Contributions to

AI$_4$FM 2015

The 6th International Workshop on the use of
AI in Formal Methods

Andrius Velykis
Gudmund Grov
Leo Freitas (Eds.)

Edinburgh, Scotland, 1st September 2015
Co-located with AVoCS 2015
List of contributions

• Preface
  Andrius Velykis, Gudmund Grov, and Leo Freitas

• The Role of Human Creativity in Mechanized Verification (invited talk)
  J Strother Moore

• Event-B and cloud provers
  Alexei Iliasov, Paulius Stankaitis, and David Adjepon-Yamoah

• First-Order Logic for the Analysis of Programs on Weak Memory Models
  Alexei Lisitsa

• Nuprl’s Inductive Logical Forms
  Mark Bickford, Robert L. Constable, Rich Eaton, and Vincent Rahli

• VERIFOLIO: A Portfolio for Software Verification
  Yulia Demyanova, Thomas Pani, Helmut Veith, and Florian Zuleger

• Learning and Exploration in Automated Theorem Proving
  Moa Johansson

• Scaling Automated Theory Exploration
  Chris Warburton and Ekaterina Komendantskaya

• Compact Explanations of Why Malware is Bad
  Wei Chen, Charles Sutton, Andrew D. Gordon, David Aspinall, Igor Muttik, and Qi Shen

• Typed meta-interpretive learning for proof strategies
  Colin Farquhar, Gudmund Grov, Andrew Cropper, Stephen Muggleton, and Alan Bundy

• Tinker: A Graph Based Proof Strategy System
  Pierre Le Bras, Yuhui Lin, and Gudmund Grov
Preface

This report contains a collection of abstracts for all the talks in the 6th International Workshop on the use of AI in Formal Methods (AI4FM 2015).

The main goal of the AI4FM workshop series is to bring together researchers from formal methods, automated reasoning and AI; aiming to address the issue of how AI can be used to support the formal software development and verification process, including requirement analysis, modelling and proof.

This workshop has been held on the 1st of September, 2015 in Edinburgh, Scotland. AI4FM 2015 is co-located with the 15th International Workshop on Automated Verification of Critical Systems (AVoCS 2015). This is the 6th workshop in this series, with previous events having been held in: Newcastle (2010), Edinburgh (2011), Schloss Dagstuhl (Germany, 2012), Rennes (France, part of ITP 2013), and Singapore (part of FM 2014).

We would like to thank all workshop contributors and participants. Furthermore, we would like to extend our thanks to Altran, D-RisQ Software Systems, Formal Methods Europe (FME), and the Scottish Informatics and Computer Science Alliance (SICSA) for sponsoring student registrations to the workshop.

Andrius Velykis, Gudmund Grov, and Leo Freitas
(AI4FM 2015 workshop organisers)
The Role of Human Creativity in 
Mechanized Verification

J Strother Moore
University of Texas at Austin

(invited talk)

Abstract
When people solve a challenging problem they often “polish” the solution to make it easier to understand and communicate. This happens in all technical fields but in this talk I focus on its occurrence in formal verification, largely by way of an example problem solved interactively with students in one of my classes. Examples from industrial applications of formal methods include revisions to models to eliminate unnecessary complexity, revisions to variable orderings to shorten state space exploration, and the process by which a key inductively provable lemma was discovered. I claim that while this natural urge to polish is important to publication it is detrimental to progress: how do we automate the creative steps in verification?
Event-B and cloud provers

Alexei Iliasov, Paulius Stankaitis, David Adjepon-Yamoah
Newcastle University, UK

Abstract. We discuss the whys and hows of remotely managing a collection of automated theorem provers supporting verification of Event-B models in the Rodin Platform [3]. We report on the current state of the work, our general aspirations and a range of technical obstacles encountered so far.

1 Overview

Event-B [1] is one of more popular notations for model-based, proof driven specification. It offers a fairly high-level mathematical language based on FOL and ZF set theory and an economical yet expressive modelling notation centred around the notion of an atomic event - a form of before-after predicate. Leaving aside the methodological qualities of Event-B, one can regard an Event-B model as a high-level notation from which a number of proof obligations may be automatically derived. Proof obligations seek to establish properties like the preservation of invariant and satisfaction of refinement obligations. A model is deemed correct when all proof obligations are successfully discharged.

Event-B employs syntactic schemas for constructing its safety and refinement proof obligations. A number of provers, some tailor-made and some external, attempt to automatically discharge an open proof obligation. Failing this, a user may provide proof hints (but not a complete proof script) to direct a combination of built-in rewrite procedures and automatic provers.

There was a concerted effort, funded by a succession of EU research projects, to make Event-B and its toolkit appealing and competitive in an industrial setting. One of the lessons of this mainly positive exercise is the general aversion of industrial users to interactive proofs. It is possible in principle (although hardly realistic in the context) to learn, through experience and determination, the ways of underlying verification tools and master refinement and decomposition to minimize proof effort. The methodological implications of avoiding manual proofs are far more serious: building a good model is always a trial and error process; any non-trivial design necessitates re-starting from a scratch or doing considerable model refactoring (i.e., re-arranging refinement layers). This means redoing manual proofs which makes time spent proving dead-end efforts seem like pointlessly wasted. Hence, proof-shy engineers too often don’t make a good use of formal specification stage as they tend to hold on to an initial, often incoherent design.

Proof economy is thus one of the worthy goals to explore. We propose to attack the problem from several directions at once.
First, we want to change the way proofs are done, at least in an industrial setting. Instead of doing an interactive proof - something that is an inherently one-off effort - we shall invite users to write and proof a schematic lemma that is strong enough to discharge an open proof obligation. Such a lemma may not refer to any model variables or types and is, in essence, a property supporting axiomatization of Event-B mathematical language. If such a lemma cannot be found or seems to difficult to prove, the model must be changed. Experience suggests that a modelling project is likely to have a fairly distinctive usage of data types and mathematical constructs; we conjecture that this leads to a distinctive set of supporting lemmas. We also hypothesise that such distinctnesses is pronounced and dictated by modeller’s experience and background as well as model subject domain. We have also observed that the style of informal requirements - structured text, hierarchical diagram, structural diagram - has an impact on a modelling style.

A schematic lemma is a tangible and persistent outcome of any modelling effort, even an abortive one. The ‘distinctnesses’ hypothesis suggests that such a lemma is likely to be useful in a next iteration and, by an extension, there is a point when all relevant lemmas are collected; then modelling, in a given context, becomes completely free of interactive proofs. We intend to demonstrate that such a point is reachable at least for a number of non-trivial existing models.

One problem we foresee is the accumulation of lemmas of which majority could be irrelevant outside of some narrow context. Adding an extra lemma in a proof context generally makes proof more difficult. It could conceivably reach a point where nothing may be proven due to the sheer number of supporting properties. Some form of filtering is thus paramount. From the outset, we make set of support lemmas bound to a specific application and private to a group of developers. To filter out irrelevant lemmas within such context we are looking into state-of-the-art techniques in hypothesis filtering. At the very basic level, we start with filtering by user-defined template expressions where a lemma is included only if the goal or some hypothesis predicate matches a given template.

Our second direction of attack is an extensive use of cheap computational resources made available by data centres and cloud computing. Provers-as-a-service is a natural direction given that provers are CPU and memory intensive; at the same time, running a collection of (distinct) provers on a same conjecture is a trivial and fairly effective way to speed up proof given plentiful resources. Usability perception of interactive modelling methods such as Event-B is sensitive to peak performance when a burst of activity (new invariant) is followed by a relatively long period of idling (modeller thinking and entering model). The cloud paradigm, where only the actual CPU time is rented, seems well suited to such scenario!}

From the technical perspective one low hanging fruit was the absence of interface between Event-B and TPTP provers. Thus hosting a collection TPTP provers on a cloud and providing an interface to them seemed a promising direction. To simplify translation we decided to use Why3 [2] umbrella prover that offers a far more palatable input notation and also supports SMT-LIB provers. A
plug-in to the Rodin Platform was realised to map between the Event-B mathematical language and Why3 theory input notation (we don’t make use of its other part - a modelling language notation). The syntactic part of the translation is trivial: just one Tom/Java class. The bulk of the effort is in the axioms and lemmas defining the properties of the numerous Event-B set-theoretic constructs. We are already at a stage where there is a working prototype able to discharge (via provers like SPASS and Alt-Ergo) a number of properties that previously required interactive proof. At the same time, we realize that axiomatization of a complex language like Event-B is likely to be an ever open problem. It is apparent that different provers prefer differing styles of operator definitions: some perform better with an inductive style (the size of an empty set is zero, adding one element to a set increases its size by one) others prefer regress to already known concepts (here exists a bijection such that ...). Since we don’t know how to define an optimal axiomatization, even for any one given prover, we offer an open translator with which a user may define, with as many cross-checks as practically reasonable, a custom embedding of Event-B into Why3.

Doing proofs on a cloud opens possibilities that we believe were previously not explored, outside, perhaps, prover contests results. The cloud service keeps a detailed record of each proof attempt along with (possibly obfuscated) proof obligations, supporting lemmas and translation rules. There is a fairly extensive library of Event-B models constructed over the past 15 years and these are a ready of source of proof obligations. Some of these come from academia and some from industry. We are now starting to put models through our prover plug-in in order to collect some tens of thousands of proof obligations. One immediate point of interest is whether one can train a classification algorithm to make useful prediction of relative prover performance. If such a prediction can yield statistically significant results, prover call order may be optimized to minimize resource utilization while retaining or improving average proof time.

References

First-Order Logic for the Analysis of Programs on Weak Memory Models

Alexei Lisitsa

Department of Computer Science, The University of Liverpool
A.Lisitsa@liverpool.ac.uk

Abstract. In this paper we present the applications of automated reasoning for classical first-order logic for the analysis of safety properties of programs running on weak memory models. The reachability within a program state space is modelled by derivability in first-order logic, and the safety verification is reduced to the refutation of a first-order formula, which is then tackled by a generic finite model finder. We present a detailed case study and report on further experimental evidence of the feasibility of the approach.

In this paper we address the problem of verification of programs running on a weak, or relaxed memory models. These models [2, 1, 7] introduce some re-ordering of the memory operations for the sake of efficiency of program execution on the modern multi-processor and multi-core architectures. There has been considerable effort to automate the verification of programs working on relaxed memory models, see e.g. [2, 1, 7].

In this paper we aim to apply the methods based on direct encoding of states and transitions of the systems of interest in classical first order logic in a way that computations of the system are faithfully modelled by the derivations in first-order logic. The safety verification technique using first-order encoding and finite model finding has been originated in earlier work on the verification of cryptographic protocols [15, 4, 5] and more recently it has been extended [9–12] to the wider classes of infinite-state and parameterized verification tasks.

The general outline of the approach is as follows: 1) the states of a system to be verified are encoded by terms (tuples of terms) over an appropriate first-order vocabulary; 2) the transitions of the system are encoded within a first-order theory $T$ such that $T \vdash R(t_1, t_2)$ holds iff the state encoded by $t_2$ is reachable from the state encoded by $t_1$ (alternatively unary reachability predicate $R$ may be used with the property $T \vdash R(t)$ iff the state encoded by $t$ is reachable from some initial state; 3) then one applies an automated theorem disprover (model finder) to establish non-reachability (safety) and/or an automated theorem prover to establish reachability (non-safety). In the papers above it was shown how to proceed with such an approach in various contexts, in most of which the systems to be verified are specified by conditional term rewriting systems (possibly modulo an equational theory) and the sets of initial and non-safe states are presented by regular sets of terms. It was shown that not only
such techniques are theoretically relatively complete wrt the methods using regular invariants (e.g. variants of regular model checking) but are also compared favourably in the practical experiments.

The definitions of concurrent programs and weak memory models are taken largely from [8]. A concurrent program is a tuple $P = (P, M, D, I)$ defined by a data domain $D$, a finite set of processes $P = \{p_1, \ldots, p_n\}$, and a finite set of memory locations $M = \{m_1, \ldots, m_k\}$. The memory locations can hold values from $D$. A function $I : M \rightarrow D$ represents an initial content of the memory. Each process $p_i$ is defined by a finite set $L(p_i)$ of control locations, an initial location $l_0(p_i)$ and a set of transitions $T(p_i)$. Each transition $t$ from $T(p_i)$ is an element of $L(p_i) \times O \times L(p_i)$ denoted by $(l, op, l')$. Here $O$ is a set of operations which contains the following memory operations:

- $store(p_i, m_j, d)$, a process $p_i$ stores value $d \in D$ to memory location $m_j$
- $load(p_i, m_j, d)$, a process $p_i$ loads the value stored in $m_j$ and checks that its value is $d$. If the stored value is different from $d$, the transition is not possible.

In PSO memory model the order between writing (store) operations to different memory locations is not preserved and during the execution and one operation may overtake another. The PSO memory model is formalized here by using store-buffer semantics [2]. In such a semantics the relaxations are modelled by assigning a buffer (a queue) $b_{ij}$ for every pair $(p_i, m_j)$, $p_i \in P$, $m_j \in M$. The buffer $b_{ij}$ keeps the store operations issued by process $i$ concerning the memory location $m_j$ which have not taken yet effect on the memory. In order to ensure a particular order of execution of the programs on weak memory models the fence commands. We extend the language of concurrent programs with a variant of fence command, that is $commit(p_i, m_j)$ [8], with an effect to commit to the memory location $m_j$ the full content of a buffer $b_{ij}$.

Safety Verification Problem we address in this paper is formulated as follows

**Given:** A concurrent program $P = (P, M, D, I)$ and a finite set $U \subseteq \Pi_{i=1}^n L(p_i)$ of unsafe vectors of local states;

**Question:** Is it true that none of states $(L, V, B)$ with $L \in U$ is reachable by execution of $P$ under PSO semantics?

In general the number of program states for a program with weak memory model semantics may be infinite as the buffers may grow indefinitely during the execution, so in general it is an infinite-state verification problem. Nevertheless in [2] it was shown to be decidable with very high non-primitive recursive complexity upper bound.

We propose a translation $P \mapsto \Phi_P$ which takes a concurrent program $P$ and yields a first-order theory $\Phi_P$ such that the following theorem holds true.

**Theorem 1.** A program state $S$ of a concurrent program $P$ is reachable, i.e. $(l_0, I, B_0) \Rightarrow_P S$, iff $\Phi_P \vdash R(\tau(S))$. 
Using this translation we present the case studies in verifications of the variants of classical mutual exclusion protocols (Burn’s, Paterson’s and Dekker’s) running under PSO semantics. In all cases when protocols are incorrect under PSO, the execution traces violating the safety are extracted from automated resolution proofs obtained by Prover9. When the protocols are fixed by inserting appropriate fence commands their safety is proved by a countermodel found by Mace4 model finder. The experimental data and results can be found in [9]. The proposed method is simple and allows to re-use existing generic methods and tools developed for the automated reasoning in first-order logic. Further research is required to investigate the applicability limits of the approach and to compare it with the existing alternatives.

References

1. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, Ahmed Rezine, Counter-Example Guided Fence Insertion under Weak Memory Models, TACAS 2012
7. Michael Kuperstein, Martin Vechev, Eran Yahav, Partial-Coherence Abstractions for Relaxed Memory Models, PLDI 2011
9. A. Lisitsa Verification via countermodel finding http://www.csc.liv.ac.uk/~alexei/countermodel/
13. A. Lisitsa, First-Order Logic for the Analysis of Programs on Weak Memory Models, extended version of this short paper, available at http://www.csc.liv.ac.uk/~alexei/countermodel/AI4FM-EXT.pdf from 03.08.2015
Nuprl’s Inductive Logical Forms

Mark Bickford, Robert L. Constable, Rich Eaton and Vincent Rahli*

Cornell University

Abstract

For more than a decade, we have been working on specifying, verifying, and synthesizing asynchronous distributed protocols using the Nuprl proof assistant. Only recently we have been able to do so for complicated protocols such as Paxos. This has been made possible thanks to the level of abstraction of our specification language called the Logic of Events (LoE). This paper discusses our main automation tool, namely our Inductive Logical Forms (ILFs), which are first order formulas that characterize the responses of a system to events in terms of observations made at causally prior events.

Nuprl’s type theory. The Nuprl proof assistant [12, 3] implements a type theory called Constructive Type Theory (CTT), which is an extensional dependent type theory à la Martin-Löf. It is based on an untyped functional programming language à la Curry. Nuprl has a rich type theory including dependent product and sum types, identity (or equality) types, a hierarchy of universes, disjoint union types, W types, quotient types [14], set types, union and (dependent) intersection types [17, 22], image types [23], PER types [4], approximation and computational equivalence types [19, 28], and partial types [16].

Type checking is undecidable but in practice this is mitigated by type inference and checking heuristics implemented as tactics. Moreover, to avoid proving well-formedness conditions (i.e., doing type checking) altogether, we often do untyped reasoning using Howe’s computational equivalence relation (an observational congruence, which we write as ∼) [19, 28]. For example, we can prove that for all terms f and t (t need not be a list), map(f, t) @ nil ∼ map(f, t). We can then rewrite map(f, t) @ nil into map(f, t) anywhere in a sequent without having to prove any well-formedness condition.

Allen developed a semantics for Nuprl where types are defined as partial equivalence relations (PERs) on closed terms [2, 1, 16]. We have implemented this semantics in Coq and verified a large number of Nuprl’s inference rules [6, 7] (Nuprl’s consistency follows from the fact that its inference rules are valid w.r.t. Allen’s PER semantics and that False is not inhabited).

We have also recently turned Nuprl into a fully intuitionistic theory following Brouwer’s principles. Using our Coq framework, we have proved Brouwer’s continuity principle for numbers [27]. Following Dummett’s “standard” classical proof [17, pp.55], we have also proved (in Prop and using classical axioms) the truth of several bar induction rules (see, e.g., [21, 11, 31]), such as bar induction on decidable bars for free choice sequences of numbers [26].

By default the Nuprl system is distributed and runs in the cloud. Alternatively, Nuprl can run locally using one of our virtual machines available at http://www.nuprl.org/vms/ Nuprl is composed of several processes: database and library processes to store and access definitions, lemmas, and proofs; refiner processes to apply inference rules; and editor processes, which are user interfaces. It also allows several users to simultaneously edit a proof, and users to simultaneously refine several subgoals in a proof.

Nuprl led to other similar proof assistants such as MetaPRL [18] and JonPRL [20]. It has been used over the years to both formalize mathematical results and develop verified software.

∗This work was partially supported by the DARPA CRASH project, award number FA8750-10-2-0238, by the SnT, and by the National Research Fund Luxembourg (FNR), through PEARL grant FNR/P14/8149128.
For example, we have recently proved a completeness result of intuitionistic first-order logic \cite{13}, and we have built distributed systems including a verified ordered broadcast service \cite{33,30,32}.

**The Logic of Events.** To specify, verify, and synthesize asynchronous distributed protocols, we use two languages, both implemented in Nuprl: (1) a high-level specification language called the Logic of Events (LoE) \cite{8,10} to specify and reason about the information flow of distributed program runs; and (2) a low-level programming language, the General Process Model (GPM) \cite{9}, to implement these information flows.

LoE, related to Lamport’s notion of causal order \cite{22}, was developed to reason about events occurring in the execution of a distributed system, where an event is an abstract entity corresponding to the receipt of a message; the message is called the *primitive information* of the event. An event happens at a specific point in space/time. The space coordinate of an event is called its location, and the time coordinate is given by a well-founded causal ordering on events that totally orders all events at the same location. Using LoE one can describe systems in terms of the causal relations among events and (ultimately) their primitive information. LoE has been used among other things to verify consensus protocols \cite{33} and cyber-physical systems \cite{5}.

We have also developed a programming language called EventML, to automatically generate both LoE specifications and GPM programs from EventML specifications \cite{29}. Once we have extracted the semantic meaning of an EventML specification in terms of a LoE formula \( F \) and a GPM program \( P \), we automatically prove that \( P \) satisfies \( F \). It remains to interactively prove that the LoE formula \( F \) satisfies the desired correctness properties.

To reason about a protocol in LoE, we reason about its possible runs. An *event ordering* is an abstract representation of one run of a distributed system; it provides a formal definition of a *message sequence diagram* as used by systems designers. It is a structure consisting of: (1) a set of events; (2) a function \( \text{loc} \) that associates a *location* with each event; (3) a function \( \text{info} \) that associates primitive information with each event; and (4) a well-founded *causal ordering* relation, \(<\), on events \cite{23}. We express system properties as predicates on event orderings. A system satisfies such a property if every execution satisfies the predicate.

The message sequence diagram on the right depicts a simple event ordering with events \( e_1 \) and \( e_3 \) happening at location \( L_1 \), and \( e_2 \) at location \( L_2 \). Event \( e_1 \) happens causally before \( e_2 \), which happens causally before \( e_3 \). We write \( e_1 < e_2, e_2 < e_3 \), and \( e_1 <_{\text{loc}} e_3 \).

In LoE, we specify systems by defining and combining *event observers* \cite{8} (which can be regarded as the combinations of *event recognizers* and *event handlers*). An event observer is a function that assigns to any event ordering \( eo \) and event \( e \) in that event ordering \( eo \), an unordered bag of outputs observed (or produced) at \( e \). For example, the following observer recognizes every event and observes its location: \( \lambda eo. \lambda e. \{ \text{loc}(e) \} \). We also have *primitive observers* to, e.g., run two processes in parallel or to build state machines.

We reason about event observers in terms of the *event observer relation*, which relates events, observers, and observations: we say that the observer \( X \) observes \( v \) at event \( e \) (in an event ordering \( eo \), and write \( v \in X(e), \) if \( v \) is a member of the bag \( X(eo) \).

Formally verifying distributed protocols is not trivial and can be time consuming. Our main automation tool to assist us in this task is called an Inductive Logical Form (ILF).

**Inductive Logical Forms.** An ILF is a first order formula that characterizes the responses of a system to events in terms of observations made at causally prior events. For example, in Paxos \cite{24,34}—a state machine replication protocol, if a leader \( L \) decides that slot \( n \) is to be filled with command \( c \), then that means that \( c \) was proposed to the leader \( L \) at an earlier event.

ILFs are automatically generated from observers using logical simplifications, and chara-
terizations of the LoE combinator. For example, one of the simplest but subtle such characterizations is the one for our parallel combinator \( "[\_\_\_\_\_]" \), which allows one to run two processes in parallel: \( v \in X \upharpoonright Y(e) \iff \langle v \in X(e) \lor v \in Y(e) \rangle \). This says that \( v \) is produced by \( X \upharpoonright Y \) iff it is produced by either of its components (see 29 for more details).

Given an observer \( X \), i.e. an LoE specification, we wrote a program that starts with a formula of the form \( v \in X(e) \), and keeps on rewriting it using equivalences such as the one presented above (and also applies various logical simplifications), to finally generate a formula of the form \( v \in X(e) \iff C \), where \( C \) is a complete declarative characterization of \( X \)'s outputs. Finally, we have built a proof tactic that automatically proves such double implications.

Rewriting using such formulas we can easily trace back the outputs of a distributed system to the states of its state machines and to its inputs. It turns out that when proving safety properties of distributed systems, much of the effort is spent tracing back outputs to inputs. ILFs helped us automate this reasoning process.

References

VERIFOLIO: A Portfolio for Software Verification

Yulia Demyanova, Thomas Pani, Helmut Veith, and Florian Zuleger*

Vienna University of Technology

Abstract. In recent work [5], we study empirical metrics for software source code, which can predict the performance of verification tools on specific types of software. Our metrics comprise variable usage patterns, loop patterns, as well as indicators of control-flow complexity and are extracted by simple data-flow analyses. We demonstrate that our metrics are powerful enough to devise a machine-learning based portfolio solver for software verification. We show that this portfolio solver would be the (hypothetical) overall winner of both the 2014 and 2015 International Competition on Software Verification (SV-COMP). This gives strong empirical evidence for the predictive power of our metrics and demonstrates the viability of portfolio solvers for software verification.

1 Introduction

The success and gradual improvement of software verification tools in the last two decades is a multidisciplinary effort – modern software verifiers combine methods from a variety of overlapping fields of research including model checking, static analysis, shape analysis, SAT solving, SMT solving, abstract interpretation, termination analysis, pointer analysis etc. The mentioned techniques all have their individual strengths, and a modern software verification tool needs to pick and choose how to combine them into a strong, stable and versatile tool. The trade-offs are based on both technical and pragmatic aspects: many tools are either optimized for specific application areas (e.g. device drivers), or towards the in-depth development of a technique for a restricted program model (e.g. termination for integer programs). Recent projects like CPA [2] and FrankenBit [9] have explicitly chosen an eclectic approach which enables them to combine different methods more easily.

There is growing awareness in the research community that the benchmarks in most research papers are only useful as proofs of concept for the individual contribution, but make comparison with other tools difficult: benchmarks are often manually selected, handcrafted, or chosen a posteriori to support a certain technical insight. Oftentimes, neither the tools nor the benchmarks are available to other researchers. The annual International Competition on Software Verification (SV-COMP, since 2012) [1] is the most ambitious attempt to remedy this situation. Now based on more than 5,500 C source files, SV-COMP has the most diverse and comprehensive collection of benchmarks available, and is a natural starting point for a more systematic study of tool performance.

In recent work [5], we demonstrate that the competition results can be explained by intuitive metrics on the source code. In fact, the metrics are strong enough to enable us to construct a portfolio solver which would (hypothetically) win SV-COMP 2014 and 2015.

* The authors are supported by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF) and by the Vienna Science and Technology Fund (WWTF) through grants PROSEED, ICT12-059, and VRG11-005.
Here, a portfolio solver is a SW verification tool which uses heuristic preprocessing to select
one of the existing tools [8].

Of course it is pointless to let a portfolio solver compete in the regular competition
(except, maybe in a separate future track), but for anybody who just wants to verify soft-
ware, it provides useful insights. Portfolio solvers have been successful (and controversial) in
combinatorially cleaner domains such as SAT solving [14], quantified boolean satisfiability
(QSAT) [12], answer set programming (ASP) [7], and various constraint satisfaction prob-
lems (CSP) [8,10]. In contrast to software verification, in these areas constituent tools are
usually assumed to be correct.

As an approach to software verification, portfolio solving brings interesting advantages:
(1) a portfolio solver optimally uses available resources, (2) it can avoid incorrect results of
partially unsound tools, (3) machine learning in combination with portfolio solving allows
us to select between multiple versions of the same tool with different runtime parameters,
(4) the portfolio solver gives good insight into the state-of-the-art in software verification.

To choose the software metrics, we consider the zoo of techniques discussed above along
with their target domains, our intuition as programmers, as well as the tool developer reports
in their competition contributions. The obtained metrics are naturally understood in three
dimensions that we only motivate informally here:

1. **Program Variables.** Does the program deal with machine or unbounded integers? Are
   the ints used as indices, bit-masks or in arithmetic? Dynamic data structures? Arrays?
   Interval analysis or predicate abstraction?

2. **Program Loops.** Reducible loops or goto programs? For-loops or ranking functions?
   Widening, loop acceleration, termination analysis, or loop unrolling?

3. **Control Flow.** Recursion? Function pointers? Multithreading? Simulink or complex branch-
   ing?

Our hypothesis is that precise metrics along these dimensions allow us to predict tool
performance. The challenge lies in identifying metrics which are predictive enough to under-
stand the relationship between tools and benchmarks, but also simple enough to be used in
a preprocessing and classification step. In [8] and [11] we describe metrics which correspond
to the three dimensions sketched above, and are based on simple data-flow analyses.

Our algorithm for the portfolio is based on machine learning (ML) using support vector
machines (SVMs) [3,4] over the metrics defined above. Figure 1 depicts our experimental
results on SV-COMP’15: Our tool TP is the overall winner and outperforms all other tools.

A machine-learning based method for selecting model checkers was previously introduced
in [13]. Similar to our work, the authors use SVM classification with weights. Our approach is
novel in that our benchmark is publicly available, and we describe the verification properties
and the machine learning weighting function. Furthermore, we use a larger set of verification
tools (22 tools vs. 3) and our benchmark is not restricted to device drivers and is 10 times
larger (49 MLOC vs. 4 MLOC in [13]).

While portfolio solvers are important, we also think that the software metrics we define
in this work are interesting in their own right. Our results show that categories in SVCOMP
have characteristic metrics. Thus, the metrics can be used to 1) characterize benchmarks
not publicly available, 2) understand large benchmarks without manual inspection, 3) un-
derstand presence of language constructs in benchmarks.

Summarizing, our work makes the following contributions:

- We define software metrics along the three dimensions – program variables, program
  loops and control flow – in order to capture the difficulty of program analysis tasks.
We develop a machine-learning based portfolio solver for software verification that learns the best-performing tool from a training set.

We experimentally demonstrate the predictive power of our software metrics in conjunction with our portfolio solver on the software verification competitions SV-COMP’14 and SV-COMP’15.

Fig. 1: Experimental results for the eight best competition participants in SV-COMP’15 Overall, plus our portfolio TP, given as arithmetic mean of 10 experiments on randomly selected 40% subsets selected for testing. The first row shows the Overall SV-COMP score and beneath it the runtime in minutes. We highlight the Overall gold, silver, and bronze medal in dark gray, light gray and white-bold font, respectively. The second row shows the number of gold/silver/bronze medals won in individual categories.

<table>
<thead>
<tr>
<th></th>
<th>blast</th>
<th>cascade</th>
<th>cbmc</th>
<th>epachecker</th>
<th>predate</th>
<th>smack</th>
<th>ultimakaj</th>
<th>ulcseq</th>
<th>TP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall</td>
<td>737</td>
<td>806</td>
<td>684</td>
<td>2228</td>
<td>389</td>
<td>1542</td>
<td>1215</td>
<td>273</td>
<td>2511</td>
</tr>
<tr>
<td>Medals</td>
<td>1/0/0</td>
<td>0/0/0</td>
<td>1/1/1</td>
<td>2/1/5</td>
<td>1/0/1</td>
<td>2/1/1</td>
<td>0/2/0</td>
<td>0/0/0</td>
<td>1/6/1</td>
</tr>
</tbody>
</table>

References

Learning and Exploration in Automated Theorem Proving

Moa Johansson
Chalmers University of Technology

Abstract

This note describes a new project with the purpose of combining the advantages of statistical machine learning with those of symbolic methods for lemma discovery, such as theory exploration. We can thus better exploit the resources in large electronic proof libraries to gain heuristic knowledge to, for example, create more efficient lemma discovery techniques or provide hints as to which induction scheme to try in an automated prover.

1 Introduction and Motivation

Machine learning and pattern recognition techniques are statistical methods which can be used for discovering similarities in data. Statistical methods are well suited for fast processing of big libraries and are generally tolerant to noise. While such statistical methods focus on extracting information from large volumes of data, they lack in capacity for conceptualisation. They can, for example, tell that a correlation between two proof patterns exists, but not why this happens nor formulate any conceptual proof hints for similar situations. Theory exploration techniques on the other hand, are concerned with exactly that: the discovery of new lemmas in new theories [5, 11, 1, 6]. However, theory exploration techniques may not cope well with very large theories when the search space becomes too big. We thus want to combine these two techniques exploiting the strengths of both.

For example, suppose we try to prove a new conjecture. We can use pattern recognition and data mining to detect if some similar theorem has been proved before, and if so, extract information about which proof technique and which lemmas were used. This information can then be fed into a theory exploration system to restrict its search space and produce suggestion of, for example, analogous lemmas applicable to the new conjecture. We have done some preliminary work in this area for the ACL2 proof assistant [4]. We used clustering methods to decide which theorems and conjectures were similar to each other based on their term structure, which functions were involved, and how similar the definitions of those functions were. In the context of inductive theorem proving, information about proofs of similar conjectures can help the prover to for example make a choice as to which induction scheme to use.

2 Machine Learning in Automated Reasoning

The main application of machine learning methods in the context of theorem proving and automated reasoning has been premise selection for first-order provers in large theories with many background facts. If presenting such a first-order prover with too many premises, its speed of operation will deteriorate. If given too few, the problem at hand might become unsolvable. Premise selection methods using machine learning techniques can help by using information from previous proofs to find correlations between theorem statements and the facts used in their proofs. When presented with a new conjecture, a suitably sized subset of premises most likely to be useful are passed to the prover. This has been implemented in for instance the provers E and SPASS [3, 9, 13]. The interactive proof assistant Isabelle allows calling external provers through the Sledgehammer tool [12]. As Isabelle’s proof library is very large, Sledgehammer uses this kind of relevance filtering to select facts estimated to be most relevant from the entire Isabelle library. The relevance filter is implemented with a naive Bayes learning algorithm [8]. The most commonly used learning algorithms for automated theorem proving integration seem to be fast but simple algorithms like naive Bayes and k-nearest neighbours. In a theorem prover, particularly interactive ones like Isabelle, the learning of new facts should interfere as little as possible with the user experience, which is presumably why these fast algorithms have been preferred.

The success and efficiency of machine learning algorithms is dependent on among other things feature selection, in our context: what characteristics of a conjecture or theorem we should focus on when trying
to assess their similarity. These features are typically translated into numerical values, and the resulting vector or matrix is used for learning. Different machine learning algorithms may work better or worse with different kinds of features, different numbers of features and so on. The most common features used in the context of theorem proving is simply which symbols occur the statements. However, many other features has been explored in various works, such as types, subterms, information about theories in which the statement occurs, just to name a few (see [7] for an overview). Recently, Kalinszyk et al. proposed to also use features based on matching and unification [7], thus basically relating statements through their generalisations. If such a generalisation is not present in the corpus, they propose to heuristically introduce such terms. They conclude that statements that have a common generalisations often have similar proofs, and show that this can improve premise selection.

3 Learning for Lemma Discovery and Inductive Proofs

We are initially interested in two areas where we believe that machine learning techniques further can assist automated theorem provers: selection of induction schemes and restricting the search space for theory exploration.

3.1 Selecting Induction Schemes

We recently added support for recursion induction\(^1\) in our theory exploration system Hipster for Isabelle/HOL [10]. We found that certain properties that Hipster could not previously prove with structural induction were possible to automate with a simple switch to recursion induction. The prover now has a choice as to which induction to apply, structural- or recursion induction. While is power is increased, the search space becomes larger and the prover is sometimes slower. The choice of induction scheme should depend on which functions are present in the conjecture, and how they are defined. We expect that machine learning can help us choosing which kind of induction to try first:

1. A new conjecture that involves functions which often has occurred in other theorems whose proof required e.g. recursion induction are likely to also require recursion induction.

2. Given a new function \(f\), and suppose its definition is judged similar to an existing function, \(g\). Suppose further that many theorems involving the function \(g\) use a particular induction scheme. It may then make sense to try the same induction scheme also for conjectures about \(f\).

The features required for realising (1) should simply be the standard symbol occurrence features used in for instance Sledgehammer. For (2) we need features also capturing the similarities in the structure of a function definition, in particular the functions recursive structure.

3.2 Theory Exploration

The problem we are interested in is different from premise selection for first-order provers. We want to be able to construct missing interesting lemmas, rather than just selecting from a given corpus, using theory exploration. A theory exploration system is typically given a set of functions, datatypes and constants and generates candidate terms using random testing, followed by automated proofs. Only theorems with non-trivial proofs are deemed interesting and presented to the user. Like automated first order provers, the performance of theory exploration systems can degrade if given too many functions as input at once (as a rough estimate, more than 20-25 can be problematic). Some areas where we think machine learning can help theory exploration include:

1. Discovering lemmas by analogy from similar proofs.

2. Speculating new properties by analogy to properties about similar functions.

3. Discovering conjectures bridging theories between similar datatypes.

For (1), we expect to follow a similar approach taken in our previous work [4], where we cluster new conjectures with existing theorems and their proofs, using features based on term structure and the similarity of functions appearing in each. The hypothesis is that lemmas used in the proof of a similar theorem can be used to extract an analogous lemma for our new conjecture. An option here is to abstract over the structure of an existing lemma, creating one or several schemas, which can then be given to a theory exploration system. Search will be required at this stage, but the search space is reduced compared to searching through all possible terms. Of interest here are also techniques similar to those suggested\(^2\).

---

\(^1\)Induction following the termination order of a function in the conjecture, rather than structural induction over datatypes.
by Kalinszyk et al. [7], if we detect several theorems in the training data have a common generalisation, such lemmas could be speculated and proofs attempted.

For (2), we expect to cluster similar function definitions together, with the hypothesis that if a function \( f \) is similar to a function \( g \), and we know some properties about \( f \), then it is worth exploring if \( g \) has some similar properties. Of course, this will not always be the case, but at least gives some guidance as to which properties might hold about a new function, thus reducing the search space.

For (3), we suppose that we have two theories about two datatypes and functions on these, and furthermore, that some properties have been proved over the two datatypes. If many of these properties are similar, it is worth speculating some bridging lemmas to connect the two theories, for instance converting one datatype into the other.

### 3.3 Inductive Proofs to Learn From

Good training data is crucial for machine learning. In addition to the existing libraries for interactive provers such as Isabelle, we have started to collect a set of benchmarks specifically for inductive provers [2]. The TIP (Tons of Inductive Problems) benchmark suite is expressed in a syntax which is an extension of SMT-LIB and currently contains a few hundred problems (we invite anyone interested to submit additional benchmarks to us). We plan to further complement this format with support for expressing at least some high-level description of proof-plans, e.g. what induction scheme and what lemmas the prover has used. This would be very useful for future machine learning applications.

### References


Scaling Automated Theory Exploration

Chris Warburton and Ekaterina Komendantskaya
University of Dundee,
http://tocai.computing.dundee.ac.uk

Abstract. We investigate the theory exploration (TE) paradigm for computer-assisted Mathematics and identify limitations and improvements for current approaches. Unlike the theorem-proving paradigm, which requires user-provided conjectures, TE performs an open-ended search for theorems satisfying given criteria. We see promise in TE for identifying new abstractions and connections in libraries of software and proofs, but realising this potential requires more scalable algorithms than presently used.

1 Introduction

Given a signature $\Sigma$ and a set of variables $V$, we call the pair $(\Sigma, V)$ a theory and use theory exploration (TE) to refer to any process $(\Sigma, V) \xrightarrow{TE} \text{Terms}(\Sigma, V)$ for producing terms of the theory which are well-formed, provable and satisfy some criterion referred to as “interesting”. These conditions give rise to the following questions, which we use to characterise TE systems:

- **Q1** How do we generate terms?
- **Q2** How do we guarantee well-formedness?
- **Q3** How do we prove terms?
- **Q4** What is considered “interesting”? 

Early implementations like THEOREMA [2] provided interactive environments, similar to computer algebra systems and interactive theorem provers, to assist the user in finding theorems. In this setting, terms are formed by the user in whichever way they find interesting, whilst the software provides support for Q2 and Q3.

Subsequent systems have investigated automated theory exploration, for tasks such as lemma discovery [7]. By removing user interaction, Q1 and Q4 must be solved by algorithms. In existing systems these are tightly coupled to improve efficiency, which makes it difficult to try different approaches independently.

As an example, QUICKSPEC [4] discovers equations about Haskell code, which are defined as “interesting” if they cannot be simplified using previously discovered equations. The intuition for such criteria is to avoid special cases of known theorems, such as $0 + 0 = 0$, $0 + 1 = 1$, etc. when we already know $0 + x = x$. Whilst Q4 is elegantly implemented with a congruence closure relation (version 1) and a term rewriting system (version 2), the term generation for Q1 is performed by brute-force.
Although QuickSpec only tests its equations rather than proving them, it is still used as the exploration component of more rigorous systems like HipSpec and Hipster.

In the following, we give an overview of the state of the art in automated theory exploration, then present potential improvements and our initial attempts at implementation.

2 Theory Exploration in Haskell

Automated theory exploration has been applied to libraries in Isabelle and Haskell, although we focus on the latter as its implementations are the most mature (demonstrated by the fact that Hipster explores Isabelle by first translating it to Haskell). Haskell is interesting to target, since its use of pure functions and algebraic datatypes causes many programs to follow algebraic laws. However, since Haskell’s type system cannot easily encode such laws, less effort is given to finding and stating them; compared to full theorem provers like Isabelle. Hence we imagine even a shallow exploration of code repositories such as Hackage could find many interesting theorems.

Currently, the most powerful TE system for Haskell is HipSpec, which uses off-the-shelf automated theorem provers (ATPs) to verify the conjectures of QuickSpec. QuickSpec, in turn, enumerates all type-correct combinations of the terms in the theory up to some depth, groups them into equivalence classes using the QuickCheck counterexample finder, then conjectures equations relating the members of these classes. This approach works well as a lemma generation system, making HipSpec a capable inductive theorem prover as well as a theory exploration system [3].

3 The ML4HS Framework

We consider Q2 and Q3 to be adequately solved by the existing use of type systems and ATPs, respectively. We identify the following potential improvements for the other questions:

Q1 Enumerating all type-correct terms is a brute-force solution to this question. Scalable alternatives to brute-force algorithms are a well-studied area of Artificial Intelligence and Machine Learning. In particular, heuristic search algorithms like those surveyed in [1] could be used. We could also use Machine Learning methods to identify some sub-set of a given theory, to prioritise over the rest.

Q4 Various alternative “interestingness” criteria have been proposed, for example those surveyed in [5]. Augmenting or replacing the criteria may be useful, for example to distinguish useful relationships from incidental coincidences; or to prevent surprising, insightful equations from being discarded because they can be simplified.
We are implementing a system called ML4HS to investigate these ideas. Its current form is a pre-processor for QUICKSPEC for prioritising theory elements. Inspired by the use of premise selection [9] to reduce the search space in ATP, we select sub-sets of the given theory to explore, chosen to try and keep together those expressions which combine in interesting ways, and to separate those which combine in uninteresting ways.

We hypothesise that similarity-based clustering of expressions, inspired by that of ML4PG [8] and related work in ACL2 [6], is an effective method for performing this separation. Future experiments will test this by comparing the throughput of QUICKSPEC with and without the ML4HS pre-processor.

Acknowledgements

Thank you to the HipSpec team at Chalmers University (Moa Johansson, Koen Claessen, Nick Smallbone, Dan Rosén and Irene Lobo Valbuena) for useful discussions of these ideas.

References

Researchers and malware analysts have identified hundreds and thousands of mobile applications as malware. These malware instances are organised into families based on some common unexpected behaviours, e.g., send premium messages, access locations, and intercept incoming messages and calls, etc. However, except some unclear online technical descriptions of several famous malware families, to the best of our knowledge, people have no idea of what exactly happens in mobile malware or what kind of behaviour of a mobile application makes it bad. This brings a challenging research problem: to automatically generate compact and precise explanations of unexpected behaviours in a mobile application if it has been identified as malware.

This research has several potential benefits, including: help people get better understanding of potential threats hidden in mobile applications; provide hints for malware analysts before more expensive investigation; support automatic generation of malware analysis reports; and provide clear and friendly references for security policy designers, etc.

Some fundamental technical questions we will answer are as follows. How could we characterise and formalise an application’s behaviour as efficiently and precisely as possible? What kind of behaviour is unexpected with respect to a specific application and how to figure it out automatically? Once a certain behaviour has been identified as unexpected, how could we automatically generate an explanation of this behaviour and in what kind of form? Finally, how could we evaluate generated explanations?

To answer these questions we develop the following approach.

**Behaviour Abstract.** We formalise an application’s behaviour as an automaton, which is actually an extension of a call-graph: valid control-dependency sequences of events, actions, and annotated API calls allowed by an application, so-called a behaviour automaton. The choice of which features to include is a trade-off between efficiency and precision. Automata are not as precise as data-flows but much more accurate than manifest information, e.g., permissions and actions, and API calls appearing in code which might contain “noise” caused by dead code and libraries. Also, compared with APIs, automata can capture more sophisticated behaviours. This is needed since some unexpected behaviours only arise when APIs are called in certain orders. On the other hand, it is much easier to generate behaviour automata for applications en masse than data-flows, in particular, people can annotate appealing APIs to generate behaviour automata more efficiently, instead of considering all data-dependency between statements. We have designed and implemented a static analysis tool to construct a behaviour automaton directly from each Android application to approximate its behaviour; to develop such a tool, we have to consider a broad range of features of the Java and the Android
framework, e.g., multi-threads, multi-entries, component life-cycle, inter-component communications, and runtime-registered listeners, etc.

**Feature Construction.** Fixing a group of applications which have been decided as malware or benign, once a behaviour automaton has been constructed for each of them, we want to figure out which behaviour (a sub-automaton) is unexpected and which is normal by applying machine learning methods. But, before any interesting pattern exploration, we have to decide which part of an application’s behaviour is exclusive to itself and which part is shared with other applications; this we call feature construction. Since the space of features, which is consisted of intersection and difference between automata, in theory, is at exponential order of the number of sample applications, we approximate this space by searching for a salient subspace. This search process is guided by behavioural difference between malware and benign applications. We combinatorially construct features from a group of malware instances by doing intersection and difference between their automata; to narrow the searching space we get rid of all non-salient features constructed in previous step, here, a feature is salient if is actually used in a classifier, which is trained from these malware instances mixed with an equal number of randomly-chosen benign applications; we combine features by doing intersection and difference between features from different groups; we design a divide-and-conquer-based multi-processing algorithm to construct and combine features in the above way more efficiently; this process stops until all sample malware instances have been considered.

**Learning Unexpected Behaviour.** An unexpected behaviour could be a common behaviour shared by malware instances and rarely seen in benign applications, e.g., intercept incoming messages and send them out via Internet connections, load classes from a hidden payload and execute commands from remote servers, and send premium messages constantly, etc. An unexpected behaviour in a group of applications could be normal in another, e.g., accessing locations is common in Jogging Tracer applications but unexpected in E-Reader applications, sending messages is the basic functionality of messenger applications but unexpected in Card Game applications, and accessing Internet can be seen in a lot of applications but unexpected for a FlashLight application, etc. These observations lead us to two methods of learning unexpected behaviours, i.e., to learn prevalent common patterns from malware families; and to identify singular behaviours of a group of malware and benign applications whose behaviours are very similar. For the first method, for each malware family, we search for a subset of all its behaviours, which maximizes the probability of a malware instance belonging to this family, if this malware instance has all behaviours in this set; we take the union of these sets as unexpected behaviours learned from malware families. As for the second method, we group applications by applying clustering methods using normalised compression distances between regular expressions of their automata, so-called context; for each group of applications, we train a classifier and choose features as unexpected behaviours by weight ranking. There are definitely some unexpected behaviours that cannot be captured using the above methods, e.g., accessing Internet is a behaviour of almost every FlashLight application although it is unexpected with respect to these applications. This case is out of range of our consideration. Except explanation learned unexpected behaviours will be used in malware detection as well: a target application is decided as malware if it has any unexpected behaviour from malware families or the classifier of its context consider it as malware.

**Generating Explanation.** We want to explain identified unexpected behaviours to a broad range of people, including: general users, malware analysts, security experts, policy designers, and researchers, etc. To satisfy different requirements for information accuracy and technical details, we produce explanations in three forms: short paragraphs, abstract and concrete behaviour automata, and statistical charts. A short paragraph is to give people rough ideas of what might happen in a malware instance, i.e., from unexpected behaviour automata, we extract actions, events, APIs, and sub-sequences of them; and feed their names as keywords
through pre-defined templates to generate paragraphs, or use these keywords to search for sentences from manually-produced malware analysis reports as explanations. This form of explanation is readable and brief but less informative or precise for malware analysts, security experts, and researchers, etc. We present on-demand to these people behaviour automata or abstract automata in which API names are replaced by permission-like phrases. Additionally, we generate prevalence pie charts and prevalence-time-series charts to show the changes of unexpected behaviours in malware samples over a long period, so as to help reveal the evolution of unexpected behaviours. Here are explanations for instances from malware families. They are generated by sentence searching.

§ This is a trojan which steals personal information from the infected device. It can be controlled over the web through http. (DroidKungfu)

§ It sends sms messages to premium rated numbers. (OpFake)

§ Allows applications to open network sockets, and uploads the data to a specific url. (Zitmo)

Some explanations generated from pair features which are learnt in application context are as follows.

§ It is declared to be a Chatting application, but, after a USB massive storage is connected, it will: retrieve a class in a runnable package; read information about networks; connect to Internet. (BaseBridge)

§ It is declared to be an Anti-Virus application, but it will: read your phone state after a phone call is made; read your phone state then connect to Internet; send SMS after a phone call is made. (Zitmo)

As a demonstration of more precise explanation, an example of behaviour automata is given as follows.

It actually captures the unexpected behaviours of malware family Zitmo.

**Evaluation.** We evaluate the quality of automatic explanation in three ways. First, by using unexpected behaviours as input features, we achieve a malware detector with good classification performance. Second, we do a user-study based on surveying randomly chosen people. The survey compares our approach with several alternative methods. Participants are invited to choose explanations they prefer. They are also asked to score each explanation to indicate to what extent it is convincing. Our results show automatic explanation can improve people’s belief on the system’s automatic decision. Third, we collect human-authored descriptions of identified malware families in online malware analysis reports from different sources. A subjective comparison shows automatic explanations are comparable to these manual descriptions.

We have presented a new approach to automatically generate explanations of unexpected behaviours for mobile applications, provided they have been automatically classified as malware. In contrast, most previous work on malware detection focused on obtaining good fits to a given training set by trying different methods and variant kinds of features [1, 3, 4, 10]. Explanations of chosen features produced by machine learning have received much less consideration. To the best of our knowledge, Drebin [2] is the first attempt to automatically generate explanations of Android malware. Drebin generates an explanation by choosing syntax-based features with top weights from a classifier and processing these features through templates to output text. But, syntax-based features cannot capture sophisticated behaviours. We overcome this limitation by adopting semantics-based features, i.e., behaviour automata.

The idea of using extended call-graphs to capture application behaviours is close to research by Chen et al. [5] and Yang et al. [9]. The former proposed permission-event graphs to capture
application behaviours for verification. The latter created two-tiered behavioural graph model to capture malware program logic for malware detection. Call-graphs were also exploited to help malware detection [7]. Among others, the method Dendroid [8] used cosine similarity between vectors of call graphs of identified malware to classify unknown malware samples into identified families. Similar ideas were applied in DroidLegacy [6] to detect piggybacking by exploiting difference between API sets of modules of malware in different families. All of these tools and methods were developed to eliminate unexpected behaviours without further explanations of why some behaviour is bad.

In future work, we want to construct anti-security policies from unexpected behaviours to help application verification and zero-day malware detection. We will focus on how to use verification results as semantics-based features to improve classification performance for zero-day malware detection.

References


Typed meta-interpretive learning
for proof strategies

Colin Farquhar¹, Gudmund Grov¹, Andrew Cropper²,
Stephen Muggleton², and Alan Bundy³

1 Heriot-Watt University, Edinburgh, UK, {cif30,G.Grov}@hw.ac.uk
2 Imperial College, London, UK {a.cropper13,s.muggleton}@imperial.ac.uk
3 University of Edinburgh, UK a.bundy@ed.ac.uk

1 Introduction
Interactive theorem provers are increasingly used in formalised mathematics and industrial software verification. However they still require a degree of manual intervention, which is time-consuming. Step-by-step user guidance is often required, where each step applies a proof tactic that splits a goal into smaller and simpler sub-goals.

These “stuck” proofs can often be grouped into families, such that a common proof strategy can be applied to them all. If we could learn and then reapply proof strategies from one (or a few) example(s) then this could significantly increase proof automation, making the overall approach more cost-effective - a key bottleneck for industrial application of software verification. Previous work focused on tactic composition, which does not capture reasons for branching, ie. when a strategy should be applied. Consequently we are faced with a choice of a large, possibly non-terminating, search space or hardcoding heuristics and possibly unintentionally ruling out some proofs. This motivated the development of the PSGraph language [2], which provides a graphical representation for proof strategies. Tactics are represented as nodes in a graph, with goal information shown as predicates labelling the edges.

Meta-Interpretive Learning (MIL) [4] was designed to learn from a small set of examples. It supports predicate invention, which is required to learn definitions of the goal predicates, ie. branching conditions, due to their rich and recursive nature. In this paper we first show (C1) that MIL is capable of learning strategies in the PSGraph language. We show that they exhibit a high degree of branching and so have a large search space, making them highly non-deterministic. We say that a strategy is deterministic if it has a single branch. We add a typing system to MIL in order to address this issue, and claim (C2) that:

“typed MIL learns more deterministic proof strategies than (untyped) MIL.”

An extended version of this paper has been presented at ILP2015 [1].

2 Framework
MIL [3] is an ILP technique aimed at supporting learning of recursive definitions. A powerful and novel aspect of MIL is that when learning a predicate definition it automatically introduces sub-definitions, allowing decomposition into a hierarchy of reusable parts. MIL is based on an adapted version of a Prolog
meta-interpreter. Normally such a meta-interpreter derives a proof by repeatedly fetching first-order Prolog clauses whose heads unify with a given goal. By contrast, a meta-interpretive learner additionally fetches higher-order metarules whose heads unify with the goal, and saves the resulting instantiated metarules to form a program. First, we extend Metagol with simple types:

**Definition 1 (Typed Meta-Interpretive Learning).** In typed MIL, each predicate and argument in the background, examples and metarules is tagged with a constant $t_i$ denoting its type. To illustrate, typing $P(X,Y)$ becomes:

$$P : t_1(X : t_2, Y : t_3).$$

To unify two predicates their types must also unify. Types for predicates, e.g. $P(X : t_2, Y : t_3)$, or arguments, e.g. $P : t_1(X,Y)$, may be omitted if they have a single type. We call these argument typed MIL and predicate typed MIL.

Our work will use predicate typed MIL. Note that in order to work the MetagolDF framework, the predicate type is represented as an additional argument: e.g. $P : t_1(X,Y)$ is internally represented as $P(t_1, X, Y)$. The argument types are not relevant to this work.

In our experiments we apply both untyped and typed MIL to proof trees generated from proofs in the Isabelle theorem prover. We aim to generalise this tree into a strategy in PSGraph which can then be applied to similar proofs.

Isabelle proof scripts are translated into Prolog clauses in order to apply Metagol to them. We introduce distinct types at this point to distinguish between tactics, wire predicates, goal data and strategies. The use of these types restricts the definitions which can be found by Metagol, which ensure that the strategy found is indeed a valid PSGraph.

### 3 Results

We have experimented with untyped and typed MIL to learn proof strategies from a collection of 15 proofs in propositional logic. The experiments were run using YAP on Ubuntu using a 3.10 GHz Intel i5-2400 CPU with 4GB RAM. The experiments were repeated with different time limits (1, 2, 4 and 8 seconds), with fig. 2 below showing mean results. For each example we consider the branching factor ($\sigma$) of the learned strategy. This indicates the number of possible proof trees which could be constructed by applying the strategy to a goal, including cases where the proof would fail. The graph in fig. 1 shows the average value of $\sigma$ for both untyped and typed MIL.

---

4 Code with all experiments available at: [www.macs.hw.ac.uk/~cif30/ilp15.zip](http://www.macs.hw.ac.uk/~cif30/ilp15.zip)
strategies compared to an optimum value. This optimum line represents Metagol learning a strategy from each example which has only one branch and thus one proof tree can be formed for each. Using untyped MIL $\sigma > 1$ initially, indicating more than one path on average, and $\sigma$ increases with time as more complex solutions are included. With typed MIL $\sigma = 1$ initially, remaining constant as time increases. These results show that as time increases untyped MIL diverges from the optimum $\sigma$, i.e. generates less efficient strategies. Conversely, typed MIL produces strategies which are optimally efficient.

<table>
<thead>
<tr>
<th></th>
<th>success</th>
<th>mean nodes</th>
<th>mean clauses</th>
<th>mean br</th>
<th>mean evals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Untyped</td>
<td>13</td>
<td>4</td>
<td>3</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>Typed</td>
<td>7</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

**Fig. 2.** Averaged learning results for untyped and typed MIL across all solutions: mean number of nodes in the strategy, mean number of clauses in Metagol’s definition, mean branching (br) and mean number of successful evaluations of other proofs.

4 Conclusion and further work

As shown in fig. 2 we have successfully learned strategies using both untyped and typed MIL, validating claim (C1). Fig. 2 also shows the difference in branching between untyped and typed MIL. Fig. 1 reinforces this, showing an increase in branching for untyped MIL with time while branching for typed MIL remains constant. Thus claim (C2) is also validated. However 2 also shows that the definitions of strategies found using typed MIL contain more clauses, which leads to an increase in run-time. This is the reason for the success rate of typed MIL being lower than that of untyped MIL (47% compared to 86%) in this time-restricted experiment.

Addressing this run-time issue will form part of our future work. We will also conduct experiments in learning a strategy from multiple proofs simultaneously. This will enable us to examine the generality of our learned strategies. We will then go on to look at more complex examples which require the invention of complex predicates.

**Acknowledgments.** This work has been supported by EPSRC grant EP/J001058/1, and the first author is supported by a James Watt scholarship. The fourth author acknowledges support from his Royal Academy of Engineering/Syngenta Research Chair.

**References**

**Tinker: A Graph Based Proof Strategy System**

Pierre Le Bras, Yuhui Lin & Gudmund Grov (Heriot-Watt University, Edinburgh, UK)

1 PSGraph & Tinker

Traditionally, tactics are used to encode common proof strategies that will work on goals by reducing it to a set of simpler subgoals. By chaining those tactics together, one can construct complex proofs. However, it is often the case that the resulting goal pipeline is hard to understand, thus making modifications and debugging a difficult task. This becomes even harder as strategies increases in size and complexity.

To address this problem, a graphical proof strategy language, called PSGraph, has been developed [1]. In PSGraph, proof strategies are represented as graphs, where tactics are represented as nodes and linked by wires. These wires can be seen as (empty) pipes where goals (also represented as nodes) can flow. Wires are also used to filter goals, either to ensure that they are sent to the correct tactic; or, if no such wire exists, to cut invalid branches. Modularity is supported by hierarchical graph nodes, where such node contains a graph. Figure 1 shows an example of this graphical representation of a proof strategy.

PSGraph has been implemented in the tool called Tinker [2]. It consists of a core and a GUI. The core interacts with the theorem prover to apply proof strategies to goals, while the GUI is meant to give the user features to develop proof strategies, and to provide an additional interactive interface to step through proofs with several debugging options. In this paper we will focus on new features and functionalities developed since [2].

2 The Graphical User Interface

![Figure 1: Example of a PSGraph.](image1)

![Figure 2: Tinker GUI](image2)

![Figure 3: Web view](image3)
A screen-shot of the new GUI is shown in Figure 2. The interface is divided into three main parts. The central part is the main window to present the current graph that can be edited and evaluated. On the top right-hand side is the graph inspector panel that allows the user to easily preview any graph if there are any hierarchical nodes. Once the user has selected a node on the graph, some information about it can be viewed on the bottom right-hand side panel. Finally, a library function is supported where each item of the library is a PSGraph that can be imported. The PSGraph files of the library are listed in a file tree on the top left-hand side and can be previewed on the bottom left-hand side. The key functionalities are divided into types discussed below.

2.1 Support for hierarchies

In PSGraph, hierarchical graphs are used to handle size and complexity of proof strategies. In the latest version of our GUI, we provide rich support for users to develop proof strategies with hierarchies. In particular, we have added support:

- to allow users to merge the selected nodes as a hierarchical tactic node:

- to provide a graph tactic inspector panel to preview hierarchical tactic nodes:

- to provide a hierarchy tree view:
2.2 Support for debugging

We have added several new features to help users debugging their proof strategies:

- to apply an atomic tactic to the selected goal (or pick an arbitrary goal);
- to apply a hierarchical tactic node to a goal by stepping into the child;
- to apply a hierarchical tactic node by stepping over the sub-graph, i.e. hide details of how the child is evaluated;
- to apply and complete the current hierarchical tactic;
- to apply and finish the current proof strategy;
- to support breakpoint, which allows user to automatically evaluate a graph but not beyond a break point (evaluation will only stop at the point where a goal node reaches the break point);
- to support flexible proof strategies development by allowing users to develop and change a proof strategy during evaluation;
- printing of goal information in the GUI;
- printing of debugging information from the core.

2.3 Support for recording, exporting and replaying

We have added functionalities to show and replay PSGraphs in different formats:

- the proof strategy the user is currently working on can be exported to a SVG format (to be included in a paper (Figure 1 is an example of this);
- (parts of) an evaluation can be recorded. This evaluation, which is a sequence of steps, can then be exported into a JSON file which can be loaded by a provided web application (written in HTML / CSS and JavaScript), to enable users to step through the file. The web application can both be viewed locally or copied to a web page. Figure 3 illustrates such a generated web page.

References
