Contribution of the AI4FM 2014 Workshop

A satellite event of FM 2014
Singapore, 13 May 2014

This report contains a collection of abstracts for all the AI4FM 2014 workshop talks. The main goal of the AI4FM workshop series is to bring together researchers in the areas of formal methods and artificial intelligence to address the issue of how artificial intelligence can be utilised to support the formal development process. This is the fifth workshop in this series, and previous events have been held in: Newcastle (2010), Edinburgh (2011), Schloss Dagstuhl (Germany, 2012) and Rennes (France, part of ITP 2010).

Gudmund Grov, Leo Freitas & Iain Whiteside
(workshop organisers)
AI4FM 2014 Programme

9:00: Introduction: **AI4FM 2014**
   Cliff Jones, Newcastle University

9:05: **Using Dafny for Programs and Proof**
   Rustan Leino, Microsoft Research

**10:00: Coffee**

10:30: **Invariants without Toil**
   Dominique Méry, LORIA and Université de Lorraine

11:00: **Specification for Termination and Non-Termination**
   Ton Chanh Le and Wei-Ngan Chin, National University of Singapore

11:30: **Building a Proof Process Capture System**
   Andrius Velykis, Newcastle University

12:00: **Discussion**

**12:30: Lunch**

14:00: **Tinkering by Theory Formation**
   Gudmund Grov (Heriot-Watt University), Colin Farquhar, Alison Pease and Simon Colton

14:30: **Using Machine Learning in the Automatic Translation of Object Propositions**
   Ligia Nistor and Jonathan Aldrich, Carnegie Mellon University

15:00: **Proof Engineering Challenges for Large-Scale Verification**
   Gerwin Klein, NICTA

15:30: **Discussion**
Using Dafny for programs and proof

K. Rustan M. Leino (Microsoft Research, Redmond, WA, USA)

Dafny is a programming language and program verifier. Reasoning about programs is done by including in the program text various assertions, like method pre- and postconditions and loop invariants, which the verifier attempts to discharge automatically. When automation falls short of completing the proofs, a user may need to supply additional information in the form of lemmas. The lemma mechanism is useful in its own right, because it lets the tool be used to formalize and prove other properties; for example, one can use Dafny to define a program semantics and prove some properties about it.

In this tutorial, I will introduce Dafny and its proof features through a series of examples. I will show the essential interaction with the tool, and show how programs are specified, how lemmas are stated, and how proofs are authored.
The *correct-by-construction* approach can be supported by a progressive and incremental process controlled by the refinement of models: the *Event-B* modelling language [1, 5] provides a toolset [2] for editing, checking and developing models using refinement. An *Event-B* model is defining a finite set of events modifying state variables and events are maintaining an invariant which is expressing strong safety properties satisfied by state variables. Rather than to define a complex state-based model, the refinement-based methodology consists in starting by a very abstract and simple model and in adding details and events, which are either new, or are strengthening former abstract events. The progressive process helps to control the complexity of proofs to discharge using the proof assistant and to enrich the invariant. Moreover, the progressive process helps in validating and tracing the details added in the refining model. Roughly speaking, refining a model $A$ by a model $C$, means that the model $C$ simulates the model $A$ and that consequently, it preserves safety properties. The main difficulty in this refinement-based process is to start by the best abstract model and to have in mind that the proof assistant is assiting you but you have to provide enough informations for helping the prover. Abstraction and refinement are two main features for modelling systems. In previous works [8, 3, 12], we have addressed the derivation of distributed algorithms and cryptographic protocols. Our quest is to facilitate the correct-by-construction approach for designing classical sequential and distributed algorithms and for deriving invariants for these algorithms.

We illustrate a simple paradigm called the call-as-event paradigm and we address the description of guidelines for the design of programs and algorithms and the integration of proof-based aspects using the RODIN platform. More precisely, we introduce several methodological steps identified during the development of case studies, and we propose auxiliary notions, such as refinement diagrams, for guiding users during problem analysis. A general structure characterizes the relationship between the contract, the *Event-B* models and the developed algorithm using a specific application of *Event-B* models and refinement. We simplify the translation of *Event-B* models into algorithmic elements by promoting the use of recursive constructs. The resulting algorithm is proved to be sound with respect to the pre/post specification, namely, the contract. Applications rely on a dynamic programming technique that illustrates the applicability of these patterns based on a call-as-event relationship. Each proof-based development is validated using the RODIN platform. Using these guidelines, we show how invariants are very simply derived from the analysis of the problem without toil.

We combine concepts of the *Event-B* method and programming paradigms by defining a proof-based pattern based on the call-as-event principle [7, 8]. Proof-based development constitutes a very powerful framework for constructing procedures, programs, and systems, but it requires the use of formal techniques and formal languages.

---

* This work was supported by grant ANR-13-INSE-0001 (The IMPEX Project [http://impex.loria.fr](http://impex.loria.fr)) from the Agence Nationale de la Recherche (ANR). The author acknowledges the Irish Research Council and Campus France for the joint funding of this research collaboration via the Ulysses scheme 2013.
Our main goal is to facilitate the use of the Event-B modelling language by proposing techniques and tools. Proofs were improved, even though RODIN has unexpected features. The technique of development is a top-down approach, which is clearly well known from the earlier works of Dijkstra[6, 11] and uses refinement to control the correctness of the resulting algorithm. It relies on a more fundamental issue related to the notion of the problem to solve. A specific diagram is used to organize the refinement, and we call it a refinement diagram [7, 8]. The call-as-event guideline is summarized by the following diagram:

- CALL is the call of the PROCEDURE.
- PREPOST is the machine containing the events stating the pre- and postconditions of CALL and PROCEDURE, and M is the refinement machine of PREPOST, with events including control points defined in CM.
- The call-as-event transformation produces a model PREPOST and a context PB from CALL.
- The mapping transformation allows us to derive an algorithmic procedure that can be mechanized.
- PROCEDURE is a node corresponding to a procedure derived from the refinement model M. CALL is an instantiation of PROCEDURE using parameters x and y.
- M is a refinement model of PREPOST, which is transformed into PROCEDURE by applying structuring rules. It may contain events corresponding to calls of other procedures.

We have used the technique in the teaching [9] of the design of algorithms. Surprisingly, students discovered invariants using the theorem prover of the RODIN platform. In fact, we think that it is the recursive nature of the development that made the discovery of invariants easier. Proof obligations were not very difficult because we inherit the recursiveness of the structure of the problem. Floyd’s algorithm is not a trivial algorithm, but proof obligations were not difficult to prove. More recently, we have combined Spec# and Event-B in a general framework [10] supported by tools.

Several non-trivial examples have been developed as the shortest path algorithm of Floyd, the insertion sort, the binary search, the CYK algorithm, ... Summing up our approach, we have proposed a one-step refinement process which is granting invariants for free or at least without toil, since the proof assistant is much more successful for discharging proof obligations.

References

Specification for Termination
and Non-Termination

Ton Chanh Le and Wei-Ngan Chin

Department of Computer Science, National University of Singapore
{chanhle, chinwn}@comp.nus.edu.sg

Abstract. We propose a unified logical framework for specifying and proving both termination and non-termination of various programs. Our framework is based on a resource logic which captures both upper and lower bounds on resources used by the programs. By an abstraction, we evolve this resource logic for execution length into a temporal logic with three predicates to reason about termination, non-termination or otherwise. We introduce a new logical entailment system for temporal constraints and show how Hoare logic can be seamlessly used to prove termination and non-termination in our unified framework. We then report on the usability and practicality of our approach by verifying both termination and non-termination properties for about 300 programs, collected from a variety of sources. This adds a modest 5-10% verification overhead to an existing partial-correctness verification system.

1 Introduction

Termination proving is an important part of correctness proofs for software systems as “so-called partial correctness is inadequate: if a program is intended to terminate, that fact must be part of its specification.” – Cliff Jones [9]. Thus, total correctness proofs, denoted by the Hoare triple $[P]\{c\}{Q}$, require the code fragment $c$ to be shown terminating in addition to meeting the poststate $Q$ after execution. The termination of a loop or a recursive method is usually proven by a well-founded termination measure given to the specification. However, such a termination measure is not part of the logical formulas for pre/post specifications. A reason for this distinction is that specification logic typically describes program states, while the termination proofs are concerned with the existence of well-founded measures to bound the execution of loops/recursions, as argued by Hehner in [7]. This distinction has been designed into state-of-the-art verification systems, such as Dafny [11]. A shortcoming of these systems is that they cannot automatically leverage richer logics that have been developed for safety properties to help us conduct more intricate termination and non-termination reasoning.

For illustration, let us use a problem under the Java Bytecode Recursive category of the annual Termination Competition [12]. In this problem, an acyclic linked list is shuffled by the shuffle method together with the auxiliary reverse method, whose source codes are shown in Fig. 1. To prove that shuffle terminates, we need to firstly show that reverse also terminates. While the termination of reverse can be easily proved by current approaches [10,2,3], proving shuffle terminates is harder because we need to know a safety-related fact that the reverse method does not change the length of the input list. Based on this fact, we can show that the linked list’s length is
also decreasing across the recursive method call `shuffle`; therefore, the method always terminates.

```java
public static List shuffle(List xs) {
    if (xs == null) return null;
    else {
        List next = xs.next;
        return new List(xs.value, shuffle(reverse(next)));
    }
}
```

```java
public static List reverse(final List l) {
    if (l == null || l.next == null) return l;
    final List nextItem = l.next;
    final List reverseRest = reverse(nextItem);
    l.next = null; nextItem.next = l;
    return reverseRest;
}
```

![Fig. 1. The Shuffle problem from the Termination Competition](image)

However, without an integration of termination specification into logics for functional correctness, the verification systems based on traditional Hoare logic for total correctness cannot specify, and thus cannot prove, the termination of `shuffle`. Note that automated termination provers, such as AProVE [5], Julia [14] and COSTA [1], are also not able to show that `shuffle` terminates. We believe that relatively complex problems, such as Shuffle, highlight the need of a more expressive logic with the ability of integration into various safety logics for termination reasoning.

Moreover, if the termination proof fails, e.g., when the input list of `shuffle` is cyclic, the program is implicitly assumed to be possibly non-terminating. That is, definite non-termination is neither explicitly stated nor proven by Hoare logic. Explicitly proving non-termination has two benefits. First, it allows more comprehensive specifications to be developed for better program understanding. Second, it allows a clearer distinction between expected non-termination (e.g., reactive systems where loops are designed to be infinite) and failure of termination proofs, paving the way for focusing on real non-termination bugs.

## 2 Our proposal

We propose integrating both termination and non-termination assertions directly into available logics for functional correctness. Our work follows Hoare and He [8] and Hehner [6], in which the termination is reasoned together with partial correctness proofs. However, these approaches are still limited with respect to non-termination reasoning.

To unify both termination and non-termination reasoning and integrate them into functional correctness proofs, we introduce a new resource logic which captures the concept of execution capacity; tracking both minimum and maximum number of calls (and loop iterations) executed by some given code. Our resource logic uses a primitive predicate $RC(l, u)$ with invariant $0 \leq l \leq u$ to capture a semantic notion of execution capacity $(l, u)$ with the lower bound $l$ and the upper bound $u$. Through this resource logic, we can specify a variety of complexity-related properties, including the notions of termination and non-termination. Termination is denoted by the presence of a finite upper bound, while non-termination is denoted by an infinite lower bound.

To support a more effective mechanism, we shall evolve a simpler temporal logic from the richer resource logic itself. We define three temporal predicates, $\text{Term } M$, $\text{NonTerm } M$, and $\text{DefTerm } M$. These predicates allow us to express both termination and non-termination properties.
Loop, and MayLoop, and associate such temporal predicates with each method in a given program to denote the termination, definite non-termination and possible non-termination of these methods, respectively. In terms of resource reasoning, these predicates represent $RC(0, \text{embed}(M))$, $RC(\infty, \infty)$ and $RC(0, \infty)$, respectively, where $M$ is a well-founded measure and $\text{embed}(M)$ is a finite bound obtained through an order-embedding of $M$ into naturals. Using the enriched specification logic, functional correctness, termination and non-termination of methods can be verified under a single modular framework.

Our research contributions can be summarized as follows:

- A new resource logic that can capture lower and upper bounds on execution capacity, together with an entailment procedure to support correctness proofs with complexity-related properties.

- A temporal logic that is abstracted from the resource logic to reason about program termination and non-termination. We introduce three new temporal constraints, its entailment and Hoare rules lifted from the resource logic.

- A successful integration of both resource and temporal logics into a separation logic based verifier [13]. The new temporal logic is expressive enough to specify and verify the termination behaviors for about 300 benchmark programs collected from a variety of sources, including the SIR/Siemens test suite [4] and problems from the Termination Competition. The prototype implementation and benchmark are available for online use and download at:

  http://loris-7.ddns.comp.nus.edu.sg/~project/hiptnt/

References

Building a Proof Process Capture System

Andrius Velykis

School of Computing Science, Newcastle University, UK
andrius.velykis@newcastle.ac.uk

Extended abstract

AL4FM research project\(^1\) aims to reduce the amount of interactive proof needed during the verification of formal specifications. The approach is to reuse the ideas from interactive proof process to discharge similar proof obligations automatically, i.e. “to learn from an expert doing proof.” Verification of formal specifications yields proof obligations (and proofs) that are quite similar to one another and can be grouped into proof families, where a single proof idea can be used to discharge all proof obligations within a family. AL4FM aims to adapt the proof strategies taken in one proof to complete the others automatically.

This research takes the approach that a reusable strategy is based on the captured high-level features of a proof process. Low-level proof steps and their details would differ within a family: e.g. the proof obligation may use different operators, predicates or data structures; new similar lemmas are needed instead of the original ones; certain proof steps are repeated, omitted or added; etc. A high-level strategy generalises over such differences and allows reuse.

This talk presents a ProofProcess system\(^2\) designed to capture comprehensive high-level proof process information that can be used to extract reusable proof strategies. The system works as an add-on to theorem proving assistants:\(^3\) it records the proof details, expert interactions and aims to infer the high-level proof process. Some of the system requirements and features are discussed below. A high-level overview of the system is available at [8,2].

Recording proof granularity and structure. Proof scripts often represent how to drive the theorem prover rather than what the proof ideas and high-level steps are. When recording the proof process, it should be captured at appropriate level of discourse: e.g. include the high-level proof plan, the actual proof tactic steps—even allow any arbitrary granularity that the expert deems appropriate. Furthermore, non-linear nature of proof should also be recorded, e.g. the proofs of base and step cases of inductive proofs should be treated as separate branches even when they are performed sequentially in a proof script.

The data structures used in the ProofProcess system accommodate the proof granularity and structure. They are similar to hiproofs [1]: distinguish between parallel branches in proof and allow grouping of lower-level proof steps. When

\(^1\) http://www.ai4fm.org
\(^2\) A prototype is developed at http://github.com/andriusvelykis/proofprocess.
parsing prover data, the system infers branching and other structural features. The expert may manually group proof steps, establishing the appropriate granularity to describe the captured proof. In the future, the ProofProcess system aims to learn these custom groupings and infer the proof structure from past examples. The system attempts to record both backwards and forwards proofs, however support for the latter is still being fine-tuned.

The proofs are represented as tree-based data structures. However, for flexibility, the ProofProcess system also uses a graph-based structure internally. It provides better proof matching and is a more powerful tool to represent complex proofs. The tree structure, however, provides better compositionality of proof steps when recording a high-level proof plan. An isomorphic translation between the graph and the tree structure is available that utilises additional linking elements within the tree.

**Recording proof history.** Developing a successful proof is rarely a straightforward exercise. Capturing the whole proof development could yield useful insights into how the final proof was reached. Failed attempts may still contain generally applicable proof steps, even though they were not successful in that particular case. Furthermore, final proof clean-up may make the proof strategy less applicable to similar proofs, whereas the original proof would represent the general idea better. Hence all versions leading to a finished proof should be captured.

The ProofProcess system identifies diverging proof attempts in the prover and records them as separate instances of the proof. However, recording the whole proof history with necessary details generates a large amount of data. To cope with scalability of such data repository, the ProofProcess system employs a database with on-demand loading and compression for detailed prover data.

**Capturing high-level proof description: intent and features.** The recorded proofs need to be supplemented with additional meta-data capturing expert’s insight. The intent is a high-level description of expert’s proof direction. It is a simple tag that gives name and description to the underlying proof reasoning. Intent can be assigned to grouped proof steps, indicating a high-level step. This allows emphasising how the proof is done abstractly, rather than what is needed to get it through the prover. Figure 1 provides an example of a proof tree with intents marked at all levels of abstraction.

The proof features is an open-ended concept to represent everything that drives the expert’s proof. It is coupled with intent to describe a proof step, i.e. intent records what the expert has done (intends to do), while proof features capture why the particular step was taken. Proof features can include the shape of the goal, the available lemma, certain datatypes or records, etc. Such information is usually readily available during the proof process, however deducing it from the finished proof is difficult. They would become triggers for the extracted strategy, i.e. indicating when the strategy can be applied for a similar proof.

Some of the proof features can be inferred and marked automatically by the system, e.g. by checking the differences between the input and output goals. Cur-
**ProofSeq**: Inductive proof
- **ProofSeq**: Prepare induction
  - **Proof Entry**: Simplify
  - **Proof Entry**: Extract inductive variable
- **ProofSeq**: Induction
  - **Proof Entry**: Apply induction rule
  - **Proof Parallel**: Induction cases
    - **ProofSeq**: Base case
      - **Proof Entry**: Simplify
      - **Proof Entry**: Use hypothesis
    - **ProofSeq**: Step case
      - **Proof Entry**: Rewrite using lemma
      - **Proof Entry**: Use hypothesis

Fig. 1. A sketch example of a proof tree with recorded intents.

Currently the **ProofProcess** system allows marking the intent and features manually, via the provided user interface.

The **ProofProcess** system is built on Java, Eclipse, and EMF. It integrates with Isabelle via Isabelle/Eclipse and with Z/EVES via the Community Z Tools. Currently the **ProofProcess** system focuses on capturing the proof process. The collected information would be used by other systems to generalise, extract or otherwise learn the strategies taken by the expert (e.g. using the PSGraph [4] system). The overall AI4FM approach and system is described in [3,5].

**References**

Tinkering by Theory Formation

Gudmund Grov & Colin Farquhar (Heriot-Watt University, Edinburgh, UK)
Alison Pease (Dundee University, UK)
Simon Colton (Goldsmiths College, UK)

1 PSGraph and the Tinker tool

Most interactive theorem provers support encoding of common proof strategies as special function called tactics. Such tactics tend to work backwards from the goal, reducing a goal to a set of simpler sub-goals. Proof strategies are then created by combining such tactics using a tactic language. Such languages are often not designed to distinguish goals in cases where tactics produce multiple sub-goals. Thus when composing tactics, one has no choice but to rely on the order in which goals arrive, thus making them brittle to minor changes. For example, consider a case where we expect three sub-goals from tactic $t_1$, where the first two are sent to $t_2$ and the last to $t_3$. A small improvement of $t_1$ may result in only two sub-goals. This “improvement” causes $t_2$ to be applied to the second goal when it should have been $t_3$. The tactic $t_2$ may then fail or create unexpected new sub-goals that cause some later tactic to fail.

As a result: (1) it is often difficult to compose tactics in such a way that all sub-goals are sent to the correct target tactic, especially when different goals should be handled differently; (2) when a large tactic fails, it is hard to analyse where the failure occurred; (3) the reliance on goal order means that learning new tactics from existing proofs has not been as successful for tactics as it has been for discovering relevant hypotheses in automated theorem provers; and (4) large complex tactics are difficult to understand and maintain. As a result it is the easiest way for a user to deal with failure is to manually guide the proof until the tactic succeeds (or becomes unnecessary), rather than correcting the weakness of the tactic itself, complicating the overall proofs.

Proof-strategy graphs (PSGraphs) [5] overcome these deficiencies following a “plumbing” approach to combine tactics, and has been implemented in the Tinker tool [4]. Here, tactics appear as nodes in a special type of directed graph with the additional property that we allow dangling edges, i.e. edges without source and/or target nodes. Edges without source or target nodes serve as inputs and outputs to the graph as a whole. Tactics are then combined by “piping” them together with an edge in the graph. One evaluates a PSGraph by placing one or more goal-nodes, each containing a single goal, on input edges of the graph, then applying tactics to consume goals on the in-edges of a tactic-node and produce sub-goals on the out-edges. As a result, the goals appear to flow through the graph, hitting tactics along the way, until they are either consumed (i.e. closed), or reach the output edges of the graph, in which case they become output goals for the overall evaluation. A tactic-node may have multiple output edges, and to decide which edge a goal should be sent to we label edges with goal-types. These encode certain properties about a goal which dictate how it should then be handled.

On the right an example PSGraph is shown, encoding a sub-part of the well known introduction tactic. Here, three introduction rules are given: conjunction introduction, implication introduction and disjunction introduction. Crucially, each of these tactics only works when the top level symbol is $\land$, $\to$ or $\lor$, respectively – and this information is encoded in the incoming goal types to ensure a goal is sent to the correct place.

*Supported by EPSRC grants EP/J001058 and EP/H023852, and a Heriot-Watt DTA scholarship.
Such goal-types enables us to abstract over goal order and numbers, and provides a more declarative view of a proof structure, helping to identify where, how and why a proof failed. Importantly, it overcome problems previous attempts to learn tactics have had (e.g. [3]): the representation can capture properties of the goals at a sufficiently abstract level in order to, for example, identify case splits and termination conditions for iteration. We also believe this is crucial when learning from single instances, as in proof by analogy.

2 A theory formation approach for learning goal types

In order to extract a sufficiently abstract proof strategy both the graph structure and the goal types on the edges must be learnt, and here we address the sub-problem of learning goal types. In addition to being a sub-problem of such a strategy language, it can help with understanding existing known tactics – hopefully improving maintainability. To illustrate, on the right a first approach to encoding the common ‘auto’ tactic of Isabelle in PSGraph is depicted. Here, some information is abstracted away, such as “first try this tactic and if it fails try this one”, as this does not help with the understanding – our ambitious goal is to know when to apply one or the other. Thus, we make an edge for each possibility in the graph. We are now left knowing the structure of the graph, however we do not know any of the goal-types. As the Isabelle library (and associated Archive of Formal Proofs) contains many examples where this tactic is used, we can try all possible paths of the ‘auto’ PSGraph and compare with the result of the standard ‘auto’ tactic. This information can then be used to label each sub-goal on each edge as a positive or negative example, and from that machine learn the goal-type.

Our goal is to understand the proof strategy, by discovering a predicate in the form of a goal-type capturing the valid goals of that path in the graph. Thus, more traditional statistical machine learning methods, which will give a classifier assigning a probability given a goal, are not desirable. Experience with the rippling proof strategy [1], encoded in PSGraph in [5, 4], has lead us to believe that this requires us to invent new predicates. In rippling, artefacts such as embedding of terms into other terms, measure reduction and wave rules are created in order to describe the goal-types. For other strategies, other artefacts may be required, and crucially these are unknown. This leads us to the artefact generation paradigm of AI, where, as apposed to the more common problem solving paradigm, the challenge is to generate new artefacts [2]. These artefacts can then be combined to describe a particular goal type.

An example of such tool is the HR Automated Theory Formation tool, and here we will outline some preliminary results of learning goal types using the latest version, called HR3 [2]. Beginning with an initial set of concepts and some examples, it applies a set of production rules to generate new concepts and conjectures. These concepts can either define objects of interest, such as integers, or a relationship between one or more such objects, for example the number of divisors of an integer. The production rules enable HR to perform operations such as negating, matching or composing concepts to generate new ones. These rules can be applied recursively to the newly discovered concepts to repeat the process until either no more can be found or the process is stopped manually.
To check for suitability, we conducted a small experiment with HR to see if it is able to learn goal types. Here, we looked at the simple introduction tactic discussed above, but removed the goal-types. This can be seen on the left where each of the goal types are replaced by a simple label, and our goal was to see if it HR could discover the required concept of top symbol from the set of simple examples. These had to be encoded in Prolog-like notation to enable HR to use them. Assuming the existence of concepts for: edge, as in the edge in the graph; symbol (as in e.g. \( \rightarrow \) written as imp), and term, to represent the terms of a goal, we start with a concept of a goal:

\[
\text{goal}(A,B,C,D) :- \text{edge}(A), \text{symbol}(B), \text{term}(C), \text{term}(D).
\]

The goal \( A \rightarrow A \) on edge \( T_1 \), for example, would be represented as

\[
\text{goal}(T_1,\text{imp},A,A).
\]

We then gave HR a handful of such examples of valid goals on edge \( T_1 \). From two production rules, instantiation and existential, we could then find the desired concept.

First, the instantiation production rule will instantiate a variable with a constant, and in for our case the symbol is instantiated, creating the concept:

\[
\text{imp_goal}(A,B,C) :- \text{edge}(A), \text{term}(B), \text{term}(C), \text{goal}(A,\text{imp},B,C).
\]

This concept captures the desired goal, however, we still need to give the terms as arguments. This can be overcome by the existential production rule, which existentially quantifies variables, represented by prefixing the name with \( _\exists \):

\[
T_1(A) :- \text{edge}(A), \text{goal}(A,\text{imp},_X1,_X2).
\]

This is the desired predicate for a goal type.

The experiment has shown feasibility for our approach. However, the example is very trivial and we expect that other common machine learning tools would have been able to discover this predicate. Next we plan to address how to test larger tactics, building up towards something like ‘auto’. In such cases we will not know in advance what concepts will be required, and it will be very interesting to see what HR may come up. A key challenge here will be to prune the number of concepts generated, by developing measures of how interesting concepts are and strategies for production rule applications.

References


\(^1\)We have simplified notation slightly, and given the new concepts suitable names to ease the reading.
Using Machine Learning in the Automatic Translation of Object Propositions

Ligia Nistor and Jonathan Aldrich
School of Computer Science, Carnegie Mellon University, USA
{l subsid, aldrich}@cs.cmu.edu

1 Introduction

Object propositions [5] are used for the modular verification of object-oriented code in the presence of aliasing, i.e., the existence of multiple references to the same object. They are associated with object references and declared by programmers as part of method pre- and post-conditions in the process of formally proving the correctness of object-oriented code. Object propositions use abstract predicates [6] to characterize the state of an object. These predicates are embedded in a logical framework and aliasing information is specified using fractions [3].

If in the system there is only one reference to an object, that reference has a fraction of 1 to the object, and thus full modifying control over its fields. If there are multiple references to an object, each reference has a fraction less than 1 to the object and each can modify the object as long as that modification does not break a predefined invariant (expressed as a predicate). In case that modification is not an atomic action (and instead is composed of several steps), the invariant might be broken in the course of the modification, but it must be restored at the end of the modification.

To verify a method using object propositions, the abstract predicate in the object proposition for the receiver object is interpreted as a concrete formula over the current values of the receiver object’s fields.

A critical mechanism in the object propositions methodology is packing/unpacking [4]. When the code modifies a field, the specification has to follow suit and unpack the predicate that contains that field (unpacking a predicate gives read/write access to the fields of that predicate). At the end of a method, the fields have been modified and after checking that a predicate holds, we are allowed to pack back that predicate.

2 Example

The code in Figure 1 represents a class DoubleCount that has two integer fields val and dbl. The predicate OK states that the value of the field dbl is double the value of the field val. This predicate is the invariant of the class. If there are two aliases r1 and r2 to an object o of type DoubleCount, r1 can assume that the invariant holds even if r2 has modified object o by calling the method
increment on it. This is ensured by the pre- and postcondition of the method increment. The precondition is this@k OK(), which states that the caller this of the method satisfies the invariant OK. After the execution of the method the same invariant holds, according to the postcondition.

class DoubleCount {
  int val;
  int dbl;
  predicate OK() ≡ ∀v : int, d : int.val → v ⊗ dbl → d ⊗ d == v + 2
  void increment()
    ∀k : int.this@k OK() → this@k OK()
    {val = val+1;
    dbl = dbl+2;}
}

Fig. 1. DoubleCount class and OK predicate

3 Translation to Boogie

Below we present the manual translation of the code and specifications of Figure 1 into Boogie [2]. In our Boogie encoding, we created a type Ref to represent references of type DoubleCount. We represented the heap by creating maps from objects to their fields: for example we represented the field val by var val: [Ref]int which maps an object of type DoubleCount to its val field of type int. We created a new map type to keep count of fractions type FractionType = [Ref, PredicateTypes] int and another map type PackedType to keep track of which predicates are packed, for a specific object. The first axiom represents the necessary conditions that have to be met for this to be packed to the predicate OK, while the second axiom is used when this is unpacked from the predicate OK.

type Ref;
  type PredicateTypes;
  type FractionType = [Ref, PredicateTypes] int;
  type PackedType = [Ref, PredicateTypes] bool;
  const null: Ref;
  const unique okP: PredicateTypes;
  var val: [Ref]int;
  var dbl: [Ref]int;
  var packed: PackedType;
  var frac: FractionType;

axiom ( forall this: Ref, val: [Ref]int, 
  dbl: [Ref]int, packed: PackedType::
    (dbl[this]−=val[this]*2) ⊃ packed[this, okP]);

axiom ( forall this: Ref, val: [Ref]int, 
  dbl: [Ref]int, packed: PackedType::
    packed[this, okP] ⊃ (dbl[this]−=val[this]*2) );
procedure increment(this: Ref)
  modifies val, dbl, packed;
  requires packed[this, okP];
  requires frac[this, okP] > 0;
  ensures packed[this, okP];
  ensures frac[this, okP] > 0;
  { val[this] := val[this] + 1;
    dbl[this] := dbl[this] + 2; }

4 Using Machine Learning

Our goal is to automatically translate the code and specifications in Figure 1 to
the Boogie code in Section 3. We are currently defining the automatic translation
rules. They are going to be the most interesting feature of our compiler which will
do the translation. While most of the translation rules are clear, there are some
places where machine learning (ML) could be useful. The compiler could generate
most of the translation but use ML, more specifically structured prediction [1],
to help the compilation in the creation of the axioms.

If our ML algorithm has access to 20¹ programs that have been manually
translated into Boogie and proved correct, the ML algorithm could learn and
predict how the axioms will look like. The output generated by the ML algo-
rithm does not need to be precise; it would be very useful if it gives multiple
possible axioms from which the programmer can choose the one that looks cor-
rect. The programmer could also run all the variants suggested by the structural
prediction algorithm and see how each performs. Let’s assume we create a new
class TripleCount that has two fields val and trpl, and the invariant states that
trpl should always be triple the value of val. The ML algorithm, given enough
training translations, among which the translation presented in Section 3, could
suggest that the first axiom be:

axiom ( forall this: Ref, val: [Ref]int,
         trpl: [Ref]int, packed: PackedType::
         (trpl[this] == val[this] * 3) => packed[this, okP] );

References

3. John Boyland. Checking interference with fractional permissions. In Static Analysis
5. Ligia Nistor, Jonathan Aldrich, Stephanie Bader, and Hannes Mehnert. Object
6. Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In POPL,

¹ or more, if we involve more people in the manual translation of object propositions
specifications into Boogie
Proof Engineering Challenges for Large-Scale Verification

Gerwin Klein
NICTA* and UNSW, Sydney, Australia
{first-name.last-name}@nicta.com.au

In this extended abstract I summarise challenges for proof engineering that we encountered in the formal verification of the seL4 microkernel [7], and its subsequent proofs of integrity [12], non-interference [10], and binary correctness [11]. I focus on problems where there is scope for automation using AI and machine-learning techniques. For more background on the seL4 verification, and an analysis of the effort spent on it, see previous work [6].

The seL4 kernel is a 3rd generation microkernel in the L4 family [9]. Such kernels provide basic operating system (OS) mechanisms such as virtual memory, synchronous and asynchronous messages, interrupt handling, and in the case of seL4, capability-based access control. The idea is that, using these mechanisms, one can isolate software components in time and space from each other, enabling separate compositional verification of trusted components as well as proof that no such correctness is required of untrusted components, because the kernel and its policy configuration already sufficiently constrain their behaviour [2].

The verification of seL4 was not a large project by industrial software development standards, but it was sizeable for an academic formal verification project. The functional correctness proof of seL4 took roughly 12 person years, the overall initial project, including tool building, libraries, and research in scalable proof techniques, usable semantics of the C programming language, etc. took about 25 person years; for a more precise analysis see [6]. This effort later paid off in the proof of high-level security properties: they were much easier to show, because they could now be established on an abstract specification instead of directly on the code. Integrity cost less than 8 person months, non-interference less than 21 person months, and updates to the kernel to add a separation scheduler cost another 21 person months, including updates to all existing proofs. Automatic binary verification for functional correctness then extended these properties down to the low-level semantics of ARMv6 machine instructions.

The largest of these proofs, the initial functional correctness verification produced about 200,000 lines of Isabelle/HOL proof scripts [7] with a team of on average 12 people over 4 years (about 7 full-time equivalent).

During the subsequent proofs, the seL4 kernel evolved. While there were no C-level defects to fix in the verified code base, changes included performance improvements, API simplifications, additional features, and occasional fixes to parts of the non-verified code base of seL4, such as the initialisation and assembly.

* NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.
portions of the kernel. Some of these changes were motivated by the security proofs, for instance to simplify them, or to add the scheduler with separation properties. Other changes were motivated by applications the group was building.

This additional work increased the overall proof size to roughly 400,000 lines of Isabelle proof script. Other projects of similar order of magnitude include the verified compiler CompCert [8], the Verisoft project [1] that addressed a whole system stack, and the four colour theorem [4].

There is little research on managing formal verification on this scale and the experience in our verification was that this scale makes a significant difference to how proofs are developed and maintained [3]. The key difference to smaller proofs is that no single person at any time understands the whole proof, or even the whole code base in detail. Only the fact the proof is machine-checked gives us confidence in the soundness of the overall result. Of course, we are not the first to recognise the issue of scale for proofs. All of the other large-scale verification projects mentioned previously make note of it, as did previous hardware verifications [5].

While many of these issues are similar to traditional software engineering, there are differences that could be exploited to increase the productivity of the verification engineer and to increase the scale at which such proofs can be applied.

In particular, large-scale proofs have the following two properties that make them more amenable to automation and assistance by AI and machine learning techniques than traditional code development:

– it is cheap and easy to check if an existing proof is correct
– in a large-scale proof there are often a large number of analogous or similar cases and proof fragments

The first property is interesting for proof refactoring; while in code refactoring it is important to be semantics preserving, in proof refactoring, there is an easy check if the refactored proof still works.

This can be exploited in techniques that deal with the second property. Assuming we were able to automatically find a similarity between a current proof goal and some previous proof fragment, a proof suggestion based on this fragment does not necessarily have to be correct. It will be checked by the prover anyway, and it may even be useful to the verification engineer in its incorrect form, because she may be able to adjust it. Specific areas where such techniques might be useful are

– finding analogies within one proof with a large number of cases, and suggesting proofs to the engineer, or optimising proof search based on previous cases;
– finding and exploiting similarity between recurring proof fragments distributed over different proofs, in particular proofs of other verification engineers;
– suggesting and automating lemma extraction for recurring proof fragments;
– automatically generalising and re-proving lemmas whose statement was needlessly specific to achieve better re-use;
– suggesting, e.g. via auto-completion, high-level proof structure that has been extracted from previous proofs with similar statements.
It is crucial for such tools not to get into the way of normal proof interaction if they are to be useful to the verification engineer. Code completion and similar techniques from traditional code-based integrated development environments that are suggestive rather than prescriptive are directly applicable.

Isabelle’s PIDE interface [13] is already making good progress in this direction, for instance providing automatic feedback on counter examples or searching for proofs in the background while the engineer goes about her work. Such techniques benefit immensely from the increasing number of cores on desktop machines, because multiple separate analyses distribute trivially over them.

With more sophisticated suggestion and analysis tools becoming available, there should not only be room for significant improvements in productivity for large-scale proofs, and but also the possibility of making such proofs more accessible to new team members. In our experience, learning Isabelle is easy. Finding your way in a large proof and code base is much more time consuming.

References