Journal of Universal Computer Science, vol. 10, no. 11 (2004), 1519-1539
submitted: 10/5/04, accepted: 20/8/04, appeared: 28/11/04 © J.UCS

Automated Support for Enterprise Information Systems

John Andrew van der Poll
(University of South Africa, South Africa
vdpolja@unisa.ac.za)

Paula Kotzé
(University of South Africa, South Africa
kotzep@unisa.ac.za)

Willem Adrian Labuschagne
(University of Otago, New Zealand
willem@cs.otago.ac.nz)

Abstract: A condensed specification of a multi-level marketing (MLM) enterprise which can be modelled by mathematical forests and trees is presented in Z. We thereafter identify a number of proof obligations that result from operations on the state space. Z is based on first-order logic and a strongly-typed fragment of Zermelo-Fraenkel set theory, hence the feasibility of using certain reasoning heuristics developed for proving theorems in set theory is investigated for discharging the identified proof obligations. Using the automated reasoner OTTER, we illustrate how these proof obligations from a real-life enterprise may successfully be discharged using a suite of well-chosen heuristics.

Key Words: automated reasoning, formal specification, heuristics, OTTER, set theory, Z

Category: D.2.4, F.3.1, F.4.1, I.2.3

1 Introduction

Among the benefits to be gained by using a formal specification language like Z [Spivey 1992, Bowen 2001] is that the specifier can prove things about the specification. The process of constructing proofs can aid in the understanding of the system, may reveal hidden assumptions and increase user confidence in the final product. A specification without proofs is untested since it may be inconsistent and describe properties that were not intended, or omit those that were [Woodcock and Davies 1996].

The huge cost and inconvenience of detecting and correcting errors only after the system has been released [Fagan 1976], justifies the effort to identify and correct errors at an early stage (e.g. specification phase). However, the readiness with which the information technology industry would accept such a methodology is likely to depend on the availability of environments that ease the burden on the specifier by automating many of the specifier's tasks. The more sophisticated the environment (and thus the greater its contribution to the partnership), the more natural becomes the inclusion of a reasoning algorithm as one component.

Reasoning about the properties of an enterprise information system at the specification level may, however, be a non-trivial task owing to the size of the system or the complexity of the structures that make up such a system. Accounts of costly, yet failed proof attempts exist, and one such prohibitively difficult attempt to generate a proof monolithically in one step from a stated property to a protocol is reported on in [Mokkedem et al. 2000]. Eventually the one step proof was abandoned, unfinished, after 18 months of effort which led to the specifiers adopting an incremental proof strategy in the end. Hence there appears to be a need for ways to increase the feasibility of the automation offered by a reasoning program and for cutting down on the average time it takes the reasoner to find a proof. Resolution is one of the automated reasoning mechanisms available to a specifier and in this paper we adopt the resolution-based theorem-proving program, OTTER for reasoning about our specification.

The aim of this paper is to describe a case study illustrating an empirical approach to improving the usefulness of OTTER for discharging the proof obligations incurred by specifying a system in Z. Heuristics to enhance the reasoning process are described, in the hope that readers may be moved to conduct their own experiments along these lines. It seems fair to anticipate that as heuristics of tried and tested utility accumulate, specifiers will become more favourably disposed to applying reasoners like OTTER.

1.1 Why Reasoning Heuristics?

Traditionally set-theoretic proofs pose demanding challenges to automated reasoning programs [Boyer et al. 1986, Quaife 1992], since unlike number theory or group theory or applications to real systems such as power stations, the denotations of terms in the context of set theory are strongly hierarchical: one object (perhaps at a very fine level of granularity) is a member of another (coarser) object, which in turn may be a member of a higher-level (even coarser) object, and so on. The possibility of moving between levels is a provocation to much irrelevant activity; intelligence would be realised by *heuristics* that limit the movement up or down to productive changes of granularity [Bundy 1999]. For example, in proving that

$$\mathbb{P}(A) \cap \mathbb{P}(B) = \mathbb{P}(A \cap B)$$

the movement from the level of elements of A up to the level of elements of $\mathbb{P}(A)$ should not be iterated to the level of elements of $\mathbb{P}(\mathbb{P}(A))$. Humans understand this intuitively and semantically – the challenge is to enunciate syntactic constraining principles.

It is furthermore an open problem as to which inference rule would accommodate the treatment of equality in set theory in as successful a way as paramodulation accommodates equality-oriented reasoning in algebra [Wos and Pieper 1999]. Paramodulation is a rule applied to a pair of clauses and requiring that at least one of the two contains a positive equality literal, and yielding a clause in which an equality substitution corresponding to the equality literal has occurred. The aim of an application of paramodulation is, therefore, to cause an equality substitution to take place from one clause into another. An example is given in [Section 2.2.4]. [Bailin and Barker-Plummer 1993] claimed to have found such a 'paramodulation rule' for set theory but were unable to solve the challenge problem which accompanies research problem #8 in [Wos 1988]. Their approach is furthermore not resolution-based.

Devising a set of heuristics appears to a suitable strategy for reasoning about set-theoretic constructs [Bundy 1999]. A promising preliminary set of heuristics was developed in [van der Poll and Labuschagne 1999] and since Z [Spivey 1992] is in part based on Zermelo-Fraenkel (ZF) set theory, it makes sense to investigate to what extent these heuristics may be used to successfully reason about the properties of a complex system specified in Z. As a challenging and relevant example of such a system we have selected an extension of the franchise concept, namely, a real-life multi-level marketing enterprise [GNLD 1997].

1.2 Structure of this Paper

An overview of some resolution principles used in this paper is presented in [Section 2], followed in [Section 3] by a brief introduction to OTTER [McCune 2003], the automated reasoner used in this work. The intention with this recapitulation is to enable the broadest possible range of readers to repeat, should they wish to, case studies similar to that reported in the remainder of this paper. A selection of heuristics for reasoning about set-theoretic structures is presented in [Section 4]. A brief Z specification of a generic multi-level marketing enterprise [GNLD 1997] is given in [Section 5]. The selected heuristics are applied in [Section 6] where two proof obligations (POs) are stated and discharged using an automated reasoner. A summary and some ideas for future work conclude this paper.

2 Resolution Inferences and Strategies

The resolution principle [Robinson 1965a, Robinson 1965b] operates on sets of clauses and is closely related to the proof technique of *reductio ad absurdum* (i.e. proof by contradiction). Essentially the idea is to determine whether a given set of clauses (say S) contains the empty clause, $\square$. If S contains $\square$ then S is unsatisfiable. If S does not yet explicitly contain $\square$, then the resolution mechanism attempts to derive $\Box$ using the clauses of S. The resolution principle is sound and refutation complete in the sense that it can always generate the empty clause from an unsatisfiable set of clauses (provided appropriate housekeeping, called factoring, is done to eliminate duplicate clauses) [Leitsch 1997].

In practice, if a goal G is believed to be provable from a set F of clauses, then we add the denial of the goal, i.e. the negation $\neg G$ of the potential theorem, to F and attempt to derive $\Box$. We illustrate this process below.

2.1 Binary Resolution

Consider the deductive database of axioms and rules in [Fig. 1] (adapted from [Date 1995]).

$$Parent(Dennis, John) \tag{1}$$
$$Parent(John, Peter) \tag{2}$$
$$(\forall x)(\forall y)(Parent(x, y) \rightarrow Ancestor(x, y)) \tag{3}$$
$$(\forall x)(\forall y)(\forall z)((Parent(x, y) \wedge Ancestor(y, z)) \rightarrow Ancestor(x, z)) \tag{4}$$

Figure 1: Initial database of rules

Suppose we want to prove that *Dennis* is an ancestor of *Peter*, i.e.

$$Ancestor(Dennis, Peter) \tag{5}$$

The first step is to rewrite the content of [Fig. 1] in clausal form. This produces the clause set in [Fig. 2]. Next we negate (5), the theorem we wish to prove, add such negated form to the clause set (6) to (9) and attempt to derive the empty clause. The negated form of (5) is simply:

$$\neg Ancestor(Dennis, Peter) \tag{10}$$

The final step is to search for a refutation and this is illustrated in [Fig. 3]. Complementary literals in the given clauses are resolved and an integral part of this process is to unify the variables and constants appearing as arguments of terms. For example: in generating clause [B1] in [Fig. 3] we resolve clause (10) with clause (9), replacing x with *Dennis* and z with *Peter* respectively in (9). The variable y in (9) is then renamed to x and this produces clause [B1] as the

$$Parent(Dennis, John) \tag{6}$$
$$Parent(John, Peter) \tag{7}$$
$$\neg Parent(x, y) \vee Ancestor(x, y) \tag{8}$$
$$\neg Parent(x, y) \vee \neg Ancestor(y, z) \vee Ancestor(x, z) \tag{9}$$

Figure 2: Clausal form of [Fig. 1]

$[B1]$	$\neg Parent(Dennis, x) \vee \neg Ancestor(x, Peter)$	[Resolvent of (10) and (9)]
$[B2]$	$\neg Ancestor(John, Peter)$	[Resolvent of $[B1]$ and (6)]
$[B3]$	$\neg Parent(John, Peter)$	[Resolvent of $[B2]$ and (8)]
$[B4]$	$\square$	[Resolvent of $[B3]$ and (7)]

Figure 3: A binary resolution proof

answer. The reader is referred to [Wos et al. 1992] for a thorough treatment of unification.

The binary resolution rule illustrated in [Fig. 3], and with which logic programmers are familiar tends toward inefficiency, and so the emphasis in resolution-based automated reasoning is on rules that permit several clauses to participate simultaneously (e.g. unit resulting resolution and hyperresolution discussed below). In the light of this we present below further rules of inference using the sample clause set in [Fig. 2].

2.2 Further Inference Rules

2.2.1 UR-resolution

Unit Resulting (UR) resolution is the process whereby n unit clauses are simultaneously resolved with a clause consisting of $(n + 1)$ literals, called the nucleus. The resolvent is a unit clause, i.e. a clause consisting of one literal only. The definition of UR-resolution allows for the resolvent to be either positive or negative. A UR-resolution proof of (5) using the clause set in [Fig. 2] is given in [Fig. 4].

$[UR1]$	$\neg Ancestor(John, Peter)$	[Resolvent of (6), (9) and (10)]
$[UR2]$	$\neg Parent(John, Peter)$	[Resolvent of $[UR1]$ and (8)]
$[UR3]$	$\square$	[Resolvent of $[UR2]$ and (7)]

Figure 4: A UR-resolution proof

In the next two sections we discuss hyperresolution, a generalization of UR-resolution.

2.2.2 Positive Hyperresolution

The object of an application of a positive hyperresolution step is to produce a *positive* clause from a set of clauses, one of which contains at least one negative literal, while the remaining are positive clauses.

Formally, positive hyperresolution is that inference rule that applies to a set of clauses, one of which must be negative or mixed (called the nucleus), the remaining (called satellites) must be positive clauses and their number must be equal to the number of negative literals in the nucleus. Every negative literal in the nucleus is resolved with exactly one satellite. These restrictions ensure that any such resolvent will be a positive clause. Positive hyperresolution is furthermore refutation complete for arbitrary unsatisfiable clause sets.

A positive hyperresolution proof of (5) using the clause set in [Fig. 2] is given in [Fig. 5].

$[PH1]$	$Ancestor(John, Peter)$	[Resolvent of (7) and (8)]
$[PH2]$	$Ancestor(Dennis, Peter)$	[Resolvent of $[PH1]$, (6) and (9)]
$[PH3]$	$\square$	[Resolvent of $[PH2]$ and (10)]

Figure 5: A positive hyperresolution proof

2.2.3 Negative Hyperresolution

Negative hyperresolution is like positive hyperresolution, but the signs of the nucleus and satellites are reversed, ensuring that all resolvents are *negative* clauses. Like its positive counterpart, negative hyperresolution is refutation complete for any given unsatisfiable set of clauses.

A negative hyperresolution proof of (5) using the clause set in [Fig. 2] produces the same refutation as for binary resolution in [Fig. 3].

2.2.4 Binary Paramodulation

Paramodulation is an extension of unification that allows one to make equality substitutions directly at the level of terms in a clause. Paramodulation is applied to two clauses and one of the clauses, called the *from* clause, is required to contain a positive equality literal. The other clause, called the *into* clause, is the one into which the equality substitution is made. The resultant clause is called a paramodulant. Paramodulation can yield clauses that might not otherwise be obtained by ordinary equality substitution. For example, from the two clauses $EQUAL(sum(0,x),x)$ and $EQUAL(sum(y,minus(y)),0)$ paramodulating from the first clause into the second yields $EQUAL(minus(0),0)$ as a valid paramodulant [Wos et al. 1992].

2.2.5 Factoring

Factoring is the process of reducing repetition in a clause by unifying two or more literals in the same clause. For example, $P(a)$ is a factor (i.e. a simplification) of the needlessly repetitive clause $P(x) \vee P(a)$.

2.3 Strategies

2.3.1 Weighting

By assigning different weights to different symbols, the reasoning algorithm may be guided towards resolving lighter clauses before heavier, in an attempt to generate lighter resolvents that are closer to the empty clause. The default weight of a clause is the sum of the number of individual constants, variables, function symbols and predicate symbols. Connectives like $\neg$, $\wedge$, $\vee$, etc. are not counted. Since it is possible to stipulate a maximum weight for clauses, the complexity of the clauses generated by the inference process can be controlled. Thus choosing a sufficiently high weight for variables or certain terms effectively combats the combinatorial explosion brought about by the generation of too many irrelevant paramodulants (i.e. by the uncontrolled substitution of equals for equals) but may in turn lead to an incomplete search for a proof.

2.3.2 Set-of-support

The set-of-support (sos) strategy involves partitioning the set S of clauses. Into one subset, the sos T, we place clauses regarded as being of particular importance, such as the denial of the goal G and any special hypotheses that characterise the specific instance of the problem at hand rather than expressing generic features of the type of problem. The sos then guides the reasoning process, since the reasoner is required not to apply a resolution step unless one of the clauses involved has support. A clause has support if either it is one of the original clauses that was placed in T or else it was obtained by applying resolution to clauses at least one of which had support.

When S is divided into the sos T and the *usable list* $S - T$, this should be done in such a way that we are confident the usable list $S - T$ is satisfiable. The set-of-support strategy then makes sense, as it prevents the reasoner from simply expanding a satisfiable set of clauses without hope of termination by discovery of a contradiction $\square$. Since the sos is involved at every step, and the denial of the goal is placed in the sos, the strategy ensures that the reasoner's attack on the problem is goal-directed.

The set-of-support strategy is refutation complete relative to binary resolution with factoring, but is not refutation complete relative to hyperresolution. We observe this in [Fig. 3] and [Fig. 5] respectively. For example, in [Fig. 3] it was possible to derive $\square$ starting with the negation of the theorem (i.e. clause (10)), but in [Fig. 5] we had to start with two other clauses. Nevertheless, the use of a sos very often leads to a quick proof, as opposed to no real time proof at all.

3 The OTTER Theorem Prover

OTTER (Organized Techniques for Theorem Proving and Effective Research) is a resolution-based theorem-proving program for first-order logic with equality and includes the inference rules binary resolution, hyperresolution (both positive and negative versions), UR-resolution and binary paramodulation. OTTER was written and is distributed by William McCune at the Argonne National Laboratory in Illinois [McCune 2003]. At the time of writing the latest version of OTTER is available at: `http://www-unix.mcs.anl.gov/AR/otter`.

OTTER can convert first-order formulae into sets of clauses, which constitute the input to the resolution algorithm. Naturally, OTTER does not accept formulae in the highly evolved notation of set theory so the user has to rewrite set-theoretic formulae in terms of a weaker first-order language having the relevant relations and functions as predicate symbols and function symbols in its alphabet. Some other capabilities of OTTER are factoring and weighting. An

OTTER program is divided into several sections, of which the most important are the *usable list* and the *set-of-support* (*sos*) *list*.

Next we introduce a number of heuristics for reasoning about set theory. These heuristics were developed to address the problems discussed in [Section 1.1].

4 Set-Theoretic Reasoning Heuristics

In analyzing the efficiency of an algorithm, a theoretical approach involving the calculation of time and space complexities [Baase and Van Gelder 2000] could be followed, or one can follow an empirical approach. The theoretical approach is well suited to algorithms in which, say, it is possible to predict roughly how often an assignment statement inside a loop will be executed. However, in the case of automated reasoning algorithms, inefficiency is primarily caused by thrashing, in other words by the exploration of the consequences of irrelevant information. The extent of such thrashing is difficult to predict theoretically. Therefore, resolution-based theorem provers are currently evaluated on empirical grounds [Bundy 1999]. The success of the reasoner is measured in terms of how many different benchmark problems it can solve as well as how quickly a proof (if any) is found [Pelletier 1986].

The heuristics detailed in this section were developed empirically through observing the behaviour of, as well as studying the format of, the clauses generated by the reasoner during a proof attempt, and preliminary reports presenting suggestive evidence of their potential usefulness may be found in [van der Poll 2000] as well as [van der Poll and Labuschagne 1999].

Heuristic #1 – *Use weights*: If the sos consists of the negation of an equality literal then assign a weight of $\mathtt{n}$, for $\mathtt{n} \in \{3, 4, 5\}$ to the variables in the input to the reasoner. Note that the default weight of a variable is 1.

Heuristic #2 – *Apply extensionality*: Apply the ZF axiom of extensionality [Enderton 1977]

$$(\forall A)(\forall B)[(\forall x)(x \in A \leftrightarrow x \in B) \rightarrow (A = B)]$$

to replace an equality in the sos by the biconditional criterion under which two sets are equal, i.e. that their elements are the same.

Heuristic #3 – *Eschew nested functors*: Avoid, if possible, the use of *nested functors* in definitions. Terms built up with the aid of function symbols (called functors) are more complex, potentially leading to difficulties with unification of terms, especially when these functors are nested inside other structures.

***Heuristic* #4** – *Divide and Conquer*: Perform two separate subset proofs whenever the problem at hand requires proving the equality of two sets. An equality in the sos implies (via Extensionality) an 'if and only if'. Hence a specifier may opt for two proofs, one for the only-if part and another for the if part. Note that this heuristic may serve as an alternative to heuristic #2 above.

***Heuristic* #5** – *Eschew multivariate functors*: Make terms in sets as simple as possible — either not involving functors at all, or else involving functors with the minimum number of argument positions taken up by variables. (The more variables occur as arguments to a functor, the greater the likelihood of thrashing caused by the unification of these variables with other terms.)

***Heuristic* #6** – *Separate out intermediate structures*: Avoid complex functor expressions by using an indirect definition for an internal structure whenever this appears less likely to produce complex functor expressions than the direct definition. In practice we simply give a name to a complex structure that is nested inside another structure and then define the inner structure externally on its own, instead of unfolding its definition directly inside the enclosing structure.

***Heuristic* #7** – *Make element structure explicit*: Define the elements of relations and functions directly in terms of ordered pairs or ordered n-tuples whenever the tuples need to be opened to find a proof. An ordered n-tuple is an example of a functor and projecting out the coordinates of the tuple often avoids the various functor problems listed above.

***Heuristic* #8** – *Search-guiding*: Generate and use half definitions, via the technique of resolution by inspection, for biconditional formulae in the usable list whenever the sos consists of a conditional formula or a single literal. A half definition is an implication (e.g. only-if) as opposed to an if and only if definition. Through inspection it is often possible to trace the initial steps a reasoner would perform starting with the conditional formula in the sos. Hence it is possible to predict which half of some definitions in the usable list would probably be needed and which 'other halves' are redundant.

***Heuristic* #9** – *Inference rule selection*: Use negative hyperresolution instead of positive hyperresolution if the sos consists of a single negative literal or whenever the combined use of positive hyperresolution and unit-resulting resolution rapidly makes the sos empty and thereby resulting in no proof. If no rapid proof results, try binary resolution. Recall that both forms of hyperresolution are capable of generating homogeneous clauses only (i.e. just positive or just negative but not mixed). Although many researches warn against the use of binary resolution [Quaife 1992] we found such rule to be occasionally useful (see [Section 6.1] below).

***Heuristic* #10** – *Resonance*: Attempt to give corresponding terms in formulae a syntactically similar structure to aid the resolution process [Wos 1995]. Not only does this apply to terms just in the usable list, but also to a term in the sos and a corresponding term in the usable list.

In the next section we introduce the enterprise information system that is used in this paper to illustrate the utility of the above heuristics.

5 A Multi-level Marketing Enterprise

A multi-level marketing (MLM) enterprise [GNLD 1997] markets consumable products through people as follows: A new distributor registers with the enterprise either as a direct associate of the company, or under an existing distributor called an *upline*. Both the upline (also called the sponsor) and the new distributor (now called a downline) then go on to each sponsor more new distributors, and so on. In this way a network of distributors is built. Hence, a MLM structure may be modelled by forests and trees [Scheurer 1994].

Distributors buy products from the company and every product carries a *point value* (pv) as well as a *business value* (bv). The business value matches the price of a product. Both the points and the business values are accumulated per distributor throughout a month. At the end of the month the total business value in the network for each distributor is calculated, and the distributor is paid (in the appropriate currency) a certain percentage (determined by the pv) of the total business value for his or her group. This is called a bonus.

A small MLM network is shown in [Fig. 6]. Distributors $A1$, $A2$ and $A3$ associated with the company directly are called the roots of the forest (or network in MLM terms).

Our MLM enterprise is specified in Z and the abstract state is (where the operation \ represents set-theoretic difference and dom and ran denote the domain and range of a relation respectively):

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{MLM}} \\
\hline
\textit{known} : \mathbb{P}\, \textit{ID} \\
\textit{NRoots} : \mathbb{P}\, \textit{ID} \\
\textit{NUplines} : \textit{ID} \leftrightarrow \textit{ID} \\
\textit{NDist} : \textit{ID} \rightarrow\!\!\!\!\!\!+\;\, \textit{Name} \times \textit{Address} \times \textit{PV} \times \textit{BV} \times \textit{Bonus} \\
\hline
\textit{known} = \mathrm{dom}\, \textit{NDist} \\
\mathrm{dom}\, \textit{NUplines} \cup \mathrm{ran}\, \textit{NUplines} \subseteq \textit{known} \\
\textit{NRoots} = \textit{known} \setminus \mathrm{ran}\, \textit{NUplines} \\
\textit{Inj}(\textit{NUplines}) \\
\hline
\end{array}
$$

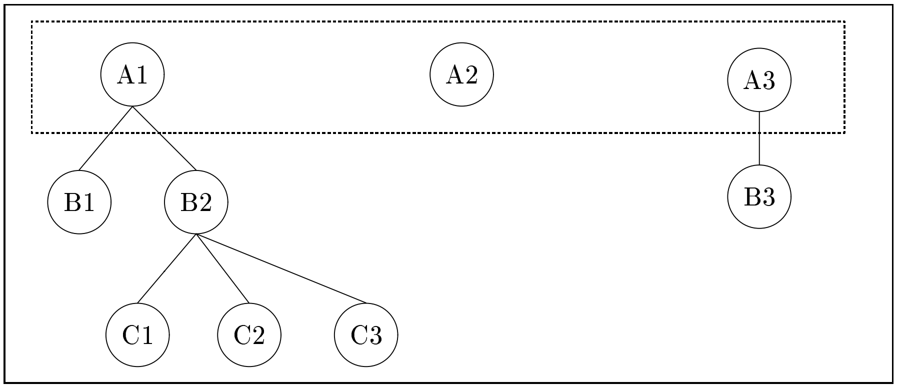

Figure 6: An example network

The set *known* contains the identity codes of all distributors in the system. *NRoots* represents all root distributors. The relation *NUplines* represents the network of distributors while the function *NDist* represents a mapping from a unique identity code to the particulars for that distributor. *NDist* is not necessarily injective since two (or more) distributors may have the same particulars (i.e. name, address, etc.). Every distributor has at most one upline, captured by the following general definition of injectivity:

$$(\forall R)(Inj(R) \leftrightarrow (\forall i)(\forall j)(\forall k)(((i,k) \in R \wedge (j,k) \in R) \longrightarrow (i = j))) \quad (11)$$

The following operation registers a new distributor $p!$ below an existing one, $q?$:

$$\begin{array}{|l}
\textit{Register_with_upline} \\
\hline
\Delta MLM \\
p!, q? : ID \\
name? : Name;\ \ addr? : Address \\
\hline
p! \notin known \wedge q? \in known \\
known' = known \cup \{p!\} \\
NUplines' = NUplines \cup \{q? \mapsto p!\} \\
NDist' = NDist \cup \{p! \mapsto (name?, addr?, 0, 0.0, 0.0)\} \\
\hline
\end{array}$$

A new identity code $p!$ is generated by the system and the new distributor is linked to $q?$ in $NUplines'$. Initial product information pertaining to $p!$ is reflected in the personal pv being 0 and both the business value and potential bonus equal to the real value 0.0.

An order placed by a distributor is given by:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Order}} \\
\Delta MLM \\
id? : ID;\ pv? : PV;\ bv? : BV \\
\hline
id? \in known \\
(\exists\, pv : PV;\ bv : BV \bullet \\
\quad pv = third(NDist(id?)) + pv? \wedge \\
\quad bv = fourth(NDist(id?)) + bv? \wedge \\
\quad NDist' = NDist \oplus \{id? \mapsto \\
\qquad (first(NDist(id?)), second(NDist(id?)), \\
\qquad\ pv, bv, fifth(NDist(id?)))\}) \\
\hline
\end{array}
$$

The functions *first*, *second*, etc. project out an element at the appropriate position in the tuple. $NDist'$ is obtained from *NDist* by replacing the tuple with first coordinate $id?$ as specified above. Many additional operations may be defined on the state but are beyond the scope of this paper. The interested reader is referred to [van der Poll and Kotzé 2003].

Next we show how some of the heuristics introduced in [Section 4] may be used to successfully discharge two proof obligations that arise from the *MLM* specification where otherwise proofs are not easily arrived at. All the proofs were done on a Pentium III with a clock speed of 600MHz and 32MB RAM. The operating system was Red Hat Linux Release 9.

6 Reasoning about the Specification

6.1 Showing $NRoots = known' \setminus \operatorname{ran} NUplines'$

In Z system operations are implicitly assumed to preserve the state invariant [Diller and Docherty 1994], hence proving that an operation preserves an invariant is not deemed necessary in Z. Nevertheless, it is a good idea to check that an operation does not incur unpredictable results as far as the invariant is concerned. In fact, in systems like the B method [Abrial 1996] ensuring that an operation preserves the invariant is considered a critical proof obligation.

A specifier may, therefore, decide to verify the following as a postcondition of schema *Register_with_upline* (Note that $NRoots' = NRoots$):

$$NRoots = known' \setminus \operatorname{ran} NUplines' \tag{12}$$

In effect the above predicate claims that the set of root elements is still equal to the new set of all distributors ($known'$) minus the new set of all downline distributors ($\operatorname{ran} NUplines'$). If we define $NewRoots = known' \setminus \operatorname{ran} NUplines'$ and pose the negation of the following equality in the sos

$$NRoots = NewRoots \tag{13}$$

then OTTER finds no proof in 20 minutes using a weight of 3, 4 or 5 and either positive or negative hyperresolution. Since neither form of hyperresolution is able to find a proof, we apply our *inference rule selection* heuristic and resort to binary resolution but still using our weight template. Now the reasoner finds a proof after just *0.66 seconds*.

Why does the reasoner fail to find a proof for (13) using hyperresolution? The sos format (13) requires the axiom of Extensionality [Enderton 1977]

$$(\forall A)(\forall B)[(\forall x)(x \in A \leftrightarrow x \in B) \rightarrow (A = B)] \tag{14}$$

to 'open' the equality in terms of elementhood to (loosely speaking) arrive at the following form of (13):

$$(\forall x)(x \in \mathit{NRoots} \leftrightarrow x \in \mathit{NewRoots}) \tag{15}$$

The negation of (15) clausifies into:

$$\$c1 \in \mathit{NRoots} \vee \$c1 \in \mathit{NewRoots} \tag{16}$$

$$\$c1 \notin \mathit{NRoots} \vee \$c1 \notin \mathit{NewRoots} \tag{17}$$

The invariant $\mathit{NRoots} = \mathit{known} \setminus \operatorname{ran} \mathit{NUplines}$ in the MLM state is unfolded in first-order notation as

$$(\forall x)(x \in \mathit{NRoots} \leftrightarrow x \in \mathit{known} \wedge x \notin \operatorname{ran}(\mathit{NUplines}))$$

and it clausifies into

$$x \notin \mathit{NRoots} \vee x \in \mathit{known} \tag{18}$$

$$x \notin \mathit{NRoots} \vee x \notin \operatorname{ran}(\mathit{NUplines}) \tag{19}$$

$$x \in \mathit{NRoots} \vee x \notin \mathit{known} \vee x \in \operatorname{ran}(\mathit{NUplines}) \tag{20}$$

Note that positive hyperresolvents can be generated by resolving the sos clause (16) with (18), but the sos clause (17) is not capable of generating a positive hyperresolvent with any of the clauses (18) - (20). The result is that a proof attempt using positive hyperresolution cannot start off correctly. A similar problem occurs with negative hyperresolution. Binary resolution creates no such problem, since binary resolvents may be mixed.

Still with this proof attempt, suppose a specifier is initially, due to the weight clause, concerned about an incomplete search for a proof. If we omit the weight template in the above binary resolution proof then the reasoner again finds no proof in 20 minutes (as opposed to a proof in 0.66 seconds). This forms the basis for a further heuristic that may be applied to our last failed proof attempt.

In the proof of (13) we unfolded the predicate $\mathit{NUplines}' = \mathit{NUplines} \cup \{q? \mapsto p!\}$ in schema *Register_with_upline* as

$$(\forall x)(x \in \mathit{NUplines}' \leftrightarrow x \in \mathit{NUplines} \vee x \in \{(q?, p!)\}) \tag{21}$$

using the following first-order definition for a singleton:

$$(\forall\, x)(\forall\, y)(x \in \{y\} \leftrightarrow x = y) \tag{22}$$

Together with definition (21), we also needed the following fact about ordered pairs from [Enderton 1977]:

$$(\forall\, u)(\forall\, v)(\forall\, w)(\forall\, x)((u, v) = (w, x) \leftrightarrow ((u = w) \wedge (v = x))) \tag{23}$$

Upon studying the clauses generated by the search for a proof, we note that (22) and (23) interact to generate literals of the form $El(ORD(x, y), Sin(ORD(u, v)))$ where x, y, u and v are variables and ORD, Sin and El are the ASCII equivalents for an ordered pair, a singleton and elementhood respectively. Clearly this literal contains *nested functors*, a practice discouraged by our heuristic #3, since it, in the absence of a weight template, leads to a large number of unnecessary unifications.

If we, therefore, rewrite (21) as

$$(\forall\, x)(x \in NUplines' \leftrightarrow x \in NUplines \vee x = (q?, p!)) \tag{24}$$

and still omit the weight template, then the reasoner again finds a proof for (13), but in *3.70 seconds*. According to our *element structure* heuristic #7 we can further rewrite (24) as

$$\begin{aligned}&(\forall\, y)(\forall\, z)\\ &\qquad((y, z) \in NUplines' \leftrightarrow (y, z) \in NUplines \vee (y = q? \wedge z = p!))\end{aligned} \tag{25}$$

which cuts the execution time of 3.70 seconds down to just *0.06 seconds*.

6.2 A Proof of Cardinality

After the execution of operation *Register_with_upline* we expect the following to hold regarding the cardinality of the set $known' = known \cup \{p!\}$:

$$\#known' = \#known + 1 \tag{26}$$

We use the following two definitions of cardinality ($Card(A, n)$ denotes $\#A = n$):

$$(\forall\, A)(Card(A, 0) \leftrightarrow A = \varnothing) \tag{27}$$

$$(\forall\, A)(\forall\, n)(Card(A, n + 1) \leftrightarrow (\exists\, x)(x \in A \wedge Card(A - \{x\}, n))) \tag{28}$$

Suppose we start with the precondition $Card(known, n)$ and pose the following question in the sos:

$$-Card(known', n + 1) \tag{29}$$

The reasoner finds no proof for (29) in 30 minutes and closer investigation reveals that the term $Card(A - \{x\}, n)$ above contains nested functors (i.e. a singleton definition inside a set difference inside the functor *Card*), a practice discouraged by our *nested functor* heuristic. As a first step we unfold definition (28) as:

$$\begin{aligned}&(\forall A)(\forall n)(Card(A, n+1) \leftrightarrow \\ &\quad (\exists B)(\exists x)(x \in A \wedge Card(B, n) \wedge (\forall y)(y \in B \leftrightarrow y \in A \wedge y \notin \{x\})))\end{aligned} \tag{30}$$

With this unfolding a proof still eludes us, but since such unfolding is in turn against the recommendation put forward by the *intermediate structure* heuristic we replace the definition of set B in (30) with

$$(\forall y)(y \in B \leftrightarrow y \in DIFF(A, \{x\})) \tag{31}$$

where x is still existentially quantified as in (30) and *DIFF* is defined by:

$$(\forall x)(x \in DIFF(known', \{p\}) \leftrightarrow x \in known' \wedge x \notin \{p\}) \tag{32}$$

With these definitions OTTER finds a short proof for (29) in just *0.21 seconds*. The input to the theorem prover for this proof attempt appears in Appendix A. Note the use of negative hyperresolution (i.e. `set(neg_hyper_res)`) as a result of the sos being a single negative literal (see Heuristic #9). Definition (32) is in line with our *multivariate functor* heuristic which advocates cutting down on the number of variables as arguments of functors. This is mainly the reason why the nested functor in this definition turns out to be harmless. For example, if we rewrite (32) above as

$$(\forall A)(\forall p)(\forall x)(x \in DIFF(A, \{p\}) \leftrightarrow x \in A \wedge x \notin \{p\}) \tag{33}$$

then the reasoner again finds no proof in 20 minutes. Replacing one of the variables (say p) in (33) above with a constant again helps OTTER to find a proof in *8.59 seconds*.

We may also fit our *search-guiding* heuristic onto the last definition of *Card* above. The technique of resolution by inspection reveals that the sos question (29) needs just the 'if-direction' of (30). If we make such adjustments we can even find a proof using (33), but in *1 minute 27 seconds*.

7 Summary and Future Work

This paper illustrated how set-theoretic reasoning heuristics may be used to discharge two proof obligations that arise from the formal specification in Z of a multi-level marketing enterprise. It is evident that the same proof obligation may be discharged in more than one way. This is significant, since if a particular heuristic fails to deliver then another one may be applied instead.

A number of challenges remain: The gross bonus earned by a distributor is normally calculated as a percentage of the total business value generated by the distributor and all downlines in the tree for that distributor. However, the net bonus for a distributor is calculated by subtracting the gross bonus for each first-level downline of the distributor from the gross bonus of the upline distributor. This generates a very important proof obligation in a MLM enterprise, namely, to show that the sum of the gross bonuses allocated to first-level downline distributors is less than or equal to the gross bonus allocated to their immediate upline.

Formally, consider a distributor, say q, with a number of first-level downlines, represented by a set, say D, such that:

$$(\forall\, d)(d \in D \leftrightarrow (q, d) \in NUplines)$$

Suppose $Bonus_i$ represents the gross bonus allocated to distributor i. The proof obligation is to show that

$$Bonus_q \; - \sum_{(q,d) \in NUplines} Bonus_d \; \geq \; 0$$

The reasoner fails to find a proof of the above property using any of our heuristics, hence more work is needed in the area of arithmetic. Of course, one way to solve this problem in one fell swoop in Z is to extend the state invariant in schema MLM, requiring that the value of every distributor's gross bonus be non-negative and greater than or equal to the sum of the gross bonuses of that distributor's immediate downlines. Nevertheless, any sensible specifier will check (using some proof mechanism) that all relevant operations preserve this property.

Further empirical work could proceed along a number of avenues: One can investigate to what extent the heuristics presented in [Section 4] are useful when using another automated resolution-based reasoner, e.g. Vampire [Voronkov 1995] to reason about the properties of a real-life enterprise such as the one presented in this paper. Alternatively, the feasibility of set-based model-checking techniques [Bérard et al. 2001], using for example ProB [Leuschel and Butler 2003] could be investigated. We also need to scale up the proofs reported on in this paper to industrial sized proof attempts and we anticipate that additional heuristics would have to be developed to address the challenges that may unfold from such experiments.

References

[Abrial 1996] Abrial, J.-R.:, "The **B** Book: Assigning Programs to Meanings", Cambridge University Press, August (1996).

[Baase and Van Gelder 2000] Baase, S., Van Gelder, A.: "Computer Algorithms: Introduction to Design & Analysis", Addison-Wesley (2000).

[Bailin and Barker-Plummer 1993] Bailin, S.C., Barker-Plummer, D.: "Z-match: An Inference Rule for Incrementally Elaborating Set Instantiations"; JAR (Journal of Automated Reasoning), 11, 3, December (1993) 391 - 428.

[Bérard et al. 2001] Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, Ph., McKenzie, P.: "Systems and Software Verification: Model-Checking Techniques and Tools", Springer-Verlag, Berlin Heidelberg, Germany (2001)

[Bowen 2001] Bowen, J.P.: Z: A Formal Specification Notation. In Frappier, M., Habrias, H., eds.: "Software Specification Methods: An Overview Using a Case Study", FACIT. Springer-Verlag, (2001), 3 - 19.

[Boyer et al. 1986] Boyer, R., Lusk, E., McCune, W., Overbeek, R., Stickel, M., Wos, L.: "Set Theory in First-Order Logic: Clauses for Gödel's Axioms"; JAR (Journal of Automated Reasoning), 2, 3, September (1986), 287 - 327.

[Bundy 1999] Bundy, A.: "A Survey of Automated Deduction", Division of Informatics, University of Edinburgh, EDI-INF-RR-0001, April (1999).

[Date 1995] Date, C.J.: "An Introduction to Database Systems", Addison-Wesley, 6th edition, The System Programming Series (1995).

[Diller and Docherty 1994] Diller, A., Docherty, R.: "**Z** and Abstract Machine Notation: A Comparison"; **Z** User Workshop: Proceedings of the Eighth **Z** User Meeting. Edited by J.P. Bowen and J.A. Hall, Workshops in Computing, Springer-Verlag, Cambridge, U.K., June (1994), 250 - 263.

[Enderton 1977] Enderton, H.B.: "Elements of Set Theory"; Academic Press, Inc. (1977).

[Fagan 1976] Fagan, M.E. "Design and Code inspections to Reduce Errors in Program Development"; ISJ (IBM Systems Journal), 15, 3 (1976), 182 - 211.

[GNLD 1997] "Distributor Business Guide: The Business Plan", Golden Neo-Life Diamite International, July (1997).

[Leitsch 1997] Leitsch, A.: "The Resolution Calculus", Springer (1997).

[Leuschel and Butler 2003] Leuschel, M., Butler, M.: "ProB: A Model Checker for B"; FME 2003: Formal Methods, edited by Araki, K., Gnesi, S. and Mandrioli, D. Springer-Verlag, LNCS 2805, (2003), 855 - 874.

[McCune 2003] McCune, W.W.: "OTTER 3.3 Reference Manual", Argonne National Laboratory, Argonne, Illinois, ANL/MCS-TM-263, August (2003).

[Mokkedem et al. 2000] Mokkedem, A., Hosabettu, R., Jones, M.M., Gopalakrishnan, G.: "Formalization and Proof of a Solution to the PCI 2.1 Bus Transaction Ordering Problem"; Formal Methods in Systems Design, 16, 1, January (2000), 93 - 119.

[Pelletier 1986] Pelletier, F.J.: "Problems for Testing Automatic Theorem Provers"; JAR (Journal of Automated Reasoning), 2 (1986) 191 - 216.

[Potter et al. 1996] Potter, B., Sinclair, J., Till, D.:, "An Introduction to Formal Specification and **Z**", Prentice-Hall, 2nd edition (1996).

[Quaife 1992] Quaife, A.: "Automated Development of Fundamental Mathematical Theories", Kluwer Academic Publishers, Automated Reasoning Series (1992).

[Robinson 1965a] Robinson J.A.: "A Machine-Oriented Logic Based on the Resolution Principle"; JACM (Journal of the Association for Computing Machinery), 12, 1, January (1965), 23 - 41.

[Robinson 1965b] Robinson J.A.: "Automatic Deduction with Hyperresolution"; IJCM (International Journal of Computer Mathematics), 1 (1965), 227 - 234.

[Scheurer 1994] Scheurer, T.: "Foundations of Computing : System Development with Set Theory and Logic", Addison-Wesley, International Computer Science Series (1994).

[Spivey 1992] Spivey, J.M.: "The **Z** Notation: A Reference Manual", 2nd edition, Prentice-Hall, London (1992).

[van der Poll and Labuschagne 1999] van der Poll, J.A., Labuschagne, W.A.: "Heuristics for Resolution-Based Set-Theoretic Proofs"; SACJ (South African Computer Journal), Issue 23, July (1999), 3 - 17.

[van der Poll 2000] van der Poll, J.A.: "Automated Support for Set-Theoretic Specifications", PhD Thesis, UNISA (University of South Africa), June (2000).

[van der Poll and Kotzé 2003] van der Poll, J.A., Kotzé, P.: "A Multi-level Marketing Case Study: Specifying Forests and Trees in Z"; SACJ (South African Computer Journal), Issue 30, June (2003), 17 - 28.

[Voronkov 1995] Voronkov, A.: "The Anatomy of Vampire: Implementing Bottom-Up Procedures with Code Trees"; JAR (Journal of Automated Reasoning), 15, 2 (1995), 237 - 265.

[Woodcock and Davies 1996] Woodcock, J., Davies, J.: "Using **Z**: Specification, Refinement, and Proof", Prentice-Hall, London (1996).

[Wos 1988] Wos, L.: *Research problem #8: An Inference rule for Set theory* in "Automated Reasoning: 33 Basic Research problems", Prentice-Hall, (1988), 137 - 138.

[Wos et al. 1992] Wos, L., Overbeek, R., Lusk, E., Boyle, J.: "Automated Reasoning: Introduction and Applications", McGraw-Hill, 2nd edition (1992).

[Wos 1995] Wos, L.:, "The Resonance Strategy"; Computers and Mathematics with Applications, (Special issue on Automated Reasoning), 29, 2, February (1995), 133 - 178.

[Wos 1998] Wos, L.: "Programs that Offer Fast, Flawless, Logical Reasoning"; CACM (Communications of the ACM), 41, 6, June (1998), 87 - 95.

[Wos and Pieper 1999] Wos, L., Pieper, G.W.: "A Fascinating Country in the World of Computing: Your Guide to Automated Reasoning", World Scientific Publishing Company (1999).

Appendix A

```
%% ----------------------------------------
%% IF -El(p,known) & (known' = known u {p})
%% THEN #known' = #known + 1
%% ----------------------------------------
set(neg_hyper_res). %% Inferences
set(ur_res).
set(factor). %% Factoring

set(para_from). %% Paramodulation settings
set(para_into).

set(order_eq). %% Ordering of equalities

%% Additional settings.
%% --------------------
set(process_input).
clear(para_from_right).
clear(para_into_right).
set(dynamic_demod_all).
set(back_demod).

assign(max_seconds,1200).      %% Allow reasoner 20 minutes.
```

```
%% Enable conciseness.
%% -------------------
clear(print_given).
clear(print_kept).
clear(print_back_sub).

weight_list(pick_and_purge).
weight(x,3). %% Weight of variables = 3.
end_of_list.

formula_list(usable).

%% Reflexivity.
%% ------------
(all x (x = x)).

%% Definition of Empty set.
%% ------------------------
-(exists x El(x,Empty)).

%% Definition of a Singleton.
%% --------------------------
(all x y ( El(x,Sin(y)) <-> (x = y) )).

%% Resonance Definition of DIFFerence.
%% -----------------------------------
(all x
 (El(x,DIFF(known1,Sin(p))) <-> (El(x,known1) & -El(x,Sin(p))))).

%% Base traditional definition:
%% ----------------------------
(all A ( Card(A,0) <-> (A = Empty) )).

%% Card after the application of 2 heuristics.
%% -------------------------------------------
(all A n
 ( Card(A,$SUM(n,1)) <->
    (exists B x
      ( El(x,A) & Card(B,n) &
        (all y (El(y,B) <-> El(y,DIFF(A,Sin(x))))) )) )).
```

```
%% Pre- and Postconditions.
%% ------------------------
(-El(p,known) & El(p,known1)).
Card(known,k).

%% known' = known u p.
%% --------------------
(all x ( El(x,known1) <-> (El(x,known) | El(x,Sin(p))) )).

end_of_list.

formula_list(sos).

%% #known' = k + 1.
%% ----------------
-Card(known1,$SUM(k,1)).

end_of_list.
```