

AI meets Formal Software Development

Edited by

Alan Bundy¹, Dieter Hutter², Cliff B. Jones³, and
J Strother Moore⁴

1 University of Edinburgh, GB, a.bundy@ed.ac.uk

2 DFKI Saarbrücken, DE, hutter@dfki.de

3 Newcastle University, GB, cliff.jones@ncl.ac.uk

4 University of Texas at Austin, US, moore@cs.utexas.edu

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 12271 “AI meets Formal Software Development”. This seminar brought together researchers from formal methods and AI. The participants addressed the issue of how AI can aid the formal software development process, including modelling and proof. There was a pleasing number of participants from industry and this made it possible to ground the discussions on industrial-scale problems.

Seminar 01.–06. July, 2012 – www.dagstuhl.de/12271

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs, I.2 Artificial intelligence (specifically I.2.3, I.2.4, I.2.6)


Keywords and phrases Learning of proof processes and strategies, Theory development, Formal software development, Automated reasoning, Formal modelling, Industrial use of formal methods

Digital Object Identifier 10.4230/DagRep.2.7.1

Edited in cooperation with Andrius Velykis (Newcastle University, GB)

1 Executive Summary

Cliff B. Jones

License  Creative Commons BY-NC-ND 3.0 Unported license
© Cliff B. Jones

This seminar brought together researchers from formal methods and AI. The participants addressed the issue of how AI can aid the formal software development process, including modelling and proof. There was a pleasing number of participants from industry and this made it possible to ground the discussions on industrial-scale problems.

Background

Industrial use of formal methods is certainly increasing but in order to make it more mainstream, the cost of applying formal methods, in terms of mathematical skill level and development time, must be reduced — and we believe that AI can help with these issues.

Rigorous software development using formal methods allows the construction of an accurate characterisation of a problem domain that is firmly based on mathematics; by applying standard mathematical analyses, these methods can be used to prove that systems satisfy formal specifications. A recent ACM computing survey [1] describes over sixty



Except where otherwise noted, content of this report is licensed under a Creative Commons BY-NC-ND 3.0 Unported license

AI meets Formal Software Development, *Dagstuhl Reports*, Vol. 2, Issue 7, pp. 1–29

Editors: Alan Bundy, Dieter Hutter, Cliff B. Jones, and J Strother Moore



Dagstuhl Reports

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

industrial projects and discusses the effect formal methods have on time, cost, and quality. It shows that with tools backed by mature theory, formal methods are becoming cost effective and their use is easier to justify, not as an academic exercise or legal requirement, but as part of a business case. Furthermore, the use of such formal methods is no longer confined to safety critical systems: the list of industrial partners in the DEPLOY project¹ is one indication of this broader use. Most methods tend to be posit-and-prove, where the user posits a development step (expressed in terms of specifications of yet-to-be-realised components) that has to be justified by proofs. The associated properties that must be verified are often called proof obligations (POs) or verification conditions. In most cases, such proofs require mechanical support by theorem provers.

One can distinguish between automatic and interactive provers, where the latter are generally more expressive but require user interaction. Examples of state-of-the-art interactive theorem provers are ACL2, Isabelle, HOL, Coq and PVS, while E, SPASS, Vampire and Z3 are examples of automatic provers.

AI has had a large impact on the development of provers. In fact, one of the first AI application was a theorem prover and all theorem provers now contain heuristics to reduce the search space that can be attributed to AI. Nevertheless, theorem proving research and (pure) AI research have diverged, and theorem proving is barely considered to be AI-related anymore.

There follows a list of background references.

References

- 1 J. Woodcock, P. G. Larsen, J. Bicarregui, and J. S. Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4), 2009.
- 2 A. Bundy. *The Computer Modelling of Mathematical Reasoning*. Academic Press, 1983. (2nd edition 1986).
- 3 A. Bundy and A. Smaill. *A Catalogue of Artificial Intelligence Techniques*. Springer, 1984. (2nd edition 1986, 3rd edition 1990, 4th revised edition 1997).
- 4 A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *J. Autom. Reasoning*, 7(3):303–324, 1991.
- 5 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.
- 6 A. Bundy. The automation of proof by mathematical induction. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 847–911. Elsevier and MIT Press, 2001.
- 7 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, June 2005.
- 8 D. Hutter and W. Stephan, editors. *Mechanizing Mathematical Reasoning, Essays in Honor of Jörg H. Siekmann on the Occasion of His 60th Birthday*, volume 2605 of *LNCS*. Springer, 2005.
- 9 D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors. *Proceedings of Current Trends in Applied Formal Methods (FM-Trends 98)*, volume 1641 of *LNCS*, Boppard, Germany, 1999. Springer.
- 10 C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A Formal Development Support System*. Springer, 1991.

¹ DEPLOY was an EU-funded “IP” led by Newcastle University; a four year project with a budget of about 18M Euros; the industrial collaborators include Siemens Transport, Bosch and SAP.

- 11 R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Formal Methods Series. Academic Press, second edition, 1997.
- 12 M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*, volume 3 of *Advances in Formal Methods*. Kluwer Academic Publishers, 2000.
- 13 M. Kaufmann and J. S. Moore. Some key research problems in automated theorem proving for hardware and software verification. *Revista de la Real Academia de Ciencias (RACSAM)*, 98(1):181–196, 2004.

Organisation of the seminar

It might be useful to organisers of future seminars to record some organisational issues. We asked participants to prepare only short talks that introduced topics and –just as we wished– a number of the talks were actually prepared at the seminar location and with the benefit of having heard other talks. This free format worked well for our exchange of ideas and in most regards we were pleased that we started with only the Monday morning actually scheduled. Perhaps the biggest casualty of the fluid organisation (coupled with so many interesting participants) was that there was no time left for Panel Sessions. However, the differing lengths of discussions (and liberal use of breaks and a “hike” for people to establish new links) led to intensive interaction.

Notes on nearly all of the talks are contained in Section 3 of the current document.

It is a pleasure to extend our thanks to everyone involved in the Dagstuhl organisation: they provide a supportive and friendly context in which such fruitful scientific exchanges can develop unhindered by distraction.

Results

It is possible to address the results under the phases of the development cycle. Requirements capture is traditionally a pre-formal exercise and is the phase where one would expect least impact from formal ideas. There is certainly scope here for the use of ontologies and some hope for help in detecting inconsistencies in requirements but little time was spent in the seminar on these topics.

Once development moves to the creation of a specification, the scope for formalism increases and with it the hope for a greater contribution from AI. Essentially, a formal specification is a model. Formal proof can be used to establish internal consistency properties or to prove that properties match expectations about the required system. Model checking approaches are often the most efficient way of detecting inconsistencies.

Steps of development (in the posit and prove approaches) essentially introduce further models which should relate in precise ways to each other. The technical details vary between development methods but the overall implications for the use of proof and the contribution of AI are similar. It is perhaps worth reemphasising here that the seminar was trying to address problems of an industrial scale.

An interesting dichotomy was explored at the seminar concerning POs that fail to discharge. One school of thought is to interpose extra models in order to cause the generation of simpler POs; the alternative is to take the POs as fixed and develop “theories” (collections of auxiliary functions and lemmas) to complete the proof process. Suffice it to say here that AI was seen to have a role in both approaches.

More generally, the whole task of refactoring models and reusing libraries of established material is another area seen as being in need of help from AI thinking.

Turning to the richest area of collaboration –that of proof itself– a prominent theme was on the ways in which machine learning can help. There are many facets of this question including analogy with previous proofs, data mining of proofs (and failures) and proof strategy languages.

One particularly important aspect of the cost of proof in an industrial setting is proof maintenance. In practical settings, many things change and it is unlikely to be acceptable to have to repeat the whole proof process after each change.

Another area that led to useful interactions between participants was the subject of failure analysis and repair. It was observed that it is useful to have strong expectations as to how proofs were meant to succeed.

In conclusion many points of contact can be seen in the material presented below. Unsurprisingly, the material ranges from hopes for future research to mature results that can be readily applied. It is not only a hope that the links between ideas and researchers made at the seminar will persist — we already have clear proof of collaborative work.

The four organisers are extremely grateful to Andrius Velykis who took on the whole of the task of collecting and tidying the input in Section 3 representing the contributions of the speakers.

2 Table of Contents

Executive Summary

<i>Cliff B. Jones</i>	1
---------------------------------	---

Overviews of Talks

Applying Formal Methods In Industry <i>Rob Arthan</i>	8
Structure Formation in Formal Theories <i>Serge Autexier</i>	8
Automated Reasoning and Formal Methods <i>Alan Bundy</i>	9
Explicit vs Implicit Search Guidance <i>Alan Bundy</i>	9
Reflections on AI meets Formal Software Development <i>Alan Bundy</i>	9
Can IsaScheme be Used for Recursive Predicate Invention? <i>Alan Bundy</i>	10
Automated Theorem Proving in Perfect Developer and Escher C Verifier <i>David Crocker</i>	10
Synthesizing Domain-specific Annotations <i>Ewen W. Denney</i>	10
Capturing and Inferring the Proof Process (Part 1: Case Studies) <i>Leo Freitas</i>	11
Learning Component Abstractions <i>Dimitra Giannakopoulou</i>	11
A strategy language to facilitate proof re-use <i>Gudmund Grov</i>	12
Glassbox vs Blackbox Software Analysis <i>Reiner Haehnle</i>	12
Beyond pure verification: AI for software development <i>Dieter Hutter</i>	12
Reasoned Modelling: Exploiting the Synergy between Reasoning and Modelling <i>Andrew Ireland</i>	13
HipSpec: Theory Exploration for Automating Inductive Proofs <i>Moa Johansson</i>	13
Formalism: pitfalls and overcoming them (with AI?) <i>Cliff B. Jones</i>	14
Languages and States: another view of "Why" <i>Cliff B. Jones</i>	14
Machine Learning for the Working Logician <i>Ekaterina Komendantskaya</i>	15

Is there a human to save the model, proof? <i>Thierry Lecomte</i>	16
Induction and coinduction in an automatic program verifier <i>K. Rustan M. Leino</i>	17
ProB Tool Demonstration and Thoughts on Using Artificial Intelligence for Formal Methods <i>Michael Leuschel</i>	17
The Use of Rippling to Automate Event-B Invariant Preservation Proofs <i>Yuhui Lin</i>	18
Discovery of Invariants through Automated Theory Formation <i>Maria Teresa Llano Rodriguez</i>	18
Abstraction in Formal Verification and Development <i>Christoph Lueth</i>	18
A study of cooperative online math <i>Ursula Martin</i>	19
TLA+ Proofs <i>Stephan Merz</i>	19
Case Based Specifications – reusing specifications, programs and proofs <i>Rosemary Monahan</i>	20
Can AI Help ACL2? <i>J Strother Moore</i>	21
Boogie Verification Debugger <i>Michał Moskal</i>	22
Formal software verification in avionics <i>Yannick Moy</i>	22
What I've Learned from the VFS Challenge thus far <i>José Nuno Oliveira</i>	23
The Need for AI in Software Engineering – A Message from the Trenches <i>Stephan Schulz</i>	23
Finding and Responding to Failure in Large Formal Verifications <i>Mark Staples</i>	24
Formal Verification of QVT Transformations <i>Kurt Stenzel</i>	25
Modeling and Proving <i>Werner Stephan</i>	25
Overview of CSP B for railway modelling <i>Helen Treharne</i>	26
AI via/for Large Mathematical Knowledge Bases <i>Josef Urban</i>	26
Capturing and Inferring the Proof Process (Part 2: Architecture) <i>Andrius Velykis</i>	27

More Abstractions
Laurent Voisin 27


Finding Counterexamples through Heuristic Search
Martin Wehrle 28

Participants 29

3 Overviews of Talks

3.1 Applying Formal Methods In Industry

Rob Arthan (Lemma 1 Ltd. – Reading, GB)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Rob Arthan


Joint work of Arthan, Rob; Jones, Roger; O'Halloran, Colin

The talk was an overview of the speaker's experience of industrial applications of formal methods mostly involving the Z Notation and the ProofPower tools for specification and verification. This included a description of the CLawZ toolset that combines ProofPower and other tools into a system for verifying code that is automatically generated from Simulink specifications. Developed by Colin O'Halloran and others of DRisQ Ltd (<http://drisq.com>), CLawZ offers an independent verification path allowing the use of an untrusted code generator in the development of safety-critical systems, such as avionics control systems.

The talk concluded with some discussion of software engineering generally and offered a challenge for AI and formal methods: software developers often need to “work around” problems in software in the field that arise as a result of errors in the development process or of change in operating environments. What help can AI and formal methods offer to engineers who have to reason about systems that include flawed components?

3.2 Structure Formation in Formal Theories

Serge Autexier (DFKI Bremen, DE)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Serge Autexier

It has been long recognized that the modularity of specifications is an indispensable prerequisite for an efficient reasoning in complex domains. Algebraic specification techniques provide appropriate frameworks for structuring complex specifications and the notion of development graphs has been introduced as a technical means to work with and reason about such structured specifications. In this work we are concerned with assisting the process of structuring specifications in order to make intrinsic structures that are hidden explicit. Based on development graphs, we present an initial methodology and a formal calculus to transform unstructured specifications into structured ones. The calculus rules operate on development graphs allowing one to separate specifications coalesced in one theory into a structured graph. The calculus can both be used to structure a flat specification into sensible modules or to restructure existing structurings. We present an initial methodology to support the process of structure formations in large unstructured specifications.

3.3 Automated Reasoning and Formal Methods

Alan Bundy (University of Edinburgh, GB)

License © ⓘ ⊖ Creative Commons BY-NC-ND 3.0 Unported license
 © Alan Bundy
URL <http://www.ai4fm.org>

We review progress in automated reasoning in the last four decades and ask whether our provers are now sufficiently mature to support formal methods in mainstream ICT system development. We look at improvements due to Moore's Law and improvements in decision procedures, automatic provers, inductive provers and interactive provers. We ask what more AI can offer to further improve the situation. In particular, we speculate about the role of machine learning, e.g., to data-mine interactive proofs to extract proof tactics to be used to increase automation.

The slides for this talk are available on the AI4FM project website (<http://www.ai4fm.org>).

3.4 Explicit vs Implicit Search Guidance

Alan Bundy (University of Edinburgh, GB)

License © ⓘ ⊖ Creative Commons BY-NC-ND 3.0 Unported license
 © Alan Bundy
URL <http://www.ai4fm.org>

Suppose we want to analyse an interactively produced proof of a proof obligation, extract the proof strategy underlying this source proof and then apply it to guide the target automatic proofs of similar proof obligations. What form should such strategies take? We compare and contrast two alternatives, which we characterise as explicit and implicit. Explicit strategies are hierarchies of proof tactics, such as those described by Gudmund Grov in his talk at this Seminar. Implicit strategies are schemas that abstract the additional lemmas that are introduced in the source proof. We can then use theory exploration tools, such as Omar Montano Rivas's IsaScheme, to generate similar lemmas for the target proof by instantiating these schemas and proving the resulting conjectures. The implicit strategies trade off a loss of fine control for increased flexibility. The hypothesis to be evaluated is whether our provers are now sufficiently autonomous to find the right way to use the newly instantiated lemmas in the target proofs.

The slides for this talk are available on the AI4FM project website (<http://www.ai4fm.org>).

3.5 Reflections on AI meets Formal Software Development

Alan Bundy (University of Edinburgh, GB)


License © ⓘ ⊖ Creative Commons BY-NC-ND 3.0 Unported license
 © Alan Bundy
URL <http://www.ai4fm.org>

These are my wrap-up slides at the end of the Seminar. I tried to itemise all the points of contact that had emerged during the meeting. I grouped these under the headings of: requirements capture, modelling, proof, and failure analysis and repair.

The slides for this talk are available on the AI4FM project website (<http://www.ai4fm.org>).

3.6 Can IsaScheme be Used for Recursive Predicate Invention?


Alan Bundy (University of Edinburgh, GB)

License  Creative Commons BY-NC-ND 3.0 Unported license
 © Alan Bundy
URL <https://dream.inf.ed.ac.uk/protected/Bluenote>

At the Dagstuhl seminar “AI meets Formal Software Development” in July 2012, several people remarked on the importance of lemmas with conditions and the creativity involved in inventing these conditions. For instance, one sometimes wants to define new (recursive) predicates to provide these conditions. It occurred to me that IsaScheme might be used to generate these conditions and invent new recursive predicates. Blue book note 1763 at <https://dream.inf.ed.ac.uk/protected/Bluenote> explains the idea. If you don’t have access to this protected area, contact me at a.bundy@ed.ac.uk for a copy.

3.7 Automated Theorem Proving in Perfect Developer and Escher C Verifier


David Crocker (Escher Technologies – Aldershot, GB)

License  Creative Commons BY-NC-ND 3.0 Unported license
 © David Crocker

We have two formal tools intended for development of high integrity software. Perfect Developer uses the specify-refine-generate paradigm, while Escher C Verifier is for formal verification of annotated hand-written MISRA-C code. Both have been used in developing industrial critical systems, and both use the same non-interactive theorem prover. I discuss different approaches to logics and theorem proving for software verification, drawing a comparison with the RISC/CISC processor wars of the 1990s, and outline the approach used by our prover. Finally, I discuss some areas where I think AI might be applied to improve the automation of our product.

3.8 Synthesizing Domain-specific Annotations


Ewen W. Denney (NASA – Moffett Field, US)

License  Creative Commons BY-NC-ND 3.0 Unported license
 © Ewen W. Denney
Joint work of Denney, Ewen; Fischer, Bernd

Verifying interesting properties on code typically requires logical annotations. If these annotations need to be added manually, this presents a significant bottleneck to automated verification. Here, we describe a language for encoding the domain knowledge needed to automatically generate such annotations for a class of mathematical properties. Interestingly, the problem of generating annotations turns out to essentially be a form of code generation.

3.9 Capturing and Inferring the Proof Process (Part 1: Case Studies)


Leo Freitas (Newcastle University, GB)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Leo Freitas

We report current work on inferring the proof process of an expert by wire-tapping various theorem proving environments (e.g. Isabelle/HOL, Z/EVES, etc). The idea is to have enough (meta-)proof information (i.e. user intent, lemmas used, points of failure and ways of recovery, various proof attempts [sub-]trees, etc.), in order to be able to do meta-level reasoning about proofs, in particular for proof reuse, but also for proof maintenance and transferability to non-expert users. For that we have worked on a number of case studies, large (e.g. EMV smart cards, Xenon security hypervisor), medium (e.g. Tokeneer ID station, Federated Cloud workflows properties, etc.), and small (e.g. Transitive closure lemma library, Union/Find Fisher/Galler problem characterisation, etc.). We are currently working on a paper describing this work to appear soon. Please visit <http://www.ai4fm.org> for more information.

3.10 Learning Component Abstractions

Dimitra Giannakopoulou (NASA – Moffett Field, US)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Dimitra Giannakopoulou

Automata learning algorithms are used increasingly in the research community to generate component abstractions or models. In this talk, we presented two of the learning-based frameworks that we have developed over the years, which use the automata learning algorithm L^* . In the first framework, L^* interacts with model checking to generate abstractions that are used as assumptions for automated compositional verification [1]. The second framework combines L^* with symbolic execution to generate component interfaces [2]; the interfaces are three-valued, capturing whether a sequence of method invocations is safe, unsafe, or its effect on the component state is unresolved by the symbolic execution engine. The latter framework is available as the `jpf-psyco` project within the JavaPathfinder open source model checker for Java bytecode.

We then discussed other uses of learning in software engineering, including model and specification mining for black box or library components, potentially based on existing code that uses the components and is available on the Web. We believe that cross-fertilization with heuristic search used in AI applications will also be beneficial because exhaustive techniques often hit scalability issues. Finally, we believe that ultimately engineers should take verification into account when designing systems, and it is possible that machine learning could help us in the detection of good design or specification patterns.


References

- 1 J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In H. Garavel and J. Hatcliff, editors, *9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, volume 2619 of *LNCS*, pages 331–346. Springer, 2003.

- 2 D. Giannakopoulou, Z. Rakamaric, and V. Raman. Symbolic learning of component interfaces. In A. Miné and D. Schmidt, editors, *19th Static Analysis Symposium (SAS 2012)*, volume 7460 of *LNCS*, pages 248–264. Springer, 2012.

3.11 A strategy language to facilitate proof re-use

Gudmund Grov (University of Edinburgh, GB)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Gudmund Grov

Joint work of Grov, Gudmund; Bundy, Alan; Komandantskaya, Katya; Dixon, Lucas

Within refinement-based formal methods, such as B, Event-B, VDM and Z, many proofs follow a similar pattern. The ability to re-use expert-provided proofs to automatically discharge proof within the same family could therefore greatly improve automation of such proofs—which is currently a bottleneck for industrial application beyond niche markets. A similar phenomenon is also observed in mathematics through proof by analogy.

An observation we have made is that in order to capture sufficiently generic strategies, both goal and proof technique properties must be captured, which is not supported by current tactic languages. Here, we introduced work-in-progress on a graph based strategy language, where nodes are proof techniques and edges contains goal properties. Evaluation is then achieved by sending actual goals down the edges of the graph, and updating the proof by applying the technique to the input goal, and sending new goals to the output edges. We outlined some properties about this language, and briefly discussed a combination of stochastic learning methods (for pattern discovery) and logic-based learning methods (for strategy extraction/generalisation) in order to learn new proof strategies represented in this language.

3.12 Glassbox vs Blackbox Software Analysis

Reiner Haehnle (TU Darmstadt, DE)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Reiner Haehnle

Many approaches to software analysis/verification/synthesis/testing can be classified as either “glassbox” or “blackbox”. Both kinds have specific advantages and disadvantages, the latter preventing wider industrial usage. We argue that there is considerable potential in a systematic combination of both and sketch the outlines of a possible way to do this.

3.13 Beyond pure verification: AI for software development

Dieter Hutter (DFKI Bremen, DE)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Dieter Hutter

Traditionally, verification is a process separated from standard development processes. We have our own methodologies and tools. Hence often Formal Methods (in particular theorem proving) is applied post mortem, i.e. after the actual development has been completed to ensure that everything went right at the end. From the viewpoint of a standard software engineer, Formal Methods are just a nuisance in the development process.

However, in the last ten years we have also seen the upcoming of semantic-based approaches to ease the software development process: model driven architecture, domain specific languages, a variety of web services and their (dynamic) composition, etc. We at DFKI have spent a lot of time in developing a change management for software development that maintain the relations between documents using semantic knowledge about them. Analysing informal documents in depth requires NL-understanding and thus a detailed ontology of the domain.

Thus, formally specified application domains are not only needed to verify the implementation but are also required to guide the overall design process, to automate (parts of) refinement process and to realize a powerful change (or development) management system. Combining these efforts in an integrated design process would multiply the benefits of applying formal methods but also ease the formal specification process becoming now incremental (starting at a simple ontology analogous to a standard glossary and ending in full fledged formal specs).

3.14 Reasoned Modelling: Exploiting the Synergy between Reasoning and Modelling

Andrew Ireland (Heriot-Watt University Edinburgh, GB)

License © ⓘ ⊖ Creative Commons BY-NC-ND 3.0 Unported license
© Andrew Ireland

Joint work of Ireland, Andrew; Grov, Gudmund; Llano, Maria Teresa; Pease, Alison

While the rigour of building formal models brings significant benefits, formal reasoning remains a major barrier to the wider acceptance of formalism within the development of software intensive systems. In our work we abstract away from the complexities of low-level proof obligations, providing high-level modelling guidance. We have achieved this through a combination of techniques from Artificial Intelligence, i.e. planning, proof-failure analysis, automated theory formation, and Formal Methods, i.e. formal modelling, proof and simulation. Our aim is to increase the accessibility and productivity of Formal Methods—allowing smart designers to make better use of their time.

3.15 HipSpec: Theory Exploration for Automating Inductive Proofs

Moa Johansson (Chalmers UT – Göteborg, SE)

License © ⓘ ⊖ Creative Commons BY-NC-ND 3.0 Unported license
© Moa Johansson

Joint work of Claessen, Koen; Johansson, Moa; Rosén, Dan; Smallbone, Nicholas

Main reference K. Claessen, M. Johansson, D. Rosén, and N. Smallbone, “HipSpec: Automating inductive proofs of program properties,” in *IJCAR Workshop on Automated Theory eXploration (ATX 2012)*, Manchester, UK, July 2012.

URL <http://web.student.chalmers.se/~danr/hipspec-atx.pdf>

HipSpec is an inductive theorem prover for proving properties about Haskell programs [1]. It implements a novel bottom-up approach to lemma discovery, where theory exploration is used to first derive a background theory consisting of potentially useful lemmas about available functions and datatypes.

HipSpec consists of several sub-systems: Hip is an inductive theorem prover. It translates Haskell function definitions to first order logic and applies induction to given conjectures.

Resulting proof obligations are passed to an off the shelf prover (for instance E or Z3). QuickSpec is responsible for generating candidate lemmas about available functions and datatypes. It generates terms which are divided up into equivalence classes using counter-example testing. From these equivalence classes, equations can be derived. These are passed to Hip for proof. Those that are proved are added to the background theory and may be used in subsequent proofs.

Initial results are encouraging, HipSpec performs very well compared to other automated inductive theorem provers such as IsaPlanner, Zeno and Dafny.


HipSpec is available online from: <http://github.com/danr/hipspect>.

References

- 1 K. Claessen, M. Johansson, D. Rosén, and N. Smallbone. HipSpec: Automating inductive proofs of program properties. In *IJCAR Workshop on Automated Theory eXploration (ATX 2012)*, Manchester, UK, July 2012.

3.16 Formalism: pitfalls and overcoming them (with AI?)

Cliff B. Jones (Newcastle University, GB)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Cliff B. Jones

This was a general, opening, talk. Based partly on recent experience in the EU-funded DEPLOY project, I described typical industrial figures where decent heuristics might automatically discharge over 90 percent of the required proof obligations (POs) but that this can still leave a large enough collection of proof tasks needing hand-assisted proofs that industrial engineers find it a disincentive to use formal tools. The issue is of course not going to go away: any set of heuristics will have their limitations.

The good news is that such collections of undischarged POs appear to fall into families and that a single idea will be the key to discharging many POs. The “ideas” are sometimes expressible as high level strategies or can be captured as lemmas (or shapes thereof). A useful approach is therefore to try to capture these key ideas whilst an expert is doing one proof from the family and to use this to then obtain automatic proofs of the remaining tasks in that family. The discovery and replay of these ideas looks like an interesting AI challenge.

There is an additional payoff when, as so often happens, specifications change and POs are regenerated.

These ideas are behind (and are evolving in) the UK research project AI4FM (see <http://www.ai4fm.org>).

3.17 Languages and States: another view of "Why"

Cliff B. Jones (Newcastle University, GB)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Cliff B. Jones

Like the view in my introductory talk, this also relates to ongoing work in the AI4FM project.

If we are to capture the key ideas in the interaction between a user and a theorem proving system, we essentially have to have a “language” for high-level strategies. The talk put the

point of view that, if one wants to design a language, the best way to start is to think about the “state” that the language constructs can manipulate. In AI4FM we talk about trying to capture the “Why” of what the user is doing. (The talk by Andrius Velykis showed an architecture for snooping on the interactions between user and theorem prover.) I sketched some points about a state for “Models of Why” but the details are less important (and are anyway still changing) than the general idea of starting thoughts about the design of our future system to learn from experts by discussing its state.


Since Ursula Martin had made kind references to the “mural” system build in the 1980s I took the opportunity to dig out some screen shots. This was an experiment in the design of of an interaction style. As [1] shows, this system was also designed from its formal specification (the cited book is now out of print but freely available on-line at <http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/mural.pdf>).

References

- 1 C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A Formal Development Support System*. Springer, 1991.

3.18 Machine Learning for the Working Logician

Ekaterina Komendantskaya (University of Dundee, GB)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Ekaterina Komendantskaya

Joint work of Komendantskaya, Ekaterina; Grov, Gudmund; Bundy, Alan

Main reference G. Grov, E. Komendantskaya, and A. Bundy, “A statistical relational learning challenge – extracting proof strategies from exemplar proofs,” in *ICML’12 Workshop on Statistical Relational Learning (SRL-2012)*, Edinburgh, UK, July 2012.

URL <http://www.computing.dundee.ac.uk/staff/katya/srl12.pdf>

The talk consisted of two parts: Part 1 discussed the motivation for using Statistical Machine Learning in Automated Theorem Proving (ATP). In particular, the following research question was chosen for discussion: how can we identify application areas within automated theorem proving where machine-learning will be both genuinely needed and trusted by the ATP users?

Part 2 addressed this question by showing results of some recent experiments on data-mining first-order proofs. Coinductive proof trees for first-order logic programs were data-mined using Neural Networks and SVMs. This proof data-mining method allowed to solve five types of proof-classification problems: recognition of well-formed proofs, proofs belonging to the same proof-family, well-typed proofs, as well as proofs from a success family and a well-typed family of proofs.

Discovery of various proof families was identified as the most promising of these. This provided my tentative answer to the research question posed in the beginning.


References

- 1 G. H. Bakir, T. Hofmann, B. Schölkopf, A. J. Smola, B. Taskar, and S. V. N. Vishwanathan. *Predicting Structured Data*. Neural Information Processing Series. MIT Press, 2007.
- 2 A. S. D. A. Garcez, K. Broda, and D. M. Gabbay. *Neural-Symbolic Learning Systems: Foundations and Applications*. Perspectives in Neural Computing. Springer, 2002.
- 3 G. Grov, E. Komendantskaya, and A. Bundy. A statistical relational learning challenge – extracting proof strategies from exemplar proofs. In *ICML’12 Workshop on Statistical Relational Learning (SRL-2012)*, Edinburgh, UK, July 2012.
- 4 J. Denzinger, M. Fuchs, C. Goller, and S. Schulz. Learning from previous proof experience: A survey. Technical report, Fachbereich Informatik, 1999.

- 5 J. Denzinger and S. Schulz. Automatic acquisition of search control knowledge from multiple proof attempts. *Inf. Comput.*, 162(1-2):59–79, 2000.
- 6 H. Duncan. *The Use of Data-Mining for the Automatic Formation of Tactics*. PhD thesis, School of Informatics, University of Edinburgh, 2007.
- 7 L. Getoor and B. Taskar. *Introduction to Statistical Relational Learning*. Adaptive Computation and Machine Learning. MIT Press, 2007.
- 8 E. Komendantskaya, R. Almaghairbe, and K. Lichota. ML-CAP: the manual, software and experimental data sets. <http://www.computing.dundee.ac.uk/staff/katya/MLCAP-man>, 2012.
- 9 J. W. Lloyd. *Logic for Learning: Learning Comprehensible Theories from Structured Data*. Cognitive Technologies. Springer, 2003.
- 10 E. Tsivtsivadze, J. Urban, H. Geuvers, and T. Heskes. Semantic graph kernels for automated reasoning. In *11th SIAM International Conference on Data Mining (SDM 2011)*, pages 795–803. SIAM / Omnipress, 2011.
- 11 J. Urban, G. Sutcliffe, P. Pudlák, and J. Vyskočil. MaLAREa SG1 – machine learner for automated reasoning with semantic guidance. In A. Armando, P. Baumgartner, and G. Dowek, editors, *4th International Joint Conference on Automated Reasoning (IJCAR 2008)*, volume 5195 of *LNCS*, pages 441–456. Springer, 2008.

3.19 Is there a human to save the model, proof?

Thierry Lecomte (ClearSy – Aix-en-Provence, FR)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Thierry Lecomte

Checking a model against properties is a demanding process, as it requires to cope with state-of-the-art demonstrators (theorem provers, tableaux-method, model checkers). In several cases, the demonstrator is not able to complete the demonstration and the human operator is in charge of finding a way to help the tool efficiently. As of today, if the demonstrator is not able to complete the proof, most of the time all the proof mechanisms have to be disabled, leaving the human operator to use this tools just as a sophisticated calculator (built-in mechanisms automatically going to a wrong direction).

However there is room for improvement at input level:

- requirements are usually not abstract enough,
- models need to be adapted to proof tools,
- the human operator needs abstraction skills to deal properly with modelling.


The exception is for data validation where ProB model-checker is strong enough to handle 100k Excel cells and 200 rules, without any human intervention.

References

- 1 T. Lecomte, L. Burdy, and M. Leuschel. Formally checking large data sets in the railways. In *Workshop on the experience of and advances in developing dependable systems in Event-B (DS-Event-B-2012)*, Kyoto, Japan, November 2012.

3.20 Induction and coinduction in an automatic program verifier

K. Rustan M. Leino (Microsoft Research – Redmond, US)

License  Creative Commons BY-NC-ND 3.0 Unported license
© K. Rustan M. Leino

Main reference K. R. M. Leino. “Dafny: An automatic program verifier for functional correctness,” in E. M. Clarke and A. Voronkov, editors, *16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16)*, volume 6355 of *LNCS*, pp. 348–370. Springer, 2010.

URL <http://research.microsoft.com/en-us/projects/dafny/>

Program verifiers are good tools for verifying programs. Theorem provers are good tools for proving theorems. But the line between the two kinds of tools often gets blurry. Theorem provers can also be used to verify programs and program verifiers can also be used to prove theorems.

In this talk, I show a number of programs whose verification also requires some lemmas. The programs and lemmas are both proved using the program verifier Dafny [1]. I also demonstrate how to manually write inductive proofs in Dafny and show Dafny’s automatic induction tactic [2]. Moreover, I show some experimental features for dealing with co-induction.


I expect that some AI techniques developed in the theorem proving and AI communities could be useful in program verifiers like Dafny.

References

- 1 K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16)*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.
- 2 K. R. M. Leino. Automating induction with an SMT solver. In V. Kuncak and A. Rybalchenko, editors, *13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2012)*, volume 7148 of *LNCS*, pages 315–331. Springer, 2012.

3.21 ProB Tool Demonstration and Thoughts on Using Artificial Intelligence for Formal Methods

Michael Leuschel (Universität Düsseldorf, DE)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Michael Leuschel


In this talk I gave a demonstration of the validation tool ProB [1, 2]. In particular, I concentrate on model checking and the constraint solving kernel and how this links with proof and intelligent search. In particular, I believe that proof, model checking and constraint solving should go hand-in-hand, and that tackling high-level (higher-order) formalisms such as B is extremely challenging, but provides more potential for intelligent search.

References

- 1 M. Leuschel and M. J. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, volume 2805 of *LNCS*, pages 855–874. Springer, 2003.
- 2 M. Leuschel, J. Falampin, F. Fritz, and D. Plagge. Automated property verification for large scale B models with ProB. *Formal Asp. Comput.*, 23(6):683–709, 2011.

3.22 The Use of Rippling to Automate Event-B Invariant Preservation Proofs

Yuhui Lin (University of Edinburgh, GB)

License  Creative Commons BY-NC-ND 3.0 Unported license

© Yuhui Lin


Main reference Y. Lin, A. Bundy, G. Grov, “The use of rippling to automate Event-B invariant preservation proofs,” in A. Goodloe and S. Person, editors, *NASA Formal Methods*, volume 7226 of *LNCS*, pages 231–236. Springer, 2012.

URL http://dx.doi.org/10.1007/978-3-642-28891-3_23

Proof automation is a common bottleneck for industrial adoption of formal methods. In Event-B a significant proportion of proof obligations which requires human interaction falls into a family called invariant preservation. In this talk we show that rippling can increase the automation of proof in this family, and extend this technique by combining two existing approaches.

3.23 Discovery of Invariants through Automated Theory Formation

Maria Teresa Llano Rodriguez (Heriot-Watt University Edinburgh, GB)

License  Creative Commons BY-NC-ND 3.0 Unported license

© Maria Teresa Llano Rodriguez

Joint work of Llano Rodriguez, Maria Teresa; Ireland, Andrew; Pease, Alison

Main reference M. T. Llano, A. Ireland, A. Pease, “Discovery of invariants through automated theory formation,” in *15th International Refinement Workshop (Refine 2011)*, volume 55 of *EPTCS*, pp. 1–19, Limerick, Ireland, June 2011.

URL <http://dx.doi.org/10.4204/EPTCS.55.1>

Refinement is a powerful mechanism for mastering the complexities that arise when formally modelling systems. Refinement also brings with it additional proof obligations – requiring a developer to discover properties relating to their design decisions. With the goal of reducing this burden, we have investigated how a general purpose theory formation tool, HR, can be used to automate the discovery of such properties within the context of Event-B. This gave rise to an integrated approach to automated invariant discovery. In addition to formal modelling and automated theory formation, our approach relies upon the simulation of system models as a key input to the invariant discovery process as well as automated proof-failure analysis.

3.24 Abstraction in Formal Verification and Development

Christoph Lueth (DFKI Bremen, DE)


License  Creative Commons BY-NC-ND 3.0 Unported license

© Christoph Lueth

Besides correctness, another aspect of formal verification is that it turns the development into a formal object which we can reason over, and which we can manipulate. One possibility is abstraction, i.e. the process of making a given proof or development applicable in different situations. We talk about three different kind of abstractions here, datatype abstraction, development abstraction, and structure abstraction, to highlight potential avenues of research and their uses.

3.25 A study of cooperative online math

Ursula Martin (Queen Mary University of London, GB)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Ursula Martin

Joint work of Martin, Ursula; Pease, Alison

Main reference A. Pease, U. Martin, “Seventy four minutes of mathematics: An analysis of the third Mini-Polymath project,” in *AISB/IACAP 2012, Symposium on Mathematical Practice and Cognition II*, pp. 19–29, Birmingham, UK, July 2012.

URL <http://homepages.inf.ed.ac.uk/apease/papers/seventy-four.pdf>

Blogs, question answering systems and “crowdsourced” proofs provide effective new ways for groups of people, who may be unknown to each other, to use the internet to conduct mathematical research. They also provide a rich resource to shed light on mathematical practice and how mathematics advances, with the internet making visible and codified matters which have heretofore been ephemeral to study.


We discuss the first steps in such a research programme, looking at two examples to see what we can learn about mathematics as practiced on the internet. Does it differ from non-internet practice, and does it support or refute traditional theories of how mathematics is made, and who makes it, in particular those of Lakatos, or does it suggest new ones.

Polymath supports “crowdsourced proofs” and provides a structured way for a number of people to work on a proof simultaneously, capturing not only the final result, but also the discussion, missteps, informal arguments and social mechanisms in use along the way. Mathoverflow supports asking and answering research level mathematical questions and provides 25 thousand mathematical conversations for analysis, again providing a record of the informal mathematical activity that goes into answering them, and the social processes underlying production, acceptance or rejection of “answers”. We look at a sample of questions about algebra, and provide a typology of the kinds of questions asked, and consider the features of the discussions and answers they generate.

We hope that this work will lead to collaboration with the formal methods community, in understanding proof or in researching proof archives to compare and contrast with the maths community.

3.26 TLA+ Proofs

Stephan Merz (INRIA – Nancy, FR)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Stephan Merz

Joint work of Cousineau, Denis; Doligez, Damien; Lammport, Leslie; Merz, Stephan; Ricketts, Daniel; Vanzetto, Hernán

Main reference D. Cousineau, D. Doligez, L. Lammport, S. Merz, D. Ricketts, H. Vanzetto, “TLA+ proofs,” in D. Giannakopoulou and D. Méry, editors, *18th International Symposium on Formal Methods (FM 2012)*, volume 7436 of *LNCS*, pp. 147–154. Springer, 2012.

URL <http://arxiv.org/abs/1208.5933>


The TLA+ Proof System (TLAPS) is mainly intended for the deductive verification of (models of) distributed algorithms. At the heart of TLAPS lies a hierarchical and explicit proof language. Leaf proof steps are discharged by different automatic backend provers. In order to ensure the overall coherence of a proof, backend provers may provide a detailed proof that can be checked within Isabelle/TLA+, an encoding of TLA+ as an object logic in the logical framework Isabelle.

The system has been designed for developing large interactive proofs. In particular, the GUI provides commands for reading and writing hierarchical proofs by letting the user focus on part of a proof. TLAPS uses a fingerprinting mechanism to store proof obligations and their status. It thus avoids reproving previously proved obligations, even after a model or a proof has been restructured, and it facilitates the analysis of what parts of a proof are affected by changes in the model.

The paper is a longer version of an article published at FM 2012.

3.27 Case Based Specifications – reusing specifications, programs and proofs

Rosemary Monahan (Nat. University of Ireland, IE)

License  Creative Commons BY-NC-ND 3.0 Unported license

© Rosemary Monahan

Joint work of Monahan, Rosemary; O'Donoghue, Diarmuid

URL <http://www.cs.nuim.ie/staff/rosemary>

URL <http://www.cs.nuim.ie/staff/dod>

Many software verification tools use the design-by-contract approach to annotate programs with assertions so that tools, such as compilers, can generate the proof obligations required to verify that a program satisfies its specification. Theorem provers and SMT solvers are then used to, often automatically, discharge the proof obligations that have been generated.

While verification tools are becoming more powerful and more popular, the major difficulties facing their users concern learning how to interact efficiently with these tools. These issues include learning how to write good assertions so that the specification expresses what the program must achieve and writing good implementations so that the program verification is easily achieved [4, 5]. In this presentation we discuss guiding the user in these aspects by making use of verifications from previously written programs. That is by finding a similar or analogous program to the one under development, we can apply the same implementation and specification approaches. Our strategy is to use a graph-based representation of a program and its specification as the basis for identifying similar programs.

Graph-matching was identified as the key to elucidating analogical comparisons in the seminal work on Structure Mapping Theory [1]. By representing two sets of information as relational graphs, structure mapping allows us to generate the detailed comparison between the two concepts involved. So given two graphs we can identify the detailed comparison using graph matching algorithms. For one application of graph matching to process geographic and spatial data see [3]. However, we may not always have an identified “source” to apply to our given problem. Thus, more recent work has taken a problem description, searching through a number of potentially analogous descriptions, to identify the most similar past solution to that problem [2].

Our work will develop a graph matching framework for program verification. The associated tools will operate on a collection of previously verified programs, identifying specifications that are similar to those under development. The program associated with this “matching specification” will guide the programmer to construct a program that can be verified as correct with respect to the given specification. Likewise, the strategy can be applied when the starting point is a program for which we need to construct a correct specification.

The core matching process can be thought of as a K+J colored graph-matching algorithm,

which coupled with analogical transfer will re-apply the old solution to a new problem. Graphs can be flow graphs, UML diagrams, parse trees or another representation of a specification. Therefore, an iterative implementation of a sigma function (say) using tail recursion will be analogous to another recursive implementation—possibly using head-recursion. Similarly, iterative calculations of the same function using while and for loops will be more analogous to one another. Identical graph matching (isomorphism) will identify exact matches, given the representation, while non-identical (homomorphic) matches will identify the best available solutions.

In summary, our work will help to make software specification and verification more accessible to programmers by guiding users with knowledge of previously verified programs. A graphical representation of the specification, coupled with graph matching algorithms, is used as the basis of an analogical approach to support reuse of specification strategies.

References

- 1 D. Gentner. Structure-mapping: A theoretical framework for analogy. *Cognitive Science*, 7(2):155–170, 1983.
- 2 D. P. O’Donoghue and M. T. Keane. A creative analogy machine: Results and challenges. In M. L. Maher, K. Hammond, A. Pease, R. Pérez y Pérez, D. Ventura, and G. Wiggins, editors, *4th International Conference on Computational Creativity (ICCC 2012)*, pages 17–24, 2012.
- 3 D. P. O’Donoghue, A. J. Bohan, and M. T. Keane. Seeing things: Inventive reasoning with geometric analogies and topographic maps. *New Generation Comput.*, 24(3):267–288, 2006.
- 4 K. R. M. Leino and R. Monahan. Automatic verification of textbook programs that use comprehensions. In *9th Workshop on Formal Techniques for Java-like Programs (FTfJP 2007), ECOOP 2007 Workshop*, Berlin, Germany, July 2007.
- 5 K. R. M. Leino and R. Monahan. Dafny meets the verification benchmarks challenge. In G. T. Leavens, P. W. O’Hearn, and S. K. Rajamani, editors, *Verified Software: Theories, Tools, Experiments (VSTTE 2010)*, volume 6217 of *LNCS*, pages 112–126. Springer, 2010.

3.28 Can AI Help ACL2?

J Strother Moore (University of Texas at Austin, US)

License © ⓘ ⊖ Creative Commons BY-NC-ND 3.0 Unported license
© J Strother Moore

Joint work of Moore, J Strother; Kaufmann, Matt; Boyer, Robert

Main reference M. Kaufmann, P. Manolios, J. S. Moore, “*Computer-Aided Reasoning: An Approach*,” volume 3 of *Advances in Formal Methods*. Kluwer Academic Publishers, 2000.


URL <http://www.cs.utexas.edu/users/moore/acl2>

ACL2 stands for “A Computational Logic for Applicative Common Lisp,” and is a fully integrated verification environment for functional Common Lisp. I briefly mentioned some of its industrial applications, primarily in microprocessor design, especially floating-point unit design, and security. ACL2 is used to prove functional correctness of industrial designs. I then demonstrated an ACL2 model of the Java Virtual Machine highlighting (a) the size and scale of the formal model, (b) the fact that it was executable and thus was a JVM engine, and (c) ACL2 can be configured so that code proofs are often automatic. I then turned to how AI could help the ACL2 user, including: facilitating proof maintenance in the face of continued evolution of designs; facilitating team interaction in team based proofs (e.g., automatically informing team member A that team member B has already proved a lemma that seems

to be related to the one A is trying to formulate); intelligent, semantic-based search of the data bases of participating users exploiting the common, formal language used to express concepts; concept and lemma formation; and inductive generalization. I concluded with the lament that many people attracted to modern AI seem to be put off by formality. This is not to say they are imprecise, only that they seem more interested in studying less rigid systems than formal proof systems. I see theorem proving as a game, like chess: there are fixed rules that every game must follow, but there is plenty of room for statistical learning, probabilistic methods, and intelligent/heuristic strategies as long as they ultimately result in the recommendation of helpful legal moves. The title of this talk suggests there is a question of whether AI could help ACL2. But in fact, there is no question that it could, if the right sort of expertise is brought to bear on the problem.

3.29 Boogie Verification Debugger

Michał Moskal (Microsoft Research – Redmond, US)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Michał Moskal

Joint work of Moskal, Michał; Leino, K. Rustan M.; Le Goues, Claire


Main reference C. L. Goues, K. R. M. Leino, M. Moskal, “The Boogie Verification Debugger (tool paper),” in G. Barthe, A. Pardo, and G. Schneider, editors, *9th International Conference on Software Engineering and Formal Methods (SEFM 2011)*, volume 7041 of *LNCS*, pp. 407–414. Springer, 2011.

URL <http://research.microsoft.com/en-us/um/people/moskal/pdf/bvd.pdf>

The Boogie Verification Debugger (BVD) is a tool that lets users explore the potential program errors reported by a deductive program verifier. The user interface is like that of a dynamic debugger, but the debugging happens statically without executing the program. BVD integrates with the program verification engine Boogie. Just as Boogie supports multiple language front-ends, BVD can work with those front-ends through a plug-in architecture. BVD plugins have been implemented for two state-of-the-art verifiers, VCC and Dafny.

3.30 Formal software verification in avionics

Yannick Moy (AdaCore, Paris, FR)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Yannick Moy

Joint work of Comar, Cyrille; Kanig, Johannes; Moy, Yannick

Main reference C. Comar, J. Kanig, Y. Moy, “Integrating formal program verification with testing,” in J. Sifakis and J. Botti, editors, *Embedded Real Time Software and Systems (ERTS² 2012)*, 2012.

URL <http://www.open-do.org/wp-content/uploads/2011/12/hi-lite-erts2012.pdf>

Certification of civilian avionics software is a very costly process, partly due to the pervasive use of testing. To reduce these costs, the avionics industry is now looking at using formal methods instead of testing, which is allowed by the new version of the certification standard (DO-178C). Based on previous experience with formal verification of avionics software, we propose a methodology and tools based on formal methods to address the DO-178C objective of verifying that code correctly implements low-level requirements. Our approach simplifies the adoption of formal verification by using the executable semantics of assertions familiar to engineers, by relying on a combination of automatic proof and testing, and by having tools that support the development of specifications. We take advantage of the latest version (2012) of the Ada language, which includes many specification features like pre- and postconditions for subprograms.

3.31 What I've Learned from the VFS Challenge thus far

José Nuno Oliveira (Universidade de Minho – Braga, PT)

License © © ⊖ Creative Commons BY-NC-ND 3.0 Unported license
© José Nuno Oliveira

Joint work of Oliveira, José Nuno; Ferreira, Miguel Alexandre

Main reference J. N. Oliveira, M. A. Ferreira, “Alloy meets the algebra of programming: A case study,” *IEEE Transactions on Software Engineering*, 99(PrePrints), 2012.

URL <http://dx.doi.org/10.1109/TSE.2012.15>

The question “Is Abstraction the Key to Computing?” (Jeff Kramer) is central to formal methods. The “yes!” answer is widely accepted, but perhaps there are other unanswered questions, for instance: “Are we using the right notation, formal language?”

In this talk I described how the experience in handling a concrete case study—the Verified File System (VFS) challenge in computing put forward by Joshi and Holzmann—changed my way of doing FMs, starting from a grand scale tool-chain involving several notations and tools (model checkers, animators, theorem-provers) to a minimalist approach, relying on quantifier-free relational notation and the Alloy model checker only.

Another question (central to the seminar) is “How can AI help?” The same experience points towards two fields where this may well happen in the future: requirements “engineering” based on semantics-rich boilerplates and ontologies of invariant theories specified around what in the talk was referred to as “the 4-relation invariant pattern”.

3.32 The Need for AI in Software Engineering – A Message from the Trenches

Stephan Schulz (TU München, DE)

License © © ⊖ Creative Commons BY-NC-ND 3.0 Unported license
© Stephan Schulz


I discuss existing structured software development processes as used by a medium sized system integrator in the field of Air Traffic Control, highlighting the variability of the processes driven by both customer capability and market demands. There often are large gaps between current practice and the prerequisites for large-scale application of formal methods.

I suggest how AI can help to bridge some of these gaps and can improve industrial software development processes as used by SMEs. One particular hotspot which could profit from AI techniques is requirements engineering. In requirements capture and engineering, AI techniques can help structuring the requirements, guiding the refinement process, and point out where specifications appear to be unclear or contradictory.

Another significant problem is the large amount of existing legacy code, often embodying decades worth of domain knowledge and testing, but usually not written to today’s standards. The ability to re-engineer existing code bases, documenting dependencies, side effects, pre- and postconditions, and invariants, would make existing libraries much more amendable to be combined with more rigorously developed new code.

3.33 Finding and Responding to Failure in Large Formal Verifications

Mark Staples (NICTA, AU)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Mark Staples

The L4.verified project has completed the formal verification, to the level of C source code, of the full functional correctness of the seL4 microkernel [2]. The project proceeded in several phases, with internal iterations, and ongoing maintenance [1]. The team can now claim there are zero bugs in the microkernel, subject to assumptions and conditions. The scale and detail in the project raise challenges of potential interest for Artificial Intelligence (AI) and Formal Methods (FM). I discuss two of these.

Firstly, changes to the specification, design, code, and invariants are almost inevitable in large formal verification projects, because of bug fixes or enhancements. Changes mean the system must be re-verified, and all lemmas must be re-proved. The automated re-proof of some lemmas may fail, because they are no longer true and must be reworked, or because the proof scripts are too fragile and must be reworked. Reverification is a management and technical challenge for which AI techniques may be relevant. Specific challenges include: In what order should lemmas be reworked? How can proof scripts be made more robust to minor changes to lemmas?

Secondly, the existence of extra-logical “gaps” between formal models and the real world (actual requirements and implementations) is well known in the FM community. Formal methods are empirical too, because the properties proved about software are claims about how it will behave and satisfy requirements in the world. Nonetheless, it is less well known how to address these gaps. The L4.verified team identified some extra-logical assumptions, including that assembly-code, C compiler, and hardware are correct. What other key assumptions might there be, and how can we identify them? Can heuristic search techniques help to identify or test such assumptions?

References

- 1 J. Andronick, D. R. Jeffery, G. Klein, R. Kolanski, M. Staples, H. Zhang, and L. Zhu. Large-scale formal verification in practice: A process perspective. In M. Glinz, G. C. Murphy, and M. Pezzè, editors, *34th International Conference on Software Engineering, (ICSE 2012)*, pages 1002–1011. IEEE, 2012.
- 2 G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010.

3.34 Formal Verification of QVT Transformations

Kurt Stenzel (*Universität Augsburg, DE*)

License © © ⊖ Creative Commons BY-NC-ND 3.0 Unported license
© Kurt Stenzel

Joint work of Stenzel, Kurt; Moebius, Nina; Reif, Wolfgang

Main reference K. Stenzel, N. Moebius, W. Reif, “Formal verification of QVT transformations for code generation,” in J. Whittle, T. Clark, and T. Kühne, editors, *14th International Conference on Model Driven Engineering Languages and Systems (MODELS 2011)*, volume 6981 of *LNCS*, pp. 533–547. Springer, 2011.

URL http://dx.doi.org/10.1007/978-3-642-24485-8_39

We present a formal calculus for operational QVT. The calculus is implemented in the interactive theorem prover KIV and allows to prove properties of QVT transformations for arbitrary meta models.

Additionally we present a framework for provably correct Java code generation. The framework uses a meta model for a Java abstract syntax tree as the target of QVT transformations. This meta model is mapped to a formal Java semantics in KIV. This makes it possible to formally prove with the QVT calculus that a transformation always generates a Java model (i.e. a program) that is type correct and has certain semantical properties. The Java model can be used to generate source code by a model-to-text transformation or byte code directly.

Finally, we report on experiences with the development of the new calculus.

3.35 Modeling and Proving

Werner Stephan (*DFKI – Saarbrücken, DE*)


License © © ⊖ Creative Commons BY-NC-ND 3.0 Unported license
© Werner Stephan

Joint work of Stephan, Werner; Cheikhrouhou, Lassaad; Langenstein, Bruno; Nonnengart, Andreas

To turn Formal Methods into an engineering discipline (beyond use in the academic community) modeling has to be taken seriously: attack the real problems, make explicit (hidden) assumptions, follow certain guidelines, provide (informal) interpretations for the interaction with non-formal parts, and allow for a third party assessment. To that end we still need modeling experts. Although their expertise might be domain specific they will be able to perform proofs in state-of-the-art interactive proof systems. However, typically they will not be able to develop complete (proof) strategies. Interactive systems offer the flexibility that is often needed for adequate models. Considerable progress has been made with respect to partial automation, user interaction, and the engineering of interactive proofs. For example our proof strategy for protocol verification selects around 85% of the steps automatically. In simple standard case it gets close to 100%. Unfortunately sophisticated real world problems in many cases tend to be ‘out of range’ of a given fixed strategy (to a varying degree). Adapting strategies still requires (sometimes a lot of) paper work and is completely out of range for users. The challenge (dream) therefore is to support the development of strategies within the system in an interactive way such that at least for simpler cases even users are able to perform these modifications themselves.

3.36 Overview of CSP||B for railway modelling

Helen Treharne (University of Surrey, GB)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Helen Treharne

Joint work of Moller, Faron; Nguyen, Hoang Nga; Roggenbach, Markus; Schneider, Steve; Treharne, Helen


CSP||B is a formal approach that has been developed at the University of Surrey over a number of years; it combines two well-established formal methods: CSP and B. At the heart of the method is a compositional verification framework. Our recent work has been using CSP||B in the verification of railway systems in collaboration with the University of Swansea. Our motivation is to develop a modelling and verification approach accessible to railway engineers: it is vital that they can validate the models and verification conditions, and—in the case of design errors—obtain comprehensible feedback. In this talk, we presented an overview of the style of formalization that we have adopted. It is aligned with the way engineers think about railway systems, and we have involved our industrial partner in detailed discussions at every stage. The railway models can become large when complex track plans are being modelled. We also discussed our ongoing work which is focusing on identifying abstractions of track plans so that model checking complex track plans is possible.

References

- 1 F. Moller, H. N. Nguyen, M. Roggenbach, S. Schneider, and H. Treharne. Combining event-based and state-based modelling for railway verification. Technical Report CS-12-02, University of Surrey, 2012.
- 2 F. Moller, H. N. Nguyen, M. Roggenbach, S. Schneider, and H. Treharne. Railway modelling in CSP||B: the double junction case study. In *12th International Workshop on Automated Verification of Critical Systems (AVOCS 2012)*, Bamberg, Germany, September 2012. (to appear).

3.37 AI via/for Large Mathematical Knowledge Bases

Josef Urban (Radboud University Nijmegen, NL)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Josef Urban

Joint work of Urban, Josef; Vyskočil, Jiří

Main reference J. Urban, J. Vyskočil, “Theorem Proving in Large Formal Mathematics as an Emerging AI Field,” Dagstuhl Preprint Archive, arXiv:1209.3914 [cs.AI], 2012.

URL <http://arxiv.org/abs/1209.3914>

The talk introduces the recently appeared large mathematical knowledge bases as a suitable repository for combining deductive and inductive AI methods. Several examples of ATP/AI systems working in this setting are given, including the Machine Learner for Automated Reasoning (MaLAREa) and the Machine Learning Connection Prover (MaLeCoP). Some lessons learned from working on such systems are discussed and some future work topics are mentioned.

3.38 Capturing and Inferring the Proof Process (Part 2: Architecture)

Andrius Velykis (Newcastle University, GB)

License © © ⊖ Creative Commons BY-NC-ND 3.0 Unported license
© Andrius Velykis

Main reference A. Velykis, “Inferring the proof process,” in C. Choppy, D. Delayahe, and K. Klai, editors, *FM2012 Doctoral Symposium*, Paris, France, August 2012.

URL <http://www.cs.ncl.ac.uk/publications/trs/papers/1344.pdf>

Interactive theorem proving can be used to verify formal models and specifications as well as justify their development process. A large portion of the proof can be automated using general heuristics available in state-of-the-art automatic theorem provers, but significant manual work still gets left for human experts.

In this talk we ask how enough information can be collected from an interactive formal proof to capture an expert’s ideas as a high-level proof process. Such information would then serve for extracting proof strategies to facilitate automation of similar proofs. We explore the question: what describes a proof process? The talk presents our take: we need structural information (e.g. proof granularity / multiple attempts) as well as proof meta-information (e.g. proof features). Furthermore, we present the architecture of a new ProofProcess framework, which is developed to support collecting and inferring the proof process (automatically, or by asking the expert). It aims to provide a generic way to capture proof process information from different theorem provers.

3.39 More Abstractions

Laurent Voisin (SYSTEREL – Aix-en-Provence, FR)

License © © ⊖ Creative Commons BY-NC-ND 3.0 Unported license
© Laurent Voisin

Based on 10+ years of formal modelling in industry, I advocate the use of domain theories and modelling patterns in system modelling.

When modelling a system, one has to somehow encode some domain data structures into some mathematical notation (e.g., set-theory for Event-B). This encoding is not trivial, except for very simple case studies. Inlining this encoding within a model makes it difficult to read and consequently difficult to prove. It is much better to separate the encoding in a separate file (i.e., a theory) which will describe the data structure, provide operators for updating it together with proof rules for reasoning about them. The model is then free from clutter and can be expressed at the same level of discourse as domain experts.

I think that AI could provide significant benefits by detecting when a model is not at the correct level of discourse and contains too much encoding. This could be detected by inspecting the model and assessing its intrinsic complexity. This would be particularly useful for beginners who usually have difficulty to separate concerns.

Another use of AI is to implement refinement plans (see paper by Grov, Ireland and Llano presented at ABZ 2012). In this setting, a failing proof is analysed with respect to some refinement patterns and the tool suggests amendment to the model that would allow to fix its proof. I think that it would be much more valuable if the refinement patterns would be in the form of generic models. The tool would then propose to instantiate the generic pattern and suggest ways to instantiate it (e.g., provide actual parameters). This would


reuse not only the pattern but also its associated proof. The user would only have to prove that the actual parameters fulfill the pattern pre-conditions.

More generally, AI could be used to mine existing models to extract generic patterns from them. This would allow to build a library of recurring patterns. As for refinement plans, AI could also be used for guiding users within the library and help them select the appropriate patterns with respect to their modelling needs.

In conclusion, using both theories and generic model patterns makes models more easy to develop, read and prove, by allowing better reuse. AI could be of great help in assisting users for making the better use of these tools.

3.40 Finding Counterexamples through Heuristic Search

Martin Wehrle (Universität Basel, CH)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Martin Wehrle

AI planning considers the task of automatically finding a sequence of actions such that applying these actions leads to a state that satisfies a given goal condition. A popular approach to solve planning tasks is based on heuristic search.

From an abstract point of view, AI planning is related to finding counterexamples in model checking. This talk shows the basic idea of computing distance heuristics automatically based on a general problem description, and shows how heuristic search techniques can be applied to finding counterexamples in model checking.

Participants

- Rob Arthan
Lemma 1 Ltd. – Reading, GB
- Serge Autexier
DFKI Bremen, DE
- Alan Bundy
University of Edinburgh, GB
- Simon Colton
Imperial College London, GB
- David Crocker
Escher Technologies –
Aldershot, GB
- Jorge Cuellar
Siemens – München, DE
- Ewen W. Denney
NASA – Moffett Field, US
- Leo Freitas
Newcastle University, GB
- Dimitra Giannakopoulou
NASA – Moffett Field, US
- Gudmund Grov
University of Edinburgh, GB
- Reiner Hähnle
TU Darmstadt, DE
- Dieter Hutter
DFKI Bremen, DE
- Andrew Ireland
Heriot-Watt University
Edinburgh, GB
- Moa Johansson
Chalmers UT – Göteborg, SE
- Cliff B. Jones
Newcastle University, GB
- Ekaterina Komendantskaya
University of Dundee, GB
- Thierry Lecomte
ClearSy – Aix-en-Provence, FR
- K. Rustan M. Leino
Microsoft – Redmond, US
- Michael Leuschel
Universität Düsseldorf, DE
- Yuhui Lin
University of Edinburgh, GB
- Maria Teresa Llano Rodriguez
Heriot-Watt University
Edinburgh, GB
- Christoph Lüth
DFKI Bremen, DE
- Ursula Martin
Queen Mary University of
London, GB
- Stephan Merz
INRIA – Nancy, FR
- Rosemary Monahan
Nat. University of Ireland, IE
- J Strother Moore
University of Texas at Austin, US
- Michał Moskal
Microsoft – Redmond, US
- Yannick Moy
AdaCore – Paris, FR
- José Nuno Oliveira
Univ. de Minho – Braga, PT
- Thomas Santen
European Microsoft Innovation
Center – Aachen, DE
- Stephan Schulz
TU München, DE
- Volker Sorge
University of Birmingham, GB
- Mark Staples
NICTA, AU
- Kurt Stenzel
Universität Augsburg, DE
- Werner Stephan
DFKI – Saarbrücken, DE
- Helen Treharne
University of Surrey, GB
- Josef Urban
Radboud Univ. Nijmegen, NL
- Andrius Velykis
Newcastle University, GB
- Laurent Voisin
SYSTEREL –
Aix-en-Provence, FR
- Martin Wehrle
Universität Basel, CH

