Integrating Isabelle/HOL with Specware

Stephen Westfold
Kestrel Institute
39 PUBLICATIONS **458** CITATIONS

Integrating Isabelle/HOL with Specware

Stephen J. Westfold

Kestrel Institute, Palo Alto, California, USA

Abstract. Isabelle/HOL is integrated with Specware in order to discharge proof obligations arising during Specware's specification and refinement process. Specware's proof obligations arise from use of predicate subtypes, termination conditions, and correctness of refinements as well as any explicit theorems. Specware specifications are structured into units called specs which correspond to Isabelle/HOL theories. Refinement is specified using spec morphisms and spec substitutions, a particular kind of colimit. We provide a system based on translating a Specware specification and its refinement into Isabelle/HOL theories, such that if Isabelle accepts all the translated theories, then the refinement is correct. Isabelle scripts for proving obligation theorems are embedded in Specware spec files so that proofs can be reused. The paper describes this translation and the issues that arise.

1 Introduction

Specware is a software specification and refinement system developed at Kestrel Institute based on higher-order logic and theory morphisms [1]. During Specware specification development and refinement to an implementation, many proof obligations are generated. Simple obligations can be discharged by Specware itself, but more complicated obligation theorems require an external theorem prover to discharge them. Previously, the main external theorem prover supported has been SNARK, an automatic first-order theorem prover [2]. This current work gives the user the alternative of proving Specware obligations using Isabelle/HOL, an interactive theorem prover for higher-order logic [3].

When SNARK succeeds in proving an obligation, everything is fine. However, often SNARK will fail to prove an obligation at first, either because the theorem is not true or the proof is too difficult for SNARK to find without further assistance. The first case, where the obligation theorem is not true, is common during the process of specification development, either because of incompleteness in the specification or because of mistakes made by the specifier. Finding the reasons for a failed proof can be an important part of the specification development process. Understanding proof failure has been difficult with SNARK, mainly because of the distance of the language of SNARK's (partial) proofs from that of Specware. This distance is because SNARK is a resolution-based prover using clausal form, whereas Specware uses higher-order logic with polymorphism.

Both Specware and Isabelle/HOL are based on higher-order logic with polymorphism, so Isabelle/HOL is a natural target for Specware. Specware declarations, definitions, axioms and theorems can be translated naturally to the corresponding Isabelle versions. The main additional feature of Specware's logic is predicate sub-typing where a type may be restricted by an arbitrary predicate.[1] Predicate subtyping can be used to restrict the domain of a function to legal inputs, and to give information about the the range of a a function. These translate to Isabelle/HOL as extra axioms on definitions involving subtypes and theorems for applications of the function. The axioms typically state that for all inputs satisfying the predicate on the input type, the function applied to the inputs satisfies the predicate of the output type. The subtype theorems state that the arguments of the function satisfy the subtype predicate assuming information extracted from the context.

Specware and Isabelle/HOL share essentially identical import mechanisms. Specware specifications are composed of *specs* which import other specs, whereas Isabelle/HOL has *theories* which import other theories. In addition Specware has the notion of *spec morphisms* [5] which Isabelle/HOL does not, although there is recent work to add them in an extended version of Isabelle/HOL [6]. A spec morphism is a mapping of op identifiers to op identifiers and type identifiers to type identifiers from a source spec to a target spec such that axioms in the source spec are theorems in the target spec. The target spec is a refinement of the source spec in that there is a target spec model for every source spec model, but not necessarily vice versa. Specware includes the colimit operation to provide a very general way of combining specs. We do not attempt to translate general colimits in this work, just *spec substitutions* which are a special case of colimit that cover most cases. A spec substitution takes a complex spec and a spec morphism where the source of the morphism is an import of the complex spec, and produces a copy of the complex spec with the source spec replaced by the target spec of the morphism. In this way, a spec substitution produces a refinement of a complex spec from a refinement of one of its components.

2 Embedding Proof Scripts in Specware

Simple extensions are made to the Specware syntax to allow Isabelle proof scripts to be embedded in Specware specs, and to allow the user to specify translation of Specware ops and types to existing Isabelle constants and types. The rest of the Specware system treats these as comments.

An embedded Isabelle proof script in a Specware spec consists of an introductory line beginning with **proof** *Isa*, the actual Isabelle script on subsequent lines terminated by the string **end−proof**. For example, the simple proof script *apply*(*auto*) **done** can be embedded as follows:

```
proof Isa
  apply(auto)
end−proof
```

[1] PVS also has predicate subtyping, but lacks polymorphism [4].

If the last command before **end−proof** is not **done**, **qed** or **sorry**, the command **done** is inserted.

The proof script should occur immediately after the theorem or definition that it applies to. If the script applies to a proof obligation that is not explicit in the spec, then the name of the obligation should appear after **proof** *Isa*, on the same line.

If the user does not supply a proof script for a theorem then the translator will supply the script *apply*(*auto*) which is all that is required to prove many simple proof obligations.

3 Translation of Specware to Isabelle/HOL

Isabelle/HOL has types, constants and variables which correspond to Specware types, ops and variables.

3.1 Identifiers

All Specware variables are explicitly bound, whereas in Isabelle/HOL top-level variables are implicitly bound. This means that the translator needs to be concerned with renaming to avoid capture of implicitly bound variables such as *comp* and *o* by Isabelle/HOL constants of the same name.

Both systems have namespaces to avoid name confusion with a dot-notation for full names, but the namespace concepts are different, so the translator does not try to use the Isabelle dot-notation. Instead, an identifier such as *a.b* is translated to *a__b*. This scheme could possibly lead to name confusion if the user uses names containing double underbars.

Specware allows some non-alphabetic characters in the identifiers of ops and types that Isabelle/HOL does not, so these must be translated. For example, *equal?* is translated to *equal_p*.

3.2 Types

Most Specware types have directly corresponding Isabelle/HOL types:

Boolean → *bool*
functions → *functions*
products → *pairs and tuples*
coproducts → *constructor types* (*datatypes*)
type *variables* → **type** *variables*
Char → *char*
List → *list*
String → *string*

There is a slight mis-match with products and tuples in that in Specware products may be of arbitrary length whereas in Isabelle/HOL a tuple of more than length 2 is really a structure of nested pairs. For construction this is transparent, but not for dereferences. For example, in Specware *x.2* translates to *snd x* if x is a product of length 2, but it translates fo *fst*(*snd x*) if x is a product of greater than length 2.

Coproducts are the same as constructor types except that a constructor with multiple arguments must be curried in Isabelle/HOL and uncurried in Specware. For example,[2]

---

type *boolex* =
  | *And* (*boolex* × *boolex*)
  | *Const Boolean*
  | *Neg boolex*
  | *Var Nat*

translates to

datatype *boolex* =
    And boolex boolex
  | *Const bool*
  | *Neg boolex*
  | *Var nat*

Isabelle/HOL does not have predicate subtypes, so these are generally mapped to the supertype. The exception is that Specware types may be explicitly mapped to existing Isabelle/HOL types as described below, and this mapping allows a supertype to be mapped to a different type from the subtype with coercions functions to map between the two types. For example, in Specware, the type *Nat* is a sub-type of *Integer*, but these are mapped to the separate Isabelle/HOL types *nat* and *integer*. Mapping *nat* to *integer* would be correct but would make many proofs harder.

3.3 Terms

Isabelle/HOL and Specware share many of the same kinds of term, including case statements, lambda expressions, let expressions, if-then-else expressions, universal and existential quantifications, and function applications both in prefix form or infix. There are examples throughout the paper. In some cases the syntax is slightly different, but they are close enough that the meaning is obvious, so I will not describe the details. Specware has a letrec expression which is described below in the subsection on local definitions.

[2] This example and some of the following are drawn from section 2.4.6 of the Isabelle/HOL tutorial [3].

3.4 Op Declarations and Definitions

A Specware definition may translate into one of three different kinds of Isabelle definitions: *defs*, *recdefs* and *primrecs* (primitive recursions). Simple recursion on coproduct constructors translates to **primrec**, but if the function has multiple arguments, only if the function is curried. Other recursion translates to *recdefs* which, in general, require a user-supplied measure function to prove termination. Non-recursive functions are translated to *defs*, except in some cases they are translated to *recdefs* which allow more pattern matching.

For example, the Specware declaration and definition:

```
op bvalue: boolex → (Nat → Boolean) → Boolean
def bvalue be env =
  case be of
    | Const b → b
    | Var x → env x
    | Neg b → ¬ (bvalue b env)
    | And(b,c) → bvalue b env ∧ bvalue c env
```

translate to

```
consts bvalue :: "boolex ⇒ (nat ⇒ bool) ⇒ bool"
primrec
  "bvalue (Const b) env = b"
  "bvalue (Var x) env = env x"
  "bvalue (Neg b) env = (¬ (bvalue b env))"
  "bvalue (And b c) env = (bvalue b env ∧ bvalue c env)"
```

Recursive functions that are translated to *recdefs* can have a measure function specified on the **proof** *Isa* line, by including it between double-quotes. For example:

```
proof Isa "measure (λ(wrd,sym). length wrd)" end−proof
```

There are examples of different kinds of definition below.

3.5 Axioms and Theorems

Specware axioms and theorems are translated naturally to Isabelle/HOL axioms and theorems. For example,

```
theorem valif is
  ∀(b,env) valif (bool2if b) env = bvalue b env
  proof Isa
    apply(induct_tac b)
    apply(auto)
  end−proof
```

translates to

```
theorem valif:
  "valif (bool2if b) env = bvalue b env"
  apply(induct_tac b)
  apply(auto)
  done
```

Annotations for theorems may be included on the **proof** *Isa* line. For example,

```
theorem Simplify_valif_normif is
  ∀(b,env,t,e) valif (normif b t e) env = valif (IF(b, t, e)) env
  proof Isa [simp]
    apply(induct_tac b)
    apply(auto)
  end−proof
```

translates to

```
theorem Simplify_valif_normif [simp]:
  "valif (normif b t e) env = valif (IF b t e) env"
    apply(induct_tac b)
    apply(auto)
  done
```

In this example we see that universal quantification in Specware becomes, by default, implicit quantification in Isabelle. This is normally what the user wants, but not always. The user may specify the variables that should be explicitly quantified by adding a clause like $\forall t\ e.$ to the **proof** *Isa* line. For example,

```
theorem Simplify_valif_normif is
  ∀(b,env,t,e) valif (normif b t e) env = valif (IF(b, t, e)) env
  proof Isa [simp] ∀t e.
    apply(induct_tac b)
    apply(auto)
  end−proof
```

translates to

```
theorem Simplify_valif_normif [simp]:
```

```
"∀(t::ifex) (e::ifex). valif (normif b t e) env = valif (IF b t e) env"
  apply(induct_tac b)
  apply(auto)
done
```

It is sometimes necessary to add explicit type annotations to allow the Isabelle type-checker to resolve overloading.

3.6 Subtype Axioms and Obligations

If the result of an op is a subtype then an axiom is produced to that effect. For example:

```
op n: {i: Nat | i > 0}
op f(x: Nat): {i: Nat | i > 0}
```

translates to

```
consts n :: "nat"
axioms n_subtype_constr:
  "n > 0"
consts f :: "nat ⇒ nat"
axioms f_subtype_constr:
  "f i > 0"
```

For references to functions with take a subtype as an argument, a subtype obligation is generated for each application of the function to ensure that the subtype predicate holds.

For example, the declaration of the *nth* function is as follows:

```
op nth: [a] {(l,i) : List a × Nat | i < length l} → a
```

Note that, to express this subtype restriction the *nth* cannot be a curried function because Specware does not have dependent types. However, we do translate *nth* to the corresponding Isabelle function *!* which is curried.

For example, from

```
op L: List Nat = [1,2,3]
proof Isa [simp] end−proof
op L2: Nat = nth(L,2)
```

we get

```
consts L :: "nat list"
```

```
defs L_def [simp]: "L ≡ [1,2,3]"
consts L2 :: "nat"
theorem L2_Obligation_subsort:
  "2 < length L"
  apply(auto)
  done
defs L2_def: "L2 ≡ L ! 2"
```

3.7 Local Recursive Functions

Local recursive functions are allowed in Specware but not in Isabelle/HOL. Therefore these functions are lifted to the top level using the technique of lambda-lifting [7] which converts free variables to extra parameters.

For example, the *tabulate* function:

```
op [a] tabulate(n: Nat, f: Nat → a): List a =
  let def tabulateAux (i : Nat, l : List a) : List a =
            if i = 0 then l
            else tabulateAux(i−1,Cons(f(i−1),l)) in
  tabulateAux(n,[])
proof Isa tabulate__tabulateAux "measure (λ(i,l,f). i)" end−proof
```

is translated to

```
recdef tabulate__tabulateAux "measure (λ(i,l,f). i)"
  "tabulate__tabulateAux(0,l,f) = l"
  "tabulate__tabulateAux(Suc i,l,f)
     = tabulate__tabulateAux(i,Cons (f i) l,f)"
consts tabulate :: "nat × (nat ⇒ 'a) ⇒ 'a list"
recdef tabulate "{}"
  "tabulate(n,f) = tabulate__tabulateAux(n,[],f)"
```

Note the translation of the **if** $i = 0$ **then** ... **else** ... into cases on *0* and *Suc i*. Specware does not allow a case split on successor because naturals are defined as a subtype of integers and not constructed from zero and successor.

3.8 Logic Morphisms

We wish to exploit existing Isabelle/HOL libraries, so we provide a mechanism for mapping Specware op and type identifiers to Isabelle/HOL constant and type identifiers. Any definitions in Specware for these ops must be theorems in the Isabelle/HOL theory for this mapping to be correct.

A translation table for Specware types and ops is introduced by a line beginning **proof** *Isa Thy_Morphism* followed optionally by an Isabelle theory (which will be imported into the translated spec), and terminated by the string **end−proof**. Each line gives the translation of a type or op. For example, for the Specware Option theory we have:

```
proof Isa Thy_Morphism
 type Option.Option → option
 Option.mapOption → option_map
end−proof
```

A type translation begins with the word **type** followed by the fully-qualified Specware identifier, → and the Isabelle identifier. If the Specware type is a sub-type, you can specify coercion functions to and from the super-type in parentheses separated by commas. Note that by default, sub-types are represented by their super-type, so you would only specify a translation if you wanted them to be different, in which case coercion functions are necessary. Following the coercions functions can appear a list of overloaded functions within square brackets. These are used to minimize coercions back and forth between the two types.

An op translation begins with the fully-qualified Specware identifier, followed by → and the Isabelle constant identifier. If the Isabelle constant is an infix operator, then it should be followed by *Left* or *Right* depending on whether it is left or right associative and a precedence number. Note that the precedence number is relative to Specware's precedence ranking, not Isabelle's. Also, an uncurried Specware op can be mapped to a curried Isabelle constant by putting *curried* after the Isabelle identifier, and a binary op can be mapped with the arguments reversed by appending *reversed* to the line.

For Specware's Integer spec we have the logic morphism

```
proof Isa Thy_Morphism Presburger
 type Integer.Integer → int
 type Nat.Nat → nat (int,nat) [+,×,div,rem,≤,<,≥,>,abs,min,max]
 Integer.+ → + Left 25
 Integer.− → − Left 25
 IntegerAux.− → −
 Integer.× → × Left 27
 Integer.div → div Left 26
 Integer.rem → mod Left 26
 Integer.≤ → \<le> Left 20
 Integer.< → < Left 20
 Integer.≥ → \<ge> Left 20
 Integer.> → > Left 20
 Integer.min → min curried
 Integer.max → max curried
end−proof
```

Note that the list of overloaded functions does not include − (minus) because

Isabelle's definition of $-$ on naturals is not the same as Specware's if the second argument is larger than the first: for Isabelle the value is *0* whereas for Specware it is a negative integer. However, if the consumer of the subtraction is expecting a natural then we know there is a proof obligation that ensures this, so in this case we do translate Specware's $-$ to Isabelle's.

3.9 Spec Morphisms and Spec Substitutions

The obligations theory in Isabelle/HOL of a spec morphism is simply a theory that imports the translation of the target spec of the morphism and includes the axioms of the source spec translated along the morphism as theorems to be proven. Named proof scripts for the theorems follow the morphism.

For example, the morphism *AB_M* from spec *A* to spec *B* in the following:

```
A = spec
      op f: Nat → Nat
      axiom f_pos is ∀(x) f x > 0
    endspec

B = spec
      op g(i: Nat): Nat = i + 2
    endspec

AB_M = morphism A → B {f ↦ g}
  proof Isa f_pos
  apply(auto simp add: g_def)
  end−proof
```

has the obligation theory:

```
theory AB_M
imports B
begin
theorem f_pos:
  "g x > 0"
  apply(auto simp add: g_def)
  done
end
```

The syntax of a spec substitution is a spec term followed by a morphism in square brackets. For example, the spec *D* is defined as the substitution of morphism *AB_M* applied to the spec *C* in the continuation of the current example:

```
C = spec
      import A
```

```
    op f2(i: Nat): Nat = f(f i)
  endspec

D = C[AB_M]
```

The translation of spec D is just spec C with A replaced by B and performing the renaming of the morphism *AB_M* giving the theory:

```
theory D
imports B
begin
consts f2 :: "nat ⇒ nat"
defs f2_def: "f2 i ≡ g (g i)"
end
```

Note that if the spec to be replaced is deeply embedded in the import structure, then it is necessary to make copies of all the imported specs that import this spec, and translate them to Isabelle/HOL theories. For a large specification and refinement there may be many substitutions applied in sequence to the top level spec, implicitly using many intermediate specs. It is desirable to generate the specs required by the final refinement and their corresponding Isabelle/HOL theories without generating all the intermediates.

4 Current Restrictions and Future Work

This initial translator has a number of limitations. It should translate all Specware specs but not all translated definitions and constructs will be accepted by Isabelle/HOL. In particular, only case expressions that involve a single level of pattern-matching on constructors are accepted. An exception, is that some nesting is allowed in top-level case expressions that are converted into definition cases. Mutual recursion is not currently supported. The translator currently targets the 2006 release version of Isabelle. The next version of Isabelle includes a new function package that should allow more Specware definitions to be translated. Also, it allows termination to be proved for a subdomain which is a natural match for a Specware function whose domain is a predicate subtype.

Bortin, Johnsen and Lüth have developed an extension of Isabelle that includes theory morphisms [6]. This would provide a natural target for a translation of Specware's spec morphisms. Morphism obligations are proved, but there is a meta-theorem about morphisms that all theorems in the source spec of the morphism, are theorems in the target spec when translated by the renaming of the morphism. We do not currently exploit this powerful property, unlike this Isabelle extension which realizes these theorems by translating proofs along the morphism. By targeting this extension we would gain this property. A concern is that the price of this extension is that low-level proofs must be stored, which could be expensive, especially for a large specification with many spec substitutions.

In the future we wish to allow results from Isabelle inference to be returned to Specware. Witness-finding can be used in a number of ways during algorithm synthesis, for example, to instantiate a function in a program schema [8]. If we find a witness for an existential during a proof in Isabelle, we need to translate it back into a Specware term, so it can be used to give a definition in a refined spec. This back-translation is straightforward, as the term language for Specware and Isabelle are very similar. The only significant issue is the different name-space rules of the two systems. In particular, translation of Specware's qualified names needs to be invertible.

5 Conclusions

Initial experience with using Isabelle/HOL to discharge proof obligations has been positive. The Isabelle/HOL translation of a Specware spec is very readable with direct correspondence between many elements. The translation includes extra theorems, mainly for sub-type obligations, but their names and location make them easy to connect to their origin. The main concern with using Isabelle/HOL compared to using SNARK was that the user would have to be concerned with giving proofs for many simple obligations. In our limited experience, the *auto* tactic is able to discharge most of the obligations that SNARK was able to discharge automatically. By making this the default tactic, we have avoided cluttering Specware specs with trivial proof scripts. In addition, the means of controlling the proof process in Isabelle/HOL are more intuitive than with SNARK, and the reasons for failure of a proof are more apparent, and it is easier to control a more complicated proof. Termination proofs, in particular, have been much easier in Isabelle/HOL, with the user typically providing a simple measure function.

We have used the Specware to Isabelle/HOL integration with tutorial examples and Specware's libraries. This has revealed several subtle bugs in the specifications. We plan to use this system to prove the correctness of more significant refinements, and to help in automatically generating correct refinements.

References

Kestrel Institute: Specware System and Documentation. (2007) http://www.specware.org/.

Stickel, M., Waldinger, R., Chaudhri, V.: A guide to SNARK. SRI International. (2000)

Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)

Rushby, J.M., Owre, S., Shankar, N.: Subtypes for specifications: Predicate subtyping in PVS. Software Engineering **24** (1998) 709–720

Burstall, R.M., Goguen, J.A.: Putting theories together to make specifications. In: Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, MA, IJCAI (1977) 1045–1058

Bortin, M., Johnsen, E.B., Lüth, C.: Structured formal development in Isabelle. Nordic J. of Computing **13** (2006) 2–21

Johnsson, T.: Lambda lifting: Transforming programs to recursive equations. In: Functional Programming Languages and Computer Architecture. Proc. of a conference (Nancy, France, Sept. 1985), New York, NY, USA, Springer-Verlag Inc. (1985)

Smith, D.R.: The role of witness-finding in software synthesis. In Fischer, B., Smith, D.R., eds.: Logic-Based Program Synthesis: State of the Art and Future Trends: Papers from the 2002 Spring Symposium, Menlo Park, CA, American Association for Artificial Intelligence. (2002) 91–94

This article was processed using the LaTeX macro package with LLNCS style