Case for support

AI4FM: using AI to aid automation of proof search in Formal Methods

Cliff B Jones and Alan Bundy (PIs)
Andrew Ireland (CI)

Previous Research

Cliff Jones (PI Newcastle)

Cliff Jones is one of the longest serving contributors to "formal methods" research having started his work in the IBM Vienna Lab in the late 1960s. He has an international reputation in both programming language semantics and program specification/development methods. Having worked on the early "operational semantics" (VDL) descriptions of programming languages, he played a key role in the move to "denotational" semantics and VDM (see [Jon01]).

Of more direct relevance to this grant application is his widely recognised contribution to development methods for programs [Jon99]. Between his two stays in Vienna, and afterwards in Belgium, he developed the more widely known aspects of VDM for program specification and design (data reification etc.).

He did a belated Doctorate with Tony Hoare in Oxford — the resulting Rely/Guarantee approach to developing concurrent programs is achieving ever wider recognition.

As well as a successful academic career (including professorships at Manchester (1981–96) and Newcastle (1999 onwards) and a Research Council *Senior Fellowship*), he has spent over twenty years in industry. His research is always fed by real problems and he still has extensive collaborations with industry.

Apart from being one of the founders of the seminal VDM research, his work most closely related to this grant is on support tools for formal methods. He built one system (FDSS) for IBM and then led the (Manchester and Rutherford Labs) effort that delivered the *mural* system for reasoning about program developments [JJLM91].

Currently, he leads the "Methodology" activities on a major EU (IP) project: *DEPLOY*, which is promoting the use of support systems in major European industries such as Bosch, SAP and Siemens. Specifically, this project is deploying and developing the *Rodin Toolset* from an earlier EU-funded Strep project (Newcastle was/is coordinating partner on both of these EU projects). His research on the "Logic of Partial Functions" is also highly relevant to this application (see WP4).

Cliff Jones is an FACM and FREng. He has written ten books (two of which are also published in translation), edited an additional eleven conference proceedings and has published over 150 papers (much of the work in industry stopped at reports rather than becoming external papers). He also has a track record of successful leadership of large projects — most recently venturing into interdisciplinary research, when he led the six year Dependability IRC.[1] He also has a Platform grant (TrAmS) on "Trustworthy Ambient Systems".

Alan Bundy (PI Edinburgh)

Alan Bundy has been active in mathematical reasoning research since 1971 and has become a world authority. His international reputation is witnessed by his being made a founding fellow of both of the international AI societies: AAAI and ECCAI, in addition to the UK society, AISB, and serving terms as Chair of both IJCAI Inc and CADE Inc. He is also a Fellow of the Royal Society of Edinburgh and the Royal Academy of Engineers. He won the SPL Insight Award in 1986, the IJCAI Distinguished Service Award in 2003, the IJCAI Research Excellence Award in 2007, the CADE Herbrand Award in 2007, was an SERC Senior Fellow (1987–92), a member of the Hewlett-Packard Research Board (1989–91), Head of the Division of Informatics at Edinburgh (1998–2001), a member of the ITEC Foresight Panel (1994–96), a member of both the 2001 and 2008 Computer Science RAE panels (1999–2001) and (2005–2008), a member of the Scottish Science Advisory Committee (2008–date) and was the founding Convener of UKCRC (2000–2005). He is the author of over 200 publications.

He is best known for the development of *proof planning* [Bun91] and *rippling* [BBHI05]. A proof plan describes the common structure of a family of proofs as a hierarchy of *proof methods*. Abstracted from a training set of example proofs, a proof plan can be used to guide the proofs of similar theorems, significantly reducing the need for search. Associated with some proof methods are collections of *proof critics* that provide common patches to failed attempts to use a method, e.g. a case split, a new induction rule or variable, an intermediate lemma or a generalisation of the theorem. Rippling is one of the most powerful proof methods used in proof planning. It applies whenever a *given*[2] can be used to prove a structurally similar *goal*. An embedding of the given in the goal is used to identify expressions that are blocking the use of the given. Rippling rewrites the goal to move these obstructive expressions out of the way allowing the given to match a subexpression of the rewritten goal. Rippling is the key to successful inductive proof, but also finds other applications. For instance, a rippling-based analysis of failed verification proofs of imperative programs can be used to suggest improved loop invariants, leading to a successful reattempted proof [SI98]. Note especially the *productive use of failure* in this technique, i.e. the analysis of an earlier failed proof attempt to inform an improved approach.

Proof plans provide an existence proof of a high-level description of a whole family of proofs. In addition, our project requires the automated abstraction of such high-level descriptions from an example proof from a family of similar proofs. Bundy's group has conducted several projects into automated learning of proof plans from exam-

[1] DIRC see `www.dirc.org.uk`

[2] E.g. a definition, axiom, assumption or previously proved theorem.

ple proofs. These projects include: the automated learning of equation-solving methods from example solutions [Sil85]; and the data-mining of a corpus of Isabelle proofs to extract rule sequences occurring more frequently than chance, then generalising these into new tactics [DBL+04].

Michael Butler (Consultant)

Michael Butler[3] has kindly agreed to donate his time on AI4FM for free — he is a Professor of Computer Science at the University of Southampton. His research interests include applications, tools and methodology for formal methods, especially refinement based methods such as B and Event-B. His work on applications of refinement have contributed to advancing the methodology of this approach. He has played a leading role in the development of several tools for B and Event-B. In the EU FP7 DEPLOY project, Butler plays the role of *tooling coordinator* ensuring the Rodin tool development is coordinated across multiple sites and is properly aligned with the needs of the DEPLOY industrial partners. Butler also played a leading role in the development of the ProB and UML-B tools. ProB is a powerful validation tool for the B method developed by Leuschel (now at Dusseldörf) and Butler. ProB's automated animation facilities allow users to gain confidence in their specifications. ProB also contains an automated model checker and refinement checker, which can be used to detect various errors in B specifications. The latest versions of ProB and UML-B are now integrated with Rodin as plug-ins.

The main aim of the DEPLOY project (led by Newcastle) is achieving deployment of Event-B and the Rodin Toolset in real industrial settings with particular emphasis on ensuring system resilience. In the DEPLOY project, Butler collaborates with engineers in Siemens Transportation (rail systems), Bosch (automotive systems), SAP (business systems) and Space Systems Finland. This involves supporting deployment of formal methods and tools and identifying further tool and method requirements. He has also worked on applying formal methods to industrial problems as part of previous EU projects (MATISSE, PUSSEE, Rodin). These projects were concerned with methods and industrial application of formal methods, which included industrial partners such as Siemens, Nokia, Volvo, Praxis and Gemplus. Butler is on the programme committees of many international conferences. He was Programme Chair of the 2007 International Conference on Formal Engineering Methods (ICFEM2007) and Co-Chair of the 2008 International Conference on ASM, B and Z (ABZ2008).

Andrew Ireland (CI Heriot-Watt)

Dr Ireland is a Reader in Computer Science within the School of Mathematical and Computer Sciences (MACS) at Heriot-Watt University. He has substantial research experience in the area of *automated reasoning* and *automated software engineering*. Dr Ireland has played a central role in the development of proof planning since 1990. He has also been involved in more applied research. As principal investigator of the GR/R24081 NuSPADE project he successfully applied the proof planning ideas to the SPARK Approach for developing safe and secure software. This involved a strong collaborative element with Praxis High Integrity Systems Ltd (Bath) which led to a follow-on Knowledge Transfer project GR/T11289. The NuSPADE project gave rise to an integrated approach to software verification. Currently Dr Ireland has two EPSRC projects: firstly, EP/F037597 which is investigating proof planning within the context of *separation logic*, a novel logic that supports scalable reasoning for the verification of pointer programs; secondly, EP/F037058 is investigating the use of Lakatos-style reasoning within the context of software modelling. During his 13 years at Heriot-Watt, Dr Ireland has maintained a close collaborative relationship with Prof. Bundy's Mathematical Reasoning Group (MRG) at the University of Edinburgh and is currently a co-investigator on Prof. Bundy's platform grant GR/S01771. Dr Ireland has organised a number of national and international workshops in automated reasoning and is a member of programme and organising committees for a series of national and international workshops and conferences (e.g. ASE, CADE and VSTTE). He reviews papers for major journals, i.e. JAR, AMAI and JLC. He has been a member of the EPSRC Peer Review College since 2003.

Gudmund Grov (RA Edinburgh)

Dr Grov finished his PhD thesis in March 2009 (and graduated in June 2009). He studied in the Dependable Systems Group (DSG) at the School of Mathematical and Computer Sciences (MACS) at Heriot-Watt University. He is currently employed as a research assistant on Dr Andrew Ireland's EP/F037597 project within DSG. From November 2008 to April 2009, he was employed as a research assistant on Prof. Bundy's platform grant "The Integration and Interaction of Multiple Mathematical Reasoning Processes" in the Mathematical Reasoning Group in Edinburgh University.

Dr Grov started his PhD in October 2004 and whilst working towards his PhD, he has been employed as an undergraduate mathematics lecturer at Narvik University College over the course of two summers. He has also tutored at both MSc and undergraduate levels and has co-supervised a MSc dissertation at MACS. Before starting his PhD in 2004, he obtained a MSc (with distinction) in "Distributed and Parallel Information Systems", also from MACS.

Dr Grov's research expertise lies in the area of formal verification and software specification. His PhD has focused on verifying and transforming software written in Hume, a new and novel programming language, using the Temporal Logic of Actions (TLA). Through his research he has experience with both model checking (e.g. Spin & TLC) and theorem proving (e.g. Isabelle, PVS, & the SPADE Proof Checker). His PhD thesis has already given rise to seven academic publications and has been nominated for the BCS Distinguished Dissertations competition.

[3] In order to save space, we have not cited publications for either the Consultant or the CI — they are both prolific authors.

1 Background

"Formal methods" use languages with known semantics to record specifications and designs of (software) systems. The gradual evolution of languages that are both tractable and expressive has led to a growth of their use in practical software development applications. The use of such formal methods is no longer confined to safety critical systems (the list of industrial partners in the EU DEPLOY project[4] is one indication of this broader use).

Most formal methods give rise to "proof obligations" (POs) which are putative lemmas that need proof. Discharging these POs can become a bottleneck in the use of formal methods in practical applications. Some techniques for reducing this bottleneck are known — it is our aim to increase the repertoire of techniques by tackling learning from proof attempts.

One key approach that maximises the use of a developer's intuition is to use methods that encourage the teasing apart and recording of design decisions. Having a formal specification and a large program that satisfies it would give rise to much less tractable proof obligations than a multi-step data reification followed by carefully decomposing post-conditions to code. The Event-B approach (see below) is an excellent example of such an approach. Where extra development steps are made, the larger number of proof obligations generated is more than compensated for by their simplicity.

Another crucial aid to reducing the bottleneck is the use of heuristics to help proof search. All theorem proving assistants provide heuristics to some extent. Undecidability shows these can never work for all POs; resource exhaustion is likely to be a more telling limitation.

There remains the problem of what to do with POs that are *not* discharged automatically. This can of course indicate that there is an error in the design step and here a user also needs support in locating the error. However, in many cases where a correct PO has not been discharged, an expert can easily see how to complete a proof. We believe that it would be acceptable to rely on such expert intervention to do one proof if this would enable a system to kill off others "of the same form". To see how useful such a facility could be, note that at the first review of DEPLOY, one of the industrial partners reported an application that gave rise to some 300 POs; of these, about 200 were discharged automatically; five really difficult proofs were done by hand; although the remaining 95 "followed the pattern" of the five, they also had to be done slowly and manually because the current tool has only fixed heuristics — it does not learn from the user.

Section 2.1 sets out our research hypothesis which indicates why we believe this situation can be improved.

1.1 Formal methods

There is no attempt here to justify the claim that formal methods are used ever more widely; there is ample evidence of this.[5] It is, however, important to acknowledge that there are differing ways in which formal methods are being deployed. To simplify the comparison, these can be thought of as top-down and bottom-up. In its simplest form, a top-down process starts with a specification, proceeds through design steps that add detail and finishes with executable code plus, perhaps more importantly, a design history. Of course, all realistic developments experience requirements change during development and, even if they didn't, designers would make false steps that have to be re-done. But the essence of "verified by construction" approaches is that there exists, in the end, a layered description of the final implementation where the layers are clearly related by formal proofs.

A similarly simplified view of the alternative "bottom-up" view is that one works from an existing body of (possibly legacy) code and tries to prove properties of that code. Even with simple results like avoiding de-referencing null pointers, the only way to avoid excessive numbers of false positive diagnostics is for someone who understands the intention of the program to add assertions. Abstract interpretation is an approach to extracting deeper properties of programs and the results can be impressive — but handling complex programs again requires that users indicate useful abstractions which can be likened to adding assertions.

Stepping away from the simplified descriptions, one can readily see that similar proof tasks can arise in either mode. Our proposal is deliberately written around the "top-down", verified-by-construction, mode if only because the DEPLOY project offers us ready access to industrial use data in this style. Although we expect the results of our project to be more widely applicable, we will focus on POs generated by formal methods being used in a verified-by-construction style. Such POs tend to be rather uniform. Unfortunately, the overall uniformity does not make the discharge of such POs simple because each application will use its own data structures and have peculiarities in the operators used in its pre and post conditions.

Our objective is to significantly decrease the human effort in doing top-down formal development. We believe this can be achieved by increasing the proportion of POs that are discharged by the system. (More detail is given in Section 2.2.)

1.2 Lessons from industry

We have significant experience of the use of formal methods in industry; furthermore we have extensive contacts who are "at the coal face" of such application. It is convenient for the moment to focus the discussion on the two EU projects: "Rodin" was an EU (STREP) project that created the Rodin Toolset that provides machine support for development in the Event-B style; "DEPLOY" is an ongoing project that is deploying the methods and tools from Rodin with industrial collaborators. A huge bonus of this project is the ability to cull realistic examples of POs that are not discharged automatically. We have the agreement of our partners to add code to the tools to extract such data.

[4] DEPLOY is an EU-funded "IP" led by Newcastle University; it is a four year project with a budget of about 18M Euros; the industrial collaborators include Siemens Transport, Bosch and SAP.

[5] A useful review is about to be published by Woodcock et al. in ACM's Computing Surveys.

The experience with the earlier industrial grouping in Rodin is being echoed in DEPLOY: industrial software engineers often struggle with formal proofs. It is of course precisely the attraction of "push button" techniques, such as model checking, that users can locate errors without needing to understand the underlying formalism. Hiding the formalism is harder with proofs but the methods already available in the Rodin Toolset are powerful enough that –with a well chosen sequence of abstraction layers– well over 80% of the generated POs are discharged automatically. The remaining cases however consume significant amounts of valuable engineer resource. Furthermore, it is wasteful and frustrating that once one PO has been proved by hand, similar POs still require manual intervention. The notion of similarity here includes the relevant data structures and possibly the form of proof obligation under consideration. If we can arrange that the system can learn from the hand discharge of one of a family of POs so that this increases the chance that others can be discharged automatically, we can significantly reduce the (personnel and time) costs involved in using formal methods.

1.3 Our chosen framework

Without questioning the value of any other approach to improving software development, we wish to be precise about the setting to which we hope to contribute.

We are interested in helping engineers discharge POs that arise in the development process from a (formal) specification to a completed design. We understand the research on the extraction of programs from proofs and work on automatic synthesis of programs but it appears to us that the scope of such methods will necessarily remain more limited than the "posit and prove" approach where a designer posits a reification step and then seeks to justify it. Furthermore, we see the redundancy inherent in posit and prove approaches like those of VDM or Event-B as invaluable in detecting mistakes. (We concede that some errors are in specifications: such errors are normally exposed by posit and prove methods but could clearly result in incorrect programs were one able to synthesise a program from an incorrect specification.)

Model checking can be invaluable for locating errors and thus clearly has a role to play in obtaining high quality software. WP2.3 discusses how we plan to use automatic tools in the detection of mistakes before fruitless proof attempts are made.

Proving the internal coherence of one level of description –and showing that a more detailed level satisfies a more abstract description– gives rise to POs. It is not possible to achieve fully automatic proofs of development steps even where specification and reification are both correct. In fact, "Proof Obligation Generation" (POG) is a way of breaking into manageable steps the bigger claim that the behaviour of the abstract and more concrete models is the same. Sadly, even with good general heuristics, not all POs will be discharged automatically in non-trivial developments. Our claim is that the percentage that can be discharged can be increased significantly if a system is designed that –in addition to having a fixed set of heuristics– can learn from users' interactions.

Experience also tells us that change is of the essence and the impact of this gives another insight into the advantages of learning. When a change is made, a user has to redo proofs. The Rodin Toolset is already good at tracing the impact of changes and reducing the proof rework. But it would be a bonus if the system could learn enough from hand proof attempts at pre-change POs to discharge automatically similar proof obligations after a change. Of course, there will always be differences that defeat this strategy.

2 Hypothesis and objectives

2.1 Main research hypothesis

The main hypothesis to be addressed in this project is:

> *enough information can be automatically extracted from a hand proof that examples of the same class can be proved automatically.*

We believe that it is possible to build a system that will learn enough from one proof attempt to significantly improve the chances of proving "similar" results automatically. By "proof attempt" we include things like the order of the steps explored by the user (not just the finished chain in the final proof). Thus it is central to our goal that we find *high-level* strategies capable of cutting down the search space in proofs. What we are looking for is at a much higher level than LCF-style tactic languages — such tactics are programs to construct proofs and are brittle in the sense that they behave differently for similar POs.

We believe that by separating information about data structures and approaches to different patterns of POs, a taxonomy begins to evolve. A PO approach might be seen to use "generalise induction hypothesis" in a specific proof about, say, sequences; a future use of this PO might involve a more complicated tree data structure — but if it has an extended induction rule (e.g. by adding an argument to accumulate values), the same strategy might work.

So our hypothesis can be recast as:

> *we believe that it is possible to (devise a high-level strategy language for proofs and) extract strategies from successful proofs that will facilitate automatic proofs of related POs.*

2.2 Research objectives

The overall objective of the project is ambitious. We therefore think it prudent to recognise three sub-objectives:

1. design of a high-level proof strategy language;
2. modify one or more systems to extract strategies during manual proof attempts;
3. modify one system to use strategies to automate production of the target proofs from the same family.

A minimal outcome of the project is to develop a language for *hand* annotation of proofs (this is anyway a stepping stone towards our real goal) — in such a fallback position, an expert would mark up a typical proof for a class of claims. We are, however, optimistic that we can achieve the second and will certainly aim for the third level objective. The strategy language developed under our first objective will be used by the tools developed within the last two objectives. The language will be used internally by the tools, thus when the user creates the "typical proof", the existing theorem provers will still be used.

Our overall goal differs from research known as "analogy": we are not striving for automatic detection of similar "claims". Nor are we expecting a fixed set of "patterns" — the system must learn how elements were deployed in the example proof. We are however interested in what might be learned from "dead ends" in proof attempts and how the expert recovered from them (as well in successful strategies).

It is important to be clear that we are not looking for some dramatic advance that will magically improve the power of a general theorem prover: we are investigating the extent to which a new source of information (culled from hand interaction) can be used in subsequent proofs. In relation to the general aims given in WP7, the method, the data structures in play (and perhaps even the POG) might provide a natural "taxonomy of proof challenges" in the sense that the form of proof expected by any particular pair of POG and data structure might succumb to similar proof strategies — ones that have been learnt by the system from user interaction.

Another important word of explanation is in order with respect to our involvement with the "Rodin Toolset". We, in fact, plan to be more general than a focus on Event-B. One reason for this is that interaction with the Rodin Toolset often conflates steps to get proofs to work with reformulations of the intermediate models. A more obvious reason is that there is a larger potential payoff if we can achieve something that is applicable to more methods. We see a real opportunity to contribute ideas that will be applicable to most "model based" development methods.[6]

2.3 Indication of level of "strategy language"

Designing the strategy language is a part of our project but it might be useful to give some indications of what we expect it to look like. Our strategy language will combine a high-level proof strategy with a "vocabulary" of terms that might be instantiated in the separate theories of data structures stored in the system. The meta-language employed in our rippling/induction proof-planning work provides an existence proof for such a strategy language, and a 'version 0' on which we would expect to build using the experience of the development examples.

The discovery/construction of this language will be a major deliverable of the project. Items that we expect to play a major part include:

- *Some 'standard' proof plans and known deviations from and patches to them.* [BBHI05] uses rippling to describe a 'standard' proof plan for inductive proofs and shows how each different pattern of failure in rippling suggests a different way of patching a failed proof attempt. We hypothesise that expert-provided proofs of undischarged POs will typically exhibit either a new proof plan or a new patch to an existing plan. The challenge in the project is to devise a strategy language that can be used to abstract the expert's proof into the new plan or patch, so that it can be applied to automate the construction of the remaining proofs from the same family.

- *Choices of unusual induction rules and variables, choices of loop invariants.* Choosing an alternative non-standard induction rule is one of the patches to the standard induction proof plan which is described in [BBHI05]; patching a failed loop invariant is the patch described in [SI98].

- *Choices of intermediate lemmas.* Designing, constructing and proving a key intermediate lemma is one of the patches to the standard induction proof plan described in [BBHI05].

- *Generalisation of the PO*[7]. [BBHI05] also describes a couple of ways of generalising the PO or the current goal to patch a failed proof.

- *Insertion of a case analysis.* Similarly, [BBHI05] describes how to cue a case analysis to fix a failed proof.

Lemmas, case splits, loop invariants, generalisations and their points of application all need to be described in an abstract form if they are to apply to all members of a family of proofs. This is because the details will vary from proof to proof, but there may be a level of abstraction at which their descriptions coincide. Rippling, for instance, provides an exemplar abstract language, since it can describe 'missing' intermediate lemmas in terms of subexpressions that must match with different parts of the current goal. We will also make use of generic taxonomies, for instance, types of induction rule, types of generalisation, etc. to support the abstraction of proofs and their subsequent application to new conjectures. For instance, the use of a two-step induction on a recursive data-structure in the source proof must first be abstracted in order to be applied to a different data-structure in the target proof. We expect to develop and use additional kinds of abstraction during the course of the project.

The major challenge is to design a sufficiently general-purpose and robust strategy language so that it can deal with unanticipated proof plans and patches that experts will devise. If we knew in advance what these plans and patches would be, we could include them in the theorem prover, so that the problematic POs would be discharged and would not require expert attention. Whether this challenge can be met is an empirical question that can only be settled by

[6] Support for the view that model-based approaches share considerable similarities can be gleaned from the eight reports in *Formal Aspects of Computing* Vol 20, No. 1, 2008: the results from widely differing approaches are amazingly uniform.

[7] Paradoxically, a more general theorem can sometimes be easier to prove.

extensive examination and analysis of undischarged POs, their expert-provided proofs and the families of POs that can be discharged in a similar way. We cannot expect 100% success, but even a modest success rate would accrue significant benefits.

3 Programme and methodology

Work on WPs 1 and 2 can begin in parallel from "day one" of the project. The lead site for each workpackage is indicated within parenthesis in its title (along with the expected start/finish months). WP1 addresses Objective 1 above and WP2 addresses Objectives 2 and 3. WP3, WP4 and WP5 further develop these objectives, WP6 disseminates the research and WP7 is an optional extension. We have been careful not to make the overall project dependent on the PhD students even though their success would add considerably to the project outcomes.

WP1: Strategy language (m1–36)

WP1.1: Analysis of proofs (Newcastle m1–18). We intend to study lots of POs/proofs (from actual program developments of differing sizes). The partners in DEPLOY have agreed to let us instrument the Rodin Toolset in order to facilitate automatic collection of such data from industrial use. In keeping with our wish to be "generic" with respect to development methods, we will also garner proofs from other systems and are already in discussion with the "Overture" project that will support VDM.[8]
Deliverables:
D1: Rodin tool to extract (POs and) proof histories
D2: Collection of POs/proof analyses

WP1.2: Development of initial strategy language (Edinburgh m1–12). From the material collected in WP1.1, we will begin to experiment with the planned strategy language (see Section 2.3). We realise that it will not be possible to settle this language in one try. It can however be begun early because we already have suitable proofs to hand. This WP will occupy the first 12 months and the outcome (D4) will be used in WP5.
Deliverables:
D3: Collection of hand annotated POs/proof analyses
D4: An initial strategy language

WP1.3: Refinement of strategy language (Edinburgh m13–36). We expect the development of the strategy language to be highly iterative and the language to evolve during the project (starting from [BBHI05]).
Deliverables:
D5: (Working deliverable) Strategy language

WP2: Tool Development (m1–48)

WP2.1: Selecting a base Theorem Prover (Newcastle m1-12). We intend to garner the POs from (at least) the use of the Rodin Toolset in the ongoing DEPLOY project. We also hope to have the possibility of getting annotated proof attempts from this source. That does not mean that we will necessarily use its theorem provers to run our annotation scripts. The Rodin Toolset is a good base because it is available as open-source software[9] but it is possible that some other Theorem Proving (TP) system will provide a better basis for interpreting annotated proofs. One option would be to use *Isabelle* but the *Coq* system also looks to be a contender partly because it too is freely available for modification but the argument goes beyond this to its dependent types approach and its general structure.
Deliverables:
D6: A reasoned choice of TP system

WP2.2: Prototype of system (Edinburgh m13–30). We plan an iterative development in the sense that experimentation with practical use will teach us a lot about progress. We therefore plan to complete a first prototype by month 30 of the project. Specifically, we will aim for a prototype that can extract high-level proof plans from on-going proofs. This will almost certainly be done within the Rodin Toolset.
Deliverables:
D7: A prototype system (tool)

WP2.3: Attempts to falsify POs (Newcastle m31–36). Some POs that fail to prove automatically are simply false: this can result from something as simple as a typo or can indicate a deep seated problem in a design. A lot of unnecessary proof work can be saved by applying lightweight counter-example generators before a proof attempt. Results indicate that many non-theorems can be dismissed in this way. QuickCheck [CH00] has been implemented for several theorem proving environments and works by arbitrarily selecting test cases using the grammar of executable specification. Complementary to QuickCheck are approaches enumerating small finite models which are applied to SAT solvers to obtain a counter-example. Whilst being more constrained by the specification size, these approaches can handle non-executable constructs (e.g. quantifiers). Examples of such tools are Alloy, and Refute and Nitpick for Isabelle. We plan to investigate QuickCheck SAT and SMT based tools in our work. The newly developed Nitpick, where existential quantifiers are partly supported, is of particular interest. Kröning et al. have suggested extending SMT-Lib to handle (finite) VDM style POs[10], which we will also explore.
Deliverables:
D8: A system which finds counter-examples (tool)

WP2.4: Refine ideas/implementation (Newcastle m31–42). In keeping with the iterative plan, a period of reconsideration is scheduled with enough time left to make revisions to the prototype system and to decide to which theorem prover(s) we will add the interpretation of the proof plans.
Deliverables:
D9: a final system (tool)

[8] See `http://www.overturetool.org`

[9] see `http://www.event-b.org`

[10] `http://www.kroening.com/smt-lib-lsm.pdf`

WP3: Formal specification of what we build (Newcastle m13–42)

It is our intention (as in the mural project [JJLM91]) to describe formally the languages that we use and the tools that we build. An important first step will be to describe a model of "incomplete proofs" (there is a "gap" where either no valid inference rule is named or some of the hypotheses required by the rule are not available). In one sense this can be achieved by removing invariants from the formal definition of "complete" proof; but the bigger challenge is to come up with a definition of "gaps" that links to the taxonomy of the proof strategy language.

Deliverables:
D10: a formal specification of "proofs with gaps"
D11: a formal specification of prototype
D12: a formal specification of final system

WP4: Newcastle PhD (m3–44)

It can be dangerous to be too prescriptive about PhD topics but one clear area that requires attention is "reasoning about partial functions". Terms that fail to denote play a significant part in program specifications and reasoning and there are several distinct approaches to handling partial functions in logic. One approach is the "Logic of Partial Functions" [BCJ84] but we are not fixed in our views here; in fact, the Rodin Toolset takes a different ("Well Defined" POs) approach. We intend to take a careful look at this issue. (A recent paper [FJ08] contains references to earlier papers on the topic by Jones and collaborators as well as comparative surveys.) Jones has one PhD student in this area but there is ample scope for another looking specifically at how the issue affects learning.

Deliverables:
D13: PhD thesis

WP5: Edinburgh PhD (m13–48)

The Edinburgh PhD will concentrate on the analysis of ways in which proof attempts fail and are subsequently repaired. This analysis will be realised in a generic, strategy language for describing and classifying common forms of failure and repair. The strategy language will, in turn, be used, firstly, to describe how experts recover from failure and, secondly, to guide recovery during automated proof search of related proofs. Example failed proofs will be harvested from (a) the data from the Rodin Toolset, (b) the manual, expert proof attempts on source theorems and (c) failed applications to target theorems of proof strategies abstracted from manual expert proofs of source theorems. Examples of repaired proof attempts will be harvested from the expert's work on failures of both types (b) and (c). This work will enhance the abilities of our prototype so it can benefit from initially failed proof attempts as well as successful ones. The PhD will start in year two of the project in order to leave time for the generation of the data needed by the student which he/she can take as an exemplar and adapt the strategy language of successful proof attempts to a strategy language for unsuccessful and repaired proof attempts.

Deliverables:
D14: PhD thesis

WP6: Disseminate (Newcastle m13–48)

We obviously plan to disseminate our ideas, as they evolve, throughout the lifetime of the project: this will be by normal publication channels. Specific objectives before the end of the project include running a Schloss Dagstuhl event and offering tutorials at major formal methods conferences. All code produced will be "Open Source".

Deliverables:
D15: Tutorial in formal methods conference 1
D16: Tutorial in formal methods conference 2
D17: Schloss Dagstuhl event
D18: Final report

WP7: Breadth of applicability (Edinburgh m37–48)

Our main objective is proofs of POs from software engineering but, if we are as successful as we hope, it will be worth looking at yet wider application of "learning from proofs". But we realise that a key issue that will come into play here is the "taxonomy" behind similarity. We expect the shape of PO and the data structures involved therein to provide a similarity principle for software development proofs. It is equally clear that mathematics as a whole is not a collection of a (small) number of proof strategies. A very interesting meeting on this topic was held at the Royal Society and is reported in *Royal Soc, Phil Trans R Soc A*, Vol 363 (both proposers spoke, Bundy was a co-organiser).

4 Related work

The most popular formal methods tools are based on model checking. These tools conduct an exhaustive search of a state space, so abstraction is required to make this search tractable, which may result in a loss of soundness and/or completeness leading to false positives or false negatives. Most applications are to the proof of particular properties, such as deadlock absence or fairness, rather than full verification. Hardware systems can sometimes be fully verified when they can be specified in a simple logic, such as propositional. The advantage of model checking is that it can be 'push button', with little need for human intervention.

We are interested in full verification of software systems, for which automated theorem provers are required to ensure correctness of complex and lengthy proofs. Most automated provers require human interaction to guide the more tricky proofs — typically a residual 10-20% of the POs, but often amounting to thousands of POs for an 'industrial strength' software system. This makes the application of formal methods within the development cycle a potentially highly skilled and time-consuming process. Many provers have a *tactic* mechanism, which provides automation for many commonly occurring, straightforward subgoals, typically using simplifiers, decision procedures, etc.

Some provers allow users to compose new tactics geared to their applications. Proof planning is a rare attempt to provide a hierarchical system of tactics that can potentially automate a whole proof and to patch failing proofs, but even this is limited in its capabilities to relatively simple proofs. For instance, experiments in [IEC+06] showed that their loop invariant discovery technique was successful 80% of the time.

5 Project management

In this project we are developing a strategy language for proofs and tools implementing it. Both of these ambitious objectives will target large industrial data. The development of the language will necessarily be highly iterative and will alone require a minimum of three years. Since we target industrial data, we will need a consolidation period once the language is fixed in order to optimise the tools and address the breadth of the applications. Thus, four years is a minimum for the overall project.

It is clear that the management of a split-site project is more difficult than one on a single site but we can only bring together the necessary expertise by combining the activity at Edinburgh and Newcastle. The two PIs have known each other for decades and are committed to a full collaboration. Specifically they will meet at least every other month with at least the RAs from both sites.

The RAs will meet each other monthly and the PhD students will be involved once they are up to speed. We will also use audio-conferencing as much as possible since this will make it easy to hook in Ireland and Butler (our experience in DIRC is that once people know each other the added value of video conferencing is not worth the inconvenience; we will however use some software package to support shared screens during "telcos").

In order to kick start the collaboration, we intend to ask each RA to spend a significant period (say one month) at the other main site during the first six months of the project. We are convinced that the expense of this will be more than balanced by better bandwidth among the research teams.

We have included Ireland as CI without RA support at Heriot-Watt (and Butler is donating his time for free). It is fair to say that their approach would be rather different focusing more specifically on the Event-B approach and gearing help to the modelling level. Having RAs at four sites would complicate communication and could reduce efficiency. We are however keen to have both act as in-project critics with Ireland's long cooperation with Bundy offering a natural focus on the AI topics and with Butler's long involvement in development methods providing critique of the program development dimension. (Jones and Butler have collaborated for about five years now under the two EU projects.) We look forward to the creative tension.

References

[BBHI05] A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*, volume 56 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2005.

[BCJ84] H. Barringer, J.H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[Bun91] 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.

[CH00] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279, 2000.

[DBL+04] H. Duncan, A. Bundy, J. Levine, A. Storkey, and M. Pollet. The use of data-mining for the automatic formation of tactics. In *Workshop on Computer-Supported Mathematical Theory Development*. IJCAR-04, 2004.

[FJ08] J. S. Fitzgerald and C. B. Jones. The connection between two ways of reasoning about partial functions. *IPL*, 107(3–4):128–132, 2008.

[IEC+06] A. Ireland, B.J. Ellis, A. Cook, R. Chapman, and J Barnes. An integrated approach to high integrity software verification. *Journal of Automated Reasoning: Special Issue on Empirically Successful Automated Reasoning*, 36(4):379–410, 2006.

[JJLM91] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A Formal Development Support System*. Springer-Verlag, 1991.

[Jon99] C. B. Jones. Scientific decisions which characterize VDM. In *FM'99 – Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 28–47. Springer-Verlag, 1999.

[Jon01] C. B. Jones. The transition from VDL to VDM. *Journal of Universal Computer Science*, 7(8):631–640, 2001.

[SI98] J. Stark and A. Ireland. Invariant discovery via failed proof attempts. In P. Flener, editor, *Logic-based Program Synthesis and Transformation*, number 1559 in LNCS, pages 271–288. Springer-Verlag, 1998.

[Sil85] B. Silver. *Meta-level inference: Representing and Learning Control Information in Artificial Intelligence*. North Holland, 1985. Revised version of the author's PhD thesis, Department of Artificial Intelligence, U. of Edinburgh, 1984.