}_3 . Equip the set of mappings in $2^3 \rightarrow \mathcal{A}$ with polyadic structure as follows; Boolean operations are defined component-wise. For $t \in 2^3 \rightarrow \mathcal{A}$ do,*

$$\begin{aligned}\sigma(t)(abc) &\mapsto \sigma t(\sigma'(abc)) \text{ when } \sigma \in \{r, p, s\} \\ c_0(t)(abc) &\mapsto \bigvee_{x \in 2} c_0(t(xbc)), \\ c_1(t)(abc) &\mapsto \bigvee_{x \in 2} c_1(t(abcx)), \\ c_2(t)(abc) &\mapsto \bigvee_{x \in 2} c_2(t(abcx)).\end{aligned}$$

Two propositions about the construction follow. Proofs are left out for a later occasion.

Proposition 4.1. *The above construction yields a \mathbf{Ps}_3 .*

Proposition 4.2. *Any finitely satisfiable sentence is satisfied in an iterate of the above construction beginning with an arbitrary \mathbf{Ps}_3 .*

4.1 Related constructions

The above construction is devised by the author. It is rather similar to what is called the cardinal multiple of theories in [FV59] (section 4.7), though, when bearing in mind that polyadic set algebras correspond to complete and consistent theories.

An essential property of the construction, is that the polyadic set algebras are closed under it, and that one gets something richer, at least in the finite case. The direct product of polyadic set algebras is not such a construction, since a sentence that is satisfiable in a product, by projection, already is satisfiable in one of the factors.

5 Some experiments with an implementation

Both the closure algorithm and a relative consistency prover have been implemented and run on a computer of the kind found in many a home now a days.

5.1 The closure algorithm

The closure algorithm has been hooked up with Mona [KM01]. Via suitable relativizations, Mona has been used as a decision procedure for the first order theories of the following structures.

- ($2, \{R\}_{R \in \mathcal{P}(2^3)}$), the full structure of a 2-element set, with one relation symbol for each subset of 2^3 .
- ($\mathbf{Q}, <$), a dense order without bounds
- ($\omega, <, x + 1$), the natural numbers with the usual ordering and successor

Note that $<$ and $x + 1$ are treated as ternary relations where the third argument makes no difference. Calculating the closures is done in an hour or two. Moving beyond theories of orderings and successors tends to exhaust the computers resources, with the implementation at hand. Also moving to algebras of dimension four does.

5.2 A relative consistency prover

The constructions given in the present paper have gone into an implementation of a relative consistency prover called *Flipper* [Rog06]. A non-greedy variant of stochastic local search ala [SLM92], was chosen for its decent performance to ease-of-implementation ratio. In what follows, Flipper has been used as a disprover using the theory for successor and the usual ordering on the natural numbers.

For testing and comparison purposes randomly generated sentences have been used, as the standard problem library for first order reasoning machinery [SS98] is to sparse in non-refuted formulae on the required form. Sporadic comparison of Flipper with state of the art model generators on finitely satisfiable sentences indicate that the latter are 10 to a 100 times faster than Flipper. The worst case here is a conjunction of some 238 sentences of the form $\forall_0 \forall_1 \exists_2 A_0 \vee A_1 \vee A_2$ where each A_i is a literal using a relationsymbol picked from a set of 10.

To convince the reader that Flipper might be worth giving a spin besides theorem provers and finite model generators (which typically are faster), when faced with a decision problem of the right form, the following infinity axiom has been chosen.

$$\forall_x \forall_y \forall_z \neg Rxx \wedge \forall_x \forall_y \exists_z Rxz \wedge \forall_x \forall_y \forall_z (Rxy \wedge Ryz) \rightarrow Rxz$$

When run with the algebras corresponding to one of the ordered structures above, Flipper terminates successfully for this sentence within a couple of seconds. This isn't much of a feat, except for the fact that the sentence, and presumably extensions of it, represent a weak spot for for the first order reasoning systems the author regards as the best. To be more specific these are each of the participants of CASC-20 [Sut05], and recent versions of each winner of the SAT division going back to and including CASC-15. Translations into appropriate formats were done with the tptp2X utility where otter-style clausification was used

when needed. See [SS98] for what this means. Except for four of these systems none terminated in reasonable time. Reasonable time one may take to mean a hundred times longer than Flipper. Two terminated with an error message, and two with a message to the effect of giving up. None of these four took part in the SAT division, which means they were not made nor tuned for establishing satisfiability.

By re-tuning one of the four, namely Otter [McC], one obtains a hyperresolution based prover, which is refutation-complete, and that terminates for the infinity axiom. By use of randomly generated sentences extending the infinity axiom it is not hard to find examples where Flipper terminates in reasonable time whereas the hyperresolution based prover does not. The following is the shortest found.

$$\begin{aligned} & \forall_x \forall_y \forall_z Rxy \vee Ryz \vee \neg Rxz \wedge \\ & \forall_x \forall_y \forall_z \neg Rxx \wedge \forall_x \forall_y \exists_z Rxx \wedge \forall_x \forall_y \forall_z \neg Rxy \vee \neg Ryz \vee Rxz \end{aligned}$$

6 Conclusion

The author hopes to have demonstrated that by working in a conservative reduction class for first order logic one can do automatic search for satisfying interpretations in given theories, provided these theories decide a sufficient yet finite portion of the corresponding polyadic algebra. This finite portion is precisely defined by the introduced notion of partial polyadic closure. By combining the enriching construction and the relative consistency prover one obtains a disprover that in principle exceeds methods relying on interpretation in finite structures alone, since the consistency of infinity axioms can also be established. Some further directions to go in are; implementation of suitable reductions for first order logic or fragments thereof, implementation of the closure algorithm to tackle theories beyond those for successor and ordering on natural or rational numbers, implementation of strategies sharper than stochastic local search, such as methods based on gradual translation to propositional problems.

7 Acknowledgement

I am indebted to Prof. Stål O. Aanderaa for help with an unpublished paper on a not entirely unrelated method for recognising finitely satisfiable sentences as well as infinity axioms.

References

- [Aan71] S. Aanderaa. On the decision problem for formulas in which all disjunctions are binary. In *Proc. 2nd Scandinavian Logic Symp.*, volume 48, pages 1–18, 1971.
- [ARR03] A. Armando, S. Ranise, and M. Rusinowitch. A rewriting approach to satisfiability procedures. *Information and Computation*, 193(2):140–164, 2003. Available from <http://www.loria.fr/~rusi/pub/longcs101.ps>.

- [Bör71] E. Börger. *Reduktionstypen in Krom- und Hornformeln*. PhD thesis, Universität Münster, 1971.
- [Büc62] J. R. Büchi. Turing machines and the entscheidungsproblem. *Math. Annalen*, 148:201–213, 1962.
- [FLS] C.G. Fermüller, A. Leitsch, and G. Salzer. Automated model building as future research topic. Available from <http://citeseer.ist.psu.edu/88447.html>.
- [FV59] S. Feferman and R. L. Vaught. The first-order properties of algebraic systems. *Fund. Math.*, 47:57–103, 1959.
- [Gur66] Y. Gurevich. On the algorithmic decision of the satisfiability of predicate logic formulas. *Algebra i Logika*, 5:25–55, 1966. (In Russian).
- [Hal62] P. R. Halmos. *Algebraic Logic*. Chelsea, New York, 1962.
- [KM01] N. Klarlund and A. Møller. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, University of Aarhus, January 2001. Notes Series NS-01-1. Available from <http://www.brics.dk/mona/>. Revision of BRICS NS-98-3.
- [Kos66] V. Kostyrko. The $\forall\exists\forall$ reduction class. *Kibernetika*, 2:17–22, 1966. (In Russian).
- [McC] W. McCune. Otter 3.3 reference manual.
- [Ném91] I. Németi. Algebraizations of quantifier logics: an introductory overview, 1991. Available from <http://citeseer.ist.psu.edu/nemeti91algebraizations.html>.
- [Rog06] A. Rognes. Flipper, 2006. Available from <http://flipper.berlios.de>.
- [Sla94] J. Slaney. Finder: Finite domain enumerator system description. In A. Bundy, editor, *Automated Deduction-CADE-12*, pages 798–801. Springer, Berlin, Heidelberg, 1994.
- [SLM92] B. Selman, H. J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In Paul Rosenbloom and Peter Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, Menlo Park, California, 1992. AAAI Press.
- [SS98] G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998. The library is available from <http://www.tptp.org>.
- [Sut05] G. Sutcliffe. The IJCAR-2004 Automated Theorem Proving Competition. *AI Communications*, 18(1):33–40, 2005.
- [Tam] T. Tammet. Finite model building: Improvements and comparisons. Available from <http://citeseer.ist.psu.edu/675660.html>.
- [Tam92] T. Tammet. *Resolution Methods for Decision Problems and Finite-Model Building*. PhD thesis, 1992.

Diagnosing a Failed Proof in Fault-Tolerance: A Disproving Challenge Problem^{*}

Lee Pike¹, Paul Miner², and Wilfredo Torres-Pomales²

¹ Galois Connections
Beaverton, Oregon, USA
leepike@galois.com

² NASA Langley Research Center
Hampton, VA, USA
{p.s.miner, w.torres-pomales}@larc.nasa.gov

Abstract. This paper proposes a challenge problem in disproving. We describe a fault-tolerant distributed protocol designed at NASA for use in a fly-by-wire system for next-generation commercial aircraft. An early design of the protocol contains a subtle bug that is highly unlikely to be caught in fault-injection testing. We describe a failed proof of the protocol’s correctness in a mechanical theorem prover (PVS) with a complex unfinished proof conjecture. We use a model checking suite (SAL) to generate a concrete counterexample to the unproven conjecture to demonstrate the existence of a bug. However, we argue that the effort required in our approach is too high and propose what conditions a better solution would satisfy. We carefully describe the protocol and bug to provide a challenging but feasible case study for disproving research.

1 Introduction

Although rarely discussed in the archival literature, many attempts to prove conjectures using interactive mechanical theorem proving fail. Provided the theorem prover is sound and the conjecture is not both true and unprovable – a possibility in mathematics – there are two possible reasons for a failed proof attempt. First, the conjecture

may be true, but the user lacks the resources or insight to prove it. Second, the conjecture may be false. It can be difficult to determine which of these is the case.

When mathematicians cannot complete a proof of a conjecture, they begin to seek a counterexample to it. Mechanical theorem proving can exacerbate this difficult task. The difficulty is partly due to theorem provers often being used to reason about algorithms and protocols. Proofs of correctness in this domain often involve nested case-analysis. A proof obligation that cannot be completed is often deep within the proof, where intuition about the system behavior – and what would constitute a counterexample – wanes. The difficulty is also due to the nature of mechanical theorem proving. The proof steps issued in such a system are fine-grained. Formal specifications make explicit much of the detail that is suppressed in informal models. The detail and formality of the specification and proof makes the discovery of a counterexample more difficult.

We present a case study that highlights this difficulty. We describe the formal verification of a distributed fault-tolerant protocol in the mechanical theorem prover PVS [2]. A conjecture about the protocol is partially verified by case-analysis, leaving a single unproven case. The case involves a complex set of fault statuses and system invariants.

In particular, the protocol investigated is an interactive consistency protocol for use in the Reliable Optical Bus (ROBUS), a state-of-the-art ultra-reliable communications bus under development at the NASA Langley Research Center and the National Institute of Aerospace. It is being developed as part of the Scalable Processor-Independent Design for Extended Reliability (SPIDER) project [3, 4]. SPIDER is a family of ultra-reliable architectures built upon the ROBUS. Currently, ROBUS implementations include a FPGA-based prototype.

The counterexample was initially discovered by the third author through “engineering insight.” The bug in the protocol design occurs only when there are two simultaneous Byzantine faults [5, 6]. As described in greater detail later in the paper, the bug arises from the interaction between the system’s fault assumptions and the local diagnoses made by nodes in the system. Local diagnoses are used in a fault-tolerant system to increase reliability and to maintain *group membership*, a group of mutually-trusted non-faulty nodes [7]. In a sense, the bug is due to the interplay of system operation (i.e., executing the protocol) and system survival

^{*} This research was supported, in part, by Research Cooperative Agreement No. NCC-1-02043 awarded to the National Institute of Aerospace while the first author was a visitor. Additional support came from NASA’s Vehicle Systems Program. This paper is a revised extended abstract of an (unrefereed) NASA technical memorandum [1].

(i.e., maintaining group membership). These concerns apply to variety of fault-tolerant embedded systems [8].

The protocol is designed to tolerate such a fault scenario. However, the subtlety of the scenario means it is extremely unlikely the bug would be caught during fault-injection testing [9]. Nevertheless, safety-critical systems like SPIDER that are designed for use in commercial aircraft must have a failure rate no higher than 10^{-9} to 10^{-12} per hour of operation [8, 10]. A design error that escapes testing could adversely affect a system’s reliability. We believe that if the bug had not been caught by insightful inspection, the only other way it would be caught is through formal analysis.

In the paper, we describe our approach to formally uncover the bug.³ Specifically, we model the failed proof obligation in a model checker, and attempt to prove it holds in a model in which parameters have been interpreted with small constants. Using the counterexample generated by the model checker, one can quickly determine that the protocol is incorrect. Furthermore, the counterexample suggests the appropriate modification to correct the bug.

Motivation Unfortunately, we feel our approach is inadequate for the following reasons:

- The approach is too interactive and onerous. It requires manually specifying the protocol and failed conjecture in a model checker and manually correcting the specification in the theorem prover.
- The approach depends on the counterexample arising by instantiating the parameters with small finite values.
- Indeed, we would like a more automated approach to verify the parameterized protocol specification in the first place than is possible using mechanical theorem proving alone.

Therefore, we offer this case study as a challenge problem to the disproving community. We believe researchers will find this problem of interest for the following reasons:

- The protocol is industrially-relevant, and the bug is genuine.
- The protocol can be described in English in just a few paragraphs (as is done in this paper), but the behavior of the protocol itself is subtle.

³ The associated files can be retrieved at http://www.cs.indiana.edu/~lepik/pub_pages/disproving.html.

- While we believe our approach is unsatisfactory, we also believe it approximates the best contemporary approach for disproving problems like the one presented (a purpose of this paper is to solicit evidence to the contrary).
- Formal specifications of the protocol in both a mechanical theorem prover and a model checker accompany this paper for reference.

Organization We describe the ROBUS Interactive Consistency (IC) Protocol as well as the architecture on which it is intended to execute in Section 2. In Section 3, we describe the kinds and number of faults under which the ROBUS IC Protocol should correctly execute. Section 4 states the correctness requirements for the protocol as well as the state invariants that must hold for the ROBUS IC Protocol to satisfy them. In Section 5, we informally describe the counterexample, discuss its origins, and provide a “fix” for it. In Section 6 we describe the conjecture attempted in PVS and then our generation of a counterexample using the Symbolic Analysis Laboratory (SAL) [11]. Section 7 outlines the specific challenge and describes some metrics for success. Concluding remarks are in Section 8.

2 The ROBUS IC Protocol

We begin by describing the background of the family of protocols from which the ROBUS IC Protocol comes. Then after describing the architecture of the ROBUS, we describe the protocol’s behavior itself.

Background Protocols like the one described in this paper are fault-tolerant consensus algorithms and are known as “interactive consistency” or “oral messages” protocols. The protocol presented here is based on a protocol designed by Davies and Wakerly [12]. Lynch’s textbook provides an introduction to these sort of protocols as well as pointers into the literature [13]. Many of these protocols have been formally verified, both by theorem proving [14–16] and by model checking [17].

Architecture The architecture of the ROBUS is a fully-connected bipartite graph of two sets of nodes, *Bus Interface Units* (BIUs) and *Redundancy Management Units* (RMUs). BIUs provide the interface between the bus and hosts running applications that communicate over the bus. The RMUs provide redundancy. The architecture for the special case of three BIUs and three RMUs is shown in Figure 1. There must be a minimum of

one BIU and one RMU, and there need not be an equal number of BIUs and RMUs.

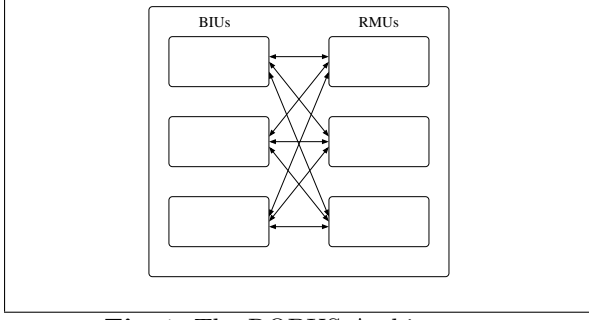


Fig. 1: The ROBUS Architecture

Diagnostic Data Understanding the protocol behavior requires a preliminary understanding of the diagnostic data collected by nodes. The protocol has a greater chance of succeeding if good nodes ignore faulty ones. Consequently, nodes maintain *diagnoses* against other nodes. These diagnoses result from mechanisms to monitor the messages received during protocol execution. Diagnostic data is accumulated over multiple protocol executions.

Each node maintains a *diagnostic function* assigning each node (including itself) to one of the following three classifications: *trusted*, *accused*, and *declared*. Every (non-faulty) node assigns every other node to exactly one class. We call the node being labeled the *defendant*. If a node labels a defendant as *trusted*, then the node has insufficient evidence that the defendant is faulty. If it labels a defendant as *accused*, then it has local evidence that the defendant is faulty, but does not know whether other good nodes have similar evidence. Once a defendant is *declared*, all good nodes know that they share the declaration.

Periodically, the RMUs and BIUs execute a *Distributed Diagnosis Protocol* in which the nodes submit the diagnoses accumulated thus far [7]. If enough good nodes have accused a defendant, then the defendant is *declared*. The Distributed Diagnosis Protocol ensures that all good nodes agree on which nodes have been declared.

2.1 Protocol Description

We distinguish one BIU as the General. The ROBUS IC Protocol is a synchronous protocol designed to reliably transmit the General’s message despite faults in the system (the formal requirements are provided in Section 4). In the following, a *benign message* is one that all nonfaulty nodes can detect came from a faulty node (see Section 3). The ROBUS IC Protocol is as follows

(the message-passing events of the protocol are illustrated in Figure 2):

1. The General, G , broadcasts its message, v , to all RMUs.
2. For each RMU, if it receives a benign message from G , then it broadcasts the special message *source error* to all BIUs. Otherwise it relays the message it received.
3. For each BIU b , if b has declared G , then b outputs the special message *source error*. Otherwise, if b received a benign message from an RMU, then that RMU is accused. b performs a majority vote over the values received from those RMUs it trusts. If no majority exists, *source error* is the result; otherwise, the majority value is the result.

3 Faults

Fault Classifications Faults result from innumerable occurrences including physical damage, electromagnetic interference, and “slightly-out-of-spec” communication [5]. We collect these fault occurrences into *fault types* according to their effects in the system.

We adopt the *hybrid fault model* of Thambidurai and Park [18]. All non-faulty nodes are also said to be *good*. A node is called *benign*, or *manifest*, if it sends only benign messages. Benign messages abstract various sorts of misbehavior. A message that is sufficiently garbled during transmission may be caught by an error-checking code and deemed benign. In synchronized systems with global communication schedules, they also abstract messages not sent (i.e., a message is expected by a receiver but is absent on a communication channel) or messages that arrive at unscheduled times. A node is called *symmetric* if it sends every receiver the same message, but these messages may be incorrect. A node is called *asymmetric*, or *Byzantine* [6], if it arbitrarily sends different messages to different receivers.

Fault Assumption A fault-tolerant protocol is designed to tolerate a certain number of faults of each fault type. For a protocol, this is specified by its *maximum fault assumption* (MFA). A proof of correctness of a protocol is of the form, “If the MFA holds, then the protocol satisfies property P ,” where P is a correctness condition for the protocol. The probability that a MFA holds is determined by reliability analysis [19].

We call the MFA for the ROBUS IC Protocol the *Interactive Consistency Dynamic Maximum*

Fault Assumption (IC DMFA). ‘Dynamic’ emphasizes that the fault assumption is parameterized by the local diagnoses of nodes, which change over time.

Definition 1 (IC DMFA). Let GB , SB , and AB denote the sets of BIUs that are good, symmetrically-faulty, and asymmetrically-faulty, respectively. Let GR , SR , and AR represent the corresponding sets of RMUs, respectively. For good BIU b , let T_b denote the set of RMUs b trusts. This is b ’s trusted set. Define T_r similarly – it is the set of BIUs that RMU r trusts. The following formulas together make up the IC DMFA. G is the General. For all good BIUs b and good RMUs r ,

1. $|GR \cap T_b| > |SR \cap T_b| + |AR \cap T_b|$;
2. $G \in AB \cap T_r$ implies $|AR \cap T_b| = 0$.

The first clause ensures that a good BIU b contains strictly more good RMUs in T_b than it does symmetrically-faulty or asymmetrically-faulty RMUs. The second clause ensures that either no good RMU r trusts an asymmetrically-faulty General, or no good BIU b trusts an asymmetrically-faulty RMU.

4 The ROBUS IC Protocol Correctness

We begin by stating the requirements for the ROBUS IC Protocol. We then state invariants that must hold in a system executing the ROBUS IC Protocol in order for it to meet these requirements.

Requirements Two requirements must hold.

Definition 2 (Agreement). All good BIUs compute the same value.

Definition 3 (Validity). If the General is good and broadcasts message v , then the value computed by a good BIU is v .

Diagnostic Assumptions In addition to constraining the number of and kind of faults, the correctness of the ROBUS IC Protocol depends on the diagnostic mechanisms satisfying certain constraints. Let b_1 and b_2 be good BIUs, let r_1 be a good RMU, and let n be either a BIU or RMU of any fault classification.

Definition 4 (Good Trusted). b_1 trusts n if n is good.

Definition 5 (Symmetric Agreement). If n is not asymmetrically-faulty, b_1 accuses n if and only if b_2 accuses n .

Definition 6 (Conviction Agreement). b_1 declares n if and only if r_1 declares n .

These properties similarly hold for any two good RMUs with respect to a defendant n .

Intuitively, *Good Trusted* ensures that diagnostic mechanisms never lead a good node to accuse another good node. *Symmetric Agreement* ensures that all good nodes that receive the same data make the same diagnosis. Note, however, that Symmetric Agreement allows a good BIU and a good RMU to make different diagnoses about a node that is asymmetrically-faulty. Finally, *Conviction Agreement* is a correctness requirement of the Distributed Diagnosis Protocol [7], and it is a precondition for the correctness of the protocol under investigation in this paper. Together, these three assumptions are called the *Diagnostic Assumptions*.

5 The Counterexample

We describe the counterexample informally and briefly describe its origins. We then describe a protocol that does not suffer from the flaw.

Counterexample The following instance of the ROBUS IC Protocol violates Agreement. Consider an architecture containing three BIUs, G , b_1 , and b_2 , and three RMUs r_1 , r_2 , and r_3 . Let the General be asymmetrically-faulty. Let RMU r_1 be asymmetrically-faulty, too, and let all other nodes be good. Suppose b_1 and b_2 either accuse or trust G (it does not matter which), and they trust all RMUs. Furthermore, suppose b_1 and b_2 trust r_1 , but no good RMU trusts G . These hypotheses satisfy the IC DMFA and the Diagnostic Assumptions. Agreement is violated if the following instance of the ROBUS IC Protocol transpires, as illustrated in Figure 2.

1. G sends message v to r_1 and r_2 , and it sends message u to r_3 , where $v \neq u$.
2. r_1 sends message v to b_1 and u to b_2 . r_2 sends message v to both b_1 and b_2 . r_3 sends message u to both b_1 and b_2 .
3. b_1 outputs v whereas b_2 outputs u .

Origins of the Flaw The flaw in the ROBUS IC Protocol was introduced when an earlier version of the protocol was amended to allow for the reintegration of transiently-faulty nodes [20]. A node becomes transiently-faulty when its state is disrupted (due, e.g., to exposure to high-intensity radiation), but the node is not permanently damaged. A node that suffers a transient fault has the

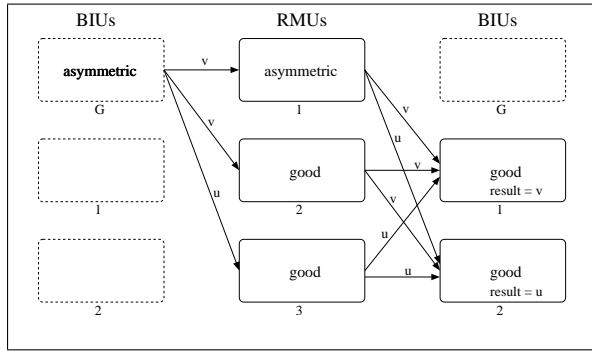


Fig. 2: An Instance of the ROBUS IC Protocol Violating Agreement

potential to *reintegrate* with the good nodes in the system by restoring consistent state with them.

In the earlier version of the ROBUS IC Protocol, an RMU would only relay a message from the General if it trusted the General. Otherwise, the *source error* message was relayed. To allow for reintegration, the messages from a previously-declared General needed to be relayed by RMUs so that the BIUs can determine whether it is fit for reintegration. However, the flaw in the ROBUS IC Protocol arose when the earlier protocol was changed so that RMUs relayed the message from the General regardless of its diagnostic status, so long as it did not send a benign message.

A New ROBUS IC Protocol In retrospect, a fix to the protocol is simple. Step 2 of the protocol description in Section 2.1 is changed so that an RMU r relays the message *source error* if it receives a benign message or if r accuses the General. If the General is declared, its message is relayed to allow BIUs to gather diagnostic data about the General. An accused General implies that the General recently suffered a fault (assuming the accuser is good), so there is no need to relay its message for reintegration purposes. The correctness of the protocol, proved in PVS, is described by Miner et al. [21].

6 Formally Deriving the Counterexample

In this section, we describe the unfinished proof obligation generated in our attempt to formally prove a conjecture about the ROBUS IC Protocol. We then describe our use of a model checker to derive a counterexample to the conjecture.

6.1 Generating the Proof Obligation

In our approach, we use the PVS theorem proving system developed by SRI International [2].

We have used PVS to specify and verify other ROBUS protocols [21, 7]. The specification language of PVS is a strongly-typed higher-order logic, and the proof system is the classical sequent calculus.

Various details about the construction of the underlying theories used to model the algorithm and ROBUS are irrelevant.⁴ A general discussion of the abstractions used in the model is provided elsewhere [22]. The following notation is used in the formal statements of the Agreement Conjecture and the unproved sequent in Figure 3 and Figure 4.

Variables and Parameters The parameters B and R are uninterpreted natural numbers. The set of BIUs and RMUs are indexed from 0 to $B - 1$ and 0 to $R - 1$, and these sets of indices are denoted $\text{below}(B)$ and $\text{below}(R)$, respectively. Thus, the PVS specification is parameterized by the number of BIUs and RMUs, and a proof of correctness holds for any instantiation of these parameters that satisfy the hypotheses of the proof. Let $b1, b2, G \in \text{below}(B)$, where G is used to designate the General. F is a variable over the set of *records* (i.e., named tuples [23]) that contain all of the diagnoses in the system. The $'$ operator provides record access. Thus, $F'RB$ denotes the collection of the BIUs' diagnoses against the RMUs, $F'BB$ denotes the BIUs' diagnoses against the BIUs, and similarly for $F'RB$ and $F'RR$. $F'RB(b1)(r)$ denotes $b1$'s diagnosis of r , and similarly for the other diagnoses. $F'RB(b1)(r)$ yields a value from the set $\{\text{trusted}, \text{accused}, \text{declared}\}$. The function b_status is a function mapping BIUs to some fault class – one of *good*, *benign*, *symmetric*, and *asymmetric*, and similarly, r_status maps RMUs to a fault class. Finally, msg is an arbitrary message being broadcast by the General.

Functions and Relations The following functions and relations appear in the sequent.

- *good?* is a predicate that takes the fault status of a node and is true if the status is *good*. *benign?*, *symmetric?*, and *asymmetric?* are similarly defined.
- *IC_DMFA* is a formal statement of the IC DMFA described in Section 3.
- *all_correct_accs?* is a predicate formally stating the Diagnostic Assumptions defined in Section 4.

⁴ The PVS models are more abstract than needed to model this protocol since the many of the same theories are generalized to model other ROBUS protocols [21].

- **declared?** is a predicate that takes the diagnosis made by one node against a defendant node and is true if the defendant is declared. Similarly, **trusted?** is true if the defendant is trusted.
- **robust_ic** is a higher-order function that functionally models the ROBUST IC Protocol, as described in Section 2.1. It takes as arguments the fault statuses of the BIUs and RMUs, the diagnoses a BIU makes of G , as well as the set of its other diagnoses. It returns another function that takes the General's identifier, the message it sends, and a BIU identifier. The function returns the message the BIU outputs after the execution of the ROBUST IC Protocol. The function is essentially the composition of two functions modeling the two rounds of message passing (recall that we are modeling the protocol in the synchronous domain, so the rounds of message passing is the granularity at which time is modeled).

It is the convention of PVS to denote skolem constants with a trailing “!n,” where n is some integer.

The Sequent The conjecture to be proved is shown in Figure 3. Assuming that $b1$ and $b2$ are both good, that the Diagnoses Assumptions hold, and that the IC DMFA holds, we attempt to prove that the result of **robust_ic** is the same when applied to $b1$ and $b2$.

```

Agreement: CONJECTURE
  good?(b_status(b1)) AND
  good?(b_status(b2)) AND
  all_correct_accs?(b_status, r_status, F) AND
  IC_DMFA(b_status, r_status, F)
=>
  robust_ic(b_status, r_status, F'BB(b1)(G), F'RB(b1))
    (G, msg, b1) =
  robust_ic(b_status, r_status, F'BB(b2)(G), F'RB(b2))
    (G, msg, b2)

```

Fig. 3: The Agreement Conjecture in PVS

Every branch of the conjecture in Figure 3 is discharged except for the branch ending in the single sequent in Figure 4 (irrelevant formulas have been omitted). PVS labels the formulas in the antecedent with negative integers, while those in the consequent are labeled with positive integers.

6.2 Model Checking the Sequent

We use the Symbolic Analysis Laboratory (SAL) [24, 11], also developed by SRI International, to model check the protocol against the undischarged sequent. SAL is a family of model checkers that includes symbolic, bounded, and

```

[-1] good?(r_status!1(r!1))
[-2] asymmetric?(b_status!1(G!1))
[-3] IC_DMFA(b_status!1, r_status!1, F!1)
[-4] all_correct_accs?(b_status!1, r_status!1, F!1)
|-----
[1]  trusted?(F!1'BR(r!1)(G!1))
[2]  declared?(F!1'BB(b2!1)(G!1))
{3}  (FORALL (p_1: below(R)):
      (trusted?(F!1'RB(b1!1)(p_1)) =>
        NOT asymmetric?(r_status!1(p_1))))
    &
    (FORALL (p_1: below(R)):
      (trusted?(F!1'RB(b2!1)(p_1)) =>
        NOT asymmetric?(r_status!1(p_1))))
[4]  declared?(F!1'BB(b1!1)(G!1))
[5]  robust_ic(b_status!1, r_status!1,
      F!1'BB(b1!1)(G!1), F!1'RB(b1!1))
    (G!1, msg!1, b1!1)
    =
    robust_ic(b_status!1, r_status!1,
      F!1'BB(b2!1)(G!1), F!1'RB(b2!1))
    (G!1, msg!1, b2!1)

```

Fig. 4: The Unproven PVS Sequent

explicit-state model checkers, among other tools. The SAL language includes constructs such as recursive function definition, synchronous and asynchronous composition operators, and quantifiers over finite types. We particularly exploit the quantifier, recursive function, and synchronous composition constructs.

Our SAL model builds on the model of Oral Messages that is explained in detail in Rushby's SAL tutorial [17]. Our model differs slightly as we must represent the local diagnoses data of each node, the Diagnosis Assumptions, and the IC DMFA, which is parametrized by the local diagnoses. Furthermore, we state these constraints explicitly rather than embedding them into the system model. We found this makes our model more perspicuous.

A sequent can be read as stating that if the conjunction of the antecedent statements is true, then the disjunction of the consequent statements is true. That is, if \mathcal{A} is the set of antecedents and \mathcal{C} is the set of consequents, a sequent is equivalent to the conditional

$$\bigwedge \mathcal{A} \Rightarrow \bigvee \mathcal{C}. \quad (1)$$

This formulation is used to express the sequent in SAL and appears in Figure 5. There, **SYSTEM** denotes the model of the ROBUST IC Protocol developed in the model checker, the symbol \models denotes the purported satisfaction relation between the model and G is the global-state operator of LTL (not to be confused with the denotation of the General).

SAL has an imperative language, so some of the predicates in the PVS sequent have been expressed equationally. Some of the functions of PVS have been converted to arrays in SAL, giving rise to the bracket notation.

```

counterex: THEOREM SYSTEM |-
G( (pc = 4 AND
   r_status[1] = good AND
   G_status = asymmetric AND
   IC_DMFA(r_status, F_RB, F_BR, G_status) AND
   all_correct_accs(r_status, F_RB,
                   G_status, F_BR, F_BB))
=>
(F_BR[1] = trusted OR
 F_BB[2] = declared OR
 (FORALL (r: RMUs): F_RB[1][r] = trusted =>
  r_status[r] /= asymmetric AND
  FORALL (r: RMUs): F_RB[2][r] = trusted =>
   r_status[r] /= asymmetric) OR
 F_BB[1] = declared OR
 robus_ic[1] = robus_ic[2]));

```

Fig. 5: The SAL Formulation of the Undischarged Sequent

Two additional statements in the LTL formulation are artifacts of how the protocol is modeled in the model checker, both of which come from Rushby’s original formulation. First, there is a program counter `pc` that represents which round of the protocol is currently executing. These rounds correspond to the three rounds described in Section 2.1. When `pc = 4`, the last round has completed. The second artifact is the imperative definition of the result of the ROBUS IC Protocol using the array called `robus_ic`.

Thus, the conjecture in Figure 5 can be read as stating that in every state reachable from the initial state of **SYSTEM**, the formulation of the unproven sequent described above is true.

A counterexample to the formula in Figure 5 is a reachable state in which the formula is false. As mentioned, that formula is derived from the conditional interpretation of a sequent in (1). The negation of (1) is equivalent to

$$\bigwedge (\mathcal{A} \cup \bar{\mathcal{C}}), \quad (2)$$

where $\bar{\mathcal{C}}$ denotes the negation of each formula in \mathcal{C} . A counterexample is therefore a reachable state in which (2) is true. Such a state matches the informal description of the counterexample in Section 5.

Using SAL’s symbolic model checker on a system with one gigabyte of memory and an AMD Athlon 2000+ processor, a counterexample like the one described in Section 5 is discovered in about 16 seconds for three RMUs and three BIUs, including the General. One may wonder whether this counterexample arises from the system having too few RMUs to relay messages. Increasing the number of RMUs quickly overwhelms the symbolic model checker. However, we obtain a similar counterexample using SAL’s bounded model checker for seven RMUs in a little over two minutes on the same system.

These concrete counterexamples demonstrate that the unproved sequent cannot be discharged because the protocol itself has an error. Changing the PVS and SAL models to include the fix suggested in Section 5 allows the Agreement proof to be completed [21], and SAL verifies the formula in Figure 5 in a model using the same number of BIUs and RMUs as used to find the counterexample (the fix is included as commented code in the SAL model available on-line³).

6.3 Remarks on our Approach

In our approach, we manually model the protocol and the requirements both in PVS and SAL. This is simultaneously advantageous and disadvantageous. Having to model the protocol and requirements in distinct languages provides an additional guard against modeling errors in each language. In particular, we wish to guard against false negatives, which are particularly easy to generate in model checkers.

A disadvantage is the additional work required to model the protocol and requirements twice. This approach is not feasible to check numerous failed proof conjectures in a proof attempt.

Some limitations of this approach are inherent to the limitations of symbolic model checking, in general. A model checker is useful when the system can be modeled as a finite-state state machine, and the requirements to be proved can be modeled in a temporal logic. As well, a counterexample may exist, but be beyond the computational limits of the model checker and the computer on which it executes.

Despite these limitations, we believe this approach approximates the current state-of-the-art to verify and discover bugs for a protocol like the one described.

Related Work Much previous work that integrates model checking and interactive theorem proving has focused on using model checking to automate proving rather than on disproving [25, 26]. Some theorem provers have embedded model checkers (both PVS and ACL2 contain embedded μ -calculus model checkers [27, 28]). Most related to our work is a study in which resolution-based theorem proving and model checking are used to discover counterexamples to proof obligations [29]. Our work differs in that we present a reasonably intricate protocol whereas a small illustrative example is presented therein. As well, the focus therein is on *automated* theorem proving; our focus is on using model checking to facilitate *interactive* theorem proving.

7 The Challenge

The challenge is as follows:

From a parameterized specification of the protocol (from which a general proof can be obtained), provide a concrete instance of the bug in a way that requires as little effort from the user as possible.

This section describes how our approach could be improved as well as speculation about other approaches.

Specification Languages In our specific case, the specification languages of PVS and SAL are similar, and it is a goal of SRI to develop translators between them [30]. Building a translator between the languages is not trivial as the languages overlap, but neither is a subset of the other. The lack of a translator required us to model the protocol twice.

More generally, efficient disproving requires better integration between proving tools (e.g., theorem provers) and disproving tools (e.g., model checkers) [31, 32] regardless of the theorem prover or model checker choice made. As noted in Section 6, some theorem-provers contain model-checkers, but it is not clear that these model checkers can handle this case study.

Parameter Interpretation Even if a translator existed, the parameterized PVS specification is not immediately amenable to finite-state model checking: the parameters must be interpreted. For a system with many parameters or a large specification, the interpretation is tedious if done manually. The tedium is compounded by the need to find parameters small enough to make model checking feasible yet large enough to expose the counterexample.

A quickcheck approach, like the one that appears in Isabelle [33], may be sufficient to demonstrate a counterexample. However, doing so requires a reformulation of the problem since our formulation in PVS uses non-executable constructs (e.g., quantifiers and Hilbert’s choice operator). How this could be done for a parameterized specification of the protocol that is not implementation-specific is unknown to the authors.

Proof Calculus Translation A minor but relevant task in our approach is the translation of a failed conjecture in the sequent calculus into an LTL formula. We hope for this task to be automated, too (for whatever proof calculus and temporal logic is used).

Proving and Disproving Ideally, both proving and disproving would be automatic. Automated disproving appears to be an easier challenge to meet than proving. That said, recent advances in *satisfiability modulo theories* (SMT) provers [34] hold promise. Fault-tolerant protocols have generally been good candidates for mechanical theorem proving given their criticality and complexity, but SMT technology may provide a more automated approach. Currently, SMT provers are not capable of proving a fully-parameterized specification of the protocol presented, but recent applications of SMT to verifying fault-tolerant and real-time protocols are promising [35, 20]. For these sort of verifications, SMT is particularly useful when combined with bounded-model checking to do highly-automated induction proofs of safety properties over infinite-state systems [36]. We hope the SMT community also takes up the protocol presented as a challenge problem in parameterized proof as well as counterexample generation.

We do not believe that automated first-order theorem provers alone can prove the correctness of the SPIDER IC Protocol. The protocol’s proof of correctness is parameterized in the number of BIUs and RMUs. Furthermore, the proof of correctness depends on reasoning about nontrivial mathematical facts (e.g., the IC DMFA). A first-order theorem prover could possibly be used to derive a counterexample for a fixed specification, replacing the model checker in our approach. Nonetheless, a similar interactive consistency protocol has been specified and verified in ACL2, an interactive first-order theorem prover [14]. Also, some initial work has been done to translate the PVS specification of the SPIDER IC Protocol into a form suitable for SAT solving [37].

Effort Requirements Given the criticality of the correct design of SPIDER and similar safety-critical embedded systems, their designers are willing to invest a great deal of effort to gain high assurance of their correctness. Therefore, the acceptable level of effort required to obtain counterexamples is relatively high, as evidenced by the use of interactive theorem proving in the first place. Certainly the upper bound on the acceptable level of effort to uncover a counterexample is the time it takes a user to diagnose the error in the failed proof attempt in the theorem prover. This effort varies according to experience with the theorem prover, expertise in the domain, the proof infrastructure, and the specific reason the proof has failed.

8 Conclusion

We have presented a subtly-flawed fault-tolerant protocol, its attempted verification by theorem proving, and our use of a model checker to demonstrate that a bug in the protocol prevents us from completing the verification. As noted, we believe our approach approximates the best possible with today's technology.

More generally, a variety of real-time fault-tolerant protocols have been designed for SPIDER and are described in detail [4]. Most of these protocols have been verified using theorem proving and model checking with SMT [38, 21, 20]. These protocols are all suitable case studies to demonstrate novel verification tools and techniques.

Acknowledgments

The PVS and SAL tools used were developed by SRI International. We used PVS theories developed by members of the NASA Langley Formal Methods Group and the National Institute of Aerospace; in particular, we thank Alfons Geser and Jeffrey Maddalon. We thank our anonymous reviewers of the DISPROVING Workshop for their helpful and detailed comments. As noted, the SAL model was adopted from work by Rushby [17].

References

1. Pike, L., Miner, P., Torres, W.: Model checking failed conjectures in theorem proving: a case study. Technical Report NASA/TM-2004-213278, NASA Langley Research Center (2004) Available at http://www.cs.indiana.edu/~lepique/pub_pages/unproven.html.
2. Owre, S., Rusby, J., Shankar, N., von Henke, F.: Formal verification for fault-tolerant architectures: Prolegomena to the design of pvs. *IEEE Transactions on Software Engineering* **21** (1995) 107–125
3. NASA Formal Methods Group: SPIDER homepage. Website (2004) Available at <http://shemesh.larc.nasa.gov/fm/spider/>.
4. Torres-Pomales, W., Malekpour, M.R., Miner, P.: ROBUS-2: A fault-tolerant broadcast communication system. Technical Report NASA/TM-2005-213540, NASA Langley Research Center (2005)
5. Driscoll, K., Hall, B., Sivencrona, H., Zumsteg, P.: Byzantine fault tolerance, from theory to reality. In Goos, G., Hartmanis, J., van Leeuwen, J., eds.: *Computer Safety, Reliability, and Security. Lecture Notes in Computer Science, The 22nd International Conference on Computer Safety, Reliability and Security SAFECOMP*, Springer-Verlag Heidelberg (2003) 235–248
6. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. *Journal of the ACM* **27** (1980) 228–234
7. Geser, A., Miner, P.: A formal correctness proof of the SPIDER diagnosis protocol. Technical Report NASA/CP-2002-211736, NASA Langley Research Center, Hampton, Virginia (2002) Technical Report contains the Track B proceedings from Theorem Proving in Higher Order Logics (TPHOLSS).
8. Rushby, J.: Bus architectures for safety-critical embedded systems. In Henzinger, T., Kirsch, C., eds.: *EMSOFT 2001: Proceedings of the First Workshop on Embedded Software*. Volume 2211 of *Lecture Notes in Computer Science*, Lake Tahoe, CA, Springer-Verlag (2001) 306–323
9. Hsueh, M.C., Tsai, T.K., Iyer, R.K.: Fault injection techniques and tools. *IEEE Computer* **30** (1997) 75–82 Available at citeseer.ist.psu.edu/hsueh97fault.html.
10. Kopetz, H.: *Real-Time Systems*. Kluwer Academic Publishers (1997)
11. de Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N., Sorea, M., Tiwari, A.: SAL 2. In Alur, R., Peled, D., eds.: *Computer-Aided Verification, CAV 2004*. Volume 3114 of *Lecture Notes in Computer Science*, Boston, MA, Springer-Verlag (2004) 496–500
12. Davies, D., Wakerly, J.F.: Synchronization and matching in redundant systems. *IEEE Transactions on Computers* **27** (1978) 531–539
13. Lynch, N.A.: *Distributed Algorithms*. Morgan Kaufmann (1996)
14. Young, W.D.: Comparing verification systems: Interactive consistency in ACL2. *IEEE Transactions on Software Engineering* **23** (1997) 214–223
15. Bevier, W., Young, W.: The proof of correctness of a fault-tolerant circuit design. In: *Second IFIP Conference on Dependable Computing For Critical Applications*. (1991) Available at <http://citeseer.ist.psu.edu/bevier91proof.html>.
16. Lincoln, P., Rushby, J.: Formal verification of an interactive consistency algorithm for the Draper FTP architecture under a hybrid fault model. In: *Compass '94 (Proceedings of the Ninth Annual Conference on Computer Assurance)*, Gaithersburg, MD, IEEE Washington Section (1994) 107–120 Available at <http://www.cs1.sri.com/papers/compass94/>.
17. Rushby, J.: SAL tutorial: Analyzing the fault-tolerant algorithm OM(1). Technical Report CSL Technical Note, SRI International (2004) Available at <http://www.cs1.sri.com/users/rushby/abstracts/om1>.
18. Thambidurai, P., Park, Y.K.: Interactive consistency with multiple failure modes. In: *7th Reliable Distributed Systems Symposium*. (1988) 93–100

19. Butler, R.W.: The SURE approach to reliability analysis. *IEEE Transactions on Reliability* **41** (1992) 210–218
20. Pike, L., Johnson, S.D.: The formal verification of a reintegration protocol. In: *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, New York, NY, USA, ACM Press (2005) 286–289 Available at http://www.cs.indiana.edu/~lepik/pub_pages/emsoft.html.
21. Miner, P., Geser, A., Pike, L., Maddalon, J.: A unified fault-tolerance protocol. In Lakhnech, Y., Yovine, S., eds.: *Formal Techniques, Modeling and Analysis of Timed and Fault-Tolerant Systems (FORMATS-FTRTFT)*. Volume 3253 of *Lecture Notes in Computer Science*, Springer (2004) 167–182 Available at http://www.cs.indiana.edu/~lepik/pub_pages/unified.html.
22. Pike, L., Maddalon, J., Miner, P., Geser, A.: Abstractions for fault-tolerant distributed system verification. In Slind, K., Bunker, A., Gopalakrishnan, G., eds.: *Theorem Proving in Higher Order Logics (TPHOLs)*. Volume 3223 of *Lecture Notes in Computer Science*, Springer (2004) 257–270 Available at http://www.cs.indiana.edu/~lepik/pub_pages/abstractions.html.
23. Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: *PVS Language Reference*. SRI International. Version 2.4 edn. (2001) Available at <http://pvs.csl.sri.com/manuals.html>.
24. SRI International: Symbolic analysis laboratory SAL (2004) Available at <http://sal.csl.sri.com/>.
25. Rushby, J.: Integrated formal verification: Using model checking with automated abstraction, invariant generation, and theorem proving. In Dams, D., Gerth, R., Leue, S., Massink, M., eds.: *Theoretical and Practical Aspects of SPIN Model Checking: 5th and 6th International SPIN Workshops*. Volume 1680 of *Lecture Notes in Computer Science*, Trento, Italy, and Toulouse, France, Springer-Verlag (1999) Available at <http://www.csl.sri.com/papers/spin99/>.
26. Havelund, K., Shankar, N.: Experiments in theorem proving and model checking for protocol verification. In: *Proceedings of Formal Methods Europe FME'96*. *Lecture Notes in Computer Science*, Springer (1996)
27. Shankar, N., Owre, S., Rushby, J.M., Stringer-Calvert, D.W.J.: *PVS Prover Guide*. SRI International. Version 2.4 edn. (2001) Available at <http://pvs.csl.sri.com/manuals.html>.
28. Manolios, P.: Chapter 7: Mu-Calculus Model-Checking. In: *Computer Aided Reasoning: ACL2 Case Studies*. Self-Published (2002)
29. Bicarregui, J.C., Matthews, B.M.: Proof and refutation in formal software development. In: *3rd Irish Workshop on Formal Methods (IWF'99)*. (1999)
30. SRI Computer Science Laboratory: Formal methods roadmap: PVS, ICS, and SAL. Technical Report SRI-CSL-03-05, SRI International, Menlo Park, CA 94025 (2003)
31. Johnson, S.D.: View from the fringe of the fringe. In Margaria, T., Melham, T., eds.: *11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Volume 2144 of *Lecture Notes in Computer Science*, Springer-Verlag (2001) 1–12
32. de Moura, L., Owre, S., Ruess, H., Rushby, J., Shankar, N.: Integrating verification components. In: *Verified Software: Theories, Tools, Experiments*. (2005)
33. Nipkow, S.B.T.: Random testing in Isabelle/HOL. In Cuellar, J., Liu, Z., eds.: *Software Engineering and Formal Methods (SEFM 2004)*, IEEE Computer Society (2004) 230–239
34. Barrett, C., de Moura, L., Stump, A.: SMT-COMP: Satisfiability Modulo Theories Competition. In Etessami, K., Rajamani, S., eds.: *17th International Conference on Computer Aided Verification*, Springer (2005) 20–23
35. Dutertre, B., Sorea, M.: Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In: *Formal Techniques in Real-Time and Fault-Tolerant Systems*. Volume 3253 of *Lecture Notes in Computer Science*, Grenoble, France, Springer-Verlag (2004) 199–214 Available at <http://fm.csl.sri.com/doc/abstracts/ftrtft04>.
36. de Moura, L., Rueß, H., Sorea, M.: Bounded model checking and induction: From refutation to verification. In Voronkov, A., ed.: *Computer-Aided Verification, CAV 2003*. Volume 2725 of *Lecture Notes in Computer Science*, Springer-Verlag (2003) 14–26
37. Sinz, C.: Propositional translation of PVS specifications. Talk slides (2004) Talk held at the National Institute of Aerospace/NASA Langley Research Center. Available at http://www-sr.informatik.uni-tuebingen.de/~sinz/pdf/prop_trans.pdf.
38. Pike, L.: Formal Verification of Time-Triggered Systems. PhD thesis, Indiana University (2006) Available at <http://www.cs.indiana.edu/~lepik/phd.html>.

Program Slicing and Middle-Out Reasoning for Error Location and Repair

Louise A. Dennis^{1,2}

*School of Computer Science and Information Technology
University of Nottingham, Nottingham, UK*

Abstract

This paper describes a proof-based approach to the location and repair of errors in functional programs. The approach is based on the use of program slicing to locate errors and middle-out reasoning to repair them.

An implementation in the *λ Clam* proof planning system is described with some preliminary results.

Key words: Proof-Directed Debugging, Proof Planning,
Middle-Out Reasoning, Program Slicing

1 Introduction

The integration of debugging and verification procedures in proof-directed debugging applications is a comparatively recent and little explored field. This paper outlines a methodology for proof-directed debugging and repair based on program slicing and middle-out reasoning.

The approach treats functional programs as sets of rewrite rules. We refer to subsets of these as *program slices*. We associate a program slice with each branch of a proof tree, representing the rewrite rules involved in that branch. It is then possible to compare the program slices from successful and unsuccessful proof branches in order to select a candidate for a repair attempt.

The repair is conducted by a technique known as *middle-out reasoning*. Middle-out reasoning is an approach to deductive synthesis problems which postpones choices about key parts of a theorem or proof for as long as possible. This is typically performed by replacing some part of the theorem goal with a

¹ Research funded by EPSRC grant GR/S01771/01 and Nottingham NLF grant 3051. My thanks to three anonymous referees whose comments greatly improved the paper.

² Email: lad@cs.nott.ac.uk

higher-order meta-variable. This meta-variable is then gradually instantiated during the course of the proof until the appropriate witness term is synthesised.

This paper shows how, once an incorrect rewrite rule has been detected by program slicing, its right hand side can be replaced by a meta-variable and the proof restarted. The new RHS is then instantiated by middle-out reasoning.

The paper is organised as follows: §2 gives background on program slicing; §3 discusses the the proof planning paradigm and middle-out reasoning; §4 discusses the proposed methodology for error location and repair in detail while §5 discusses an implementation; §6 presents some preliminary results; §7 discusses further work and §8 surveys related techniques.

2 Program Slicing

Program slicing is a technique from debugging which is used to locate errors within programs. In debugging applications program slicing works on an execution trace of the program. Traditional program slicing [19] identifies a variable of interest at some point in a program (the *slicing criterion*) and then extracts a fragment of the program (a *program slice*) containing, for instance, all those statements upon which the value of the variable at that point depended. Program slicing techniques for imperative languages use graph-based representations of programs with statements represented as nodes and a program slice as a set of nodes.

The HAT debugging tool [3] applies program slicing to Haskell programs in order to direct a user’s attention to relevant parts of the program’s source. HAT constructs an *evaluation dependency tree* showing the computation process of the program on some input. The nodes in this tree are equations indicating the reduction of a redex to a value. Implicitly the edges of this tree are the cases of the program’s functional definition used to perform the reductions. HAT uses a polling system based on superimposing correct and incorrect branches of evaluation dependency trees to generate heuristic scores for the cases in order to identify errors [5]. This paper proposes to use a similar process except with branches in a proof tree used to generate appropriate slices rather than branches in a debugging trace.

3 Proof Planning

This paper uses proof planning as a framework within which the automatic detection of erroneous rewrite rules by program slicing and their repair using middle-out reasoning can be conducted.

Proof planning [1] is an Artificial Intelligence based technique for the automation of proof. *Proof methods* are the main planning operators used by proof planning systems. Proof method application is governed by preconditions and by a *proof strategy* which, among other things, restricts the methods available at any point in a proof.

A proof strategy provides a guide to which proof methods should be applicable. Knowing which method is expected to apply gives additional information should the system fail to apply it. *Proof critics* [10] can then be employed to analyse the reasons for failure and propose alternative choices. Critics are expressed in terms of preconditions and patches. The preconditions examine the reasons why the method has failed to apply. Traditionally the proposed patch suggests some change to the proof tree or strategy. In this paper we propose extending critics so they may also propose a change to the theorem.

Proof planning also employs the concept of a *proof context*. The proof context is a repository which stores additional heuristic information. The context may be global to an entire plan or local to goals.

3.1 Middle-Out Reasoning

The use of meta-variables to delay commitment in proof search is commonplace (eg. it is used in Lego and Isabelle [14,16]). This form of middle out reasoning has been widely investigated in proof planning. In particular it has been used in synthesis proofs of the form $\exists x.P(x)$ to generate a witnesses for the existentially quantified variable [9,12] and in work with critics to synthesise new lemmas and generalisations [10]. The instantiation and unification of such variables is naturally higher order. The central idea is that unknown parts of the proof goal (eg. the witness term in a synthesis proof) are replaced by a meta-variable which is instantiated as needed during the proof.

The introduction of meta-variables into a proof process increases the search space involved and it becomes necessary to control their instantiation carefully. The rippling heuristic (a form of rewriting constrained to be terminating by meta-logical annotations) [2,18] is used to control this instantiation in the step cases. After every rewriting step an attempt is made to instantiate all meta-variables appearing in the theorem goal. These meta-variables appear in our situation as functions applied to a list of candidate arguments. Instantiation attempts investigate their projection onto one of their arguments (eg. F appearing as $F(x, y, z)$ is instantiated in turn as $\lambda x \lambda y \lambda z. x$, $\lambda x \lambda y \lambda z. y$ and $\lambda x \lambda y \lambda z. z$). Candidate projections are filtered through a simple counter-example finder to exclude false instantiations.

4 A Methodology for Error Location and Repair

The central idea in this paper is that program slicing in the style of the HAT system can be used within a proof context to select a rewrite rule for modification. A proof critic can use a comparison of the program slices for each proof branch to nominate a rewrite rule for repair and to restart the proof with a new version of this rewrite rule which contains meta-variables. The new rewrite rule is then instantiated by middle-out reasoning in a re-run of the proof. The use of proof planning implicitly assumes that the user has

sufficient prior knowledge about the proof structure to provide an appropriate strategy. The proof strategy for induction used in this work has been widely studied and can be considered of general applicability in many situations.

4.1 *Adapting Program Slicing to Verification*

We treat the functional cases of a program definition as nodes in program slices. The slicing criterion is the nodes that have been used in a branch of a proof. If we think of the functional cases as rewrite rules then the slicing criterion creates a program slice for each branch of the proof tree consisting only of those rewrite rules used in the derivation of that branch.

In order for such a system to work reasonably within a verification context a user is expected to limit the set of rewrite rules which may appear in program slices. This is done by nominating some set of rules as “suspect”. This prevents every definition and theorem in the system appearing in program slices, making the choice of rule for repair more constrained. It is assumed that the pre-existing definitions and theorems of the system within which a verification takes place are correct and it is only those elements introduced by the user³ which are available for correction.

4.2 *Adapting Critics and Middle-Out Reasoning for Program Repair*

Our previous work [6,8] identified two typical places in which program errors become “obvious” to a human during a proof attempt. These are when a false goal is derived (typically during symbolic evaluation) or when the induction hypothesis can not be used (referred to in the proof planning literature as *fertilisation*). Therefore new critics can be attached to these two methods. The first only fires if a goal has been reached which contradicts an axiom while the second fires whenever fertilisation fails. These critics inspect the program slice information available for the whole proof tree to choose a particular rewrite rule for correction. We assume this rewrite rule has been derived from the case breakdown of a functional program. In general the pattern on the LHS of such rules are clear and errors are generally picked up by static analysis. Therefore we assume the LHS is correct and that it is the RHS that is incorrect. The RHS for a new version the rule is constructed as an application of a meta-variable to any variables appearing on the LHS and also to an additional recursive argument to represent the potential recursive structure of the proof.

The proof is then restarted with this new rule available instead of the presumed incorrect version.

³ which could potentially include elements of the specification as well as the program.

5 An Implementation of Error Location and Repair

A prototype system which uses program slicing and middle-out reasoning to locate and repair erroneous rewrite rules has been implemented in $\lambda Clam$ [17] – a proof planning system written in $\lambda Prolog$. It uses a modification of $\lambda Clam$'s proof strategy for induction enhanced by two new critics.

$\lambda Clam$ works by using depth-first planning with proof methods and critics. Each node in the search tree is a subgoal⁴. The planner checks the preconditions for the available proof methods and critics at each node and applies those whose preconditions succeed to, in the case of methods, create the child nodes or, in the case of critics, modify the current proof (or, in this new extension, theorem) and proof strategy. In general proof strategies are constructed so that critics are attempted by the depth-first search only after all the available methods have failed. $\lambda Clam$ associates a heuristic proof context with individual goals.

$\lambda Clam$'s proof strategy for induction is shown in figure 1. The diagram

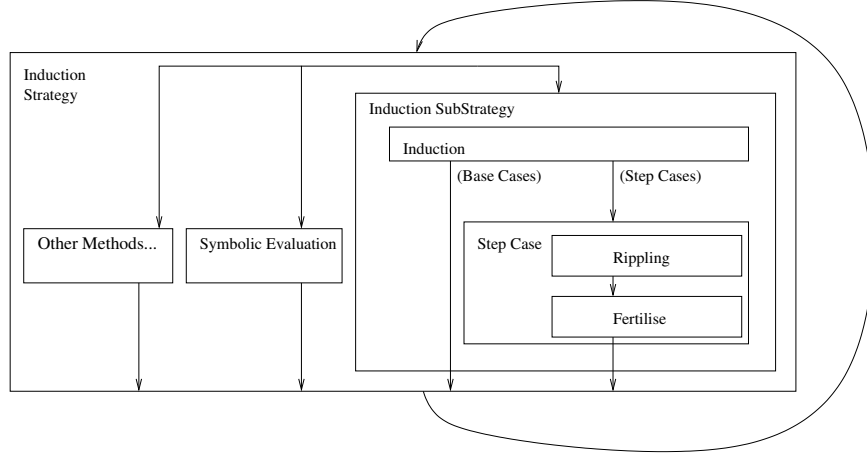


Fig. 1. The Proof Strategy for Induction

shows a top level repeat which attempts a disjunction of methods. These include tautology checking, generalisation of common subterms and also symbolic evaluation (rewriting) and the induction strategy. Within the induction strategy, the induction method chooses an induction scheme and produces subgoals for base and step cases. The top level strategy is re-applied to the base cases. The step cases are handled using rippling and then fertilisation which uses the induction hypothesis to simplify the conclusion further. The results are then passed out to the top level strategy again if necessary. The process terminates when all subgoals have been reduced to *true*.

⁴ in general this means that the search tree can be considered to represent the proof tree although a separate global proof tree structure built up by the search process is also maintained which can be inspected and potentially modified by proof critics.

5.1 Examples

The discussion of this implementation is centered around two examples both derived from the list append function and a proof of its associativity. These are artificial examples based on theorems $\lambda Clam$ is known to be able to prove. The errors are based on errors found in novice ML programs [8] to which we ultimately intend to apply the system (see §7). We use here the $\lambda Prolog$ convention where variables start with an upper case letter and constants with a lower case letter. The faulty definitions are:

Erroneous Basis Case (app1)

definition app11: app1([], Z) = [0].

definition app12: app1(H::T, Z) = H::(app1(T, Z)).

Erroneous Recursive Case (app2)

definition app21: app2([], Z) = Z.

definition app22: app2(H::T, Z) = H::(H::app2(T, Z)).

5.2 Extensions to the Proof Strategy for Induction

The preceding analysis requires that two extensions be made to the proof strategy for induction. Firstly a mechanism is required for constructing program slices and associating these with proof branches (§5.2.1). Then new critics need to be introduced (§5.2.2) to analyse these slices; choose a rewrite rule for correction; construct a new version of this rule containing meta-variables (§5.2.3) and then restart the proof. The subsequent process of middle-out reasoning was already implemented in $\lambda Clam$ and is based on the work of Ireland and Bundy [10] but was trivially extended to symbolic evaluation as well as rippling.

5.2.1 A Data Structure for Proof Branch Based Program Slices

The first extension required was a mechanism for constructing program slices associated with proof branches. This was achieved by introducing a new data structure into the local proof context associated with individual nodes in the proof tree. The basic data structure is a tuple relating a syntax constant (the suspect program) with a list of tracking information for the rewrite rules associated with that constant.

The tracking information is a further tuple of the name of the rewrite rule, a *good/bad tree*, a position in that tree, and a flag showing whether the rule has been used in this branch of the proof tree. The good/bad tree is a standard tree datatype representing the current proof branching structure, whose leaves are labelled either good, bad or unused.

Consider a proof by induction. Let us suppose that we are interested in constructing program slices for our app2 program above. There are two

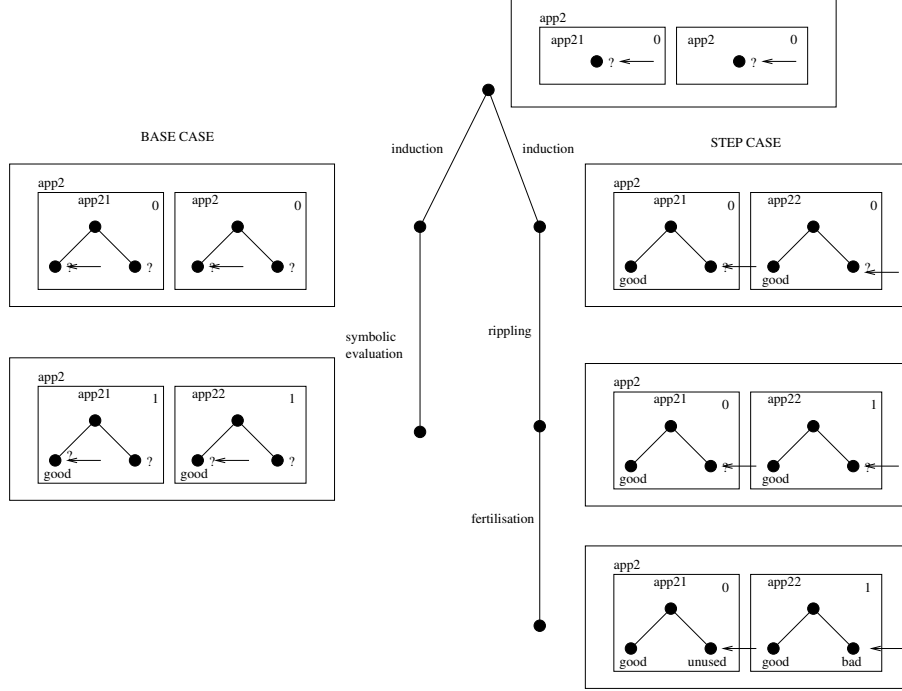


Fig. 2. Constructing Program Slices as a proof progresses

rewrite rules associated with this program, `app21` and `app22`. The good/bad trees for these rules are instantiated dynamically as a proof progresses. When a proof branch is successfully closed the relevant branch in the good/bad tree can be instantiated with a `good` or `unused` leaf node. Where a proof critic detects a problem with a proof branch then the good/bad tree can be instantiated with a `bad` or `unused` leaf node.

Figure 2 shows a schematic version of this new contextual information and how it is developed as a simple induction proof progresses. The figure shows the proof tree with nodes representing subgoals and edges labelled with the proof method that generates the child goals. We've omitted the goals from the node labels but have shown the proof context. The context consists of program slice information about `app2` consisting of individual information about `app21` and `app22`. At the start (the top node in the proof tree) of the proof nothing is known about the proof structure so the good/bad tree associated with both rewrite rules is a single node labelled with a question mark. There is a pointer to the top node of this tree, indicating where this proof node is in that structure. Neither rule has been used yet so the flags are set to 0. Induction is applied and the proof tree splits into a base case and a step case. The new context information for `app2` at these nodes has been extended. The good/bad tree now shows two branches. The pointers in the base case point to the left hand branch while those in the step case point to the right hand branch. Let us assume the base case is attempted first and concludes successfully having used both rewrite rules during symbolic

evaluation. The leaf nodes for the left hand branch are therefore instantiated to `good` for both rules. The step case is then attempted and uses only `app22` during rippling. The result is judged to be incorrect at the fertilisation stage and so the right hand branches are instantiated `bad` for `app22` and unused for `app21`.

In *λClam* the dynamic instantiation of the good/bad trees as the proof progresses is handled using prolog variables on the nodes in the tree structures. This should not be confused with the later use of middle-out reasoning where variables are used to represent unknowns in goals.

The program slices associated with each branch of the proof structure can thus be deduced from the final good/bad trees. So the program slice associated with the base case branch is `[app21, app22]` while that associated with the step case branch is `[app22]`.

5.2.2 Using Program Slices in Critics

As discussed in §4.2 two new critics were introduced. The first fires if symbolic evaluation produces a goal which contradicts an axiom while the second fires whenever fertilisation fails.

These critics compute a score for each rule from its good/bad tree. This score is a tuple of the number of bad nodes and the number of good nodes in the tree. These scores are then ordered by \succ where

$$(b_a, g_a) \succ (b_b, g_b) \iff (b_a > b_b) \vee ((b_a = b_b) \wedge (g_a < g_b))$$

and $<$ and $>$ represent the standard order relation on natural numbers. The rule with the highest score is selected. So a rule which has 2 bad nodes in its good/bad tree will be chosen over one which has only 1 bad node. However when comparing two rules both with 2 bad nodes then one with only 1 good node will be chosen over one with 2 good nodes.

5.2.3 Middle-Out Reasoning for Rewrite Rule Repair

In order to use middle-out reasoning it was necessary to adapt the old, erroneous rules to contain meta-variables replacing the “bad bits”. It is presumed that the case structure on the LHS of these rules is correct so these are left unchanged. The RHSs are then constructed as an application of a meta(prolog)-variable to the variables appearing on the LHS with an additional recursive argument including further meta-variables to represent a potential recursive structure for the new rule (i.e. $\forall x_1. \text{app1}([], x_1) = [0]$ becomes $\forall x_1. \text{app1}([], x_1) = F(\text{app1}(G(x_1), H(x_1)), x_1)$).

The *λClam* proof context was also extended to maintain a list of “banned” rules. Rules nominated for revision by the critics can therefore be added to the banned list. Once this is done the proof is restarted.

We illustrate the middle-out reasoning process with two examples.

5.2.4 Incorrect Basis Case

Consider our example of an incorrect base case. $\lambda Clam$ synthesises a new candidate rule, $\forall x_1. \text{app1}(\square, x_1) = F(\text{app1}(G(x_1), H(x_1)), x_1)$, bans the previous rule, app1 , and restarts the proof. This proceeds by induction, the step case is discharged using the new rule version. The base case,

$\vdash \forall x_1. \forall x_2. \text{app1}(\text{app1}(\square, x_1), x_2) = \text{app1}(\square, (\text{app1}(x_1, x_2)))$,
is rewritten using the new version to

$$\vdash \forall x_1. \forall x_2. \text{app1}(F(\text{app1}(G(x_1), H(x_1)), x_1), x_2) = \text{app1}(\square, (\text{app1}(x_1, x_2))).$$

$\lambda Clam$ immediately tries to find a projection for F , G and H suggesting

$$\forall x_1. \text{app1}(\square, x_1) = (\lambda x_2. \lambda x_3. x_3) (\text{app1}(\lambda x_4. x_4(x_1), \lambda x_4. x_4(x_1)), x_1)$$

and the new rule beta-reduces to

$$\forall x_1. \text{app1}(\square, x_1) = x_1.$$

This makes the goal

$$\vdash \forall x_1. \forall x_2. \text{app1}(x_1, x_2) = \text{app1}(\square, (\text{app1}(x_1, x_2))).$$

No counter-examples are found and so the instantiation is accepted. Further rewriting then successfully finishes the proof.

5.2.5 Incorrect Recursive Case

In the example of an incorrect recursive case $\lambda Clam$ suggests the replacement rule

$$\forall x_1. \forall x_2. \forall x_3. \text{app2}(x_3 :: x_2, x_1) = F(\text{app2}(G(x_3, x_2, x_1), H(x_3, x_2, x_1)), x_3, x_2, x_1).$$

The proof is restarted and induction applied giving the step case

$$\begin{aligned} \forall x_1. \forall x_2. \text{app2}(\text{app2}(t, x_1), x_2) &= \text{app2}(t, \text{app2}(x_1, x_2)) \\ \vdash \forall x_1. \forall x_2. \text{app2}(\text{app2}(h :: t, x_1), x_2) &= \text{app2}(h :: t, \text{app2}(x_1, x_2)). \end{aligned}$$

Rippling rewrites this with the new rule version. G and H are immediately instantiated to $\lambda W_1. \lambda W_2. \lambda W_3. W_2$ and $\lambda W_1. \lambda W_2. \lambda W_3. W_3$ respectively to give the goal:

$$\begin{aligned} \forall x_3. \forall x_4. \text{app2}(\text{app2}(t, x_3), x_4) &= \text{app2}(t, \text{app2}(x_3, x_4)) \\ \vdash \forall x_1. \forall x_2. \text{app2}(F(\text{app2}(t, x_1), h, t, x_1), x_2) &= \text{app2}(h :: t, \text{app2}(x_1, x_2)). \end{aligned}$$

There isn't room in this paper for a full discussion of the rippling heuristic. Briefly, it operates in two modes – outward and inward – and it attempts outward first (shown above). Outward rippling attempts to move differences between the induction hypothesis and conclusion towards the root of the term tree of the conclusion. In this case before the sub-expression $\text{app2}(h :: t, x_1)$ is rewritten the difference between it and the induction hypothesis is $h ::$ which

appears under `app2`. If G and H are not instantiated then not only is a new difference created “higher up” the tree (F) but there remains a difference on the first argument of `app2` (G), and a new one has appeared on the second argument (H). In order to maintain differences in the “right places” for outward rippling, instantiation of G and H is forced. If rippling outward is unsuccessful then rippling inward is attempted in which case such differences would be allowed to remain under certain conditions.

An attempt is made to find a projection for F but in this case no suggestions get past the counter-example finder. The new rule is again used to rewrite the term. In this case it instantiates F to `::` in order for the sub-expression `app2 (F (app2 (t, x1), h, t, x1), x2)` to match the LHS of the rule. However there remains a choice of arguments for `::` from `app2 (t, x1)`, h , t and x_1 . Once again the rippling heuristic helps. Rippling insists that the induction hypothesis is embedded in the induction conclusion. This forces the selection of the recursive argument `app2 (t, x1)`. Type checking forces the selection of the other argument instantiating the new rule to:

$$\forall x_1. \forall x_2. \forall x_3. \text{app2}(x_3 :: x_2, x_1) = x_3 :: \text{app2}(x_2, x_1)$$

thus the goal becomes:

$$\begin{aligned} &\forall x_3. \forall x_4. \text{app2}(\text{app2}(t, x_3), x_4) = \text{app2}(t, \text{app2}(x_3, x_4)) \\ &\vdash h :: \text{app2}(\text{app2}(t, x_1), x_2) = \text{app2}(h :: t, \text{app2}(x_1, x_2)) \end{aligned}$$

The rest of the proof proceeds in a straightforward fashion.

6 Results

The following tables show the performance of the current system in four scenarios. It is being tested on standard simple theorems from the *λClam* benchmarks. Details of the theorems can be found in appendix A. In all cases the user has nominated the definitions of plus, times and exponentiation as “suspect”. Theorems are only shown in a table where they were actually false given the error.

Scenario 1: $0 + X = 1$ (rule plus1)

Theorem	Rule Chosen	Instantiation	Notes
simple	plus1	$0 + X = X$	
assp	plus1	$0 + X = X$	
comp	plus1	$0 + X = X$	
plus2right	plus1	$0 + X = X$	
zeroplus	-	-	loops
comm	times1	$0 * X = X$	

Theorem	Rule Chosen	Instantiation	Notes
dist	times1	$0 * X = X$	
zerotimes	-	-	loops
times2right	plus1	$0 + X = X$	
expplus	exp1	-	heap overflow

Scenario 2: $s(Y) + X = s(s(Y))$ (rule plus2)

Theorem	Rule Chosen	Instantiation	Notes
assp	plus2	$s(Y) + X = X + Y$	heap overflow
comp	plus1	-	heap overflow
plus1lem	plus2	$s(Y) + X = s(X + Y)$	
plusxx	plus2	$s(Y) + X = s(X + Y)$	
comm	plus1	-	heap overflow
distr	plus2	$s(Y) + X = s(X + Y)$	
assm	plus2	$s(Y) + X = s(X + Y)$	heap overflow
expplus	exp2	$X^{s(Y)} = X^Y$	

Scenario 3: $0 * X = X + X$ (rule plus3)

Theorem	Rule Chosen	Instantiation	Notes
comm	times1	-	can't synthesise a constant
dist	times1	-	can't synthesise a constant
distr	times1	-	can't synthesise a constant
assm	times1	-	can't synthesise a constant
zerotimes	-	-	loops
times2right	times1	-	can't synthesise a constant
expplus	times1	-	can't synthesise a constant

Scenario 4: $s(Y) * X = (Y * X) + Y$ (rule plus4)

Theorem	Rule Chosen	Instantiation	Notes
comm	times1	-	heap overflow
dist	plus1	-	heap overflow

Theorem	Rule Chosen	Instantiation	Notes
distr	plus1	-	heap overflow
assm	plus1	-	heap overflow
zerotimes	-	-	loops
times2right	plus2	-	heap overflow

Out of 31 test runs the offending rewrite rule was correctly identified in 16 cases. In 3 further cases although a different rewrite rule was identified the system nevertheless managed to manufacture a patch that allowed the theorem to go through. Of the 16 correctly identified rules the system managed to synthesise the correct patch in 12 cases and complete the proof in 11 of these.

6.1 Discussion

The above results are, to an extent, discouraging. However analysis of the problems and some pen-and-paper working suggests several approaches.

Firstly, it is clear that the existing middle-out reasoning technology in *λClam* is incapable of synthesising a constant when a function has been implicitly suggested. This should not be too hard to correct. Secondly, choosing essentially random theorems in order to verify a program is, unsurprisingly, not terribly accurate. Furthermore, pen-and-paper working suggests the results are likely to improve dramatically if several proofs (or even all branches in one proof – since *λClam* works depth-first the critics are fired on the first “bad” proof branch encountered rather than inspecting the entire tree) were checked before a rule were selected for revision. This is technically difficult in *λClam* since Teyjus λProlog does not perform garbage collection and is inclined to run out of heap when attempting two proofs in a row.

The experiment also suggests that potential rule instantiations should be screened to disallow trivial suggestions (eg. the instantiation of plus2 to $s(Y) + X = X + Y$ in the attempt to prove `assp` in scenario 2).

Lastly a few bugs in *λClam* were revealed causing unnecessary looping and heap overflows.

7 Further Work

The experiment reported above reveals some improvements and bug-fixes required in the prototype system. This then needs to be evaluated on a wider range of examples. Our target data set is the examples reported in [8]. These contain several erroneous programs where either the error or the necessary repair does not strictly conform to the rippling heuristic. This would mean that the pre-existing *λClam* middle-out reasoning techniques would be unable to synthesise such rules. Investigation of the ways in which the heuristic can

be relaxed and/or the technique extended is therefore necessary. An obvious avenue to explore is the best-first rippling of Johansson [11] which allows the preconditions of the ripple method to be treated as soft rather than hard constraints on the process.

It is also desirable to work at a finer level than whole rewrite rules when generating program slices. Program slicing applied to functional programming typically uses function application/redexes as the nodes in program slices. Working at this level would have the advantage of allowing only sub-expressions of the RHS of rules to be replaced which would again reduce the search space required for correct instantiations.

8 Related Work

Monroy [15] has previously used middle-out reasoning in the context of proof planning for repairing theorems. His work attempts to synthesize a *corrective predicate* in the course of proof. This predicate is represented by a meta-variable, P , such that $P \rightarrow G$ where G is the original (non)theorem. P is instantiated during the course of a proof planning attempt. In the presence of incorrect definitions this system may synthesise the predicate `False` since it has no mechanism for excluding rewrite rules, an example of this is shown in [7]. At best, when faced with an incorrect rule, Monroy’s system synthesises a domain restriction limiting the theorem to those cases which are correct rather than altering the rule itself.

Colton and Pease [4] use techniques inspired by Imre Lakatos’s ‘Proofs and Refutations’ [13] to modify theorems. As with Monroy’s system the techniques focus on restricting the domain of the theorem by barring particular counter-examples (eg. “All primes *except 2* are odd”); identifying some property common to all counter-examples; or identifying some property common to all the examples. The revision process is driven by the discovery of counter-examples rather than an analysis of proof failure.

Both these approaches assume that it is the theorem statement that is in error not the definitions of the constants appearing in the statement. This latter scenario is more likely to be the case in proof-directed debugging.

9 Conclusion

This paper has put forward a proof-based methodology for locating and repairing errors in programs based upon program slicing (for locating errors) and middle-out reasoning (for repairing these errors). The approach is intended for use in proof-directed debugging applications.

A prototype implementation has been built and run on some simple examples. This demonstrates that the methodology can, in principle at least, correctly locate errors and that it is possible to achieve sufficient control over the middle-out reasoning process to synthesise appropriate patches.

References

- [1] A. Bundy. A science of reasoning. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 178–198. MIT Press, 1991.
- [2] A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-Level Guidance for Mathematical Reasoning*, volume 56 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2005.
- [3] O. Chitil. Source-based trace exploration. In C. Grelck, F. Huch, G. J. Michaelson, and P. Trinder, editors, *IFL 2004*, LNCS 3474, pages 126–141. Springer, March 2005.
- [4] S. Colton and A. Pease. Lakatos-style methods in automated reasoning. In *Proceedings of the IJCAI’03 workshop on Agents and Reasoning, 2003.*, 2003.
- [5] T. Davie and O. Chitil. One right does make a wrong. In H. Nilsson and M. van Eekelen, editors, *TFP 2006*, pages 27–40, 2006.
- [6] L. A. Dennis. The use of proof planning critics to diagnose errors in the base cases of recursive programs. In W. Ahrendt, P. Baumgartner, and H. de Nivelle, editors, *IJCAR 2004 Workshop on Disproving: Non-Theorems, Non-Validity, Non-Provability*, pages 47–58, 2004.
- [7] L. A. Dennis, R. Monroy, and P. Nogueira. Proof-directed debugging and repair. In H. Nilsson and M. van Eekelen, editors, *TFP 2006*, pages 131–140, 2006.
- [8] L. A. Dennis and P. Nogueira. What can be learned from failed proofs of non-theorems? In J. Hurd, E. Smith, and A. Darbari, editors, *TPHOLs 2005: Emerging Trends Proceedings*, pages 45–58, 2005. Technical Report PRG-RP-05-2, Oxford University Computer Laboratory.
- [9] J. Hesketh, A. Bundy, and A. Smaill. Using middle-out reasoning to control the synthesis of tail-recursive programs. In D. Kapur, editor, *CADE 11*, volume 607 of *LNAI*, pages 310–324, 1992.
- [10] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996.
- [11] M. Johansson. A best-first planner for rippling. 4th Year Project Report, Division of Informatics, University of Edinburgh, 2005.
- [12] D. Lacey, J. D. C. Richardson, and A. Smaill. Logic program synthesis in a higher order setting. In J. Lloyd, V. Dahl, U. Furbach, and P. J. Stuckey, editors, *CL 2000*, LNCS 1861. Springer, 2000.
- [13] I. Lakatos. *Proofs and Refutations*. Cambridge University Press, 1976.
- [14] Z. Luo and R. Pollack. Lego proof development system: User’s manual. Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh, May 1992. See also <http://www.dcs.ed.ac.uk/home/lego>.

- [15] R. Monroy. Predicate synthesis for correcting faulty conjectures: The proof planning paradigm. *Automated Software Engineering*, 10(3):247–269, 2003.
- [16] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
- [17] J. D. C. Richardson, A. Smaill, and I. Green. System description: Proof planning in higher-order logic with lambda-clam. In C. Kirchner and H. Kirchner, editors, *CADE-15*, LNCS 1421, pages 129–133. Springer, 1998.
- [18] A. Smaill and I. Green. Higher-order annotated terms for proof search. In J. von Wright, J. Grundy, and J. Harrison, editors, *TPHOLs’96*, LNCS 1275, pages 399–414. Springer-Verlag, 1996.
- [19] M. D. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

A Test Theorems

Theorem Name	Theorem Statement
simple	$0 = 0 + (0 + 0)$
assp	$\forall x, y, z. (x + y) + z = x + (y + z)$
comp	$\forall x, y. x + y = y + x$
plus2right	$\forall x, y. x + s(y) = s(x + y)$
plus1lem	$\forall x. x + s(0) = s(x)$
plusxx	$\forall x. s(x) + x = s(x + x)$
zeroplus	$\forall x, y. x = 0 \wedge y = 0 \rightarrow x + y = 0$
comm	$\forall x, y. x * y = y * x$
dist	$\forall x, y, z. x * (y + z) = (x * y) + (x * z)$
distr	$\forall x, y, z. (x + y) * z = (x * z) + (y * z)$
assm	$\forall x, y, z. (x * y) * z = x * (y * z)$
zerotimes	$\forall x, y. x = 0 \vee y = 0 \rightarrow x + y = 0$
times2right	$\forall x, y. x * s(y) = x + (x * y)$
expplus	$\forall x, y, z. x^{y+z} = x^y * x^z$

Satisfiability Solving for Program Verification: towards the Efficient Combination of Automated Theorem Provers and Satisfiability Modulo Theory Tools

Silvio Ranise^{1,2}

¹ LORIA-INRIA-Lorraine, Nancy, France

² Department of Computer Science, Università degli Studi di Milano, Italy
`ranise@loria.fr` & `http://www.loria.fr/~ranise`

Abstract. Many approaches to software verification require to check the satisfiability of first-order formulae. For such techniques, it is of crucial importance to have satisfiability solvers which are both *scalable*, *predictable* and *flexible*. We describe our approach to build solvers satisfying such requirements by combining equational theorem proving, Boolean solving, Arithmetic reasoning, and some transformations on the proof obligations. The viability of the approach has been shown by successfully discharging proof obligations arising in a variety of verification problems with an implementation of the proposed techniques, in a system called **haRVey**. Finally, we sketch how *model building* can be integrated in the proposed approach in order to enhance counter-example generation or to generalize the Nelson-Oppen combination schema.

1 Introduction

Many approaches to software verification, ranging from applications of Hoare logic (see, e.g., [15]) to software model checking (see e.g., [4]) and, more recently, to program analysis [22], require to discharge some proof obligations, i.e. checking that some formula, usually, of first-order logic with equality (FOLE) is satisfiable in a given theory modeling the user-defined data types of the software system under scrutiny, the memory model used by the programming language, its type system, and so on. For such verification techniques, it is of crucial importance to have *satisfiability solvers* which are both *scalable*, *predictable*, and *expressive*, i.e. capable of automatically discharging the largest possible number of proof obligations coming from the widest range of verification problems. Indeed, this task is far from simple.

In the talk, we will describe our approach to build satisfiability solvers for software verification by tackling the above challenges using a careful integration of the following three techniques:

- Equational theorem proving which allows us to provide support for a wide range of theories formalizing data structures and coping with quantifiers which may occur in some proof obligations (for expressiveness);

- Boolean solving to handle the control flow of software systems (for scalability);
- Arithmetic reasoning combined with equational provers *à la* Nelson-Oppen [23] (for expressiveness);
- Heuristics to reduce the search space of the satisfiability problem (for predictability).

Finally, we will see how to integrate model building (see, e.g., [8]) with the techniques introduced above in order to

- compute counter-examples for undischarged proof obligations or
- refine the Nelson-Oppen combination schema [23] for checking the satisfiability in unions of theories.

2 An Example

To illustrate some of the most important problems related to the implementation of satisfiability solvers for software verification, we consider a set of proof obligations which has been considered difficult in [13].

Consider a program *Prg* which manipulates a square matrix *r* of dimension *n*. Assume that *Prg* updates only the elements on the diagonal of *r*. We would like to check that if the matrix *r* is symmetric before the execution of *Prg*, then the matrix *r'* (obtained by the execution of *Prg* with *r* as input) is also symmetric.

To reduce this to a satisfiability problem in FOLE, we need to formalize the bidimensional matrix data structure. For this, we use the theory of arrays (see, e.g., [2]) which is axiomatized by:

$$\forall A, I, E. \text{wr}(A, I, E)[I] = E \quad (1)$$

$$\forall A, I, J, E. I \neq J \Rightarrow \text{wr}(A, I, E)[J] = A[J], \quad (2)$$

where $a[i]$ returns the elements stored at index *i* in the array *a* and $\text{wr}(a, i, e)$ returns an array *b* such that $b[k] = e$ if $k = i$ and $b[k] = a[k]$ for every $k \neq i$. To formalize bidimensional matrices, it is sufficient to have pairs, which are axiomatized as follows:

$$\forall I, J. \pi_1(\langle I, J \rangle) = I \quad (3)$$

$$\forall I, J. \pi_2(\langle I, J \rangle) = J \quad (4)$$

$$\forall P. \langle \pi_1(P), \pi_2(P) \rangle = P \quad (5)$$

where $\langle i, j \rangle$ takes indexes *i* and *j* and returns their pair, $\pi_1(P)$ and $\pi_2(P)$ return the first and second element of the pair, respectively. We are now in the position to define what it means for a matrix *m* of dimension *n* to be symmetric:

$$\forall r, n. \text{symm}(r, n - 1) \Leftrightarrow \forall i, j. (\text{inrange}(0, i, j, n - 1) \Rightarrow r[\langle i, j \rangle] = r[\langle j, i \rangle]), \quad (6)$$

where $\text{inrange}(0, i, j, n - 1)$ is an abbreviation for $0 \leq i, j \leq n - 1$ and \leq is the usual ‘less-than-or-equal’ relation over integers. For $n = 2$, the formula expressing the fact the program *Prg* returns a symmetric matrix *r'* if it takes as input

a symmetric matrix r and it changes only the elements on the diagonal of r is the following:

$$\text{symm}(r, 1) \wedge r' = \text{wr}(\text{wr}(r, \langle 0, 0 \rangle, e_0), \langle 1, 1 \rangle, e_1) \Rightarrow \text{symm}(r', 1). \quad (7)$$

To show the validity of (7), we can reason by refutation and check the unsatisfiability of its negation together with the axioms (1), (2), (3), (4), (5), and definition (6) of **symm**. To solve this satisfiability problem, we can take one of the many state-of-the-art equational theorem provers (such as SPASS [29] or E [28]) and type it in. Unfortunately, as already observed in [13], this does not work: already for the proof obligation (7), some of the available theorem provers fail. SPASS succeeds for $n = 2$, but it fails as soon as we consider the same problem for $n = 3$.

2.1 Discussion

By analyzing the behavior of SPASS and E, it is possible to identify two main problems.

Term depth: the depth of the term describing r' , see (7), increases with n .

This gives an exponential explosion of the possible instances of the axioms (1) and (2): this dramatical enlargement of the search space prevents the equational provers to scale up significantly.

Definition expansion: the available first-order provers usually assume the input formulae to be in conjunctive normal form. It is known that this transformation does not allow to unfold definitions as one can do during hand proofs and unnecessarily enlarge the search space (see, e.g., [19]). In our case, the definition of the predicate **symm** generates 6 clauses which allow provers to derive a large number of unnecessary facts which do not contribute to the proof of unsatisfiability of the clause.

Is there other reasoning tools available which would allow us to solve the satisfiability problem stated above? Recently, a new generation of tools based on the integration of SAT solvers and combinations of decision procedures for theories such as those of arrays, pairs, and integers has emerged under the name of Satisfiability Modulo Theory (SMT) tools (see, e.g., [16, 5, 3]). For the problem above, it would be sufficient to use an SMT system featuring a decision procedure for the combination of the theory of arrays, the theory of pairs, and integers. Unfortunately, most SMT tools are capable of handling only quantifier-free formulae and this is not sufficient for the problem above since the negation of (7) is not ground, even if we expand the two occurrences of the definition of **symm**, we obtain the following quantified formula:

$$\begin{aligned} & \forall i, j. (\text{inrange}(0, i, j, 1) \Rightarrow r[\langle i, j \rangle] = r[\langle j, i \rangle]) \\ & \wedge r' = \text{wr}(\text{wr}(r, \langle 0, 0 \rangle, e_0), \langle 1, 1 \rangle, e_1) \\ & \wedge \text{inrange}(0, i_0, j_0, 1) \\ & \wedge r'[\langle i_0, j_0 \rangle] \neq r'[\langle j_0, i_0 \rangle]), \end{aligned} \quad (8)$$

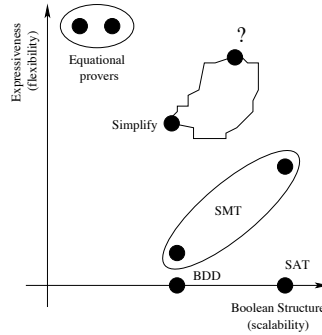


Fig. 1. Satisfiability solvers: Scalability vs. Flexibility

where i_0, j_0 are Skolem constants. An SMT system which supports the handling of quantified formulae is *Simplify* [14], the theorem prover of the ESC/Java program analysis tool. In fact, *Simplify* solves quite easily (under a second) even large instances ($n = 100$) of the satisfiability problem above. Although such a system is quite powerful for the kind of problem we are interested in, it suffers two main limitations with respect to equational provers and the SMT tools mentioned above. First, its heuristic mechanism to instantiate quantifiers is sensitive to the syntactic structure of the formula and often requires the user to provide hints to successfully discharge proof obligations which are relatively easy for equational theorem provers, see [11] for more on this issue. Second, *Simplify* does not scale up to handle large (quantifier-free) formulae with a complex Boolean structure, see [10] for an experimental analysis supporting this claim.

To summarize, Figure 1 depicts the situation of the tools available for satisfiability solvers. There is a sharp trade-off between expressiveness (usually not predictable) and scalability (usually very predictable) with *Simplify* standing in-between but biased to expressiveness rather than scalability. In the following, we describe techniques which allow us to re-use available tools in order to build satisfiability solvers (identified by ‘?’ in the Figure) which offer a better trade-off between expressiveness, flexibility, and scalability.

3 Our Approach

Our starting point is the SMT architecture, which is based on an extension of Boolean satisfiability: a Boolean solver is modified to enumerate Boolean assignments and modified with a procedure capable of checking the satisfiability of sets of literals in a given theory T (see, e.g., [11] for more details). When T is the combination of several theories, a combination schema is used to modularly build a procedure for T out of those for the component theories (see, e.g., [26] for an overview and pointers to the literature).

Equational provers. Instead of using *ad hoc* procedures for selected theories possibly glued together by a combination method, we use an equational theorem

prover such as E. This allows us to retain the scalability of the SMT architecture (where the Boolean structure of the formula is handled by the Boolean solver) while gaining the greater expressiveness of the equational prover. Indeed, we assume that the theory can be finitely axiomatized (as it is the case for several theories of data structures) or a finite approximation is available. Furthermore, the rewriting approach in [2, 1] guarantees that equational provers based on the superposition calculus [24] behave as decision procedures for the satisfiability problem of the theories of many useful data structures (such as lists, arrays, and their combination).

Flattening. Replacing an *ad hoc* procedure with an equational prover does not guarantee to obtain reasonable performances. The crucial problem is the depth of terms (as discussed in Section 2.1). Our solution is to *flatten* atoms, i.e. adding constants to name sub-terms so that each term has depth at most 1. For example, the atom $r' = \text{wr}(\text{wr}(r, \langle 0, 0 \rangle, e_0), \langle 1, 1 \rangle, e_1)$ in (7) would be transformed into $i_0 = \langle 0, 0 \rangle \wedge i_1 = \langle 1, 1 \rangle \wedge r_0 = \text{wr}(r, i_0, e_0) \wedge r' = \text{wr}(r_0, i_1, e_1)$, where i_0, i_1, r_0 are fresh constants. (Notice that flattening preserves satisfiability.) Flattening is also the key to obtain a predictable and (more) scalable behavior of the equational prover. For instance, for the case of the problem considered in Section 2, E scale up to instances with $n = 50$ while, without flattening, was not able to go beyond $n = 5$.

A related problem is the interplay between the Boolean solver and the prover. In fact, a naïve implementation of an SMT tool can always enumerate exponentially many (in the number of atoms in the input formula) Boolean assignments so that the first-order procedure is invoked a huge number of times, thereby degrading performances unacceptably. The standard solution to this problem is to extend the capabilities of the first-order procedure so that, besides deciding satisfiability, it also returns a small unsatisfiable sub-set of the assignment (called the *conflict set*) which can be used, in practice, to dramatically prune the search space of the Boolean solver. Now, many available equational provers (such as E) are capable of computing a proof of the unsatisfiability of an input set of clauses. By analyzing such a proof, it is possible to extract a conflict set which can be used by the Boolean solver (see again [11] for details).

Quantifier instantiation. It is possible to extend the SMT architecture so to handle formulae containing quantifiers. The key idea is to minimize the scope of quantifiers by using well-known techniques of FOLE, to abstract away quantified sub-formulae by using fresh propositional letters, and then add to the theory to be sent to the equational prover a suitable formula defining the newly introduced propositional letters. For example, formula (8) can be transformed to

$$q \wedge r' = \text{wr}(\text{wr}(r, \langle 0, 0 \rangle, e_0), \langle 1, 1 \rangle, e_1) \wedge \\ \text{inrange}(0, i_0, j_0, 1) \wedge r'[\langle i_0, j_0 \rangle] \neq r'[\langle j_0, i_0 \rangle],$$

and $q \Leftrightarrow \forall i, j. (\text{inrange}(0, i, j, 1) \Rightarrow r[\langle i, j \rangle] = r[\langle j, i \rangle])$ is added to axioms (1)-(4).³ Notice that the availability of the equational prover to handle a more complex theory is the key to handle quantifiers, while still exploiting the capability of a Boolean solver to handle large formulae. (For more details, the interested reader is referred to [9].)

Definition expansion. Definitions are an important mechanism to structure specifications. Unfortunately, it is well-known (see, e.g., [19]) that they are problematic for the predictability of equational reasoners since they enlarge dramatically the search space. For example, definition (6) is the reason of the failure to scale up (for $n \geq 2$) of both SPASS and E on the satisfiability problem considered in Section 2. Another problem related to definitions arising in software verification is caused by predicates defined by case-analysis on the values of their arguments. This is so because the equational provers show poor performances to handle the case analysis required by such a kind of definitions. For instance, consider the following alternative definition of the predicate `inrange` introduced in Section 2:

$$\forall i, j. \text{inrange}(0, i, j, n-1) \Leftrightarrow \left(\begin{array}{l} (i = 0 \vee \dots \vee i = n-1) \wedge \\ (j = 0 \vee \dots \vee j = n-1) \end{array} \right) \quad (9)$$

for some fixed value of $n > 1$. If we consider again formula (8), then $n = 2$ and the conjunctive normal form of (9) consists of 6 clauses, which must be added to the theory passed on to the equational prover thereby significantly enlarging the search space. Notice that such clauses essentially encodes a case-split on the values of the pair of indexes i and j : the equational reasoner is therefore responsible for case splitting for the second occurrence of `inrange` in (8), namely `inrange(0, i_0 , j_0 , 1)`. However, the Boolean solver available in the SMT architecture would be much more suitable to perform this task. Thus, if the splitting is performed at the propositional level, the produced arithmetic literals would be directed to the proper decision procedure by the combination schema. Indeed, for larger value of n , the gain in using the available Boolean solver to perform case-splitting would be more important. To overcome such problems and improve predictability, we unfold definitions using some of the heuristics in [25].

Equational provers and arithmetic reasoning. There are decidable theories (e.g., various fragments of Arithmetic) for which the rewriting approach does not apply. Since Arithmetic is ubiquitous in virtually any verification effort, it is crucial to develop efficient methods to combine equational provers and decision procedures for Arithmetic. In [20], using the Nelson-Oppen combination method, where satisfiability procedures cooperate by exchanging entailed (disjunction of) equalities between variables, it is shown that the superposition calculus deduces

³ To check satisfiability, it is sufficient to consider the implication $q \Rightarrow \dots$ and not the biconditional $q \Leftrightarrow \dots$ which would yield a more complex formula for the equational prover (see, e.g., [19] for a discussion of this problem).

sufficiently many such equalities for convex theories (e.g., the theory of equality and the theory of lists) and disjunction of equalities for non-convex theories (e.g., the theory of arrays) to guarantee the completeness of the combination method. It is also shown how to make satisfiability procedures built by superposition both incremental and resettable by using a hierarchic variant of the Nelson-Oppen method. A generalization of these results has been recently given in [21].

Theory reduction. Software verification often need to consider hundreds of axioms, but the proofs are usually shallow and consist of analyzing a large number of cases. Equational provers are known to have quite impressive performances on problems consisting of few axioms and conjectures (usually extracted from mathematical problems) whose proofs are quite deep and contain simple case analyses. Hence, it is not surprising that equational provers do not perform well in the context of software verification (see, e.g., [13]), as they explore too a large search space generated by a large number of axioms, most of which are irrelevant.

In [27], a technique to find out an approximation of the relevant axiom set for the proof of a conjecture is presented. We use an adaptation of such a technique supported by syntactic constructs for structuring the set Ax of axioms into (sub-)theories which may extend (import) previously defined sub-theories. For example, one can group axioms (1)–(2) into a theory of arrays, axioms (3)–(5) into a theory of pairs, and definition (6) can put in a new theory extending both previous theories. For a given conjecture φ , a tailored theory is extracted from a set of axioms Ax by considering the set S of theories interpreting the symbols in φ as well as those in the image of S with respect to the transitive closure of the relation “import.”

3.1 Implementation: **haRVey**

We have implemented the techniques described above in **haRVey**, which is available at the following address: <http://harvey.loria.fr>.

The system has two incarnations. The former (called **haRVey-FOL**) integrates Boolean solvers with an automated theorem prover as described above, see [11, 12]. The latter (called **haRVey-SAT**) integrates Boolean solvers with a combination of decision procedures for the theory of uninterpreted function symbols and Linear Arithmetic based on the Nelson-Oppen schema and techniques to handle quantifiers and lambda-expressions (see [17]). Furthermore, **haRVey-SAT** can produce proofs which can then be independently checked by the proof assistant Isabelle (see [18]).

While **haRVey-FOL** offers a high degree of flexibility and automation for a variety of theories, **haRVey-SAT** is usually faster on problems with simpler background theories and ensures a high degree of certification by its proof checking capability. Along the lines hinted in [20], our current work aims at merging the two incarnations in one system which retains the flexibility and high-degree of automation for expressive theories of **haRVey-FOL** and provides better performances on simpler problems as **haRVey-SAT**.

haRVey has been successfully used to discharge the proof obligations arising in many verification problems as described in [11, 9, 20, 18].

4 Integrating Model Building

We consider two situations in which model building [8] can be used to enhance the techniques described above.

4.1 Counter-example generation

In [9], we presented a technique to prove invariants of model-based specifications in a fragment of set theory. Proof obligations containing set theory constructs were translated to first-order logic with equality augmented with (an extension of) the theory of arrays with extensionality. Besides using **haRVey** to automate the verification of the proof obligations obtained by the translation, [9] also discussed how a sub-formula can be extracted from a failed proof attempt and used by a model finder to build a counter-example. We summarize such an integration below by emphasizing its independence of the particular application.

If an assignment β^B of the Boolean abstraction ϕ^B of the proof obligation ϕ is found satisfiable by **haRVey**, then we can use β as the starting point to build a model for ϕ , i.e. a counter-example for the formula to be proved valid (which is the negation of ϕ). Notice that this is an advantage w.r.t. building a model for the whole formula ϕ since the assignment is usually much smaller.

In preliminary investigations, we have tried to build a model of β by using state-of-the-art model builders (such as MACE⁴ and SEM⁵) without success. The extension of the theory of arrays used in this application generates a search space which is too large to be treated in a reasonable amount of time. In order to overcome these difficulties, we have used CLPS [7], a constraint solver for the theory of Hereditarily Finite Sets with Atoms, which was capable of building all necessary counter-examples. The main drawback was the need to translate β back to a formula of the fragment of set theory because the input language of CLPS is based on set theory. More careful experiments with model builders are necessary: Peltier reported success on some of the problems extracted from our application using more sophisticated techniques (see [8] for details).

4.2 Model Building for Satisfiability Solving

Recently, [6] strengthened the Nelson-Oppen decidability result [23], by showing that it applies to theories over disjoint signatures, whose satisfiability problem, in either finite or infinite models, is decidable. Furthermore, this result covers decision procedures based on equational theorem proving [2], generalizing recent work [1] on combination of theories in the rewrite-based approach to satisfiability.

⁴ <http://www-unix.mcs.anl.gov/AR/mace2/>

⁵ <http://www.cs.uiowa.edu/~hzhang/sem.html>

In fact, in [6], it is observed that state-of-the-art implementations of the superposition calculus, such as SPASS [29] or E [28], fail to terminate on theories admitting only finite models. For example, both SPASS and E were able to handle only the trivial theory with trivial models. Already for the theory admitting models of cardinality at most 2, the provers seem to go on indefinitely (or, better, they do not terminate in a reasonable amount of time) although we experimented with various settings.

Fortunately, this problem can be solved by the following two observations. First, an equational prover may not terminate on instances of the constraint satisfiability problem of the form $T \cup F$, where F is a constraint and T does not admit infinite models, one of the results in [6] ensures that a clause C of particular syntactic form constraining the cardinality of the models will eventually be derived in a finite amount of time. Second, when such a clause C is derived from $T \cup F$, a bound on the cardinality of the domains of any model can be immediately obtained by the cardinal associated to C . It is then possible to use such a bound with a finite model builder to check the satisfiability of the clauses $T \cup F$ in finite models of cardinality up to N .

Acknowledgements

This research would have been impossible without the work and insights of the following people: D. Déharbe, P. Fontaine, S. Ghilardi, E. Nicolini, C. Ringeissen, D.-K. Tran, and D. Zucchelli.

References

1. A. Armando, M. P. Bonacina, S. Ranise, and S. Schulz. On a rewriting approach to satisfiability procedures: extension, combination of theories and an experimental appraisal. In *Proc. of the 5th Int. Workshop on Front. of Comb. Sys. (FroCos'05)*, volume 3717 of *LNCS*, pages 65–80, 2005.
2. A. Armando, S. Ranise, and M. Rusinowitch. A Rewriting Approach to Satisfiability Procedures. *Info. and Comp.*, 183(2):140–164, June 2003.
3. G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions. In *Proc. Int. Conf. on Automated Deduction*, pages 195–210, 2002.
4. T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *SPIN 2001*, volume 2057 of *LNCS*, pages 103–122, 2001.
5. C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proc. 16th Intl. Conf. Comp. Aided Verif.*, 2004.
6. M. P. Bonacina, S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Decidability and Undecidability Results for Nelson-Oppen and Rewrite-Based Decision Procedures. In *In Proc. of IJCAR'06*, 2006. To appear.
7. F. Bouquet, B. Legeard, and F. Peureux. CLPS-B - A Constraint Solver for B. In *Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems, TACAS'02*, volume 2280 of *LNCS*, pages 188–204, 2002.
8. R. Caferra, A. Leitsch, and N. Peltier. *Automated Model Building*. Kluwer Academic Publishers, 2005.

9. J.-F. Couchot, D. Déharbe, A. Giorgetti, and S. Ranise. Scalable automated proving and debugging of set-based specifications. *J. of the Brazilian Computer Society*, 9(2):17–36, November 2003.
10. L. de Moura and H. Ruess. An experimental evaluation of ground decision procedures. In *Proc. of Comp. Aided Verif., CAV'04.*, LNCS, 2004.
11. D. Déharbe and S. Ranise. Light-Weight Theorem Proving for Debugging and Verifying Units of Code. In IEEE Comp. Soc. Press, editor, *Proc. of the Int. Conf. on Software Engineering and Formal Methods (SEFM03)*, 2003.
12. D. Déharbe and S. Ranise. Satisfiability Solving for Software Verification. In *Proc. of IEEE/NASA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation (ISoLA'05)*, 2005.
13. E. Denney, B. Fischer, and J. Schumann. Using automated theorem provers to certify auto-generated aerospace software. In *Proc. of Int. Joint Conf. On Automated Reasoning (IJCAR'04)*, volume 3097 of LNCS, 2004.
14. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A Theorem Prover for Program Checking. Technical Report HPL-2003-148, HP Lab., 2003.
15. J.-C. Filliâtre and C. Marché. Multi-Prover Verification of C Programs. In *6th Int. Conf. on Form. Eng. Methods*, volume 3308 of LNCS, pages 15–29, 2004.
16. J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated Canonization and Solving. In *Proc. of CAV'2001*, volume 2102 of LNCS, pages 246–249, 2001.
17. P. Fontaine. *Techniques for verification of concurrent systems with invariants*. PhD thesis, Institut Montefiore, Universit de Lige, Belgium, September 2004.
18. P. Fontaine, J.-Y. Marion, S. Merz, L. Prensa Nieto, and A. Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In *Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems, TACAS'06*, volume 3920 of LNCS, pages 167–181, 2006.
19. H. Ganzinger and J. Stuber. Superposition with equivalence reasoning and delayed clause normal form transformation. *Inf. Comput.*, 199(1-2):3–23, 2005.
20. H. Kirchner, S. Ranise, C. Ringeissen, and D.-K. Tran. On Superposition-Based Satisfiability Procedures and their Combination. In *Proc. of the 2nd Int. Coll. on Theoretical Aspects of Computing*, volume 3722 of LNCS, pages 594–608, 2005.
21. H. Kirchner, S. Ranise, C. Ringeissen, and D.-K. Tran. Automatic combinability of rewriting-based satisfiability procedures. In *Proc. of LPAR'06*, 2006. To appear.
22. V. Kuncak and M. Rinard. An overview of the Jahob analysis system: Project Goals and Current Status. In *NSF Next Generation Software Workshop*, 2006.
23. Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. on Progr. Lang. and Sys.*, 1(2):245–257, October 1979.
24. R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In A. Robinson and A. Voronkov, editors, *Hand. of Automated Reasoning*. 2001.
25. A. Nonnengart and C. Weidenbach. *Handbook of Automated Reasoning*, chapter Computing Small Clause Normal Forms. Elsevier Science, 2001.
26. S. Ranise, C. Ringeissen, and D.-K. Tran. Nelson-Oppen, Shostak and the Extended Canonizer: A Family Picture with a Newborn. In *First Int. Coll. on Theoretical Aspects of Computing - ICTAC 2004*, LNCS, China, September 2004.
27. W. Reif and G. Schellhorn. *Automated Deduction—A Basis for Applications*, volume 1, chapter Theorem Proving in Large Theories. Kluwer Academic Pub., 1998.
28. S. Schulz. System Abstract: E 0.61. In *Proc. of the 1st IJCAR, Siena*, number 2083 in LNAI, pages 370–375. Springer, 2001.
29. C. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Hand. of Automated Reasoning*. 2001.

A Fast Disprover for \checkmark eriFun

Markus Aderhold, Christoph Walther, Daniel Szallies, and Andreas Schlosser

Fachgebiet Programmiermethodik, Technische Universität Darmstadt, Germany
{aderhold, chr.walther, szallies, schlosser}@pm.tu-darmstadt.de

Abstract. We present a disprover for universal formulas stating conjectures about functional programs. The quantified variables range over freely generated polymorphic data types, thus the domains of discourse are infinite in general. The objective in the development was to quickly find counter-examples for as many false conjectures as possible without wasting too much time on true statements. We present the reasoning method underlying the disprover and illustrate its practical value in several applications in an experimental version of the verification tool \checkmark eriFun.

1 Introduction

As a common experience, specifications are faulty and programs do not meet their intention. Program bugs range from easily detected simple lapses (such as not excluding division by zero or typos when setting array bounds) to deep logical flaws in the program design which emerge elsewhere in the program and therefore are hard to discover.

But programmers' faults are not the only source of bugs. State-of-the-art verifiers *synthesize conjectures* about a program that are needed (or are at least useful) in the course of verification: Statements may be generalized to be qualified for a proof by induction, the verifier might generate termination hypotheses that ensure a procedure's termination, or it might synthesize conjectures justifying an optimization of a procedure. Sometimes these conjectures can be faulty, i.e. over-generalizations might result, the verifier comes up with a wrong idea for termination, or an optimization simply does not apply.

Verifying that a program meets its specification is a waste of time in all these cases, and therefore one should begin with testing the program beforehand. However, as testing is a time consuming and boring task, machine support is welcome (not to say needed) to relieve the human from the test-and-verify cycle. Program testing can be reformulated as a verification problem: A program conjecture ϕ fails the test if the negation of ϕ can be verified. However, for proving these negated conjectures, a special verifier—called a *disprover*—is needed.

In this paper, we present such a disprover for statements about programs written in the functional programming language \mathcal{L} [14], which has been integrated into an experimental version [12] of the interactive verification tool \checkmark eriFun [15, 16]. The procedures of \mathcal{L} -programs operate over freely generated

```

structure bool <= true, false
structure  $\mathbb{N}$  <= 0,  $^+(\cdot : \mathbb{N})$ 
structure list[@A] <=  $\varepsilon$ , [infix] :: (hd : @A, tl : list[@A])
function [outfix] | (k : list[@A]) :  $\mathbb{N}$  <=
  if k =  $\varepsilon$  then 0 else  $^+(|tl(k)|)$ 
function [infix] <>(k, l : list[@A]) : list[@A] <=
  if k =  $\varepsilon$  then l else hd(k) :: (tl(k) <> l) end
function rev(k : list[@A]) : list[@A] <=
  if k =  $\varepsilon$  then  $\varepsilon$  else rev(tl(k)) <> hd(k) ::  $\varepsilon$  end
lemma rev <> <=
   $\forall k, l : list[@A] \text{ rev}(k \text{ <> } l) = \text{rev}(k) \text{ <> } \text{rev}(l)$ 

```

Fig. 1. A simple \mathcal{L} -program

polymorphic data types and are defined by using recursion, case analyses, let-expressions, and functional composition. The data types *bool* for Boolean values and \mathbb{N} for natural numbers \mathbb{N} as well as equality $= : @A \times @A \rightarrow bool$ and a procedure $> : \mathbb{N} \times \mathbb{N} \rightarrow bool$ deciding the $>$ -relation on \mathbb{N} are predefined in \mathcal{L} . Figure 1 shows an example of an \mathcal{L} -program that defines a polymorphic data type *list*[@A], list concatenation $<>$, and list reversal *rev*. In this program, the symbols *true*, *false* are *constructors* of type *bool*, $^+(\dots)$ is the *selector* of the \mathbb{N} -constructor $^+(\dots)$, and *hd* and *tl* are the *selectors* of constructor $::$ for lists. Subsequently, we let $\Sigma(P)$ denote the signature of all function symbols defined by an \mathcal{L} -program P , and $\Sigma(P)^c$ is the signature of all *constructor* function symbols in P . An operational semantics for \mathcal{L} -programs P is defined by an interpreter $eval_P : \mathcal{T}(\Sigma(P)) \mapsto \mathcal{T}(\Sigma(P)^c)$ which maps ground terms to constructor ground terms of the respective monomorphic data types using the definition of the procedures and data types in P , cf. [10, 14, 18].

In \mathcal{L} , statements about programs are given by expressions of the form **lemma** *name* <= $\forall x_1 : \tau_1, \dots, x_n : \tau_n \ b$ (cf. Fig. 1), where *b*—called the *body* of the lemma—is a Boolean term built with the variables x_i (of type τ_i) from a set \mathcal{V} of typed variables and the function symbols in $\Sigma(P)$, where case analyses (like in procedure definitions) and the truth values are used to represent connectives. Hence the general form of the proof obligations we are concerned with are universal formulas $\phi = \forall x_1 : \tau_1, \dots, x_n : \tau_n \ b$. Disproving such a formula ϕ is equivalent to proving its negation $\neg\phi \approx \exists x_1 : \tau_1, \dots, x_n : \tau_n \ \neg b$, and as the domain of each type τ_i can be enumerated, disproving ϕ is a *semi-decidable* problem. A *disproof* of ϕ (also called a *witness* of $\neg\phi$) can be represented by a *constructor ground substitution* σ such that $eval_P(\sigma(b)) = false$. Consequently, disproving ϕ can be viewed as solving the (semi-decidable) equational problem $b \doteq false$.

To solve such an equation, we develop two *disproving calculi* that constitute the two phases of our disprover. The inference rules of both calculi are inspired by the calculus proposed in [6] by Comon and Lescanne for solving equational problems. As disproving is semi-decidable, a *complete* disprover can

be developed. However, as truth of universal formulas ϕ is not semi-decidable by Gödel's first incompleteness theorem, *disproving* ϕ is *undecidable*. Therefore a complete (and sound) disprover need not terminate. But the use in an interactive environment—such as the one **veriFun** provides—requires termination of all subsystems, hence completeness must be sacrificed in favor of termination. The use in an interactive environment also demands *runtime performance*, so particular care is taken to achieve early failure on non-disprovable conjectures.

In Section 2, we explain how we disprove universal formulas ϕ . In Section 3, we demonstrate the practical use of our disprover when **veriFun** employs it in different disproving applications. We compare our proposal with related work in Section 4 and conclude with an outlook on future work in Section 5.

2 Disproving Universal Formulas

Our disproving method proceeds in two phases. The first phase is based on the *elimination calculus* (\mathcal{E} -calculus for short). Its *language* is given by $\mathbb{L}_{\mathcal{E}} := \{\langle E, \sigma \rangle \in L_{\mathcal{E}} \times Sub \mid \mathcal{V}(E) \cap dom(\sigma) = \emptyset\}$: $L_{\mathcal{E}}$ is the set of clauses in which atoms are built with terms from $\mathcal{T}(\Sigma(P), \mathcal{V})$ and the predicate symbols \doteq of type $@A \times @A$ and $>$ of type $\mathbb{N} \times \mathbb{N}$; negative literals are written $t_1 \neq t_2$ or $t_1 \not> t_2$, respectively. Sub denotes the set of all *constructor ground substitutions* σ , i.e. $\sigma(v) \in \mathcal{T}(\Sigma(P)^c)$ for each $v \in dom(\sigma)$. The *inference rules* of the \mathcal{E} -calculus (defined below) are of the form “ $\frac{\langle E, \sigma \rangle}{\langle \lambda(E'), \sigma \circ \lambda \rangle}$, if COND”, where COND stands for a side condition that has to be satisfied to apply the rule, and $\lambda \in Sub$. An \mathcal{E} -*deduction* is a sequence $\langle E_1, \sigma_1 \rangle, \dots, \langle E_n, \sigma_n \rangle$ such that for each i , $\langle E_{i+1}, \sigma_{i+1} \rangle$ originates from $\langle E_i, \sigma_i \rangle$ by applying an \mathcal{E} -inference rule, and $\langle E_1, \sigma_1 \rangle \vdash_{\mathcal{E}} \langle E_n, \sigma_n \rangle$ denotes the existence of such an \mathcal{E} -deduction.

The second phase of our disproving method uses the *solution calculus* (\mathcal{S} -calculus for short). It operates on $\mathbb{L}_{\mathcal{S}} := \{\langle E, \sigma \rangle \in L_{\mathcal{S}} \times Sub \mid \mathcal{V}(E) \cap dom(\sigma) = \emptyset\}$: $L_{\mathcal{S}} \subset L_{\mathcal{E}}$ is the set of clauses in which atoms are formed with predicate symbols \doteq , $>$ and terms from $\mathcal{T}(\Sigma(P'), \mathcal{V})$, where $\Sigma(P')$ emerges from $\Sigma(P)$ by removing the function symbols *if* and $=$ as well as all procedure function symbols. The form of the \mathcal{S} -inference rules (defined below) and deduction $\vdash_{\mathcal{S}}$ are defined identically to the \mathcal{E} -calculus, and $\langle E, \sigma \rangle \vdash_{\mathcal{S} \circ \mathcal{E}} \langle E'', \sigma'' \rangle$ denotes the existence of a composed deduction $\langle E, \sigma \rangle \vdash_{\mathcal{E}} \langle E', \sigma' \rangle \vdash_{\mathcal{S}} \langle E'', \sigma'' \rangle$.

A substitution σ is an *E-substitution* for a clause $E \in L_{\mathcal{E}}$, $\sigma \in Sub_E$ for short, iff $\sigma(v) \in \mathcal{T}(\Sigma(P)^c)$ for each $v \in \mathcal{V}(E)$. We write $\sigma \models l$ if an $\{l\}$ -substitution σ *solves* an \mathcal{E} -literal l , defined by $\sigma \models t_1 \doteq t_2$ iff $eval_P(\sigma(t_1)) = eval_P(\sigma(t_2))$ and $\sigma \models t_1 > t_2$ iff $eval_P(\sigma(t_1)) > eval_P(\sigma(t_2))$. An \mathcal{E} -clause E is solved by $\sigma \in Sub_E$, $\sigma \models E$ for short, iff $\sigma \models l$ for each $l \in E$. Both calculi are *sound* in the sense that $\langle E, \sigma \rangle \vdash_{\dots} \langle E', \sigma' \rangle$ entails $\theta \models \sigma'(E)$ for each $\theta \in Sub_{E'}$ with $\theta \models E'$, and $\sigma \subseteq \sigma'$.

To disprove a conjecture $\phi = \forall x_1 : \tau_1, \dots, x_n : \tau_n \ b$, we search for a deduction $\langle \{b \doteq false\}, \varepsilon \rangle \vdash_{\mathcal{S} \circ \mathcal{E}} \langle \emptyset, \sigma \rangle$.¹ Hence σ represents a disproof of ϕ , as $\sigma \models b \doteq false$.

¹ Since the domain of each data type is at most countably infinite, we actually use *monomorphic* types τ'_i instead of the *polymorphic* types τ_i in ϕ without loss of gen-

- $$\begin{aligned}
(1) & \frac{\langle E \uplus \{l\}, \sigma \rangle}{\langle E \cup \{l[\pi \leftarrow t_2], t_1 \doteq \text{true}\}, \sigma \rangle \mid \langle E \cup \{l[\pi \leftarrow t_3], t_1 \doteq \text{false}\}, \sigma \rangle} \quad , \\
& \text{if } l|_\pi = \text{if}(t_1, t_2, t_3) \\
(2) & \frac{\langle E \uplus \{l\}, \sigma \rangle}{\langle E \cup \{l[\pi \leftarrow w], w \doteq f(\dots)\}, \sigma \rangle} \quad , \text{ if } f \in \Sigma^{\text{proc}}, l|_\pi = f(\dots), \text{ and } \pi \notin \{1, 2\} \\
(3) & \frac{\langle E \uplus \{l\}, \sigma \rangle}{\langle E \cup \{l[\pi \leftarrow \rho(r)]\} \cup E_{\text{cond}}, \sigma \rangle} \quad , \text{ if } \begin{cases} l|_\pi = f(t_1, \dots, t_n) \text{ for some } \pi \in \{1, 2\} \\ \text{and } \langle C, \overline{C}, r \rangle \in D_f \text{ for } f \in \Sigma^{\text{proc}} \end{cases} \\
& \text{where } E_{\text{cond}} = \bigcup_{c \in C} \{\rho(c) \doteq \text{true}\} \cup \bigcup_{c \in \overline{C}} \{\rho(c) \doteq \text{false}\} \text{ and} \\
& \rho := \{x_1/t_1, \dots, x_n/t_n\} \text{ for the formal parameters } x_i \text{ of } f \\
(5) & \frac{\langle E \uplus \{v \doteq \text{cons}(t_1, \dots, t_n), v \doteq \text{cons}(t'_1, \dots, t'_n)\}, \sigma \rangle}{\langle E \cup \{v \doteq \text{cons}(w_1, \dots, w_n)\} \cup \bigcup_{j=1}^n \{w_j \doteq t_j, w_j \doteq t'_j\}, \sigma \rangle} \\
(6) & \frac{\langle E \uplus \{v \doteq \text{cons}(t_1, \dots, t_n), v \not\doteq v', v' \doteq \text{cons}(t'_1, \dots, t'_n)\}, \sigma \rangle}{\langle E \cup \{v \doteq \text{cons}(w_1, \dots, w_n), v' \doteq \text{cons}(w'_1, \dots, w'_n)\} \cup \{w_i \not\doteq w'_i\} \cup \bigcup_{j=1}^n \{w_j \doteq t_j, w'_j \doteq t'_j\}, \sigma \rangle} \quad , \text{ if } i \in \{1, \dots, n\} \\
(7) & \frac{\langle E \uplus \{l\}, \sigma \rangle}{\langle E \cup \{l[\pi \leftarrow w_i], w \doteq \text{cons}(w_1, \dots, w_n), w \doteq t\}, \sigma \rangle} \quad , \text{ if } l|_\pi = \text{sel}_i(t) \quad ^2 \\
(8) & \frac{\langle E \uplus \{^+(t_1) \odot ^+(t_2)\}, \sigma \rangle}{\langle E \cup \{t_1 \odot t_2\}, \sigma \rangle} \quad , \text{ if } \odot \in \{\succ, \not\succ\}
\end{aligned}$$

Fig. 2. Inference rules of the \mathcal{E} -calculus

The inference rules of both calculi are given in the subsequent paragraphs. The most important rules are formally defined whereas others (denoted by rule numbers in italics) are only informally described for the sake of brevity. In order to reduce the depth of the terms in \mathcal{E} - and \mathcal{S} -literals, some of the rules introduce fresh variables (called “auxiliary unknowns” in [6]), which we denote by w and w' . Terms are written as t , t_1 and t_2 , and v and v' denote variables.

2.1 Inference rules of the \mathcal{E} -calculus

The \mathcal{E} -calculus consists of the inference rules (1)–(3) and (5)–(8) of Fig. 2 plus rules (4) and (9)–(10) described informally. The purpose of the \mathcal{E} -inference rules is to eliminate all occurrences of if , $=$, and of procedure function symbols so that some $\langle E, \sigma \rangle \in \mathbb{L}_{\mathcal{S}}$ is obtained by an \mathcal{E} -deduction $\langle \{b \doteq \text{false}\}, \varepsilon \rangle \vdash_{\mathcal{E}} \langle E, \sigma \rangle$.³ All rules are supplied with an additional side condition (*) demanding $E \notin L_{\perp}$ for each $\langle E, \sigma \rangle$ they apply to, where L_{\perp} is the set of all \mathcal{E} -clauses containing *evident*

erality. Type τ'_i originates from type τ_i by instantiating each type variable in τ_i with type \mathbb{N} . E. g., to disprove $\forall k, l: \text{list}[\text{@}A] \ k \text{ <> } l = l \text{ <> } k$, the monomorphic instance $\forall k, l: \text{list}[\mathbb{N}] \ k \text{ <> } l = l \text{ <> } k$ is considered.

² Assuming $t \doteq \text{cons}(\dots)$ is sound, as well-typedness is demanded.

³ This elimination is possible whenever $b \doteq \text{false}$ is solvable, as each procedure call needs to be unfolded by rule (3) only finitely many times.

$$\begin{aligned}
(15) \quad & \frac{\langle E \uplus \{t_1 \odot t_2\}, \sigma \rangle}{\langle E \cup \{w_1 \doteq t_1, w_2 \doteq t_2, w_1 \odot w_2\}, \sigma \rangle} \quad , \text{ if } t_1, t_2 \notin \mathcal{V} \\
(16) \quad & \frac{\langle E \uplus \{v \neq t\}, \sigma \rangle}{\langle E \cup \{v \neq t, v \doteq \text{cons}'(w_1, \dots, w_n)\}, \sigma \rangle} \quad , \text{ if } t \in \mathcal{V} \text{ or } t = \text{cons}(\dots) \\
(17) \quad & \frac{\langle E \uplus \{t_1 \not\geq t_2\}, \sigma \rangle}{\langle E \cup \{t_1 \doteq t_2\}, \sigma \rangle \mid \langle E \cup \{t_2 \succ t_1\}, \sigma \rangle} \quad , \text{ if } t_1, t_2 \notin \mathcal{V} \\
(18) \quad & \frac{\langle E \uplus \{t_1 \neq t_2\}, \sigma \rangle}{\langle E \cup \{t_1 \succ t_2\}, \sigma \rangle \mid \langle E \cup \{t_2 \succ t_1\}, \sigma \rangle} \quad , \text{ if } t_1, t_2 \in \mathcal{T}(\Sigma(P'), \mathcal{V})_{\mathbb{N}}
\end{aligned}$$

Fig. 3. Inference rules of the \mathcal{S} -calculus

contradictions such as $\{t \neq t, \dots\}$, $\{0 \succ t, \dots\}$ or $\{t \doteq 0, t \doteq 1, \dots\}$. This proviso corresponds to the *elimination of trivial disequations* and the *clash* rule for equations in [6].

Rule (1) eliminates an *if*-conditional and rule (2) eliminates an *inner* procedure call from a literal.⁴ Rule (3) unfolds a call of procedure f that occurs as a *direct* argument in a literal. A procedure f is represented here by a set D_f of triples $\langle C, \overline{C}, r \rangle$ such that r is the *if*-free result term in the procedure body of f obtained under the conditions $C \cup \{-c \mid c \in \overline{C}\}$, where C and \overline{C} consist of *if*-free Boolean terms only. E.g., $D_{<>}$ consists of two triples, viz. $d_1 = (\{k = \varepsilon\}, \emptyset, l)$ and $d_2 = (\emptyset, \{k = \varepsilon\}, hd(k) :: (tl(k) <> l))$, for procedure $<>$ of Fig. 1.

A further rule (4) translates inequations and equations expressed with symbols from $\Sigma(P)$ into \mathcal{E} -literals; e.g., “ $t_1 > t_2 \doteq \text{false}$ ” is translated into “ $t_1 \not\geq t_2$ ”. Rule (6) is like the *decomposition* rule from [6] for inequations, but restricted to constructors. Another rule (9)—corresponding to the *elimination of trivial equations* and *clash* for inequations in [6]—removes *trivial literals* such as $t \doteq t$ or $0 \not\geq t$ from a clause $E \in L_{\mathcal{E}}$ and supplies arbitrary values in λ for variables that *disappear* from the clause. Finally, literals are simplified by rule (10), which replaces subterms of the form $sel_i(\text{cons}(t_1, \dots, t_n))$ with t_i . This rule does not exist in [6] and accounts for data type definitions with selectors.

2.2 Inference rules of the \mathcal{S} -calculus

The \mathcal{S} -calculus consists of the inference rules (5)–(19), to which the additional side condition (*) applies as well. Rules (5)–(10) are the same as in the \mathcal{E} -calculus, rules (11)–(14) are “structural” rules to merge (in)equations, to replace variables with terms, and to solve equations $v \doteq t$ with $t \in \mathcal{T}(\Sigma(P)^c)$ by substitutions $\lambda := \{v/t\}$.

Rules (15)–(18) are given in Fig. 3. Rule (15) removes non-variable arguments from an \mathcal{S} -literal, rule (16) is basically a case analysis on v using some

⁴ For a literal $l = t_1 \odot t_2$, $l|_{\pi}$ is the subterm of l at occurrence $\pi \in \text{Occ}(l)$, and $l[\pi \leftarrow t]$ is obtained from l by replacing $l|_{\pi}$ in l with t . We use “ \mid ” as a shorthand for the succedents of different rules with the same premises and side conditions. All rules are applied “*modulo symmetry*” of \doteq and \neq if possible (e.g., see rule (5)).

constructor $cons'$, and rules (17) and (18) eliminate negative literals. Rule (19) invokes a *constraint solver*.⁵ We call an \mathcal{S} -literal $t_1 \odot t_2$ a *constraint literal* iff at least one of the t_i is a variable of type \mathbb{N} , $\Sigma(t_1) \cup \Sigma(t_2) \subseteq \{0, +, -\}$, and $\odot \in \{=, >\}$; \mathcal{C} is the set of all constraint literals. When none of the other \mathcal{S} -rules is applicable, rule (19) passes the constraint literals $E \cap \mathcal{C}$ to a modified version of INDIGO [5]: To terminate on cyclic constraints like $\{x > y, y > x\}$, we simply limit the number of times a constraint can be used by the number of variables in $E \cap \mathcal{C}$. If $E \cap \mathcal{C}$ can be satisfied, we get a solving assignment $\lambda \in Sub_{E \cap \mathcal{C}}$; otherwise rule (19) fails.⁶

2.3 Search Heuristic and Implementation

By the inherent indeterminism of both calculi, search is required for computing \mathcal{E} - and \mathcal{S} -deductions: An \mathcal{E} -clause E to be solved defines an *infinite \mathcal{E} -search tree* $T_{\mathcal{E}}^{(E, \varepsilon)}$ whose nodes are labeled with elements from $\mathbb{L}_{\mathcal{E}}$. The root node of $T_{\mathcal{E}}^{(E, \varepsilon)}$ is labeled with $\langle E, \varepsilon \rangle$, and $\langle E'', \sigma'' \rangle$ is a successor of node $\langle E', \sigma' \rangle$ iff $\langle E'', \sigma'' \rangle$ originates from $\langle E', \sigma' \rangle$ by applying some \mathcal{E} -inference rule. The leaves of $T_{\mathcal{E}}^{(E, \varepsilon)}$ are given by the \mathcal{E} -success and the \mathcal{E} -failure nodes: $\langle E', \sigma' \rangle$ is an *\mathcal{E} -success node* iff $E' \in L_{\mathcal{S}} \setminus L_{\perp}$, and $\langle E', \sigma' \rangle$ is an *\mathcal{E} -failure node* iff $E' \in L_{\perp}$. A path from the root node to an \mathcal{E} -success node is called an *\mathcal{E} -solution path*. All these notions carry over to *\mathcal{S} -search trees* $T_{\mathcal{S}}^{(E, \sigma)}$ by replacing \mathcal{E} with \mathcal{S} literally, except that an \mathcal{S} -node labeled with $\langle E', \sigma' \rangle$ is an *\mathcal{S} -success node* iff $E' = \emptyset$, and $\langle E', \sigma' \rangle$ is an *\mathcal{S} -failure node* iff $E' \neq \emptyset$ and no \mathcal{S} -inference rule applies to $\langle E', \sigma' \rangle$.⁷

An \mathcal{E} -clause E to be solved defines an *infinite $\mathcal{S} \circ \mathcal{E}$ -search tree* $T_{\mathcal{S} \circ \mathcal{E}}^E$, which originates from $T_{\mathcal{E}}^{(E, \varepsilon)}$ by replacing each \mathcal{E} -success node $\langle E', \sigma' \rangle$ with the \mathcal{S} -search tree $T_{\mathcal{S}}^{(E', \sigma')}$. An *$\mathcal{S} \circ \mathcal{E}$ -solution path* in $T_{\mathcal{S} \circ \mathcal{E}}^E$ is an \mathcal{E} -solution path p followed by an \mathcal{S} -solution path that starts at the \mathcal{E} -success node of path p .

To disprove a conjecture $\phi = \forall x_1 : \tau_1, \dots, x_n : \tau_n \ b$, the $\mathcal{S} \circ \mathcal{E}$ -search tree $T_{\mathcal{S} \circ \mathcal{E}}^{\{b \neq false\}}$ is explored to find an $\mathcal{S} \circ \mathcal{E}$ -solution path. In order to guarantee termination of the search, only a *finite* part of $T_{\mathcal{S} \circ \mathcal{E}}^{\{b \neq false\}}$ may be explored. Additional side conditions for the \mathcal{E} - and \mathcal{S} -inference rules ensure that one rule does not undo another rule's work. Rules that do not require a choice, e.g. rule (5), are preferred to those that need some choice, e.g. rules (6) and (16).

The most significant restriction in exploring $T_{\mathcal{S} \circ \mathcal{E}}^{\{b \neq false\}}$ (supporting termination at the cost of completeness) comes from an additional side condition $(**)$ of rule (3), called the *paramodulation rule* in [8]: Definition triples $d := (C, \overline{C}, r) \in D_f$ with $f \notin \Sigma(r)$, called *non-recursive* definition triples, can be applied as often

⁵ We use $>$ and \neq in $L_{\mathcal{E}}$ only to handle calls of the predefined procedure $>$ more efficiently by a constraint solver.

⁶ In our setting there is no need to assign priorities to constraints, so we can simplify the algorithm by treating all constraints as “required” constraints.

⁷ $E' \in L_{\perp}$ is sufficient but not necessary for $\langle E', \sigma' \rangle$ being an \mathcal{S} -failure node, as the constraint solver called by rule (19) might fail on some \mathcal{S} -clause $E' \in L_{\mathcal{S}} \setminus L_{\perp}$.

as possible. However, if $f \in \Sigma(r)$, we need to limit the usage of d . Side condition $(**)$ demands that *recursive* definition triples be used at most once on each side of a literal in each branch of $T_{\mathcal{E}}^{\langle\{b \doteq \text{false}\}, \varepsilon\rangle}$. This leads to a fast disprover that works well on simple examples, cf. Sect. 3. We call this restriction *simple paramodulation*.

To increase the deductive performance of the disprover, we can allow more applications of a recursive definition triple d by considering the “context” of f -procedure calls. Each procedure call $f(\dots)$ in the original formula ϕ is labeled with $(N, \dots, N) \in \mathbb{N}^k$ for a constant $N \in \mathbb{N}$, e. g. $N = 2$, and $k = |D_f|$. *Context-sensitive paramodulation* modifies side condition $(**)$ in the following way: A *recursive* definition triple $d_i := (C_i, \overline{C}_i, r_i) \in D_f$ may only be used if procedure call $f(\dots)$ is labeled with (n_1, \dots, n_k) such that $n_i > 0$. The recursive calls of f in r_i are labeled with $(n_1, \dots, n_{i-1}, n_i - 1, n_{i+1}, \dots, n_k)$, and the other procedure calls in r_i are labeled with (N, \dots, N) . Context-sensitive paramodulation still allows only finitely many applications of rule (3), as the labels decrease with each rule application. Section 3 gives examples that illustrate the difference between these alternatives in practice. Note that simple paramodulation is *not* a special case of context-sensitive paramodulation (by setting $N = 1$), because it does not distinguish between different occurrences of a procedure as context-sensitive paramodulation does.

For efficiency reasons (wrt. memory consumption), we explore $T_{\mathcal{E}}^{\langle\{b \doteq \text{false}\}, \varepsilon\rangle}$ with a *depth-first* strategy, whereas $T_{\mathcal{S}}^{\langle E', \sigma' \rangle}$ is examined *breadth-first* to avoid infinite applications of rule (16). Two technical optimizations considerably speed up the search for an $\mathcal{S} \circ \mathcal{E}$ -solution path. Firstly, caching allows to prune a branch that has already been considered in another derivation. The cache hit rates are about 20 %. Secondly, while exploring $T_{\mathcal{E}}^{\langle\{b \doteq \text{false}\}, \varepsilon\rangle}$ we can already start a subsidiary \mathcal{S} -search on \mathcal{S} -literals from a clause $E \notin L_{\perp}$ (even though node $\langle E, \sigma \rangle$ of $T_{\mathcal{E}}^{\langle\{b \doteq \text{false}\}, \varepsilon\rangle}$ is not an \mathcal{E} -success node) and feed the results back to the \mathcal{E} -search node $\langle E, \sigma \rangle$. For instance, if we derive $x \doteq^+(y)$ from E , we can discard \mathcal{E} -branches that consider the case $x \doteq 0$. In conjunction with *simple paramodulation*, this (empirically) leads to early failure on unsolvable examples.

3 Using the Disprover

In this section we illustrate the use and the performance of our disprover when it is employed as a subsystem of $\check{\text{VeriFun}}$ [12]. Unless otherwise stated, we use simple paramodulation. We distinguish between conjectures provided by the user and conjectures speculated by the system.

3.1 User-Provided Conjectures

Before trying to verify a program statement, it is advisable to make sure that it does not contain lapses that render it false. E. g., in arithmetic we are often interested in cancelation lemmas such as $x^y = x^z \rightarrow y = z$. However, the disprover

finds the witness $\{x/0, y/0, z/1\}$ falsifying the conjecture. Excluding $x=0$ does not help, as now the witness $\{x/1, y/0, z/1\}$ is quickly computed. But excluding $x=1$ as well causes the disprover to fail, hence we are expectant that verification of $\forall x, y, z: \mathbb{N} \ x \neq 0 \wedge \neg(x) \neq 0 \wedge x^y = x^z \rightarrow y = z$ will succeed. If we conjecture the associativity of exponentiation, $(x^y)^z = x^{(y^z)}$, the disprover finds the witness $\{x/2, y/0, z/0\}$. For the injectivity conjecture of the factorial function, i. e. $\forall x, y: \mathbb{N} \ x! = y! \rightarrow x = y$, the disprover comes up with the witness $\{x/1, y/0\}$ and fails if we demand $x \neq 0 \wedge y \neq 0$ in addition. For $\forall k, l: list[@A] \ k <> l = l <> k$, the solution $\{k/0::\varepsilon, l/1::\varepsilon\}$ is computed.

All conjectures from above are disproved within less than a second.⁸ One might argue that these disproofs are quite simple, so they should be easy to find. **veriFun**'s old disprover [1] basically substitutes the variables with values (or value templates like $n::k$ for lists) of a limited size and uses a heuristic search strategy to track down a counter-example quickly if one exists. However, such a strategy does *not* lead to *early* failure on *true* conjectures: The old disprover fails after 46s on the conjecture that procedure *perm* (deciding whether two lists are a permutation of each other) computes a symmetric relation.⁹ The new disprover fails after just a second.

The disprover also helps to find simple flaws in the definition of lemmas or procedures. For instance, it disproves lemma “*rev <>*” (cf. Fig. 1), yielding $\{k/0::\varepsilon, l/1::\varepsilon\}$. Also, the termination hypothesis for *<>* is disproved at once if one inadvertently writes $tl(l)$ in the recursive call of *<>* (instead of $tl(k)$). Similar errors are the use of \geq instead of $>$ in program conjectures and procedure definitions.

To illustrate the consequences of *simple paramodulation*, consider formula $\forall k: list[@A] \ rev(k) = k$. As the smallest solution is $\{k/0::1::\varepsilon\}$, we need to open *rev* twice. Thus the disprover fails to find this witness with *simple paramodulation*, but succeeds with *context-sensitive paramodulation*. The same effect is observed with lemma “*rev <>*” or with $\forall x: \mathbb{N} \ 2^x > x^2$. However, as most conjectures do not need extensive search, we prefer to save time and offer this alternative only as an option to the user who is willing to spend more time on the search for a disproof.

3.2 Conjectures Speculated by the System

When generalizing statements by machine, a disprover is needed to detect *over-generalizations*. E. g., **veriFun**'s generalization heuristic [1] tries to generalize $\phi = \forall k, l: list[@A] \ half(|k <> l|) = half(|l <> k|)$ to $\phi' = \forall k, l: list[@A] \ |k <> l| = |l <> k|$ and then ϕ' to $\phi'' = \forall k, l: list[@A] \ k <> l = l <> k$. Our disprover quickly fails on ϕ' and succeeds on ϕ'' (see above), hence generalization ϕ' is a good candidate for a proof by induction, whereas ϕ'' is recognized as an over-generalization of ϕ .

⁸ All timing details refer to our single-threaded JAVA implementation on a 3.2 GHz hyper-threading CPU, where the JAVA VM was assigned 300 MB of main memory.

⁹ The old disprover examined the conjecture for lists of length ≤ 2 and natural numbers between 0 and 2.

Another example of such a generate-and-test cycle is *recursion elimination*: For user-defined procedures, **veriFun** synthesizes so-called *difference* and *domain procedures* which represent information that is useful for automated analysis of termination [13, 17] and for proving absence of “exceptions” [18] (caused by division by 0, for example). Both kinds of procedures may contain unnecessary recursive calls, which complicate subsequent proofs. Therefore the system generates *recursion elimination formulas* [13] justifying a sound replacement of some recursive calls with truth values. For those formulas that the system could not prove, the user has to decide whether to support the system either by interactively constructing a proof or by giving a witness to disprove the conjecture. He can also ignore the often unreadable conjectures (which most users do), not being aware that missing a true recursion elimination formula means much more work in subsequent proofs.

For example, for the domain procedure of a tautology checker (cf. procedure \neq in [14]), **veriFun** generates 62 recursion elimination formulas. Our disprover falsifies all of them within 33 s. Without a disprover, we wasted four times longer on futile proof attempts from which we cannot conclude anything. With the old disprover, it took more than five times longer to disprove 59 formulas; it failed on the others. For other domain or difference procedures, the disprover performs equally well, so in the vast majority of cases the user does not need to worry about recursion elimination any more. This is a tremendous improvement in user-friendliness.

4 Related Work

The problem of automatically disproving statements in the context of program verification has been tackled in various research projects.

Protzen [8] describes a calculus to disprove universal conjectures in the INKA system [4]. While it apparently performs quite well on false system-generated conjectures, it has a rather poor performance on true ones; if the input conjecture is true, it searches until it reaches an explicit limit of the search depth.

A disprover for KIV is presented in [9]. The existing proof calculus is modified so that it is able to construct disproofs. This interleaves the incremental instantiation of variables and simplifying proof steps. For solvable cases “good results” are reported, whereas performance on unsolvable problems is not communicated.

Ahrendt has developed a *complete* disprover for free data type specifications [2]. Since the interpretation of function symbols is left open in this *loose semantics* approach, one needs to consider *all* models satisfying the axioms when proving the non-consequence of a conjecture ϕ . Similarly, the ALLOY modeling system [7] can investigate properties of under-specified models. The corresponding constraint analyzer checks only models with a bounded number of elements in each primitive type, so (like our disprover) it is incomplete. Differently from these approaches, we consider only a fixed interpretation of function symbols (given by the interpreter *eval_P*) in our setting.

Isabelle supports a “quickcheck” command [3] to test a conjecture by substituting random values for the variables several times. A comparison of the success rates and the performance of this approach with our results is planned as future work.

CORAL [11] is a system designed to find non-trivial flaws in security protocols. It is based on the Comon-Nieuwenhuis method for proof by consistency and uses a parallel architecture for consistency checking and so-called induction derivation to ensure termination. Finding an attack on a protocol may take several hours with CORAL.

5 Conclusion

In the design of our disprover we tried to minimize the time wasted on true conjectures. We achieved this by limiting the application of the paramodulation rule. Apart from this, we do *not* need any explicit depth limits. In particular, there is no explicit limit on the size of a witness. We also reduce the cost of simplifications by restricting them to selector and constructor calls. By incorporating a constraint solver [5] for inequalities on the predefined data type \mathbb{N} for \mathbb{N} , we further improved the performance.

We identified several applications of our disprover that considerably improve the productivity when working with the $\sqrt{\text{eriFun}}$ system. The main application of our disprover is bulk processing (such as recursion elimination) or automatic generalization. While it is possible to approximate completeness arbitrarily well to find deeper flaws in a program (conjecture), this would tremendously increase the time wasted on true conjectures. The advantage of our disprover is that it is successful in most solvable cases and quickly gives up in unsolvable cases, as practical experiments reveal.

In future work, we intend to investigate further heuristics for the paramodulation rule, which primarily controls the power of the disprover. We also intend to examine whether the use of verified lemmas supports the disproving process. Finally, it would be interesting to look at combinations of various disproving strategies. When we are aware of the strengths and weaknesses of different strategies, we could possibly decide beforehand which one is most suitable for a specific problem.

References

1. Markus Aderhold. Formula generalization in $\sqrt{\text{eriFun}}$. Diploma thesis, Technische Universität Darmstadt, 2004.
2. Wolfgang Ahrendt. Deductive search for errors in free data type specifications using model generation. In A. Voronkov, editor, *Proc. of the 18th International Conference on Automated Deduction*, volume 2392 of *LNCS*. Springer, 2002.
3. Stefan Berghofer and Tobias Nipkow. Random testing in Isabelle/HOL. In *Software Engineering and Formal Methods*, pages 230–239. IEEE Computer Society, 2004.

4. Susanne Biundo, Birgit Hummel, Dieter Hutter, and Christoph Walther. The Karlsruhe induction theorem proving system. In J. Siekmann, editor, *Proc. of CADE-8*, volume 230 of *LNCS*, pages 672–674. Springer, 1986.
5. Alan Borning, Richard Anderson, and Bjorn N. Freeman-Benson. Indigo: A local propagation algorithm for inequality constraints. In *ACM Symposium on User Interface Software and Technology*, pages 129–136, 1996.
6. Hubert Comon and Pierre Lescanne. Equational problems and disunification. *Journal of Symbolic Computation*, 7:371–425, 1989.
7. Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.
8. Martin Protzen. Disproving conjectures. In D. Kapur, editor, *Proc. of CADE-11*, volume 607 of *LNAI*, pages 340–354. Springer, 1992.
9. Wolfgang Reif, Gerhard Schellhorn, and Andreas Thums. Flaw detection in formal specifications. In *Proc. of IJCAR-1*, pages 642–657. Springer, 2001.
10. Stephan Schweitzer. *Symbolische Auswertung und Heuristiken zur Verifikation funktionaler Programme*. Doctoral dissertation, TU Darmstadt, to appear 2006.
11. Graham Steel and Alan Bundy. Attacking group protocols by refuting incorrect inductive conjectures. *Journal of Automated Reasoning*, pages 1–28, 2005.
12. Daniel Szallies. Ein Werkzeug zur automatischen Widerlegung von Aussagen in \checkmark eriFun. Diplomarbeit, Technische Universität Darmstadt, 2006.
13. Christoph Walther. On proving the termination of algorithms by machine. *Artificial Intelligence*, 71(1):101–157, 1994.
14. Christoph Walther, Markus Aderhold, and Andreas Schlosser. The \mathcal{L} 1.0 Primer. Technical Report VFR 06/01, Technische Universität Darmstadt, 2006.
15. Christoph Walther and Stephan Schweitzer. About \checkmark eriFun. In F. Baader, editor, *Proc. of CADE-19*, volume 2741 of *LNCS*, pages 322–327. Springer, 2003.
16. Christoph Walther and Stephan Schweitzer. Verification in the classroom. *Journal of Automated Reasoning*, 32(1):35–73, 2004.
17. Christoph Walther and Stephan Schweitzer. Automated termination analysis for incompletely defined programs. In F. Baader and A. Voronkov, editors, *Proc. of LPAR-11*, volume 3452 of *LNAI*, pages 332–346. Springer, 2005.
18. Christoph Walther and Stephan Schweitzer. Reasoning about incompletely defined programs. In G. Sutcliffe and A. Voronkov, editors, *Proc. of LPAR-12*, volume 3835 of *LNAI*, pages 427–442. Springer, 2005.

Predicting Failures of Inductive Proof Attempts

Mahadevan Subramaniam¹, Deepak Kapur², and Stephan Falke²

¹ Computer Science Department,
University of Nebraska at Omaha
Omaha, NE 68182, USA
`msubramaniam@mail.unomaha.edu`

² Computer Science Department
University of New Mexico
Albuquerque, NM 87131, USA
`{kapur|spf}@cs.unm.edu`

Abstract. Reasoning about recursively defined data structures and functions defined on them typically requires proofs by induction. Despite advances made in automating inductive reasoning, proof attempts by theorem provers frequently fail while performing inductive reasoning. A user of such a system must scrutinize a failed proof attempt and do intensive debugging to understand the cause of failure.

The failure of proof attempts could be because of a number of reasons even when a conjecture is believed to be valid. One reason is that an induction scheme used in a proof attempt is not powerful enough to yield useful induction hypotheses which can be applied effectively. Or the proof attempt might need intermediate lemmas. The focus of the research reported in this paper is to analyze possible failures of proof attempts due to inapplicability of induction hypotheses and predict failure a priori before even attempting a proof, so as to avoid failed attempts.

Definitions of functions appearing in a conjecture are analyzed to determine whether their interaction in the conjecture guarantees a proof attempt to get stuck. The analysis relies on the concept of blocking of a function definition by another function definition. If, in a conjecture, a function g appears as an argument to another function f such that when the definition of g is expanded, f blocks a function symbol resulting from the definition of g , then a proof attempt of the conjecture based on expanding the definition of g is likely to get stuck. The concept of a flawed induction scheme is introduced capturing this idea. It is shown that if a proof of a conjecture is attempted using only flawed induction schemes, then, under certain conditions, such proof attempts are guaranteed to fail. The analysis can be easily automated and is illustrated on several examples.

1 Introduction

Induction plays an important role in reasoning about many applications that involve computation with recursive data structures. Among other things, this

includes software, parameterized hardware and protocols, and, more recently, static program analysis. A number of theorem provers supporting procedures to mechanize induction have been successful in verifying properties of several applications. However, induction proof attempts in theorem provers frequently fail, often requiring user intensive debugging of failed proof attempts. Automated methods that aid users in handling failures of inductive proof attempts are of crucial importance.

In this paper, we describe a novel approach that a priori determines the failure of a theorem prover to establish a given conjecture using induction. Based on such an approach, a theorem prover can provably predict its own failure even before attempting an induction proof of the conjecture. Syntactic conditions on the conjectures and on the definitions of the functions appearing in the conjecture are identified to achieve this goal. Based on such an analysis, a prover can provide useful feedback without the users having to resort to the tedious task of analyzing failed proof transcripts.

The framework used for performing inductive reasoning is that of explicit induction using the cover set method as implemented in our theorem prover *RRL* [8,12]. Typically, to prove a conjecture by induction, a subterm of the form $f(x_1, x_2, \dots)$ in the conjecture with a defined function f is selected to automatically identify the induction variables and generate an induction scheme from f 's terminating definition; for simplicity and without any loss of generality, it is assumed that f is defined by recursing over the first two arguments, and the remaining arguments remain invariant. The induction scheme specifies how the induction variables are instantiated to generate subgoals in an inductive proof attempt of the conjecture (base cases and induction step cases), and these instantiations are generated from the definition of f .

The main reason for using an induction scheme from the definition of an innermost function appearing in a conjecture is that when the conjecture is instantiated in each subgoal, it can be simplified using the definition of f , hopefully leading to a formula which can be either be shown to be valid without any use of the induction hypotheses (base cases) or on which an induction hypothesis(es) is applicable. Different induction schemes generated from different subterms of a conjecture lead to different proof attempts. Provers like *RRL* [8], *Nqthm*, *ACL2* [2,10] implement several heuristics for choosing among these schemes, with the hope that the selected induction scheme will lead to a successful proof attempt.

There can be several reasons for an induction proof attempt to fail even when the conjecture is valid. The most obvious cause of failure is that an induction hypothesis from a selected induction scheme is not applicable¹. To a priori predict failures due to inapplicability of induction hypotheses in an induction proof

¹ In [11], conditions have also been identified for predicting proof attempt failures where the formula, even after the successful application of induction hypotheses, cannot be reduced to *true*. Another reason for failure is the need for intermediate lemmas which must be used to make progress in a proof attempt. We do not discuss that type of analysis here due to lack of space. An interested reader can consult [11] for details.

attempt generated using an induction scheme, the context of the induction variables (i.e., other function symbols around f) in the conjecture becomes critical. Interaction among function symbols in the context is analyzed using the concept of *blocking* of a functions definition by another function definition. If, in a conjecture, a function g appears as an argument to another function f such that when the definition of g is expanded, f blocks a function symbol resulting from the definition of g , then a proof attempt of the conjecture based on expanding the definition of g is likely to get stuck. This concept is then used to define a *flawed* induction scheme for the conjecture, using the terminology of Boyer and Moore. It is shown that if a proof of a conjecture is attempted using only flawed induction schemes, then, under certain conditions, such proof attempts are guaranteed to fail. In the next subsection, we illustrate the proposed approach for predicting failures using a simple example over numbers.

Typically, several induction schemes are available to attempt an induction proof of a conjecture and several heuristics are used to pick the “best” one among them. These heuristics may be significantly improved by avoiding schemes declared flawed based on blocking. The above analyses can also lead to the discovery of potentially useful bridge lemmas that may be used to *unblock* function definitions so that a proof attempt that is predicted to fail may succeed with the help of such lemmas. In fact, such feedback can be generated automatically from the analysis and this can be more meaningful to a user than what can be obtained by tedious manual analyses of failed proof attempts.

This work is complementary to our work reported in [7,6], where conditions on function definitions and conjectures are identified in order to ensure that the inductive validity of a subclass of conjectures can be decided automatically without any user interaction. Compatibility among function definitions plays a critical role there, albeit positively, in the sense that if functions appearing in a conjecture have compatible definitions, then, under certain conditions, the inductive validity of the conjecture can be decided.

The paper is organized as follows. In the next subsection, a simple example on numbers is informally discussed to motivate and illustrate the key ideas of the proposed approach. Related work on failure analyses of inductive theorem provers is discussed in Section 2. Section 3 provides a brief overview of the cover set method that is used to mechanize induction in the theorem prover Rewrite Rule Laboratory (*RRL*) [8]. In Section 4 the concepts of equational blocking and definitional blocking are introduced and these concepts are used to define flawed induction schemes. In Section 5, we show how failures of certain oriented conjectures based on flawed induction schemes are guaranteed to get stuck because an induction hypothesis cannot be applied. This main result of the paper is proved by identifying conditions on oriented conjectures and definitions of function symbols appearing in the conjecture such that an inductive proof attempt of the conjecture fails. The result is illustrated using examples.

1.1 A Simple Illustrative Example

Example 1: Consider proving the conjecture²

$$C_1 : \text{even}((n + n) * m) == \text{true},$$

from the following rules defining functions $+$, $*$, and even on natural numbers:

1. $x + 0 \rightarrow 0$,
2. $x + s(y) \rightarrow s(x + y)$,
3. $x * 0 \rightarrow 0$,
4. $x * s(y) \rightarrow x + (x * y)$,
5. $\text{even}(0) \rightarrow \text{true}$,
6. $\text{even}(s(0)) \rightarrow \text{false}$,
7. $\text{even}(s(s(x))) \rightarrow \text{even}(x)$,

with 0 and s (successor) as the constructors for natural numbers.

The conjecture C_1 cannot be decided by equational reasoning from the above rules. An inductive proof of C_1 can be attempted. There are two variables (n and m) which serve as candidates for performing induction. Let us first consider m for performing induction. Using the principle of mathematical induction, m is instantiated to be 0 for the base case; for the induction step case, m is instantiated to be $s(y)$, with a hypothesis generated by instantiating m to be y .³

The base case simplifies to *true* using rules 3, 5. Focussing on the step case, the conclusion in the subgoal is

$$\text{even}((n + n) * s(y)) == \text{true},$$

with the induction hypothesis being

$$\text{even}((n + n) * y) == \text{true}.$$

The conclusion simplifies using rule 4 to $\text{even}(((n + n) + ((n + n) * y))) == \text{true}$, to which the hypothesis cannot be applied. Therefore, the proof attempt fails.

It will be shown in this paper that such a failure of an inductive proof attempt can be predicted a priori (without attempting the proof) by analyzing the interaction among the rules in the definition of $*$ and even . When $*$ as an argument to even is expanded using the recursive rule 4, the extra function $+$ between even and $*$ cannot be eliminated using the existing rules. In this case, even is said to definitionally block $*$. Consequently, the corresponding induction scheme is said to be *flawed*.

If induction is performed on n , then $*$, surrounding $n + n$, cannot be simplified when its second argument is a variable; instead another proof by induction must be performed.

It is thus futile to attempt a proof of C_1 without defining the function even in terms of $+$, or without proving a lemma about $*$ with a variable as its second argument. However, if we first prove lemmas about even , such as $(\text{even}(x) == \text{true} \wedge \text{even}(y) == \text{true}) \implies \text{even}(x + y) == \text{true}$ and $\text{even}(x + x) == \text{true}$, that

² We use the symbol “==” to denote the equality predicate in conjectures.

³ In the next section, we discuss the cover set induction method [12] for generating induction schemes from the terminating definitions of functions. Using the terminating definition of $*$, the cover set method will generate the same induction scheme.

capture the interaction between `even` and `+`, the conjecture C_1 can be proved to be an inductive theorem using the scheme generated from the variable m .

In this paper, we identify conditions on conjectures and definitions of function symbols appearing in such conjectures such that it can be predicted a priori that a proof attempt will not succeed.

2 Related Work

The manual overhead of analyzing failed proof attempts of inductive theorem provers is a well-known problem. Methods to aid users in analyses of failed inductive proof attempts, ranging from the development of interactive proof browsers [9], automatically patching faulty conjectures based on proof planning [5,4], and using rippling techniques [3], have been investigated earlier. The approach discussed in this paper was proposed first in [11] and is radically different from the above approaches in its attempt to predict guaranteed failures, thereby avoiding wasteful analyses of failed proof attempts. Flawed induction schemes for conjectures were defined using the concept of definitional blocking in [11]. The basic idea of a flawed scheme is first described in [2] for developing heuristics to select induction schemes in theorem prover Nqthm. The key result of this paper is from [11], with the difference that function definitions were assumed to be shown terminating using a recursive path ordering and its variants in [11]. In contrast, we only require in this paper that function definitions are given using an arbitrary terminating set of rewrite rules. As a result, we are able to extend the class of conjectures from those considered in [11].

3 Generating Induction Schemes from Function Definitions

In this section we first briefly review the cover set method [12] for automating induction. We assume familiarity with the concepts of term rewriting [1].

Let $T(F, X)$ denote the set of terms built using a set of function symbols F and a set of variables X . Let $t|_p$ stand for the subterm of t at position p . A substitution σ is a finite map from variables to terms, written $\{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$. Function definitions are given using a finite set \mathcal{R} of ground-convergent, sufficiently complete, terminating rewrite rules of the form $l \rightarrow r$. The left side l is assumed to be of the form $f(s_1, \dots, s_k)$, where s_1, \dots, s_k are constructor terms. The right side r contains zero or more recursive calls to f and to other functions. Inductive arguments of a function f are arguments differ between the left side l and one of the recursive calls to f .

In the cover set method [12], induction schemes are automatically generated for a conjecture C from its subterms $f(x_1, \dots, x_n, t_1, \dots, t_m)$ where the x_i are distinct variables occurring on the inductive positions of f and not in the t_i . Each subterm $f(x_1, \dots, x_n, t_1, \dots, t_m)$ in a conjecture, called a φ -term, produces an induction scheme φ . Let $IndVar(\varphi)$ be the induction variables of φ , i.e., $IndVar(\varphi) = \{x_1, \dots, x_n\}$.

An induction scheme φ is a collection of pairs $\langle \sigma, \{\vartheta_1, \dots, \vartheta_n\} \rangle$ of substitutions where the ϑ_i produce the hypotheses and σ produces the conclusion. For a φ -term $s = f(x_1, \dots, x_n, t_1, \dots, t_m)$, a pair $\langle \sigma, \{\} \rangle$ in the scheme φ is obtained corresponding to each non-recursive rule $l \rightarrow r$ defining the function f ; σ is the *mgu* (most general unifier) of l and s . A pair $\langle \sigma, \{\vartheta_1, \dots, \vartheta_n\} \rangle$ is produced corresponding to each recursive rule $l \rightarrow r$ where σ is the mgu of l and s and each ϑ_i is the mgu of l and the i^{th} recursive call to f in r .

In an induction proof attempt of C based on scheme φ , an induction base case $C\sigma$ is generated from each $\langle \sigma, \{\} \rangle$. An induction step case with conclusion $C\sigma$ and n hypotheses $C\vartheta_i$, is generated for each pair $\langle \sigma, \{\vartheta_1, \dots, \vartheta_n\} \rangle$ in the scheme φ . As an example, in Section 1.1, the induction proof attempt of conjecture C_1 based on the induction scheme $\varphi = \{\langle n \mapsto 0, \{\} \rangle, \langle n \mapsto s(y), \{\{n \mapsto y\}\} \rangle\}$ produces one base case by instantiating $n = 0$ in C_1 ; it produces a step case where the conclusion is obtained by instantiating $n = s(y)$ in C_1 and the hypothesis is obtained by instantiating $n = y$ in C_1 .

To ensure that an induction proof attempt actually does get stuck whenever the generated hypotheses cannot be applied, we make some further assumptions about conjectures and induction schemes obtained from function definitions. We assume that the conclusion in the subgoal of an inductive step case cannot be simplified to *true* without using an induction hypothesis. We also assume that the simplification of the hypotheses does not affect their applicability to the conclusion. These requirements are met whenever the function definitions and the substitutions in the induction schemes are irreducible, which is commonly the case in several provers including *RRL*. Induction schemes generated from such definitions have been called *well-behaved* induction schemes in [11]; more details about well-behaved schemes and their properties can be found in [11]. Henceforth, all induction schemes are assumed to be well-behaved.

4 Blocking

The interaction among the definitions of function symbols in the conjecture is captured using the concept of *blocking*. Below, we define two kinds of blocking. *Equational blocking* captures the property that the relative function positions are left invariant by simplification by a set of rules. A somewhat related but orthogonal property of *definitional blocking* is introduced next, where relative function positions produced by the expansion of a function definition are left invariant.

Definition 1 (Equational Blocking). *The definition of a function g is equationally blocked by the definition of another function f as its k^{th} argument iff there is no rule $l \rightarrow r \in \mathcal{R}$ and no substitution σ such that $l\sigma$ has the form $f(\dots, g(\dots), \dots)$.*

As an example, the definition of the function *even* in Example 1 in Section 1.1 equationally blocks the definition of the functions $+$ as its argument with

respect to the rules 1-7. The term $\text{even}((n+n) + ((n+n) * y))$ is also said to be equationally blocked.

Based on equational blocking, we can determine a priori which of the subterms at a particular position in a term, if any, will be left unchanged when the term is simplified using \mathcal{R} .

Lemma 2. *Let $s = f(\dots, t, \dots)$ with $t = g(\dots)$. If f equationally blocks g and the term t is irreducible by \mathcal{R} , then any simplified form of s has the form $f(\dots, t, \dots)$.*

Proof sketch. Since t is irreducible by \mathcal{R} , no rewrite rule applies to the term s at or inside the subterm t . Since f equationally blocks g , no rewrite rule applies to s at the root position. Hence, the only possible rewrite steps can occur inside the remaining arguments to f , i.e., each simplified form of s has the form $f(\dots, t, \dots)$. \square

To predict information about the simplified forms of conclusions in induction step cases, we extend equational blocking to rules defining functions used to generate induction schemes.

Let H_g denote the set of function symbols h different from g such that h is the outermost function symbol of a right-hand side of a rule in the definition of the function g , i.e., $H_g = \{\text{outermost}(r) \mid l \rightarrow r \in \mathcal{R}_g\} - \{g\}$, where $\mathcal{R}_g \subseteq \mathcal{R}$ contains all rules defining g .

Definition 3 (Definitional Blocking). *The definition of a function g is definitional blocked by the definition of another function f on its k^{th} argument iff there is an $h \in H_g$ such that h is equationally blocked by f as its k^{th} argument.*

As an example, the definition of **even** definitionally blocks the definition of $*$ as its argument since $+$ $\in H_*$ and $+$ is equationally blocked by **even** as its argument.

5 Predicting Failures of Oriented Conjectures

Definitional blocking is the key idea to predict failure of an inductive proof attempt based on an induction scheme, as illustrated by the example in Section 1.1. In conjecture C_1 , consider the subterm $\text{even}((n+n) * m)$ in which **even** has $*$ as its argument. The reader can easily verify that **even** both equationally as well as definitionally blocks $*$. The right hand side of rule 4 has $+$ as its outermost symbol; however, **even** is not defined on $+$ as its argument, i.e., **even** equationally blocks $+$. The induction scheme generated from the definition of $*$ produces a step case where the conclusion contains the symbol $+$ as the argument of **even**. However, in the hypothesis, the argument of **even** will remain to be $*$, the same as in the conjecture. Therefore, the hypothesis cannot be applied due to this difference, as illustrated in Section 1.1.

We now generalize the above discussion as follows. Consider a conjecture that has a subterm s of the form $f(\dots, g(\dots), \dots)$. Suppose that f definitionally

blocks g and that we generate an induction scheme using the definition of g . An induction proof attempt of the conjecture based on such a scheme will produce an induction step case in which the conclusion will contain a term of the form $f(\dots, h(\dots, g(\dots), \dots), \dots)$ where h belongs to H_g and is equationally blocked by f (the function symbol h appears as an outermost symbol of the right hand-side of one of the recursive rules defining g). The argument to f is thus likely to be an h -term in all simplified forms of the conclusion. However, in the hypotheses of this induction step case, the corresponding argument of f will be a subterm $g(\dots)$, since the hypotheses are generated by unifying with the recursive calls. So, it is likely that the hypotheses cannot be applied to the conclusion due to this difference in the terms.

Among the induction schemes suggested from the definitions of function symbols appearing innermost in a given conjecture, *flawed* induction schemes are identified based on the immediate context of an innermost function symbol in the conjecture. If the function symbol f surrounding an innermost function symbol g definitionally blocks g , then the induction scheme generated from the definition of g is *flawed*. Proof attempts based on flawed schemes are highly likely to fail. We discuss additional conditions on the conjecture to provably predict the failure of a proof attempt based on a flawed scheme.

Definition 4 (φ -context Term). *A term s is a φ -context term for an induction scheme φ if the k^{th} argument of s is a φ -term.*

As an example, the φ -context term containing the φ -term $(n + n) * m$ in conjecture C_1 in Section 1.1 is $\text{even}((n + n) * m)$. The φ -context term for the other φ -term $n + n$ of C_1 is $(n + n) * m$.

Definition 5 (Flawed Induction Scheme). *An induction scheme φ is flawed in the conjecture $l == r$ if*

1. *there is a φ -context term s , with the outermost function symbol f , in $l == r$ whose k^{th} argument is the φ -term t with the outermost symbol g , such that f definitionally blocks g , and*
2. *every proper superterm of s is equationally blocked.*

The induction scheme suggested by the φ -term $(n + n) * m$ in Example 1 in Section 1.1 is flawed since even definitionally blocks $*$. Since the φ -context term is the left-hand side of the conjecture, it trivially follows that every proper superterm containing the φ -context term is equationally blocked.

The requirements in the above definition of a flawed induction scheme ensure that the difference between the φ -context term in the conclusion and the hypothesis is preserved by every simplification step of the conclusion. Since f definitionally blocks g , there is an outermost symbol in the definition of g which cannot be eliminated by any simplification of s ; since every term containing s is equationally blocked, this outermost symbol cannot be removed by simplifying any superterm containing s either.

The above conditions are sufficient to predict failures of certain *oriented* conjectures $l == r$ where, w.l.o.g., we assume that $\mathcal{R} \cup \{l \rightarrow r\}$ terminates and

l contains a φ -term. In the inductive proof attempt of an oriented conjecture, the hypotheses are applicable from left-to-right only, i.e., in each induction case with conclusion $l\sigma == r\sigma$, a hypothesis $l\vartheta_i == r\vartheta_i$ is applicable only if $l\sigma \rightarrow_{\mathcal{R}}^* l'$ and l' contains $l\vartheta_i$ as a subterm.

The following Theorem is the main result of this paper. It states the conditions under which the failure of an inductive proof attempt of an oriented conjecture due to the inapplicability of induction hypotheses can be provably predicted.

Theorem 6 (Inapplicability Failures for Oriented Conjectures). *An inductive proof attempt of an oriented conjecture $l == r$ based on an induction scheme φ fails due to an inapplicability failure if the following conditions hold:*

- φ is flawed in l for the φ -context term s , and
- no $x \in \text{IndVar}(\varphi)$ occurs outside the φ -context term s in l .

Proof sketch. Let φ be flawed in l for the φ -context term $s = f(\dots, g(\dots), \dots)$, where the subterm $g(\dots)$ is the φ -term. Thus, there is an $h \in H_g$ such that f equationally blocks h . Consider an induction case of φ that is generated by a rule whose outermost function symbol on the right side is h . For this induction case, the conclusion (after application of that rule) contains the term $s' = f(\dots, h(\dots, g(\dots), \dots), \dots)$ instead of s . Since f equationally blocks h , every simplified form of s' has the same form by Lemma 2. Thus, none of the inductive hypotheses is applicable to a simplified form of s' . Since every proper superterm of s is equationally blocked and induction variables don't appear outside s in them, no rewrite rule applies to any proper superterm of s' , either. This makes use of the assumptions that the conjecture is irreducible and the induction scheme is well-behaved. Furthermore, none of the inductive hypotheses is applicable to those terms since they don't contain induction variables. For subterms of l that don't contain s as a subterm the same argument applies. \square

The conjecture $C_1 : \text{even}((n + n) * m) == \text{true}$ in Section 1.1 is an oriented conjecture. The scheme suggested by the φ -term $(n + n) * m$ is flawed in l as was shown earlier. The induction variable m does not occur outside the φ -context term in l . Hence, by Theorem 6, the proof attempt of C_1 based on φ generated from the φ -term $(n + n) * m$ and the rules 1-7 given in Section 1.1 is predicted to fail, which is indeed the case as illustrated in Section 1.1.

6 Examples

We illustrate the use of Theorem 6 on the following three examples. Examples 3 and 4 cannot be handled using the methods of [11] since termination cannot be shown using a recursive path ordering. In both of these examples, the right side of the rules include a ternary boolean operator, $\text{if}(c, s, t)$, which equals s if c is true and equals t otherwise. Whenever such a rule is used for simplification in an induction subgoal, case analysis automatically produces two subgoals corresponding to the first argument being true or false.

Example 2: Consider the following rules defining the factorial function on natural numbers:

5. $\text{fac}(0) \rightarrow \text{s}(0)$, 6. $\text{fac}(\text{s}(x)) \rightarrow \text{s}(x) * \text{fac}(x)$,
7. $\text{facit}(0, y) \rightarrow y$, 8. $\text{facit}(\text{s}(x), y) \rightarrow \text{facit}(x, \text{s}(x) * y)$,

along with rules 1-4 in Section 1.1 defining $+$ and $*$. Assume we are trying to prove the following oriented conjecture relating the tail recursive and recursive factorial functions facit and fac :

$$C_2 : \text{facit}(x * y, k) == \text{fac}(x * y) * k.$$

Assume that the induction scheme $\varphi = \{\langle y \mapsto 0, \{\} \rangle, \langle y \mapsto \text{s}(y'), \{\{y \mapsto y'\}\} \rangle\}$ is generated from the φ -term $x * y$ appearing on the left side of C_3 . The scheme φ is flawed in C_3 since facit definitionally blocks $*$ as its first argument. The φ -context term on the left side is $\text{facit}(x * y, k)$, hence no induction variable occurs outside it in the left side. Thus, all conditions of Theorem 6 are satisfied and the inductive proof attempt of C_2 based on the scheme φ is predicted to fail, which can be verified. Since $x * y$ is the only φ -term in the left side of C_2 , no other induction scheme is generated, and therefore any proof attempt of conjecture C_2 by induction is guaranteed to fail.

Example 3: As the next example, given function definitions

5. $x - 0 \rightarrow x$, 6. $0 - y \rightarrow 0$, 7. $\text{s}(x) - \text{s}(y) \rightarrow x - y$,
8. $\text{rem}(0, y) \rightarrow 0$, 9. $\text{rem}(\text{s}(x), 0) \rightarrow 0$,
10. $\text{rem}(\text{s}(x), \text{s}(y)) \rightarrow \text{if}(y \leq x, \text{rem}(x - y, \text{s}(y)), \text{s}(x))$,
11. $\text{div}(x, 0) \rightarrow 0$, 12. $\text{div}(0, \text{s}(y)) \rightarrow 0$,
13. $\text{div}(\text{s}(x), \text{s}(y)) \rightarrow \text{if}(x < y, 0, \text{s}(\text{div}(x - y, \text{s}(y))))$,

and rules 1-4 in Section 1.1 defining $+$ and $*$ consider proving

$$C_3 : \text{rem}(x, y * z) == x - ((y * z) * \text{div}(x, y * z)),$$

an oriented conjecture, using the induction scheme $\varphi = \{\langle \{z \mapsto 0\}, \{\} \rangle, \langle \{z \mapsto \text{s}(z')\}, \{\{z \mapsto z'\}\} \rangle\}$ generated from the φ -term $y * z$ in the left side of C_3 . The scheme φ is flawed since rem definitionally blocks $*$. The induction variable z only occurs within the φ -context term $\text{rem}(x, y * z)$ in the left side. All conditions of Theorem 6 are satisfied, and therefore the proof attempt of C_3 using scheme φ is predicted to fail, which can be verified. Since there are no other φ -terms in the left side C_3 , any proof attempt of C_3 by induction will fail.

Example 4: Given function definitions

5. $\text{odd}(0) \rightarrow \text{false}$, 6. $\text{odd}(\text{s}(0)) \rightarrow \text{true}$, 7. $\text{odd}(\text{s}(\text{s}(x))) \rightarrow \text{odd}(x)$,
8. $\text{half}(0) \rightarrow 0$, 9. $\text{half}(\text{s}(0)) \rightarrow 0$, 10. $\text{half}(\text{s}(\text{s}(x))) \rightarrow \text{s}(\text{half}(x))$,
11. $\text{exp}(x, 0) \rightarrow \text{s}(0)$, 12. $\text{exp}(x, \text{s}(y)) \rightarrow x * \text{exp}(x, y)$,
13. $\text{expit}(x, 0, z) \rightarrow z$,
14. $\text{expit}(x, \text{s}(y), z) \rightarrow \text{if}(\text{odd}(\text{s}(y)), \text{expit}(x, y, x * z), \text{expit}(x * x, \text{half}(\text{s}(y)), z))$,

along with rules 1-4 in Section 1.1 defining $+$ and $*$, consider proving the following oriented conjecture relating the tail recursive and recursive exponentiation functions expit and exp on natural numbers:

$$C_4 : \text{expit}(x, y * z, k) == \text{exp}(x, y * z) * k.$$

Assume that the induction scheme $\varphi = \{\langle z \mapsto 0, \{\} \rangle, \langle z \mapsto s(z'), \{\{z \mapsto z'\}\} \rangle\}$ is generated from the φ -term $y * z$ appearing on the left side of C_4 . The scheme φ is flawed in C_4 since expit definitionally blocks $*$ as its second argument. The φ -context term on the left side is $\text{expit}(x, y * z, k)$, hence no induction variable occurs outside it in the left side. Thus, all conditions of Theorem 6 are satisfied and the inductive proof attempt of C_4 based on the scheme φ is predicted to fail, which can be verified. Since $y * z$ is the only φ -term in the left side of C_4 , no other induction scheme is generated, and therefore any proof attempt of conjecture C_4 by induction is guaranteed to fail.

7 Concluding Remarks

An approach for provably predicting failures of mechanized induction proof attempts of a large class of conjectures is proposed. Syntactic conditions capturing the interaction among the definitions of functions appearing in a conjecture are identified in order to predict failures of induction proof attempts due to inapplicability of hypotheses. The notions of equational blocking and definitional blocking are introduced to characterize induction schemes as being flawed. Failures of proof attempts for oriented conjectures are provably predicted if function symbols leading to flawed induction schemes appear in contexts that definitional block the function whose definition is used for generating an induction scheme. The effectiveness of the approach is illustrated using several examples. The approach in this paper predicts failures only for a single application of induction. We plan to generalize this approach to multiple applications of induction, which is common in practice.

References

1. F. Baader and T. Nipkow, *Term Rewriting and All That*. Cambridge University Press, 1998.
2. R. S. Boyer and J S. Moore, *A Computational Logic*. ACM Monographs in Computer Science, 1979.
3. A. Bundy, The Automation of Proof by Mathematical Induction, *Handbook of Automated Reasoning*, 2001.
4. A. Bundy, Planning and Patching Proof, *Proc. AISC '04*, LNCS 3249, 26–37, 2004.
5. A. Ireland, Productive Use of Failure in Inductive Proof, *Journal of Automated Reasoning*, 16(1–2):79–111, 1996.
6. D. Kapur, J. Giesl and M. Subramaniam, Induction and Decision Procedures, *Rev. R. Acad. Cien. Serie A: Mat.*, 2004.
7. D. Kapur and M. Subramaniam, Extending Decision Procedures with Induction Schemes, *Proc. CADE '00*, LNAI 1831, 324–345, 2000.

8. D. Kapur and H. Zhang, An Overview of Rewrite Rule Laboratory (RRL), *Proc. RTA '89*, LNCS 355, 559–563, 1989.
9. M. Kaufmann, An Extension of the Boyer-Moore Theorem prover to Support First-Order Quantification, *Journal of Automated Reasoning*, 9(3):355–372, 1992.
10. M. Kaufmann, P. Manolios, and J S. Moore, *Computer-Aided Reasoning: An Approach*, Kluwer, 2000.
11. M. Subramaniam, *Failure Analyses in Inductive Theorem Provers*, Ph.D. Thesis, Department of Computer Science, University of Albany, New York, 1997.
12. H. Zhang, D. Kapur, and M. S. Krishnamoorthy, A Mechanizable Induction Principle for Equational Specifications, *Proc. CADE '88*, LNCS 310, 162–181, 1988.

Computing Finite Models by Reduction to Function-Free Clause Logic

Peter Baumgartner¹, Alexander Fuchs², Hans de Nivelle³, and Cesare Tinelli²

¹ National ICT Australia (NICTA), Peter.Baumgartner@nicta.com.au

² The University of Iowa, USA, {fuchs,tinelli}@cs.uiowa.edu

³ Max-Planck-Institut für Informatik, Germany, nivelle@mpi-inf.mpg.de

Abstract. Recent years have seen considerable interest in procedures for computing finite models of first-order logic specifications. One of the major paradigms, MACE-style model building, is based on reducing model search to a sequence of propositional satisfiability problems and applying (efficient) SAT solvers to them. A problem with this method is that it does not scale well, as the propositional formulas to be considered may become very large.

We propose instead to reduce model search to a sequence of satisfiability problems made of function-free first-order clause sets, and to apply (efficient) theorem provers capable of deciding such problems. The main appeal of this method is that first-order clause sets grow more slowly than their propositional counterparts, thus allowing for more space efficient reasoning.

In the paper we describe the method in detail and show how it is integrated into one such prover, Darwin, our implementation of the Model Evolution calculus. The results are general, however, as our approach can be used in principle with any system that decides the satisfiability of function-free first-order clause sets.

To demonstrate its practical feasibility, we tested our approach on all satisfiable problems from the TPTP library. Our methods can solve a significant subset of these problems, which overlaps but is not included in the subset of problems solvable by state-of-the-art finite model builders such as Paradox and Mace4.

1 Introduction

Methods for model computation can be classified as those that directly search for a finite model, like the extended PUHR tableau method [8], the methods in [6, 10] and the methods in the SEM-family [14, 16, 13], and those based on transformations into certain fragments of logic and relying on corresponding readily available systems (see [4] for a recent approach).

The latter approach includes the family of MACE-style model builders [13]. These systems search for finite models, essentially, by searching the space of interpretations with domain sizes $1, 2, \dots$, in increasing order, until a model is found. The MACE-style model builder with the best performance today is perhaps the Paradox system [9]. We present in this paper a new approach in the MACE/Paradox tradition which however capitalizes on new advances in instantiation-based first-order theorem proving, as opposed to advances in propositional satisfiability as in the case of MACE and Paradox. The general idea in our approach is the same: to find a model with n elements for a

given a clause set possibly with equality, the clause set is first converted into a simpler form by means of the following transformations.

1. Each clause is flattened.
2. Each n -ary function symbol is replaced by an $n + 1$ -ary predicate symbol and equality is eliminated.
3. Clauses are added to the clause set that impose totality constraints on the new predicate symbols, but over a domain of cardinality n .

The details of our transformation differ in various aspects from the MACE/Paradox approach. In particular, we add no functionality constraints over the new predicate symbols. The main difference, however, is that we eventually reduce the original problem to a satisfiability problem over function-free clause logic (without equality), not over propositional logic. As a consequence of the different target logic, we do not use a SAT solver to look for models. Instead, we use a variant of *Darwin* [2], our implementation of the Model Evolution calculus [5], which can decide satisfiability in that logic.

While we do take advantage of some of the distinguishing features of *Darwin* and the Model Evolution calculus, especially in the way models are constructed, our method is general enough that it could use without much additional effort any other decision procedure for function-free clause logic, for instance, any implementation of one of the several instance-based methods for first-order reasoning that are currently enjoying a growing popularity.

In this paper we illustrate our method in some detail, presenting the main translation and its implementation within *Darwin*, and discussing our initial experimental results in comparison with Paradox itself and with Mace4 [13], a competitive, non-MACE-like (despite the name) model builder. The results indicate that our method is rather promising as it can solve 1074 of the 1251 satisfiable problems in the TPTP library [15]. These problems are neither a subset nor a superset of the sets of 1083 and 802 problems respectively solved (under the same experimental settings) by Paradox and Mace4.

2 Preliminaries

We use standard terminology from automated reasoning. We assume as given a signature $\Sigma = \Sigma_F \cup \Sigma_P$ of function symbols Σ_F (including constants) and predicate symbols Σ_P . As we are working (also) with equality, we assume Σ_P contains a distinguished binary predicate symbol \approx , used in infix form, with $\not\approx$ denoting its negation. Terms, atoms, literals and formulas over Σ and a given (denumerable) set of variables V are defined as usual. A clause is a (finite) implicitly universally quantified disjunction of literals. A *clause set* is a finite set of clauses. We use the letter C to denote clauses and the letter L to denote literals.

For a given atom $P(t_1, \dots, t_n)$ (possibly an equation) the terms t_1, \dots, t_n are also called the *top-level terms* (of $P(t_1, \dots, t_n)$).

With regards to semantics, we use the notions of (first-order) *satisfiability* and *E-satisfiability* in a completely standard way. If \mathcal{I} is an (E -)interpretation then $|\mathcal{I}|$ denotes the domain (or universe) of \mathcal{I} . Recall that in E -interpretations the equality relation is interpreted as the *identity relation*, i.e. for every E -interpretation \mathcal{I} it holds $\approx^{\mathcal{I}} =$

$\{(d, d) \mid d \in |\mathcal{I}|\}$. We are primarily interested in computing *finite* models, which are models (of the given clause set) with a finite domain.

In the remainder of the paper, we assume that M is a given (finite) clause set over signature $\Sigma = \Sigma_F \cup \Sigma_P$, where Σ_F (resp. Σ_P) are the function symbols (resp. predicate symbols) occurring in M .

3 Finite Model Transformation

In this section we give a general description of the transformations we apply to the input problem to reduce it to an equisatisfiable problem in function-free clause logic without equality. We do that by defining various transformation rules on clauses.

In the rules, we write $L \vee C \rightsquigarrow C' \vee C$ to indicate that the clause $C' \vee C$ is obtained from the clause $L \vee C$ by (single) application of one of these rules.

3.1 Basic Transformation

(1) Abstraction of positive equations.

$$\begin{aligned}
 s \approx y \vee C &\rightsquigarrow s \not\approx x \vee x \approx y \vee C && \text{if } s \text{ is not a variable and } \\
 &&& x \text{ is a fresh variable} \\
 x \approx t \vee C &\rightsquigarrow t \not\approx y \vee x \approx y \vee C && \text{if } t \text{ is not a variable and } \\
 &&& y \text{ is a fresh variable} \\
 s \approx t \vee C &\rightsquigarrow s \not\approx x \vee t \not\approx y \vee x \approx y \vee C && \text{if } s \text{ and } t \text{ are not variables and } \\
 &&& x \text{ and } y \text{ are fresh variables}
 \end{aligned}$$

These rules make sure that all (positive) equations are between variables.

(2) Flattening of non-equations.

$$(\neg)P(\dots, s, \dots) \vee C \rightsquigarrow (\neg)P(\dots, x, \dots) \vee s \not\approx x \vee C \quad \text{if } P \neq \approx, s \text{ is not a variable, and } x \text{ is a fresh variable}$$

(3) Flattening of negative equations.

$$f(\dots, s, \dots) \not\approx t \vee C \rightsquigarrow f(\dots, x, \dots) \not\approx t \vee s \not\approx x \vee C \quad \text{if } s \text{ is not a variable and } x \text{ is a fresh variable}$$

(4) Separation of negative equations.

$$s \not\approx t \vee C \rightsquigarrow s \not\approx x \vee t \not\approx x \vee C \quad \text{if neither } s \text{ nor } t \text{ is a variable, and } x \text{ and } y \text{ are fresh variables}$$

This rule makes sure that at least one side of a (negative) equation is a variable.

Notice that this property is also satisfied by the transformations (2) and (3).

(5) Removal of trivial negative equations.

$$x \not\approx y \vee C \rightsquigarrow C\sigma \quad \text{where } \sigma = \{x \mapsto y\}$$

(6) Orientation of negative equations.

$$x \not\approx t \vee C \rightsquigarrow t \not\approx x \vee C \quad \text{if } t \text{ is not a variable}$$

For a clause C , let the *basic transformation of C* , denoted as $\mathcal{B}(C)$, be the clause obtained from C by applying the transformations (1)-(6), in this order, each as long as possible.⁴ We extend this notation to clause sets in the obvious way, i.e., $\mathcal{B}(M)$ is the clause set consisting of the basic transformation of all clauses in M .

The two flattening transformations alone, when applied exhaustively, turn a clause into a *flat* one, where a clause is *flat* if:

1. each top-level term of each of its negative equations is a variable or has the form $f(x_1, \dots, x_n)$, where f is a function symbol, $n \geq 0$, and x_1, \dots, x_n are variables;
2. each top-level term of each of its non-equations is a variable.

Similar flattening transformations have been considered before as a means to deal more efficiently with equality within calculi for first-order logic without equality [7, 1].

The basic transformation above is correct in the following sense.

Lemma 1 (Correctness of \mathcal{B}). *The clause set M is E -satisfiable if and only if $\mathcal{B}(M)$ is E -satisfiable.*

Proof. That flattening preserves E -satisfiability (both ways) is well-known (cf. [7]). Regarding transformations (1), (4), (5) and (6), the proof is straightforward or trivial. \square

3.2 Conversion to Relational Form

It is not hard to see that, for any clause C , the following holds for the clause set $\mathcal{B}(C)$:

1. each of its positive equations is between two variables,
2. each of its negative equations is flat and of the form $f(x_1, \dots, x_n) \not\approx y$, and
3. each of its non-equations is flat.

After the basic transformation, we apply the following one, turning each n -ary function symbol f into a (new) $n + 1$ -ary predicate symbol R_f .

(7) Elimination of function symbols.

$$f(x_1, \dots, x_n) \not\approx y \vee C \rightsquigarrow \neg R_f(x_1, \dots, x_n, y) \vee C$$

Let $\mathcal{B}_R(M)$ be the clause set obtained from an exhaustive application of this transformation to $\mathcal{B}(M)$.

Recall that an $n + 1$ -ary relation R over a set A is *left-total* if for every $a_1, \dots, a_n \in A$ there is an $b \in A$ such that $(a_1, \dots, a_n, b) \in R$. The relation R is *right-unique* if whenever $(a_1, \dots, a_n, b) \in R$ there is no other tuple of the form (a_1, \dots, a_n, b') in R .

Because of the above properties (1)–(3) of $\mathcal{B}(M)$, the transformation $\mathcal{B}_R(M)$ is well-defined, and will produce a clause set with no function symbols. This transformation however is not unsatisfiability preserving unless one considers only left-total interpretations for the predicate symbols R_f . More formally:

⁴ It is easy to see that this process always terminates.

Lemma 2 (Correctness of \mathcal{B}_R). *The clause set M is E -satisfiable if and only if there is an E -model \mathcal{I} of $\mathcal{B}_R(M)$ such that $(R_f)^\mathcal{I}$ is left-total, for every function symbol $f \in \Sigma_F$.*

Proof. The direction from left to right is easy. For the other direction, let \mathcal{I} be an E -model of $\mathcal{B}_R(M)$ such that $(R_f)^\mathcal{I}$ is left-total for every function symbol $f \in \Sigma_F$.

Recall that functions are nothing but left-total and right-unique relations. We will show how to obtain from \mathcal{I} an E -model \mathcal{I}' of $\mathcal{B}_R(M)$, that preserves left-totality and adds right-uniqueness, i.e., such that $(R_f)^{\mathcal{I}'}$ is both left-total and right-unique for all $f \in \Sigma_F$. Since such an interpretation is clearly a model of $\mathcal{B}(M)$, it will follow immediately by Lemma 1 that M is E -satisfiable.

We obtain \mathcal{I}' as the interpretation that is like \mathcal{I} , except that $(R_f)^{\mathcal{I}'}$ contains exactly one element (d_1, \dots, d_n, d) , for every $d_1, \dots, d_n \in |\mathcal{I}|$, chosen arbitrarily from $(R_f)^\mathcal{I}$ (this choice exists because $(R_f)^\mathcal{I}$ is left-total). It is clear from the construction that $(R_f)^{\mathcal{I}'}$ is right-unique and left-total. Trivially, \mathcal{I}' interpretes \approx as the identity relation, because \mathcal{I} does, as \mathcal{I} is an E -interpretation. Thus, \mathcal{I}' is an E -interpretation, too.

It remains to prove that with \mathcal{I} being a model of $\mathcal{B}_R(M)$ then so is \mathcal{I}' . This follows from the fact that every occurrence of a predicate symbol R_f , with $f \in \Sigma_F$, in the clause set $\mathcal{B}_R(M)$ is in a negative literal. But then, since $(R_f)^{\mathcal{I}'} \subseteq (R_f)^\mathcal{I}$ by construction, it follows immediately that any clause of $\mathcal{B}_R(M)$ satisfied by \mathcal{I} is also satisfied by \mathcal{I}' . \square

The significance of this lemma is that it requires us to interpret the predicate symbols R_f as left-total relations, *but not necessarily as right-unique ones*. Consequently, right-uniqueness will not be axiomatized below.

3.3 Addition of Finite Domain Constraints

To force left-totality, one could add the Skolemized version of axioms of the form

$$\forall x_1, \dots, x_n \exists y R_f(x_1, \dots, x_n, y)$$

to $\mathcal{B}_r(M)$. The resulting set would be E -satisfiable exactly when M is E -satisfiable.⁵ However, since we are interested in finite satisfiability, we use finite approximations of these axioms. To this end, let d be a positive integer, the *domain size*. We consider the expansion of the signature of $\mathcal{B}_r(M)$ by d fresh constant symbols, which we name $1, \dots, d$. Intuitively, instead of the totality axiom above we can now use the axiom

$$\forall x_1, \dots, x_n \exists y \in \{1, \dots, d\} R_f(x_1, \dots, x_n, y) .$$

Concretely, if f is an n -ary function symbol let the clause

$$R_f(x_1, \dots, x_n, 1) \vee \dots \vee R_f(x_1, \dots, x_n, d)$$

be the *d -totality axiom for f* , and let $\mathcal{D}(d)$ be the set of all d -totality axioms for all function symbols $f \in \Sigma_F$. The set $\mathcal{D}(d)$ axiomatizes the left-totality of $(R_f)^\mathcal{I}$, for every function symbol $f \in \Sigma_F$ and interpretation \mathcal{I} with $|\mathcal{I}| = \{1, \dots, d\}$.

⁵ Altogether, this proves the (well-known) result that function symbols are “syntactic sugar”. They can always be eliminated in an equisatisfiability preserving way, at the cost of introducing existential quantifiers.

$$\begin{array}{cc}
\begin{array}{c}
R_{c_1}(1) \vee \dots \vee R_{c_1}(d) \\
R_{c_2}(1) \vee \dots \vee R_{c_2}(d) \\
\vdots \\
R_{c_m}(1) \vee \dots \vee R_{c_m}(d)
\end{array}
&
\begin{array}{c}
R_{c_1}(1) \\
R_{c_2}(1) \vee R_{c_2}(2) \\
\vdots \\
R_{c_d}(1) \vee \dots \vee R_{c_d}(d) \\
R_{c_{d+1}}(1) \vee \dots \vee R_{c_{d+1}}(d) \\
\vdots \\
R_{c_m}(1) \vee \dots \vee R_{c_m}(d)
\end{array}
\end{array}$$

(a) (b)

Fig. 1. Totality axioms for constants and their triangular form

3.4 Symmetry Breaking

Symmetries have been identified as a major source for inefficiencies in constrain solving systems. *Value symmetry* applies to a problem when a permutation of the values (or better, value vector) assigned to variables constitutes a solution to the problem, too. A dual symmetry property may apply to the (decision) variables of a problem, giving rise to *variable symmetry*. Breaking such symmetries has been recognized as a source for considerable efficiency gains.

It is easy to break some value symmetries introduced by assigning domain values to constants. Suppose Σ_F contains m constants c_1, \dots, c_m . Recall that $\mathcal{D}(d)$ contains, in particular, the axioms shown in Figure 1(a). Similarly to what is done with Paradox, these axioms can be replaced by the more “triangular” form shown in Figure 1(b). This form reflects symmetry breaking of assigning values for the first d constants. In fact, one could further strengthen the symmetry breaking axioms by adding (unit) clauses like $\neg R_{c_1}(2), \dots, \neg R_{c_1}(d)$. We do not add them, as they do not constrain the search for a model further (they are all pure).

In the sequel we will refer to the clause set as described here as $\mathcal{D}(d)$.

3.5 Putting all Together

Since we want to use clause logic *without* equality as the target logic of our overall transformation, the only remaining step is the explicit axiomatization of the equality symbol \approx over domains of size d —so that we can exploit Lemma 2 in the (interesting) right-to-left direction. This is easily achieved with the clause set

$$\mathcal{E}(d) = \{i \not\approx j \mid 1 \leq i, j \leq d \text{ and } i \neq j\}.$$

Finally then, we define the *finite-domain transformation of M* as the clause set

$$\mathcal{F}(M, d) := \mathcal{B}_R(M) \cup \mathcal{D}(d) \cup \mathcal{E}(d).$$

Putting all together we arrive at the following first main result:

Theorem 3 (Correctness of the Finite-Domain Translation). *Let d be a positive integer. Then, M is E -satisfiable by some finite interpretation with domain size d if and only if $\mathcal{F}(M, d)$ is satisfiable.*

Proof. Follows from Lemma 2 and the comments above on $\mathcal{D}(d)$ and $\mathcal{E}(d)$, together with the observation that if $\mathcal{F}(M, d)$ is satisfiable it is satisfiable in a Herbrand interpretation with universe $\{1, \dots, d\}$.

More precisely, for the only-if direction assume as given a Herbrand model \mathcal{I} of $\mathcal{F}(M, d)$ with universe $\{1, \dots, d\}$. It is clear from the axioms $\mathcal{E}(d)$ that \mathcal{I} assigns false to the equation $(d' \approx d'')$, for any two different elements $d', d'' \in \{1, \dots, d\}$. Now, the model \mathcal{I} can be modified to assign true to all equations $d' \approx d'$, for all $d' \in \{1, \dots, d\}$ and the resulting E -interpretation will still be a model for $\mathcal{F}(M, d)$. This is, because the only occurrences of negative equations in $\mathcal{F}(M, d)$ are those contributed by $\mathcal{E}(d)$, which are still satisfied after the change.⁶ It is this modified model that can be turned into an E -model of M . \square

This theorem suggests immediately a (practical) procedure to search for finite models, by testing $\mathcal{F}(M, d)$ for satisfiability, with $d = 1, 2, \dots$, and stopping as soon as the first satisfiable set has been found. Moreover, any reasonable such procedure will return in the satisfiable case a Herbrand representation (of some finite model).

Indeed, the idea of searching for a finite model by testing satisfiability over finite domains of size $1, 2, \dots$ is implemented in our approach and many others (Paradox [9], Finder [14], Mace [12], Mace4 [13], SEM [16] to name a few).

4 Implementation

We implemented the transformation described so far within our theorem prover *Darwin*. In addition to being a full-blown theorem prover for first-order logic without equality, *Darwin* is a decision procedure for the satisfiability of function-free clause sets, and thus is a suitable back-end for our transformation. We call the combined system FM-*Darwin* (for Finite Models *Darwin*).

Conceptually, FM-*Darwin* builds on *Darwin* by adding to it as a front-end an implementation of the transformation \mathcal{F} (Section 3.5), and invoking *Darwin* on $\mathcal{F}(M, d)$, for $d = 1, 2, \dots$, until a model is found. In reality, FM-*Darwin* is built *within Darwin* and differs from the conceptual procedure described so far in the following ways:

1. The search for models of increasing size is built in *Darwin*'s own restarting mechanism. For refutational completeness *Darwin* explores its search space in an iterative-deepening fashion with respect of certain *depth* measures. The same mechanism is used in FM-*Darwin* to restart the search with an increased domain size $d + 1$ if the input problem has no models of size d .
2. FM-*Darwin* implements some obvious optimizations over the transformation rules described in Section 3. For instance, the transformations (1)–(4) are done in parallel, depending on the structure of the current literal. Transformation (6) is done implicitly

⁶ Notice, in particular, that $\mathcal{B}_R(M)$ contains only positive occurrences of equations, if any.

as part of transformation (7), when turning equations into relations. Also, when flattening a clause, the same variable is used to abstract different occurrences of a subterm.

3. Because the clause sets $\mathcal{F}(M, d)$ and $\mathcal{F}(M, d + 1)$, for any d , differ only in their subsets $\mathcal{D}(d) \cup \mathcal{E}(d)$ and $\mathcal{D}(d + 1) \cup \mathcal{E}(d + 1)$, respectively, there is no need to re-generate the constant part, and this is not done.

4. Similarly to SAT solvers based on the DPLL procedure, *Darwin* has the ability to learn new (entailed) clauses—or *lemmas*—in failed branches of a derivation, which is helpful to prune search space in later branches [3]. Some of the learned lemmas are independent from the current domain size and so can be carried over to later iterations with larger domain sizes. To do that, each clause in $\mathcal{D}(d + 1)$ is actually *guarded* by an additional literal M_d standing for the current domain size. In FM-*Darwin*, lemmas depending on the current domain size d , and only those, retain the guard M_d when they are built, making it easy to eliminate them when moving to the next size $d + 1$.

5. Recall from step (7) in the transformation (Section 3.2) that every function symbol is turned into a predicate symbol. In our actual implementation, we go one step further and use a meta modeling approach that can make the final clause set produced by our translation more compact, and possibly speed up the search as well, thanks to the way models are built in the Model Evolution calculus. The idea is the following.

For every $n > 0$, instead of generating an $n + 1$ -ary relation symbol R_f for each n -ary function symbol $f \in \Sigma_F$ we use an $n + 2$ -ary relation symbol R_n , for all n -ary function symbols. Then, instead of translating a literal of the form $f(x_1, \dots, x_n) \approx y$ into the literal $\neg R_f(x_1, \dots, x_n, y)$, we translate it into the literal $R_n(f, x_1, \dots, x_n, y)$, treating f as a zero-arity symbol. The advantage of this translation is that instead of needing one totality axiom per relation symbol R_f with $f \in \Sigma_F$ we only need one per function symbol *arity* (among those found in Σ_F).⁷ The d -totality axioms then take the more general form

$$R_n(y, x_1, \dots, x_n, 1) \vee \dots \vee R_n(y, x_1, \dots, x_n, d)$$

where the variable y is meant to be quantified over the (original) function symbols in Σ_F . Note that the zero-arity symbols representing the original function symbols in the input are in addition to the domain constants, and of course never interact with them.⁸

6. Like Paradox, FM-*Darwin* performs a kind of sort inference in order to improve the effectiveness of symmetry breaking. Each function and predicate symbol of arity n in Σ is assigned a type respectively of the form $S_1 \times \dots \times S_n \rightarrow S_{n+1}$ and $S_1 \times \dots \times S_n$, where all sorts S_i are initially distinct. Each term in the input clause set is assigned the result sort of its top symbol. Two sorts S_i and S_j are then identified based on the input clause set by applying a union-find algorithm with the following rules. First, all sorts of different occurrences of the same variable in a clause are identified; second, the result sorts of two terms s and t in an equality $s \approx t$ are identified; third, for each term or atom

⁷ Consequently, this translation is actually applied for a given arity only if there are at least two symbols of that arity.

⁸ They are intuitively of a different sort S . Moreover, by the Herbrand theorem, we can consider with no loss of generality only interpretations that populate the sort S precisely with these constants, and no more.

of the form $f(\dots, t, \dots)$ the argument sort of f at t 's position is identified with the sort of t .

All sorts left at the end are assumed to have the same size. This way, when a sorted model is found (with all sorts having some size d), it can be translated into an unsorted model by an isomorphic translation of each sort into a single domain of size d . This implies that one can conceptually search for a sorted model and apply the symmetry breaking rules independently for each sort, and otherwise do everything else as described in the previous section. In addition to generally improving performance, this makes the whole procedure less fragile, as the order in which the constants are chosen for the symmetry breaking rules can have a dramatic impact on the search space.

7. Splitting clauses. Paradox and Mace2 use transformations that, by introducing new predicate symbols, can split a flat clause with many variables into several flat clauses *with fewer variables*. For instance, a clause of the form

$$P(x, y) \vee Q(y, z)$$

whose two subclauses share only the variable y can be transformed into the two clauses

$$P(x, y) \vee S(y) \quad \neg S(y) \vee Q(y, z)$$

where the predicate symbol in the *connecting* literal $S(y)$ is fresh. This sort of transformation preserves (un-)satisfiability. Thus, in this example, where the number of variables in a clause is reduced by from 3 to 2, procedures based on a full ground instantiation of the input clause set may benefit from of having to deal with the $O(2n^2)$ ground instances of the new clauses instead of $O(n^3)$ ground instances of the original clause.⁹

As it happens, reducing of the number of variables per clause is not necessary helpful in our case. Since (FM-)Darwin does not perform an exhaustive ground instantiation of its input clause set, splitting clauses can actually be counter-productive because it forces the system to populate contexts with instances of connecting literals like $S(y)$ above. Our experiments indicate that this is generally expensive unless the connecting literals do not contain any variables. Still, in contrast to Darwin, where in general clause splitting is only an improvement for ground connecting literals, for FM-Darwin splitting in all cases gives a slight improvement.¹⁰

8. Naming subterms. Clauses with deep terms lead to long flat clauses. To avoid that, deep subterms can be extracted and named by an equation. For instance, the clause set

$$P(h(g(f(x)), y)) \quad Q(f(g(z)))$$

can be replaced by the clause set

$$P(h_2(x, y)) \quad Q(h_1(x)) \quad h_2(x, y) = h(h_1(x), y) \quad h_1(x) = g(f(x))$$

⁹ A similar observation was made in [11] and exploited beneficially to solve planning problems by reduction to SAT.

¹⁰ In our experiments on the TPTP (Section 5.2) it helped to solve eight additional satisfiable problems.

where h_1 and h_2 are fresh function symbols. When carried out repeatedly, reusing definitions across the whole clause set, this transformation yields to shorter flattened clauses.

We tried some heuristics for when to apply the transformation, based on how often a term occurs in the clause set, and how big the flattened definition is (i.e., how much it is possible to save by using the definition). The only consistent improvement on TPTP problems was achieved when introducing definitions only for ground terms. This solves 16 more problems, 14 of which are Horn. Thus, currently only ground terms are flattened by default with this transformation in FM-*Darwin*.

5 Experimental Evaluation

5.1 Space Efficiency

Our reduction to clause sets encoding finite E -satisfiability is similar to, and indeed inspired by, the one in Paradox [9]. The most significant difference is, as we mentioned, that in Paradox the whole counterpart of our clause set $\mathcal{F}(M, d)$ is grounded out, simplified and fed into a SAT solver (Minisat). In our case, $\mathcal{F}(M, d)$ is fed directly to a theorem prover capable of deciding the satisfiability of function-free clause sets. This has the advantage of often being more space-efficient: in Paradox, as the domain size d is increased, the number of ground instances of a clause grows exponentially in the number of variables in the clause [9]. In contrast, in our transformation no ground instances of the clause set \mathcal{F} are produced. The subsets \mathcal{D} and \mathcal{E} do grow with the domain size d ; however, the number of clauses in $\mathcal{D}(d)$ remains constant in d while their length grows only linearly in d . The number of clauses in $\mathcal{E}(d)$, which are all unit, grows instead quadratically.

As far as preprocessing the input clause set is concerned then, our approach already has a significant space advantage over Paradox's. This is crucial for problems that have models of a relatively large size (more than 6 elements, say, for functions arities of 10), where Paradox's eager conversion to a propositional problem is simply unfeasible because of the huge size of the resulting formula. A more accurate comparison, however, needs to take the dynamics of model search into account. By using *Darwin* as the back-end for our transformation, we are able to keep space consumption down also during search. Being a DPLL-like system, *Darwin* never derives new clauses.¹¹ The only thing that grows unbounded in size in *Darwin* is the *context*, the data structure representing the current candidate model for the problem. With function-free clause sets the size of the context depends on the number of possible ground instances of input *literals*, a much smaller number than the number of possible ground instances of input *clauses*. In addition, our experiments show that the context basically never grows to its worst-case size.

The different asymptotic behaviours between FM-*Darwin* and Paradox can be verified experimentally with the following simple problem.

Example 4 (Too big to ground). Let p be an n -ary predicate symbol, c_1, \dots, c_n (distinct) constants, and x, x_1, \dots, x_n (distinct) variables. Then consider the clause set consisting

¹¹ Except for lemmas of which, however, it keeps only a fixed number during a derivation.

n	FM-Darwin			Mace4 Time	Paradox		
	Max. ctxt	Mem	Time		# Vars	# Clauses	Time
3	14	1	< 1	< 1	14	0	< 1
4	24	1	< 1	< 1	301	123	< 1
5	37	1	< 1	< 1	3192	534	< 1
6	53	1	< 1	< 1	46749	7919	< 1
7	72	1	< 1	178	823666	46749	12
8	94	1	5.1	Fail at size 7	Inconclusive, size ≥ 7		36
9	119	1	50	Fail at size 6	Inconclusive, size ≥ 5		9.6
10	147	1	566	Fail at size 4	Inconclusive, size ≥ 4		3.6

Table 1. Comparison of Darwin and Paradox on Example 4, for $n = 3, \dots, 9$. All **Time** results are CPU time in seconds. Specific column entries for **FM-Darwin**: **|Max. ctxt|** – maximum context size needed in derivation; **Mem** – required memory size in megabytes. Column entry for **Mace4**: “Fail at size d ” – Memory limit of 400 MB exhausted during search for a model with size d . Specific column entries for **Paradox**: **# Vars** – the number of propositional variables of the translation into propositional logic for domain size n ; **# Clauses** – likewise, the number of propositional clauses; “Inconclusive, size $\geq d$ ”: Paradox gave up after the time stated.

of the following $n \cdot (n - 1)/2 + 1$ unit clauses, for $n \geq 0$:

$$\begin{aligned}
& p(c_1, \dots, c_n) \\
& \neg p(x_1, \dots, x_{i-1}, x, x_{i+1}, \dots, x_{j-1}, x, x_{j+1}, \dots, x_n) \quad \text{for all } 1 \leq i < j \leq n
\end{aligned}$$

The first clause just introduces n constants. Any (domain-minimal) model has to map them to at most n domain elements. The remaining clauses force the constants to be mapped to pairwise distinct domain elements. Thus, the smallest model has exactly n elements. This clause set is perhaps the simplest clause set to specify a domain with n elements. \square

We ran the example for $n = 3, \dots, 10$ on FM-Darwin, Mace4 and Paradox and obtained the results in Table 1. These results confirm our expectations on FM-Darwin’s greater scalability with respect to space consumption. The growth of the (propositional) variables and clauses within Paradox clearly shows exponential behaviour. In contrast, Darwin’s contexts grow much more slowly.

5.2 Comparative Evaluation on TPTP

We evaluated the effectiveness of our approach on all the satisfiable problems of the TPTP 3.1.1 in comparison to Paradox 1.3 and Mace4.¹² All tests were run on Xeon 2.4Ghz machines with 1GB of RAM, with the imposed limits of 300s of CPU time and 512MB of RAM. FM-Darwin was run with the *grounded* learning option and with an upper limit of 500 lemmas (see [3] for more details on these options), Paradox and Mace4 in the default configuration.

¹² Since Darwin native input language is clausal, we used the eprover 0.91 to convert non-clausal TPTP problems into clause form.

Problem Type		Problems	FM-Darwin		Mace4		Paradox 1.3	
Horn	Equality		Solved	Time	Solved	Time	Solved	Time
no	no	607	575	3.9	394	3.0	578	0.9
no	yes	383	312	4.3	190	7.8	264	0.4
yes	no	65	51	17.5	37	0.2	59	2.1
yes	yes	196	136	7.0	181	3.6	182	5.3
all		1251	1074	5.1	802	4.1	1083	1.6

Table 2. Comparison of FM-Darwin, Mace4, and Paradox 1.3 over all satisfiable TPTP problems, also grouped based on being Horn and/or containing equality. **Solved Problems** gives the number of problems solved by a configuration, **Time** the average time used to solve these problems.

The results given in Figure 2 show that in terms of solved problems FM-Darwin significantly outperforms Mace4. Overall, our system is almost as good as Paradox, outperforming it over the non-Horn problems in the set. More precisely, FM-Darwin solves 328 problems that Mace4 cannot solve—Mace4 runs out of time for 169 problems and out of memory for the remaining ones—and solves 82 problems that Paradox can not solve—on all these problems Paradox runs out of memory or gives up. We sampled some of these problems and re-ran Paradox without memory and time limits, but to no avail. For problem NLP049-1, for instance, about 10 million (ground) clauses were generated for a domain size of 8, consuming about 1 GB of memory, and the underlying SAT solver could not complete its run within 15 minutes.

In contrast, on all problems FM-Darwin never uses more than 200 MB of memory, and in most cases less than 50 MB. In conclusion then, both the artificial problem in Example 4 and the more realistic problems in the TPTP library support our thesis that FM-Darwin scales better on bigger problems, that is, problems with a larger set of ground instances for non-trivial domain sizes.

On the other hand, Paradox and to a lesser extent Mace4 tend to solve problems faster than FM-Darwin. We expect, however, that the difference in speed will decrease in later implementations of our system as we refine and improve our approach further.

6 Conclusions

Recent years have seen considerable interest in procedures for computing finite models of first-order logic specifications. In this paper we overcome a major problem with established, leading methods—embodied by systems like Paradox and Mace4—which do not scale well with the required domain size of the (smallest) models. These methods are essentially based on propositional reasoning. In contrast, we proposed instead to reduce model search to a sequence of satisfiability problems made of function-free first-order clause sets, and to apply (efficient) theorem provers capable of deciding such problems.

In this paper we presented our approach in some detail and argued for its correctness. We then provided results from a comparative evaluation of our prover, Mace4 and Paradox, demonstrating that the expected space advantages do indeed occur. The evaluation also shows that FM-Darwin, our initial implementation of our approach built on

top of the *Darwin* theorem prover, is already competitive with state-of-the-art model builders.

We believe that the performance of FM-*Darwin* has still considerable room for improvement. One main opportunity of improvement is that currently there is no explicit symmetry breaking mechanism for function symbols of arity greater than zero. Another is that the disequality of domain elements is still explicitly axiomatized by ground axioms over the domain constants. In future work, we intend to explore the possibility of adapting existing first-order level symmetry breaking techniques to our method, and building-in equality over domain constants into FM-*Darwin*.

While FM-*Darwin* scales better memory-wise than the other systems considered, it generally struggles like all other finite model-finders with problems (such as the TPTP problem LAT053-1) whose smallest model is relatively large (20 or more elements). Increasing the scalability towards larger domain sizes is then certainly a main area of further research.

Acknowledgements. We thank the reviewers for their helpful comments.

References

1. Leo Bachmair, Harald Ganzinger, and Andrei Voronkov. Elimination of equality via transformation with ordering constraints. In Claude Kirchner and Hlne Kirchner, editors, *Automated Deduction — CADE 15*, LNAI 1421, Lindau, Germany, July 1998. Springer-Verlag.
2. Peter Baumgartner, Alexander Fuchs, and Cesare Tinelli. Implementing the model evolution calculus. *International Journal of Artificial Intelligence Tools*, 15(1):21–52, 2006.
3. Peter Baumgartner, Alexander Fuchs, and Cesare Tinelli. Lemma learning in the model evolution calculus. Submitted, Mai 2006.
4. Peter Baumgartner and Renate Schmidt. Blocking and other enhancements for bottom-up model generation methods. In *Proc. International Joint Conference on Automated Reasoning (IJCAR)*, 2006. To appear. Long version.
5. Peter Baumgartner and Cesare Tinelli. The Model Evolution Calculus. In Franz Baader, editor, *CADE-19 – The 19th International Conference on Automated Deduction*, volume 2741 of *Lecture Notes in Artificial Intelligence*, pages 350–364. Springer, 2003.
6. M. Bezem. Disproving distributivity in lattices using geometry logic. In *Proc. CADE-20 Workshop on Disproving*, 2005.
7. D. Brand. Proving theorems with the modification method. *SIAM Journal on Computing*, 4:412–430, 1975.
8. François Bry and Sunna Torge. A Deduction Method Complete for Refutation and Finite Satisfiability. In *Proc. JELIA*, LNAI. Springer, 1998.
9. Koen Claessen and Niklas Sörensson. New techniques that improve mace-style finite model building. In Peter Baumgartner and Christian G. Fermüller, editors, *CADE-19 Workshop: Model Computation – Principles, Algorithms, Applications*, 2003.
10. Hans de Nivelle and Jia Meng. Geometric resolution: A proof procedure based on finite model search. In *Proc. International Joint Conference on Automated Reasoning (IJCAR)*, 2006. To appear.
11. Henry Kautz and Bart Selman. Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search. In *Proceedings of the 13th National Conference on Artificial Intelligence*, Portland, OR, USA, 1996.

12. W. McCune. A davis-putnam program and its application to finite first-order model search: Qusigroup existence problems. Technical report, Argonne National Laboratory, 1994.
13. W. McCune. Mace4 reference manual and guide. Technical Report ANL/MCS-TM-264, Argonne National Laboratory, 2003.
14. John Slaney. Finder (finite domain enumerator): Notes and guide. Technical Report TR-ARP-1/92, Australian National University, Automated Reasoning Project, Canberra, 1992.
15. G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
16. Hantao Zhang. Sem: a system for enumerating models. In *IJCAI-95 — Proceedings of the 14th International Joint Conference on Artificial Intelligence, Montreal*, pages 298–303, 1995.