Proof and Refutation in Formal Software Development

Juan C. Bicarregui and Brian M. Matthews
Information Technology Department
Rutherford Appleton Laboratory
Chilton, Didcot, Oxfordshire, OX11 OQX, UK.
{jcb,bmm}@inf.rl.ac.uk

June 16, 1999

Abstract

In this paper we describe investigations into the use of automatic theorem proving technology in the refutation of proof obligations. Specifically, we discuss the use of resolution based theorem proving and model checking to find false obligations and counterexamples. These techniques can be used as basis of an automatic method for finding faults in design during the formal development of software. This approach is complementary to verifcation by proof as such proofs can only be completed when all faults have been corrected. We give a simple example using the B formal development method to demonstrate its potential.

1 Introduction

Formal specifications have long been advocated as a means of precisely capturing system requirements and hence as a reference relative to which correctness of an implementation can be proven. In practice however, although formal techniques are increasingly being adopted particularly in certain critical application domains, obtaining a proof from first principles of correctness remains an expensive activity.

Despite advances in theorem proving technology which have enabled many proof tasks to be completed automatically, when a prover fails to complete a proof, the developer is left uncertain as to whether the failure is due to the inability of the prover to find a proof or simply because the conjecture is actually false, that is, because there is in fact some defect in the design.

In this situation, the developer is often left with the highly specialised task of inspecting the failed proof to try to discover which of these is the case. In our experience in proof of real-life formal developments [7, 5] using the B-Method and tools, although automatic proof may discharge the large majority of the many thousands of proof obligations generated by even a simple design, the fact that several hundred proof obligations may remain unproven, means that even cursory inspection of them is highly expensive.

Experience shows however, that when the failed proofs *are* inspected, some are easily seen to be unprovable. Often, there are simply no hypotheses which could possibly be used to justify the conclusion: a situation which typically indicates that the precondition of an operation needs to be strengthened. The fault in the specification being relatively obvious once the false proof obligation has been detected. Correction of the fault then leads to automatic proof succeeding for this proof obligation.

In this paper we describe investigations into the use of automatic theorem proving technology in the refutation of proof obligations in order to find faults in design during the formal development of software.

This activity can be seen as the formal counterpart to software testing which is, after all, the industry's standard way of demonstrating correctness of software. A false proof obligation is akin to a failed test in that it indicates a fault in the design, the correction of which, by definition, improves the correctness of the software. An individual true proof obligation, on the other hand, rather like a test success, in itself does little to demonstrate correctness. Although, refutation, like testing, can never be used to demonstrate absolutely the absence of errors, successive detection and correction of faults can help increase confidence in that correctness. Furthermore, each fault corrected, improves the success of automatic proof and hence lessens the cost of completing the overall proof task.

It has been said that testing can only show the presence of bugs, not their absence (Dijkstra [12]). Whilst this is true, it does not tell the whole story, and in practical software development, the difference between testing and proof is not so clear cut. The formal development of a new software system is akin to developing a new theory in mathematics, with axioms describing the properties of the system. Lakatos argues in his work on the logic of mathematical discovery [19] that the full development of a mathematical theory requires a parallel search for proof and for counter-examples. These latter, while seeming to refute the theory actually contribute a vital part in its development, modifying the assumptions and limitations of the theory, as well its consequences, and leading ultimately to a better defined and often a more widely applicable or more elegant theory.

We propose that this methodology is transferred to the formal development of software through a methodical search for counter-examples as well as the formal proof of the development. This would not only demonstrate the presence of bugs at an early stage in the development, but also through demonstrating weaknesses in the assumptions of the system, improve the specification of its requirements, and ultimately help produce a better software solution. The view has also been noted within the Irish School of VDM [20].

2 Formal Software Development using the B method

The B method [1] is a "model-oriented" formal method providing notations and support tools for many of the activities in the software development lifecycle. It is currently being used in several industrial organisations [15, 3, 4, 16, 9, 6].

Modularity is central to the B method and this is achieved by structuring specifications and developments into "Abstract Machines". Machines define an encapsulated state with operations. The state is defined by the construction of a set theoretic model. Similar constructors to those of other model-oriented notations are available, although, in practice, one tends to use a number of variables each of simple type, rather than building more complex, user-defined, types as encouraged by VDM [18] or Z [25].

The invariant and other predicates are given in first order predicate calculus and set theory. The underlying logic is untyped and typing constraints appear as set memberships in the invariant along with the usual relationships between variables. The foundations are based on Zermelo set theory with an axiom of choice, an axiom of infinity, and an axiomatic definition of Cartesian product.

Operations are defined as "Generalised Substitutions". This departure from the before-after predicates of VDM and Z, yields the same expressive power whilst giving the language a more programmatic feel and thus making it more accessible to those with programming, rather than mathematical, experience. For example, a number of constructs are available which mimic the usual notation for assignment, $x := y$ for x becomes equal to y, to give loose specifications such as $x :\in S$ for x becomes any member of S. The semantics of operations are given by weakest preconditions.

The overall specification is structured by using machine composition. Specifications can be built incrementally by using the "sees" and "includes" mechanisms which respectively allow a read-only and read-write extension of a machine by new variables and operations. Data reification is provided by "refinements" and compositional development by "imports". Low-level machines, "implementations", can be written in an executable subset of the language and a library of "base" machines which can be automatically translated into C code.

Validation is supported by an animation facility which allows the developer to interactively "execute" a specification by providing input to simplify non-executable constructs or to resolve non-determinism. Verification is supported through the generation and discharge of proof obligations which ensure the consistency of specifications and the correctness of refinements.

The emphasis on modularity is also applied to proof. The motivation here is that the overall proof task should, as far as possible, be decomposed into proofs concerning individual machines. Once a machine has been proven consistent and correct, those proofs should be valid in any context in which this machine is used as part of a more complex specification. Indeed, it is this aim that has determined the structuring mechanisms available for machines. Thus, a highly compositional method is provided for proof so that, although numerous, proof obligations are mostly simple and the majority can be discharged automatically.

The B-Toolkit, a support system for B [2], provides two approaches to proving : the "autoprover" is used to automatically discharge the majority of proof obligations using a "rulebase" of built-in rules and tactics; and the "interprover" which is used to interactively explore the failed proof attempt and extend the rulebase with user defined "theories" which provide problem specific rules and tactics.

Proof is performed in a cycle of automation and interaction. Firstly the *autoprover* is used to discharge obligations using the built-in rulebase. Then the user browses the remaining unproven obligations and selects one to analyse. The leaves of its failed proof search tree are examined and the user selects a leaf which is believed to be valid. This is asserted as a lemma which proves the selected proof obligation. Lemmas thus generated are then proved interactively using the interprover by adding rules to the users rule set. Future invocations of the autoprover will now pick up the proof rules developed interactively in order to make the proof of this (and possibly other) proof obligations automatic. This cycle is repeated until all the proof obligations are discharged. Each iteration of this cycle introduces a new 'level' of user theory thus allowing the addition of only those rules which are necessary to prove the current obligations.

Automated proof is based upon a large *rule base* of built-in rules and associated control tactics. This rule base is not normally visible to the user, but rather provides a number of 'hooks' whereby user rules for forward and backward proof are called from the automatic process. Within the rule base, rules are organised into 'theories', linear collections of rules which are searched in sequence. Each rule can include a tactic call which directs the prover in the proof search, including tactics which encode the dependencies between rule sets. Thus the *theories* are not logical theories in the usual sense, as they are characterised by their use in proofs rather than logical form, and provide a control strategy for guiding the automatic proof.

Although the combination of autoproof and interproof constitutes a reasonably cost-effective way of completing the proofs of the large number of proof obligations generated by the method, to date it has been tuned to the proof of the universally quantified obligations which are typical, and is completely ineffective at refuting them by proving their negation.

3 Automatic proof using Resolution

3.1 Resolution

Resolution is perhaps the most widely investigated technique for automated theorem proving. First developed by Robinson in the mid 1960's [23], it provides a partial decision procedure for the (classical) first order predicate calculus. It is refutationally complete. That is, if the original premises are not satisfiable, it will eventually uncover a contradiction. We give a brief review of resolution.

To show that a conclusion, A, follows from a set of premises Γ, we negate the conclusion and show that the set of formulae $\Gamma \cup \neg A$ is unsatisfiable. The formulae are Skolemised and converted to clausal form. This gives a set of clauses, $\{C_i\}_{i=1}^{n}$, which may not have the same models as $\Gamma \cup \neg A$, however, by a theorem due originally to Skolem, the clausal form is equivalent as far as satisfiability is concerned. If $\{C_i\}_{i=1}^{n}$ is unsatisfiable, then so is $\Gamma \cup \neg A$. As it is the unsatisfiability of $\Gamma \cup \neg A$, and so the validity of A given Γ, we are interested in, this is sufficient.

In $\{C_i\}_{i=1}^n$, each clause is in the form of a disjunction of literals, $P_1 \vee P_2 \vee \ldots \vee P_m$, where each literal P_j is either an atom or the negation of an atom. With the formulae in this form, resolution is described by a single rule of inference. Given two clauses:

$$P(\bar{x}) \vee Q \qquad\qquad \neg P(\bar{y}) \vee R$$

where Q and R represent the remaining literals in the respective clauses, resolution adds a new clause generated from these two in the following fashion. First the literals $P(\bar{x})$ and $\neg P(\bar{y})$ are unified with their most general unifier θ (renaming variables as necessary) and then the two clauses under this unifier are combined while deleting the contradictory literals. So the resulting clause has the form

$$\theta(Q) \vee \theta(R)$$

The intuition behind this step is that there is an instance of the two literals, given by θ, under which they cannot both be satisfied. Thus, in any model, for the original clauses to be satisfied, the remaining literals under that substitution must hold. Clauses continue to be generated in this fashion using the extended clause set. Eventually, if the clauses are unsatisfiable, the the clause set will contain clauses P and $\neg P$ for some atom P. Using the resolution rule this results in the clause with no literals.

The resolution principle gives us a simple yet powerful way of performing proofs. It is relatively simple to establish the soundness of the procedure, and also its refutational completeness[8]. It is simple to automate the process. However it suffers from a severe drawback: for a problem of any complexity, it produces a very large search space. There has been a large effort to devise strategies to control the number of resolvents generated while maintaining the soundness and refutational completeness of the procedure.

Paramodulation The most basic way to handle the equality predicate in first order logic is to include amongst the clauses an axiomatisation of the four laws of equational logic. This leads to an explosion in the number of clauses needed to express a particular problem, with the corresponding greatly increased search space. In order to handle the equality predicate more simply, the resolution procedure is augmented by the technique of *paramodulation*. Paramodulation adds an extra inference rule to the resolution method. Given a clause $L(\ldots a \ldots) \vee C$ and a clause containing the equality predicate, $(a = b) \vee D$, we can derive the paramodulant $L(\ldots b \ldots) \vee C \vee D$. If used with the addition of the reflexivity axiom $(x = x)$, resolution and paramodulation combined is refutationally complete [8].

The paramodulation method can be combined with some of the normal strategies used in resolution, but care must be taken to preserve completeness. While a fairly successful technique for handling the equality predicate, the search space generated is still very large. It does not have the power that a reduction method using rewrite rules has to rapidly decide the equality of terms. Completion procedures, such as that of Knuth and Bendix can also be included to augment the power of the rewriting by deriving additional rules.

3.2 Otter and Mace

Otter and Mace are a complementary pair of tools from Argonne National Laboratory. Otter [21] is a resolution-based automated deduction system. It has a front end which reduces formulae in first-order logic with equality to clausal form by Skolemisation. As well as resolution and paramodulation, it includes facilities for term rewriting, term orderings, Knuth-Bendix completion, weighting, and strategies for directing and restricting searches for proofs.

MACE [22] is a program that can be used to search for (small) finite models of first-order statements. It takes the same input format as Otter and, for a given number, searches exhaustively for models of that size. The size has to be very small:in our problem we found that models of more than 10 elements became infeasible.

4 Theories and models

We summarise part of the theory of first-order logic (following [8]) and then discuss how this can be used to support the search for refutations in formal software development.

We assume a basic knowledge of first-order predicate logic, with constant, function and predicate symbols, with free and bound variables. We begin with a discussion the of the interpretation of first-order formulae. (We omit for brevity the standard consideration of valuations of free variables.)

Definition 1 Interpretation *An interpretation I of a first-order formula Φ is a set D, together mapping μ_I from the symbols of Φ to D such that:*

1. *for each constant symbol c, $\mu_I(c) \in D$*
2. *for each n-ary function symbol f, $\mu_I(f) : D^n \to D$*
3. *for each n-ary predicate symbol P, $\mu_I(P) : D^n \to \{\mathbf{T}, \mathbf{F}\}$*

Under the interpretation, Φ can be evaluated to **T** or **F** according to the rules for predicate logic.

Definition 2 Model. *If given a first-order formula Φ and an interpretation m, $\mu_m(\Phi)$ evaluates to* **T**, *then m is a model for Φ. The set of models for a formula Φ is given by $[\Phi]$.*

Definition 3 Satisfiable. *A first-order formula Φ is satisfiable if there exists a model for Φ. In this case, the model m satisfies Φ.*

A set of formula Γ is a (consistent)*Theory* if there is some interpretation m such that m is a model for each formula in Γ.

Definition 4 Validity. *A first-order formula Φ is valid if every interpretation satisfies Φ. This is written $\models \Phi$.*

Definition 5 Logical Consequence. *Given a set of first-order formulas Γ, a first-order formula Φ is a logical consequence of Γ (or is valid with respect to Γ) if every interpretation I which satisfies Γ also satisfies Φ (or in other words $[\Gamma] \subseteq [\Phi]$). This is written $\Gamma \models \Phi$.*

The usual process of theorem proving is to prove the validity of a formula with respect to a set of initial formulas or axioms, that is to develop a sound proof theory.

Definition 6 Soundness. *A logical entailment system $\vdash$ is sound if whenever Φ is provable from Γ, $\Gamma \vdash \Phi$, then $\Gamma \models \Phi$.*

4.1 Refutation

A refutation by proof of Φ is proof of $\neg\Phi$.

Proposition 1 *If Γ is consistent and $\Gamma \models \neg\Phi$ then $\Gamma \not\models \Phi$.*

This is clearly true as for any model for Γ, $\neg\Phi$ holds and therefore Φ cannot hold. Thus to achieve a refutation, the proof procedure (e.g. resolution) can attempt to show $\Gamma \vdash \neg\Phi$.

Refutation by proof may not be possible, even if a theorem is not provable. Refutation by proof would show $\neg\Phi$ in all models, whereas, Φ is not provable if there is *some* model for the axiomatisation Γ such that Φ does not hold. That is, there exists some counter-example model which does not satisfy the formula.

Proposition 2 $\Gamma \not\models \Phi$ *if there exists a counter-example, that is a model* m *for* Γ *which is not a model for* Φ.

This proposition suggests two methods for generating counter-examples to the validity of a formula Φ. One is from the following corollary. Note that we informally extend the notation for models to include sets of formulae and conjunctions.

Corollary 1 $\Gamma \not\models \Phi$ *if there exists a model* m *such that* $m \in [\Gamma \wedge \neg\Phi]$.

This is true since if $m \in [\neg\Phi]$ then $m \notin [\Phi]$.

Thus we can show the invalidity of a formula by generating counterexamples. Model generators such as Mace can generate finite models for a set of first-order clauses. Thus a refutational proof method is to present $\Gamma \wedge \neg\Phi$ to the model generator and look for models.

The second method to refute Φ by counterexample comes from the observation that if $\Gamma \models \Phi$ then $[\Gamma] \subseteq [\Phi]$. Thus if $[\Gamma \wedge \Phi] \subset [\Gamma]$, then there is a model $m \in [\Gamma]$ such that $m \notin [\Phi]$ and thus $\Gamma \not\models \Phi$. Hence if $[\Gamma \wedge \Phi]$ is strictly contained within $[\Gamma]$ then the models in $[\Gamma] - [\Gamma \wedge \Phi]$ are counterexamples to $\Gamma \models \Phi$.

Thus model generation and checking provides a means of producing counter-examples to refute (show the invalidity of) a formula under an axiomatisation.

4.2 Weaker theories

In general, the axiomatisation Γ of the base theory of a method like B is likely to be too rich for exhaustive model generation. This is because:

1. Γ is too large for model generation with reasonable time and space restrictions.
2. Second-order axioms, such as induction which are not realisable in a first-order system.

Thus in general, if we are to use resolution, we have to carry out theorem proving in a weaker theory, ie. under a subset $\Theta \subseteq \Gamma$. For proof this is not a problem since validity in classical logic is monotonic:

Proposition 3 **Monotonicity** *If* $\Theta \models \Phi$ *then* $\Theta \cup \Delta \models \Phi$.

However, for searching for counterexamples, this is not the case. If we find a model m such that $m \in [\Theta \wedge \Phi]$, it is not necessarily the case that $m \in [\Gamma \wedge \Phi]$, since $[\Gamma] \subseteq [\Theta]$, and so perhaps $m \notin [\Gamma]$. However, we can make the following statement concerning the weaker theory:

Proposition 4 *Given an interpretation* m *such that* $m \in [\Theta]$ *and* $m \notin [\Phi]$ *(or alternatively* $m \in [\neg\Phi]$ *), if* $m \in [\Gamma - \Theta]$ *then* $\Gamma \not\models \Phi$.

At first sight this seems to be to no advantage as the model has to satisfy the whole of Γ. However, it can give rise to a counter-example generation procedure. The model-generator generates models which satisfy

$\Theta \wedge \neg\Phi$. These models then become *candidates* for counter-examples for $\Gamma \wedge \Phi$, and can be tested via model-checking against $\Gamma - \Theta$.

These candidates are akin to test cases which *may* show up design faults. Thus, this method combines features of proof with features of testing.

4.3 Extending Theories

Having weakened the theory, we can extend it to a larger theory if the larger theory does not negate any existing theory, that is if the extension is conservative.

Definition 7 Conservative Extension Δ *is a conservative extension to* Θ *if for any* Φ*,* $\Theta \models \Phi$ *if and only if* $\Theta \cup \Delta \models \Phi$*, where* Φ *is in the language of* Θ*.*

Then any model of Θ can be extended to a model[1]of Γ. In this case, we can be sure that a counter-example in Θ will also yield a counter-example in $\Gamma = \Theta \cup \Delta$. (The test is sure to fail.) A similar result applies for Definitional Extensions. Note that, in particular, *induction* gives a conservative extension and so inductive theorems can be disproved by finding a finite counter-example.

5 Example

5.1 A simple B development

We demonstrate the approach on an simple example development in the B method. Even on this extremely small example we see the added value which is possible by considering refutation as well as proof in development, although we do not claim this result to be anything but a demonstration of potential.

The example specification and refinement are presented in Figure 1. The abstract specification (on the left hand side) defines a set-valued variable, ss, with operations to add an element to the set and to remove an arbitrary element from it. It also defines an underspecified function which chooses and returns an element of the set.

The concrete specification attempts to implement the abstract by choosing an element will always return the last element added, that element having been stored in the variable, $rr2$, since it was added.

The reader may note that, as it stands, there is an error in the refinement. If the last element added is removed before it is chosen, then the result of the next invocation of choose will not be from the set. Note that because of the underspecification of the remove operation, this error could be difficult to find by testing alone.

The proof of consistency of the abstract specification gives four proof obligations, all of which are proved automatically. The proof of the validity of the refinement comprises nine proof obligations of which seven are discharged by the autoprover. Of the remaining two, inspection quickly reveals that one is false, whilst the status of the other is less clear. The false proof obligation is given below.

```
∀ ss, rr, ee (
    (
    ((ss ≠ {} ) ⇒ (rr ∈ ss)) ∧
    ee ∈ ss ∧
    ss ≠ {} ∧
```

[1]Strictly speaking this should be for any model of Θ, there is an *elementary equivalent model* which can be extended to a model of Γ [27, 10]. For the purposes of this paper, this can be ignored.

```
MACHINE    aset

SETS    ENUMSET

VARIABLES    ss

INVARIANT    ss ⊆ ENUMSET

INITIALISATION    ss := {}

OPERATIONS

    addelem ( ee )    ≙
        PRE     ee ∈ ENUMSET
        THEN     ss := ss ∪ { ee }
        END ;

    remelem    ≙
        PRE     ss ≠ {}
        THEN
            ANY     ee
            WHERE     ee ∈ ss
            THEN
                ss := ss − { ee }
            END
        END ;

    oo ⟵ chooseelem    ≙
        PRE     ss ≠ {}
        THEN
            oo :∈ ss
        END

END
```

```
REFINEMENT    asetR

REFINES    aset

VARIABLES    ss2 , rr2

INVARIANT
    ss2 ⊆ ENUMSET ∧
    rr2 ∈ ENUMSET ∧
    ( ss2 ≠ {} ⇒ rr2 ∈ ss2 ) ∧
    ss2 = ss

INITIALISATION
    ss2 := {}  ||
    rr2 :∈ ENUMSET

OPERATIONS

    addelem ( ee )    ≙
        PRE     ee ∈ ENUMSET
        THEN
            ss2 := ss2 ∪ { ee }  ||
            rr2 := ee
        END ;

    remelem    ≙
        PRE     ss2 ≠ {}
        THEN
            ANY     ee
            WHERE     ee ∈ ss2
            THEN
                ss2 := ss2 − { ee }
            END
        END ;

    oo ⟵ chooseelem    ≙
        PRE     ss2 ≠ {}
        THEN
            oo := rr2
        END

END
```

Figure 1: Abstract and Concrete specification of a set valued variable.

$rr \in ss \wedge$
$\exists\, x \cdot (\, x \in ss \wedge x \neq ee\,)$
$)$
$\Rightarrow$
$(rr \in ss) \wedge (\, rr \neq ee\,)$
$)$

This can be seen to be false since although there some x necessarily different from ee, there is no reason why this x should be equal to rr, and so it is possible that $rr = ee$.

Of course, in a larger example, with many hundreds of unproved obligations, even this cursory inspection would be very costly. In what follows, we show how resolution based theorem proving techniques can be used to automatically refute the false obligation and discover a counterexample to it, thus leading to a low cost means of finding the fault in the design.

5.2 Axiomatisation in Otter

In order to investigate the validity of this unproven proof obligation, we employ a mixture of proof and counter-example search using Otter and Mace. To use these tools one has to provide them with an axiomatisation of the specification in first order logic.

```
set(auto).
formula_list(usable).

all x -(isaSet(x) & isaElem(x)).
-isaElem(undef) & -isaSet(undef).

isaElem(e1).
isaElem(e2).
e1!=e2.
isaSet(empty).
isaSet(ENUM).
empty!= ENUM.
ENUM = add(e1, add(e2,empty)).

% empty and universe
all x ( isaElem(x) -> -member(x,empty) & member(x,ENUM)).

% add form
all x s (isaSet(s) & isaElem(x) -> isaSet(add(x,s))).
all x s (-(isaSet(s) & isaElem(x)) -> (undef=add(x,s))).

% comm add
all x y s (isaElem(x)& isaElem(y)& isaSet(s)
              -> (add(x,add(y,s))= add(y,add(x,s)))).

% idem add
all x s (isaElem(x)& isaSet(s)
                     -> (add(x,add(x,s))= add(x,s))).

% member definition
all x y s (isaElem(x)& isaElem(y)& isaSet(s)
       -> (member(x,add(y,s))<-> x=y | member(x,s))).
end_of_list.
```

Figure 2: Axioms for a doubleton set world in Otter.

Figure 2 presents the axiomatisation in Otter format of finite set theory which we used. It defines two types, sets and elements, and constructors for finite sets `add` and `empty`. Note that we know that the counterexample we are looking for will be a finite set and so we can work in the simpler logic.

In order to give a tightly defined set of models, in this axiomatisation, we have been very specific in our requirements on possible models for finite sets. We require two distinct elements `e1` and `e2` which are not sets, and also that no sets are members of other sets. We add an extra element `undef` representing non-well-formed terms which reduces the number of models by forcing a single interpretation for many undefined terms.

```
====================== Model #1 at 3.21 seconds:
 isaElem :
        0 1 2 3 4 5 6
    -----------------
        F F F F F T T

 isaSet :
        0 1 2 3 4 5 6
    -----------------
        F T T T T F F

 undef: 0
 empty: 1
 ENUM: 4
 e1: 5
 e2: 6

 add :
      | 0 1 2 3 4 5 6
    --+--------------
    0 | 0 0 0 0 0 0 0
    1 | 0 0 0 0 0 0 0
    2 | 0 0 0 0 0 0 0
    3 | 0 0 0 0 0 0 0
    4 | 0 0 0 0 0 0 0
    5 | 0 2 2 4 4 0 0
    6 | 0 3 4 3 4 0 0

 member :
      | 0 1 2 3 4 5 6
    --+--------------
    0 | - - - - - - -
    1 | - - - - - - -
    2 | - - - - - - -
    3 | - - - - - - -
    4 | - - - - - - -
    5 | - F T F T - -
    6 | - F F T T - -
```

Figure 3: A model of the doubleton set produced by Mace.

This axiomatisation produces simple models of finite set theory with a minimum of seven elements, *empty*, *e1*, *e2*, $\{e1\}$,$\{e2\}$,$\{e1,e2\}$, *undef*, eliminating many models which construct sets from other sets. In this simple case, we were guided by our understanding of the proof obligation that a two element set would be sufficient to refute the obligation. In general, a more systematic method of generating models with 1, 2, 3 and more objects could be envisaged.

Figure 3 presents one of the models for the finite set theory above as produced by Mace. In fact, 5040 models are produced by Mace which are all simply renaming permutations of this one ($5040 = 7!$). The number of such models can be reduced by explicitly assigning domain elements to terms. This also improves efficiency. In this example, we can easily use this technique to give a unique model.

5.3 Refutation by Proof

In the specification given above, there were two unproven proof obligations. Our intention here is to demonstrate that the status of the false one can be uncovered automatically, leading to the correction of the error and the full proof of refinement.

The (false) Proof Obligation is easily refuted by resolution using Otter. The false proof obligation in Otter format is:

```
(all ss2 rr2 ee
   ((isaSet(ss2) & isaElem(rr2) & isaElem(ee)) ->
    (
      - ( (ss2 != empty) ->  member(rr2, ss2) )|
      - member(ee, ss2)|
      - (ss2 != empty) |
      - member(rr2, ss2)|
      - (exists X (isaElem(X) -> (member(X,ss2) & (X!=ee)))) |
     (member(rr2, ss2) & (rr2 != ee))
 ))).
```

```
-----> EMPTY CLAUSE at 8.53 sec ---->
---------------- PROOF ----------------
1 [] -isaElem(undef).
2 [] -isaSet(undef).
3 [] e1!=e2.
4 [copy,3,flip.1] e2!=e1.
5 [] -isaSet(x)| -isaElem(y)|isaSet(add(y,x)).
8 [] empty!=ENUM.
16 [] -isaElem(x)| -member(x,empty).
17 [] -isaElem(x)|member(x,ENUM).
20 [] -isaElem(x)| -isaElem(y)| -isaSet(z)| -member(x,add(y,z))|
          x=y|member(x,z).
24 [] -isaSet(x)| -isaElem(y)| -isaElem(z)| -member(z,x)|
          x=empty| -member(y,x)| -member(u,x)|u=z|y!=z.
26 [factor,20,1,2] -isaElem(x)| -isaSet(y)| -member(x,add(x,y))|
          x=x|member(x,y).
30 [factor,24,2,3,factor_simp] -isaSet(x)| -isaElem(y)|
          -member(y,x)|x=empty| -member(z,x)|z=y|y!=y.
32 [] isaSet(empty).
33 [] isaSet(ENUM).
34 [] isaElem(e1).
35 [] isaElem(e2).
36 [] ENUM=add(e1,add(e2,empty)).
37 [copy,36,flip.1] add(e1,add(e2,empty))=ENUM.
55 [hyper,34,17] member(e1,ENUM).
72 [hyper,35,17] member(e2,ENUM).
74 [hyper,35,5,32] isaSet(add(e2,empty)).
110 [para_from,37.1.1,26.3.2,unit_del,34,74,55] e1=e1|
          member(e1,add(e2,empty)).
472 [hyper,110,20,34,35,32,unit_del,4] e1=e1|member(e1,empty).
492 [hyper,472,16,34] e1=e1.
498 [hyper,492,30,33,34,55,72,unit_del,8,4] $F.
------------ end of proof -------------
```

Figure 4: A proof of the falsehood of the proof obligation

On adding this to the axiomatisation, resolution finds a contradiction very quickly thus demonstrating the falsehood of the proof obligation, resulting in a proof, reconstructed by Otter as in Figure 4.

5.4 Refutation by Model Generation

The same result can be achieved by running the model generator on this augmented theory. As expected there are now no models of size 7. The fact that any models are invalidated by addition of the proof obligation implies that the proof obligation is not true on all models and is therefore false.

Interestingly, this means of achieving a refutation is successful even without the axiom $ENUM = add(e1, add(e2, empty))$ although it is slower than the refutation by proof described above.

5.5 Finding counter examples by Model Generation

Although refuting the proof obligation indicates a fault in design, it does not, in itself, lead us to identify the source of problem. We now show how one can employ the model generator to find a counter example to the proof obligation and hence uncover the bug.

Naturally, as the obligation is false, running the model generator on axiomatisation with the proof obligation yields no models. However, if we negate the proof obligation (thus making it satisfiable) and run the model generator, we get the same models as for the axiomatisation but now 4 new skolem constants $\$c1, \ldots, \$c4$ are added. These correspond to the variables existentially quantified in the negated proof obligation.

```
======================= Model #1 at 3.24 seconds:
 isaElem :
        0 1 2 3 4 5 6
    -----------------
        F F F F F T T

 isaSet :
        0 1 2 3 4 5 6
    -----------------
        F T T T T F F

 undef: 0
 empty: 1
 ENUM: 4
 e1: 5
 e2: 6

 add :
      | 0 1 2 3 4 5 6
    --+--------------
    0 | 0 0 0 0 0 0 0
    1 | 0 0 0 0 0 0 0
    2 | 0 0 0 0 0 0 0
    3 | 0 0 0 0 0 0 0
    4 | 0 0 0 0 0 0 0
    5 | 0 2 2 4 4 0 0
    6 | 0 3 4 3 4 0 0

 member :
      | 0 1 2 3 4 5 6
    --+--------------
    0 | - - - - - - -
    1 | - - - - - - -
    2 | - - - - - - -
    3 | - - - - - - -
    4 | - - - - - - -
    5 | - F T F T - -
    6 | - F F T T - -

 $c4: 4
 $c3: 5
 $c2: 5
 $c1: 6
```

Figure 5: The Model Generator is used to find a counter example

One such counter-example is given in Figure 5 as it presented by Mace. From this the counter-example can be deduced in B format. Translated back into the original problem syntax, this model attributes the following instantiations to the quantified variables, which indicates the counterexample we are searching for.

$$ss2 = ENUM, \quad rr2 = e1, \quad ee = e1, \quad X = e2$$

This can then be checked by testing it within the B-tookit, and the problem with the refinement uncovered.

On correcting the error, all proof obligations are easily proved, and therefore we have avoided the necessity to even inspect the last of the original proof obligations, which would require more work to prove or disprove.

6 Discussion

We have shown how automatic theorem proving techniques can be used to show the falsehood of proof obligations and so can uncover design faults in formal development using the B method. Although the techniques used are standard, their application to refutation in this type of formal development is novel and we believe that the transfer of this technology could significantly improve the cost effectiveness of use of such methods.

Naturally, there is still much work to be done before this can be achieved. Significantly, only a tiny fragment of the B language was axiomatised for this example and there is a considerable piece of work to be done to extend this. Furthermore, we have not shown any formal correspondence between the axiomatisation we used and that of the B language, although as first-order classical systems they are clearly similar. Also this is not necessarily a problem as faults when discovered are tested within the original B context and so any difference in semantics would not lead to erroneous conclusions.

The problem of scale is perhaps the most significant. In the presented example, with no attempt at optimisation, the refutation proof was found in 8 seconds and the counter example model in 3 seconds. However, limitations on the size of models produced by the model generator render this tool effectively useless for more realistic problems. We plan to investigate what can be achieved with other model checkers. Larger examples also require a more skillful use of resolution and term-rewriting techniques but, here at least, it should be possible to tune the tools to the requirements of the application. Experience will show how far this can be developed.

As discussed above, the techniques presented are in some ways akin to testing. Over the last two decades, research has been undertaken into the possibility of systematically deriving tests from formal specifications. This has been extensively researched for Algebraic specifications and Process based specifications (eg. [13] [14]). Error detecting in process algebras has also been considered in [24]. Far less research has been done on finding faults in model-oriented specifications, but we note [11] which proposes techniques to derive test cases from VDM specifications, the PROST-Objects project [26] and the "Nitpick" tool [17] which is designed to find faults in Z specifications

Human directed proof is an expensive and highly specialised activity. In B, modularity mechanisms which have been tuned to make proof compositional result in very many (but relatively simple) proof obligations. Depending on the level of assurance that is required by the application, typically, only some of these are proved from first principles. Others are reduced to acceptable "lemmas", whilst others may simply be accepted without proof. In the absence of a complete proof from first principles of all the proof obligations, the purpose of the proof activity becomes bug finding rather than absolute verification and value comes therefore from the finding of false obligations rather than true ones. As in all science, the validation of a theory comes from the failure to find counterexamples rather than through the observation of confirming instances. From this perspective, there is little point in proving 90% of the obligations which are true, if the status of the false ones among the remaining 10% is not uncovered.

References

[1] J-R. Abrial. *The B-Book*. ISBN 0521496195, Cambridge University Press, 1996.

[2] B-Core (UK) Ltd. The B-Toolkit. Welcome page URL <http://www.b-core.com>

[3] D. Bert (Ed) Proc. of the 2nd International B Conference, Montpellier, France, April 22-24, 1998. Springer Verlag, LNCS 1393.

[4] J. C. Bicarregui et al. Formal Methods into Practice: case studies in the application of the B Method IEE Proceedings on Software Engineering, vol 144, issue 2 pp.119-133, April 1997.

[5] J.C. Bicarregui and B.M. Matthews, Formal Methods in Practice: a comparison of two support systems for proof. SOFSEM'95: Theory and Practice of Informatics, Bartosek et al. (Eds), LNCS 1012, Springer-Verlag.

[6] J.C. Bicarregui, A.J.J. Dick, B.M. Matthews and E. Woods, Making the most of formal specification through Animation, Testing and Proof, Science of Computer Programming, Volume 29/1-2, pp. 55 - 80, Elsevier-Science, June 1997.

[7] D. Clutterbuck, J.C. Bicarregui and B.M. Matthews, Experiences with Proof in a Formal Development Proceedings of 1st International Conference on B, Institut de Recherche en Informatique de Nantes, France, November 1996.

[8] C.-L. Chang and R. C.-T. Lee, Symbolic Logic and Mechanical Theorem Proving, Academic Press, 1973

[9] B. Dehbonei and F. Mejioa. Formal Methods in the railways signaling industry. In M. Naftalin and T. Denvir, editors, Proceedings of Formal Methods Europe '94, Lecture Notes in Computer Science, pp. 26-34, Springer-Verlag, 1994.

[10] R. Diaconescu, J. Goguen and P. Stefaneas, Logical support for modularisation, 2nd BRA Logical Frameworks Workshop, Edinburgh, 1991

[11] A.J.J.Dick and A. Faivre, Autometing the Generation and Sequencing of Test Cases from Model-Based Specifications. Proc. FME'93, Industrial Strength Formal Methods, Springer-Verlag, LNCS 670, 1993.

[12] E Dijkstra. A discipline of programming, Prentice-Hall 1976.

[13] M-C Gaudel, Testing can be formal too. TAPSOFT'95,: Theory and Practice of Software Development, eds. PD Mosses, M Nielsen, MI Schwartzbach, LNCS 915, Springer-Verlag, 1995.

[14] M-C Gaudel and P. R. James, Testing Algebraic Data Types and Processes: a Unifying Theory 3rd International Workshop on Formal Methods for Industrial Critical Systems, Amsterdam, May 1998.

[15] H. Habrias (Ed.), Proceeding of 1st International Conference on B, Institut de Recherche en Informatique de Nantes, France, November 1996

[16] J. Hoare, J. Dick, D. Neilsen and I. Sorensen. Applying the B technologies to CICS, Proceeding of FME'96, Third International Symposium of Formal Methods Europe, ISBN: 3-540-60973-3, pp. 74-84, Springer-Verlag (LNCS 1051), March 1996.

[17] D. Jackson and C. A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7):484–495, July 1996.

[18] C.B. Jones. Systematic Software Development using VDM. Prentice Hall, 1990.

[19] I. Lakatos Proofs and Refutations Cambridge University Press, 1976.

[20] M. Mac an Airchinnigh, Tutorial Lecture Notes on the Irish School of the VDM. In Proceedings of VDM'91, eds. Prehn, Toetenel, LNCS 552, Springer-Verlag, 1991.

[21] W. McCune, The Otter 3.0 Reference Manual and Guide. Argonne National Laboratory, ftp://info.mcs.anl.gov/pub/Otter/Papers/otter3_manual.ps.gz

[22] W. McCune, A Davis-Putnam Program and its Application to Finite First-Order Model Search: Quasigroup Existence Problems. Argonne National Laboratory, ftp://info.mcs.anl.gov/pub/Otter/www-misc/anldp.ps.Z

[23] J.A. Robinson Logic: Form and Function, Elsevier North-Holland, 1979.

[24] A W Roscoe and G Lowe, Using CSP to detect errors in the TMN protocol. IEEE Transactions on Software Engineering, 23(10):659-669, 1997.

[25] J.M. Spivey, The Z notation: a reference manual, Prentice Hall 1989.

[26] S. Stepney. Testing as abstraction. ZUM'95: the Z User Meeting, LNCS 967, Springer-Verlag, 1995.

[27] P.A.S. Veloso and S.R.M. Veloso, Some remarks on conservative extensions: a Socratic dialogue, Bulletin of European Association of Theoretical Computer Science, vol. 43 pp. 189-198, 1991.