SRI International

October 8, 1998

Final Report

Cooperation among Theorem Provers

Prepared for:

Dr. Michael Lowry
NASA Ames Research Center
M/S 269-2
Moffett Field, CA 94035-1000

Prepared by:

Dr. Richard J. Waldinger
Principal Scientist
Artificial Intelligence Center

The following individuals are authorized to conduct negotiations on behalf of SRI International:

Technical matters:
Dr. Richard J. Waldinger, Principal Scientist
(650) 859-2216 e-mail: waldinger@ai.sri.com Fax: (650) 859-3735

Contractual Matters:
Donna Linné, Sr. Contracts Administrator
(650) 859-2004 e-mail: linne@sri.com Fax: (650) 859-6171

1 Background

This is a final report on the "Cooperation among Theorem Provers" project, which supports NASA's PECSEE (Persistent Cognizant Software Engineering Environment) effort and complements the Kestrel Institute project "Inference System Integration via Logic Morphisms". The ultimate purpose of the project is to develop a superior logical inference mechanism by combining the diverse abilities of multiple cooperating theorem provers.

In many years of research, a number of powerful theorem-proving systems have arisen with differing capabilities and strengths. Resolution theorem provers (such as Kestrel's KITP or SRI's SNARK) deal with first-order logic with equality but not the principle of mathematical induction. The Boyer-Moore theorem prover excels at proof by induction but cannot deal with full first-order logic. Both are highly automated but cannot accept user guidance easily. The PVS system (from SRI) in only automatic within decidable theories, but it has well-designed interactive capabilities; furthermore, it includes higher-order logic, not just first-order logic. The NuPRL system from Cornell University and the STeP system from Stanford University have facilities for constructive logic and temporal logic, respectively—both are interactive.

It is often suggested—for example, in the anonymous "QED Manifesto"—that we should pool the resources of all these theorem provers into a single system, so that the strengths of one can compensate for the weaknesses of others, and so that effort will not be duplicated. However, there is no straightforward way of doing this, because each system relies on its own language and logic for its success. Thus, SNARK uses ordinary first-order logic with equality, PVS uses higher-order logic, and NuPRL uses constructive logic.

The purpose of this project, and the companion project at Kestrel, has been to use the category-theoretic notion of logic morphism to combine systems with different logics and languages. Kestrel's SPECWARE system has been the vehicle for the implementation.

2 SPECWARE

Kestrel's SPECWARE is a category-theory-based software development environment. It exploits category theory to capture two fundamental notions: the refinement of specifications into code and the composition of software components.

The fundamental objects of SPECWARE are specifications and the morphisms between them. SPECWARE uses the word "specification" in an uncommonly general way—it includes theories and code as well as high-level descriptions of software. A morphism is a kind of mapping between specifications that indicates how one specification—the source—can be viewed as a subtheory of another specification—the target. A morphism identifies each symbol of the source theory with a corresponding symbol in the target theory, in such a way that the theorems of the source theory are mapped into theorems of the target theory. For SPECWARE to verify a morphism, it must prove that each axiom of the source theory is mapped into a theorem of the target theory. For this purpose, it contains its own theorem prover, KITP.

To capture the notion of software refinement, SPECWARE uses the category-theoretic concept of an interpretation. The SPECWARE notion of software composition is based on the

category-theoretic concept of a colimit. Both of these concepts are based on morphisms. For instance, a refinement is a morphism from a source theory into a mediator—not the target theory itself but an extension of the target in which new concepts have been defined.

The essence of the Kestrel approach to combining two theorem provers is to build a refinement between their logical systems. However, in an ordinary refinement, both theories must have the same logical inference system, while different theorem provers are based on different logical systems. To smooth over these anomalies, Kestrel employs the notion of an inter-logic refinement, based on Meseguer's concept of a logic morphism.

Although Kestrel has formulated the principles of the inter-logic refinement, SPECWARE does not support this refinement in the way it supports the single-logic refinement. In support of this project, a new inter-logic morphism must be built between SPECWARE and each theorem prover to be connected. The first system to be connected (after Kestrel's own KITP) has been SRI's SNARK. Since the inter-logic morphism is not yet implemented, this has been done using primitives of the underlying implementation language, REFINE. This experiment will guide future development of the implementation of the inter-logic morphism within SPECWARE.

3 SNARK

SNARK is SRI's theorem prover for first-order logic with equality, based on resolution, paramodulation, and term-rewriting inference rules. SNARK has built-in associative and commutative unification algorithms, and facilities for inserting new unification algorithms—this means SNARK can deal efficiently with theories with commutative or associative operators. It also has a built-in decision procedure for temporal reasoning, using the temporal primitives of James Allen. SNARK has a sort structure, and its unification algorithms are cognizant of that structure so that terms of mismatched sorts cannot be unified. SNARK strategies can easily be altered or replaced by the SNARK user. In particular, its agenda-ordering strategy can be changed by providing a new LISP function, and SNARK can follow a user-supplied symbol ordering so that "bad" symbols will tend to be replaced by "good" symbols.

SNARK has been employed by NASA as the fundamental reasoning component of the Amphion system. It has been augmented by NASA in several ways; in particular, in Meta-Amphion, facilities have been introduced for identifying sets of axioms that can be removed and replaced by decision procedures, with dramatic improvements of efficiency.

Part of the reason for wanting to combine SNARK and SPECWARE is to use SPECWARE's notion of morphism to formalize the search for a decidable subtheory. If SPECWARE is given a library of decidable theories, we can look for morphisms from any of those theories into the theory at hand—the image of a decidable theory will be a decidable subtheory.

4 The SPECWARE-SNARK interface

SRI has collaborated with the Kestrel Institute in building a SPECWARE-SNARK interface. SRI's part of the effort has consisted of improving SNARK's own interface, working with

Kestrel personnel on the design of the SPECWARE-SNARK interface, and selecting default values for SNARK parameters when it is invoked by SPECWARE. Our ideal has been that the theorem prover should be invisible to the naive user, but that the knowledgeable user should have full access to SNARK's capabilities.

For instance, the initial parameter settings cause the search for a proof, and the proof itself, to be invisible to the user, who simply receives a report that the theorems have been proved and the morphism verified. The selection of defaults varies according to the application; for instance, in normal theorem proving, hyperresolution is employed, but if a witness is to be extracted from the proof, binary resolution is invoked instead. This is because hyperresolution is more effective in general, but is incompatible with witness-finding.

Should the proof fail, the user may change the settings to exhibit the failed proof attempt and, if necessary, alter the inference rules, parameter settings, or strategy. But this requires a more educated user.

5 A Verification Example

Let us examine an example of a verification problem. We are given the specification for a bit.

```
spec BIT is
  sort Bit
  op bit-0 : Bit
  op bit-1  : Bit
  axiom (not (equal bit-0  bit-1))
  axiom (fa (x : Bit) (or (equal x bit-0) (equal x bit-1)))
  constructors {bit-0, bit-1} construct BIT

  %-------------------------------------------
  op bit-plus : Bit, Bit -> Bit
  definition of bit-plus : Bit, Bit -> Bit is
    axiom (equal (bit-plus bit-0 bit-0) bit-0)
    axiom (equal (bit-plus bit-0 bit-1) bit-1)
    axiom (equal (bit-plus bit-1 bit-0) bit-1)
    axiom (equal (bit-plus bit-1 bit-1) bit-0)
    end-definition

  %-------------------------------------------
  op bit-carry : Bit, Bit -> Bit
  definition of bit-carry is
    axiom (equal (bit-carry bit-0 bit-0) bit-0)
    axiom (equal (bit-carry bit-0 bit-1) bit-0)
    axiom (equal (bit-carry bit-1 bit-0) bit-0)

    axiom (equal (bit-carry bit-1 bit-1) bit-1)
```

```
  end-definition

  op bit-and : Bit, Bit -> Bit
  definition of bit-and is
   axiom (equal (bit-and bit-0 bit-0) bit-0)
   axiom (equal (bit-and bit-0 bit-1) bit-0)
   axiom (equal (bit-and bit-1 bit-0) bit-0)
   axiom (equal (bit-and bit-1 bit-1) bit-1)
  end-definition

  theorem bit-and-carry is (equal (bit-and b1 b2) (bit-carry b1 b2))

  theorem bit-and-carry-iff is
     (iff (equal (bit-and b1 b2) bit-1)
          (equal (bit-carry b1 b2) bit-1))

  theorem bit-and is
     (iff  (equal (bit-and b1 b2) bit-1)
    (and (equal b1 bit-1) (equal b2 bit-1)))

  <Some operations and theorems omitted.>
 end-spec
```

In other words, there are two bit constants, `bit-0` and `bit-1`, each of sort `Bit`. The first axiom asserts that these two bits are distinct. The second asserts that every bit is either `bit-0` or `bit-1`. The `constructors` statement amounts to an induction principle, which states that to prove a property for all bits, it suffices to prove that it holds for `bit-0` and `bit-1`. This is a degenerate version of induction, in which there are two base cases and no induction step. (Actually the second axiom follows from this induction principle, and so could have been stated as a theorem.)

Then comes the definition of several functions on bits. The function `bit-plus` is binary addition of bits; for example,

```
axiom (equal (bit-plus bit-1 bit-1) bit-0)
```

says that the result of adding `bit-1` to itself is `bit-0`.

The function `bit-carry` gives the bit that is carried when two bits are added. In fact, this bit is `bit-1` when `bit-1` is added to itself; in all other cases, it is `bit-0`.

The function `bit-and` gives the logical conjunction of two bits, regarded as truth values. Thus, the conjunction of `bit-0` and `bit-1` is `bit-0`, and so on.

A sequence of theorems about bit addition and other functions is included in the specification. The first states that the bit carry function is actually identical to the logical conjunction of bits:

```
theorem bit-and-carry is (equal (bit-and b1 b2) (bit-carry b1 b2))
```

Let us follow the interaction that allows this theorem to be proved.

First we inform SNARK (by setting its focus to verification) that only theorem proving is required, no witness generation. This is done by selecting an item from a menu. We do not change the default settings—hyperresolution, paramodulation, recursive-path ordering (the term-ordering strategy), etc. for a verification proof. Menus for SNARK feature selection do not exist in pure SNARK; they were implemented as part of the interface, because the SPECWARE user expects analogous interfaces for SNARK and KITP.

We customize the interface; in particular, we elect to choose which conjectures from the specification are to be proven; otherwise, the system will attempt to prove all of them. We also elect to chose proof options for each batch of conjectures—this will enable us to decide what form of induction principle to use, and to set certain strategic controls if we choose to. We indicate how detailed we would like the trace of the proof to appear.

In setting the SPECWARE focus, we choose to verify the specification BIT. We elect to do an induction proof, by induction on b1 over {bit-1,bit-0}—this is one of two options offered. After this, the proof is automatic. In the trace below, which shows the proof of one of the two base cases, we show more than the naive user would see—normally, one would only learn that the conjecture had been verified.

```
.> ;;; Verify conjecture
;;;   fa(b2: Bit, b1: Bit) bit-and(b1, b2) = bit-carry(b1, b2)
;;;   by induction on b1 over {bit-1,bit-0}

Warning: Setting *PRINT-PRETTY* to NIL.

gc: done

The current SNARK option values are
   use-hyperresolution = T
   use-paramodulation = T
   use-factoring = T
   use-term-ordering = :RECURSIVE-PATH
   use-replacement-resolution-with-x=x = T
   agenda-length-before-simplification-limit = 10000
   agenda-length-limit = 3000
   agenda-ordering-function = ROW-WEIGHT+DEPTH
   pruning-tests = (ROW-WEIGHT-ABOVE-LIMIT-P)
   pruning-tests-before-simplification =
     (ROW-WEIGHT-BEFORE-SIMPLIFICATION-ABOVE-LIMIT-P)
   use-clausification = T
   use-and-splitting = T


Refutation:
1: (= ?X ?X)
                                                    assertion
```

```
3: (OR (= ?X SPEC::BIT-0) (= ?X SPEC::BIT-1))
                                                        assertion
8: (= (SPEC::BIT-CARRY SPEC::BIT-0 SPEC::BIT-0) SPEC::BIT-0)
                                                        assertion

10: (= (SPEC::BIT-CARRY SPEC::BIT-1 SPEC::BIT-0) SPEC::BIT-0)
                                                        assertion
11: (= (SPEC::BIT-CARRY SPEC::BIT-1 SPEC::BIT-1) SPEC::BIT-1)
                                                        assertion
14: (= (SPEC::BIT-AND SPEC::BIT-0 SPEC::BIT-0) SPEC::BIT-0)
                                                        assertion
16: (= (SPEC::BIT-AND SPEC::BIT-1 SPEC::BIT-0) SPEC::BIT-0)
                                                        assertion
17: (= (SPEC::BIT-AND SPEC::BIT-1 SPEC::BIT-1) SPEC::BIT-1)
                                                        assertion
18: (NOT (= (SPEC::BIT-AND SPEC::BIT-1 #:SK1)
            (SPEC::BIT-CARRY SPEC::BIT-1 #:SK1)))
                                                        ~conclusion
21: (OR (= ?X SPEC::BIT-1) (= (SPEC::BIT-AND SPEC::BIT-1 ?X) ?X))
                                                        paramodulate 16 by 3
23: (OR (= ?X SPEC::BIT-1) (= (SPEC::BIT-AND ?X ?X) ?X))
                                                        paramodulate 14 by 3
24: (OR (= ?X SPEC::BIT-1) (= (SPEC::BIT-CARRY SPEC::BIT-1 ?X) ?X))
                                                        paramodulate 10 by 3

26: (OR (= ?X SPEC::BIT-1) (= (SPEC::BIT-CARRY ?X ?X) ?X))
                                                        paramodulate 8 by 3
68: (OR (= #:SK1 SPEC::BIT-1)
        (NOT (= #:SK1 (SPEC::BIT-CARRY SPEC::BIT-1 #:SK1))))
                                                        paramodulate 18 by 21
102: (= (SPEC::BIT-AND ?X ?X) ?X)
                                                        paramodulate 17 by 23
144: (= (SPEC::BIT-CARRY ?X ?X) ?X)
                                                        paramodulate 11 by 26
287: (= #:SK1 SPEC::BIT-1)
                                                        hyperresolve 68,24
288: FALSE
                                                        rewrite 18 by 1, 144,
                                                        102, 287

;; Summary of computation:
;;    529 formulas have been input or derived (from 43 formulas).
;;    288 (54%) were retained.  Of these,
;;        30 (10%) were simplified or subsumed later,
```

```
;;         0 ( 0%) were deleted later because the agenda was full, and
;;       258 (90%) are still being kept.

;; Run time by activity in seconds
;; excluding printing time:
;;      0.11   0%   Resolution
;;      5.05  13%   Paramodulation
;;      0.01   0%   Factoring
;;     20.10  52%   Forward subsumption
;;      3.65   9%   Backward subsumption
;;      3.04   8%   Forward simplification
;;      0.12   0%   Backward simplification
;;      0.12   0%   Equality ordering
;;      6.24  16%   Other
;;     38.44        Total

 Snark result  PROOF-FOUND PROOF-FOUND.
;;;  Verified conjecture Bit-And-Carry.
;;; Verified the 1 conjecture attempted.
```

Note that the SPECWARE syntax has been translated into SNARK syntax (via REFINE rewriting). Thus the negation of one of the base cases of the SPECWARE conjecture,

```
(not (fa (b2 : Bit)
  (equal (bit-and bit-1 b2)
         (bit-carry bit-1 b2))))
```

has been translated into

```
(NOT (= (SPEC::BIT-AND SPEC::BIT-1 #:SK1)
        (SPEC::BIT-CARRY SPEC::BIT-1 #:SK1)))
```

The SPECWARE `equal` has been translated into the SNARK `=`. Specware symbols have been prefixed by `SPEC::` to avoid name clashes. Also, the quantifier `fa` has been removed and the quantified variable `b2` has been replaced by the skolem constant `#:SK1`, by SNARK skolemization, not by the interface.

This interface has been completed and tested successfully on several SPECWARE theories, including the theorems for a specification for bit-vectors, a theory of pictures for the automated construction of visualizations of structures, and a formulation of semi-lattice theory for JAVA byte-code verification. SRI has also collaborated with Kestrel and NASA personnel on the design of hooks to allow the Meta-Amphion application to invoke SNARK via SPECWARE.

6 Remaining Work

With experience we may see ways to improve the SNARK-SPECWARE interface. Some of these problems are as follows:

higher-order functions. SPECWARE logic is higher order, while SNARK is a first-order theorem prover. The interface makes no attempt to translate higher-order functions and predicates into first-order logic, e.g., via reification.

sorts. The interface translates SPECWARE sorts into SNARK sorts, which is economical. However, there is no attempt to deal with such complex SPECWARE sorts as the quotient, product, or co-product, for which SNARK has no equivalent; presumably, this should be done by sort axioms, as in KITP. Also, SPECWARE subsorts are mapped into distinct sorts, not into SNARK subsorts. (In SPECWARE, as in all category theory, an element of a subsort is not regarded as an element of its supersort; in SNARK, it is.

associative-commutative unification. If associative and commutative axioms are provided, SNARK will recognize them, remove them from the axiom base, and use associative-commutative unification instead. Other than this, there is no mechanism by which a user may declare a function or predicate symbol to be associative or commutative.

witness-finding. SNARK has a witness-finding capability, and a LISP function has been provided by which a SPECWARE user may invoke SNARK witness-finding for a SPECWARE theorem. However, SPECWARE has no mechanism for referring to a witness within the specification itself, and theoretical obstacles exists for introducing such a mechanism, unless the theorem establishes uniqueness.

strategic controls. The SPECWARE user is given no way of introducing a term ordering, an agenda-ordering function, or symbol weights for specializing SNARK to a particular subject domain; he or she must be happy with the defaults.

treatment of logic morphisms. Whereas the ordinary morphism is a construct supported by SPECWARE, the inter-logic morphism is not. Construction of the morphism between SPECWARE and SNARK required a lot of ad hoc programming in REFINE. While some of this is unavoidable (e.g., menu design), much of it can be systematized (e.g., the translation between languages). The plan is to introduce the inter-logic morphism into SPECWARE as a first-class citizen.

Interfaces between SPECWARE and other theorem-provers, such as PVS, remain to be constructed. This effort will be facilitated if the results of the experience of integrating SPECWARE and SNARK can inform the development of an implementation of the inter-logic morphism within SPECWARE.