Would you ever risk a non-monomorphic specification?*

Arno Schönegge
Institut für Logik, Komplexität und Deduktionssysteme
Universität Karlsruhe
D-76128 Karlsruhe, Germany
email: `schoenegge@ira.uka.de`

Abstract

In this paper we illustrate the risk of non-monomorphicity[1] of requirement specifications[2] by an example from formal software development. This example was completely carried out in the KIV system.[3]

1 Introduction

Formal software development starts with making up a formal requirement specification that describes the features required of the software system to be developed. Requirement specifications may be(come) an essential part of a contract between a customer, who wants to get bug-free software for his (safety-critical) application, and the software developer: the customer assures to accept the software if (and only if) it meets this specification.

The requirement specification must not be monomorphic. Quite the reverse holds: in order to provide the software developer with freedom that

*This work was supported under grants no. Me 672/6-2,3 by the Deutsche Forschungsgemeinschaft as part of the focus program "Deduktion".

[1] A specification is called *monomorphic* if any two models of it are isomorphic. So a monomorphic specification is either inconsistent or it determines some algebra uniquely up to isomorphism.

[2] In this paper we regard a requirement specification to be a formal description of the expected external behavior of the software system. So, non-behavioral requirements (concerning efficiency, reliability, maintainability, portability, etc.) are not the subject of our discussion.

[3] KIV stands for Karlsruhe Interactive Verifier, which is an advanced tool for the development of correct software. It supports the entire design process starting from formal specifications and ending with verified code [9, 13].

may facilitate more efficient implementations, it may be desired to specify only the relevant features. However, on the other hand, non-monomorphicity (i.e. ambiguity) can be very dangerous, especially if one is unaware of the (whole extent of the) gaps in the specification.

The aim of this paper is to illustrate two points: firstly, that in general, it is in no way trivial to recognize gaps in a specification. Secondly, that ambiguity of a requirement specification can have far-reaching, dire consequences, even if this ambiguity is caused by gaps which are neither intended nor easy to see. This illustration is done via a well-known[4] example, namely the formal development of a prover for propositional logic.

We start out from the definition of validity in propositional logic given in [4] and formalize it using a first-order[5] specification language with (ultra-) loose semantics.[6] This is done in the next section. In section 3 an executable implementation is presented that meets this specification, but which behaves not as intended. In section 4 we discuss the reason for this bad surprise: concealed ambiguity of (parts of) the requirement specification. Section 5 illustrates the difficulties in removing such ambiguity. Finally, in the last section we draw conclusions and briefly indicate directions for future work.

2 Specifying a Propositional Logic Prover

In this section we build up a requirement specification for a propositional logic prover, i.e. a specification of syntax and semantics of propositional logic. As basis we take the definitions given in ([4], p. 4-7) and formalize them (as literally as possible) in a first-order specification language (enlarged with generation principles). We start with the syntax of propositional formulas, which is defined in ([4], p. 5) as follows:[7]

> “We now set up the propositional logic as a formal language. The symbols of our language are as follows:
>
> connectives $\wedge$ (and), $\neg$ (not);
> parentheses),(;
> a nonempty set $\mathcal{S}$ of signature symbols

[4] This example was already treated (however, with another aim in mind) e.g. in [3, 19], and also in the KIV system [17].

[5] enlarged with generation principles (a kind of higher-order axioms).

[6] In the ultra-loose approach to algebraic specification *all* models of a specification are considered; whereas in the loose approach exactly those models are considered which do not contain junk, i.e. all models whose elements can be denoted by ground terms. However, for the specifications presented in this paper the loose and the ultra-loose semantics coincide since the specifications contain generation principles for all their sorts. The loose semantics approach to algebraic specification is widely accepted as appropriate for specification of software systems (see for instance [6, 20, 11]).

[7] Here and in what follows we quote [4] almost literally. Modifications concern the replacement of ‘sentential’ by ‘propositional’, and ‘statement’ by ‘formula’.

> Intuitively, the signature symbols stand for atomic formulas, and the connectives $\wedge$, $\neg$ stand for the words used to combine atomic formulas into compound formulas. Formally, the *formulas* of $\mathcal{S}$ are defined as follows:
>
> (i). Every signature symbol S is a formula.
>
> (ii). If φ is a formula then $(\neg\varphi)$ is a formula.
>
> (iii). If φ and ψ are formulas then $(\varphi \wedge \psi)$ is a formula.
>
> (iv). A finite sequence of symbols is a formula only if it can be shown to be a formula by a finite number of applications of (i)-(iii)."

In order to formalize this definition, we (slightly) restrict it by demanding that the propositional signature $\mathcal{S}$ (i.e. the set of atomic formulas) is infinite and enumerable, or even more concrete, we put

$$\mathcal{S} \;:=\; \{\, S_n \mid n \in \mathbb{N} \,\}$$

where we assume $S_i \neq S_j$ whenever $i \neq j$. Thus, we are allowed to specify the syntax of propositional formulas as follows:[8]

specification FORMULA

using NAT

sorts formula

functions
S : nat $\rightarrow$ formula
$\neg' \cdot$: formula $\rightarrow$ formula
$\cdot \wedge' \cdot$: formula $\times$ formula $\rightarrow$ formula

variables φ, ψ : formula

axioms formula **freely generated by** S, $\neg'$, $\wedge'$

end specification

Propositional formulas are represented by terms of the sort formula, which are built from the constructor functions S, $\neg'$ and $\wedge'$. In order to rule out conflicts with the logical connectives of first-order logic (which is a subset of the specification language used here), we have attached a prime ($'$) at the symbols $\neg'$ and $\wedge'$. The dots ($\cdot$) after $\neg'$, and before and after $\wedge'$ indicate that $\neg'$ will be used in prefix notation and $\wedge'$ in infix notation. The sole axiom is a strengthening of the generation principle

formula **generated by** S, $\neg'$, $\wedge'$

which is a kind of higher-order axiom. It restricts the class of models of the specification to those in which each object of the domain can be denoted by

[8] The specification NAT of natural numbers used here is given below.

a constructor term, i.e. by a ground term built from the constructors S, $\neg'$, $\wedge'$ (plus the constructors for the sort nat). The addition **freely** means that any two distinct constructor terms denote distinct objects.

The specification NAT of natural numbers used here can be specified as follows (the symbol +1 denotes the successor function, written postfix):

```
specification NAT
    sorts        nat
    functions     0    :        → nat
                 · +1  : nat    → nat
    variables    n, m  : nat
    axioms       nat freely generated by 0, +1
end specification
```

This concludes the specification of the syntax of propositional logic. We now turn to its semantics. In ([4], p. 4) we find the following [motivation for the] definition of the class of models for propositional logic.

> "At the most intuitive level, an intended interpretation of these formulas is a 'possible world', in which each formula is either true or false. We wish to replace these intuitive interpretations by a collection of precise mathematical objects which we may use as our models. The first thing which comes to mind is a function F which associates with each atomic formula S one of the truth values 'true' or 'false'. Stripping away the inessentials, we shall instead take a model to be a subset A of $\mathcal{S}$; the idea is that $S \in A$ indicates that the atomic formula S is true, and $S \notin A$ indicates that the atomic formula S is false.
>
> By a *model A of $\mathcal{S}$* we simply mean a subset A of $\mathcal{S}$."

Again, we slightly modify this definition in order to fit our purpose. For the first modification we make use of the fact that we are interested in validity of (finite sets of) finite formulas only (see rest of this section). Therefore (cf. [4]), it is sufficient to regard *finite* models only, i.e. finite subsets of atomic formulas. The second modification concerns representation issues only. Since we have restricted ourself to $\mathcal{S} = \{S_n \mid n \in \mathbb{N}\}$ with $S_i \neq S_j$ whenever $i \neq j$, we can represent a atomic formula S_n by its index n. So, instead of taking a model to be a subset of atomic formulas, it is more convenient for us to represent a model A as the set of indices of atomic formulas. Thus, we specify a model A to be a finite subset of natural numbers. As operations on models we take the constructors[9] empty set ($\emptyset$) and insertion of an element ($\odot$), and

[9] Notice, that the data type of models (i.e. finite sets) is *not freely* generated: for instance, the distinct ground terms $0 \odot \emptyset$ and $0 \odot 0 \odot \emptyset$ denote the same object.

the membership predicate ($\in$), where $n \in A$ indicates that S_n is true in A and $n \notin A$ indicates that S_n is false in A.

```
specification MODEL
    using        NAT
    sorts        model
    functions      ∅     :                  →   model
                 · ⊙ ·   : nat × model      →   model
    predicates   · ∈ ·   : nat × model
    variables     A, B   : model
    axioms       model generated by ∅, ⊙
                 ∅ ≠ n ⊙ A                       (1)
                 n ⊙ m ⊙ A = m ⊙ n ⊙ A           (2)
                 n ⊙ n ⊙ A = n ⊙ A               (3)
                 n ∈ A ↔ n ⊙ A = A               (4)
end specification
```

The first axiom of this specification states the generatedness by $\emptyset$ and insertion ($\odot$). The other axioms can be read as:

(1) By inserting an arbitrary element to an arbitrary set we get a set distinct from the empty set.

(2) The order of insertion is irrelevant.

(3) Inserting an element twice yields the same result as inserting it once.

(4) An element is member of a set if and only if this set is not changed by insertion of the element.

It is quite obvious, that all these axioms hold in the intended algebra (which is the algebra of finite sets on natural numbers).

After we have specified what we mean by a model, it remains to formalize the notion of validity of a formula. As above we quote the definition from ([4], p. 7):

> "We are now ready to build a bridge between the language $\mathcal{S}$ and its models, with the definition of the truth of a formula in a model. We shall express the fact that a formula φ is true in a model A succinctly by the special notation
>
> $$A \models \varphi.$$
>
> The relation $A \models \varphi$ is defined as follows:

> (i). If φ is a signature symbol S, then $A \models \varphi$ holds if and only if $S \in A$.
>
> (ii). If φ is $\psi \wedge \theta$, then $A \models \varphi$ if and only if both $A \models \psi$ and $A \models \theta$.
>
> (iii). If φ is $\neg\psi$, then $A \models \varphi$ iff it is not the case that $A \models \psi$.
>
> When $A \models \varphi$, we say that [...] A is a *model of* φ.
>
> A formula φ is called *valid*, in symbols $\models \varphi$, iff φ holds in all models for $\mathcal{S}$, that is, iff $A \models \varphi$ for all A."

The translation of this definition into first-order logic is completely straightforward.[10] In fact, the individual parts of it have one-to-one equivalents in the following specification.

```
specification VALIDITY
   using FORMULA, MODEL
   predicates    · ⊨ ·      : model × formula
                 · is_valid : formula
   axioms        A ⊨ S(n) ↔ n ∈ A
                 A ⊨ (¬′ φ) ↔ ¬ A ⊨ φ
                 A ⊨ (φ ∧′ ψ) ↔ A ⊨ φ ∧ A ⊨ ψ
                 φ is_valid ↔ ∀ A . A ⊨ φ
end specification
```

This concludes our specification of syntax and semantics of propositional logic. Its structure is illustrated in figure 1.

We hope, that the reader agrees that we have built up the requirement specification very carefully. Especially, one could (or even should) be convinced that any prover for propositional logic, which can be verified against this specification is a correct one.

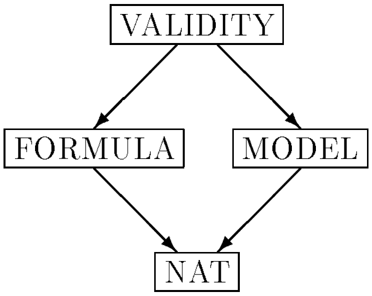


Figure 1: structure of the requirement specification.
(The arrows indicate the 'using' relation.)

[10] Here we benefit from using *full* first-order logic. A formalization without quantifiers would be more complicated (cf. [3]).

3 Implementing the Specified Prover

In this section we put ourselves in the place of the software developer. In doing so, we no longer have to think about what it means for an implementation to be correct. This is already formalized in the requirement specification, i.e. an implementation is regarded as correct whenever it meets this specification. What the software developer has to think about is optimization of qualities like verifiability and efficiency.[11]

A clever software developer may state the following procedures as part[12] of an implementation of the requirement specification:

```
VALIDITY-CHECKER (phi : formula ; var b : boolean)
begin
   var $b_1$ , $b_2$ : boolean in
   VALIDITY-CHECKER-H (true , phi ; $b_1$);
   VALIDITY-CHECKER-H (false , phi ; $b_2$);
   b := $b_1$ and $b_2$
end.

VALIDITY-CHECKER-H ($b_0$ : boolean , phi : formula ; var b : boolean)
begin
   if phi is_atomic then b := $b_0$
   else if phi is_negation then
     begin
       VALIDITY-CHECKER-H ($b_0$ , phi.negated_formula ; b);
       b := not b
     end
   else if phi is_conjunction then
     begin
       var $b_1$ , $b_2$ : boolean in
       VALIDITY-CHECKER-H ($b_0$ , phi.first_conjunct ; $b_1$);
       VALIDITY-CHECKER-H ($b_0$ , phi.second_conjunct ; $b_2$);
       b := $b_1$ and $b_2$
     end
   else abort
end.
```

These procedures might not look as the reader expected, but concerning verifiability, efficiency and correctness this implementation turns out to be a good one. This claim will be substantiated in some detail below. First we

[11] Often ease of verifiability and efficiency contradict each other, and one has to be content with a compromise.

[12] The complete implementation is given in appendix A.

have to explain the data structures the procedures work on. We describe data structures as so-called data specifications. Data specifications are specifications of freely generated data types with selectors for each parameter of a constructor and optional predicates for each construct. In a highly compact notion the data specifications used in the implementation look as follows:

```
data specification FORMULA-DATA
   using NAT-DATA
   formula =
            S      ( · .index              : nat      ) with · is_atomic
        |  $\neg'$ ·   ( · .negated_formula    : formula  ) with · is_negation
        | · $\wedge'$ · ( · .first_conjunct     : formula ,
                     · .second_conjunct    : formula  ) with · is_conjunction
   variables $\varphi$, $\psi$ : formula
end data specification
```

```
data specification NAT-DATA
   nat =   0
         | · +1 ( · −1 : nat )
   variables n, m : nat
end data specification
```

```
data specification BOOL-DATA
   boolean = true | false
   variables b : boolean
end data specification
```

To illustrate the semantics of such data specifications we point out that the specification FORMULA-DATA can be regarded as a compact notion for the specification one gets from enriching the above specification FORMULA by the following selector functions and predicates:

```
functions   · .index              : formula  → nat
            · .negated_formula    : formula  → formula
            · .first_conjunct     : formula  → formula
            · .second_conjunct    : formula  → formula

axioms      S(n).index = n
            ($\neg'$ $\varphi$).negated_formula = $\varphi$
            ($\varphi$ $\wedge'$ $\psi$).first_conjunct = $\varphi$
            ($\varphi$ $\wedge'$ $\psi$).second_conjunct = $\psi$
```

predicates	· is_atomic	: formula
	· is_negation	: formula
	· is_conjunction	: formula
axioms	$S(n)$	is_atomic
	$\neg\ (\neg' \varphi)$	is_atomic
	$\neg\ (\varphi \wedge' \psi)$	is_atomic
	$\neg\ S(n)$	is_negation
	$(\neg' \varphi)$	is_negation
	$\neg\ (\varphi \wedge' \psi)$	is_negation
	$\neg\ S(n)$	is_conjunction
	$\neg\ (\neg' \varphi)$	is_conjunction
	$(\varphi \wedge' \psi)$	is_conjunction

Data specifications are consistent per construction, and code implementing them can be automatically generated.[13] So our implementation is in fact executable. Furthermore the implementation meets the requirement specification. This claim is substantiated in the next section; in appendix A we sketch a formal correctness proof, which was carried out in the KIV system. We can hope that the verification of our implementation causes no difficulties, because the procedures are quite simple. In fact, as you can look up in appendix A, this hope was proved to be well-founded: the verification with the KIV system could be done with a high degree of automation and in about half an hour. Even concerning the efficiency aspect, our implementation looks quite nice: the amount of time required by VALIDITY-CHECKER is linear in the size of the input formula.[14]

So we have done our job and the customer will be enjoyed about the results, especially concerning efficiency and the surprisingly small effort of time (which means money) needed for carrying out the verification. However, he will be frustrated about the fact that his *verified* prover insists that the formula $\neg\ (S_0 \wedge \neg\ S_1)$ is a valid one. Presumably, this causes a lasting damage of his confidence in formal methods for software engineering.

4 What has gone Wrong?

We see us confronted with the unpleasant situation that we have formally specified and verified a program, but which turns out to behave not as intended. What has gone wrong? Several potential reasons come to mind:

[13] Cf. e.g. the `defstruct` construct in the programming language LISP.

[14] However, this insight should make suspicious, since the validity check problem of propositional logic is known to be NP-complete (see e.g. [2]).

1. There is a defect in the informal specification we started from.
2. The formal specification does not correspond to the informal specification.
3. The verification is not correct, i.e. some 'proofs' are not real proofs.

We hope that the reader agrees with us, that the first potential reason can be ruled out: we have adopted the informal specification from ([4], p. 4-7) with very little modifications. Concerning the last potential reason we point out that a formal verification has been completely done in the KIV system (cf. appendix A). This gives us reason to trust the proofs.[15]

For the second potential reason we call to mind that we have constructed the formal specification very carefully, and quite close along the informal specification (see section 2). Indeed, reviewing the specification (again and again) leads to the conviction that there isn't any faulty axiom. So everything looks fine.

However, and this is just the thing we want to illustrate on the example, checking the *correctness* of the axioms is not enough for ensuring adequateness of the specification, but a kind of *completeness* has to be guaranteed too. To be more concrete, one has to make sure that the specification does not permit any non-intended interpretations. Otherwise there is no guarantee that an implementation which is proved to meet the specification, behaves actually as desired. This is exactly what has gone wrong in our formal development of the propositional logic prover: the implementation embodies a non-intended interpretation of the requirement specification.

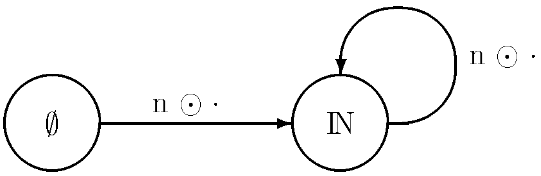
But the worst thing about it is, that (see end of section 2) we have been unaware of the whole extent of this ambiguity of the specification, and its far-reaching consequences. The ambiguity is caused by gaps, which are far from being completely obvious to see.[16]

In our example the root of all evil is in the specification of models, i.e. of finite sets on natural numbers. It permits non-intended interpretations, for instance an interpretation $\mathcal{A}_{\mathrm{MODEL}}$, which has exactly the two elements[17] $\emptyset$ and $\mathbb{N}$ in its domain. The insert operation $\odot$ is interpreted by $\mathcal{A}_{\mathrm{MODEL}}$ as follows: 'inserting' any element n in $\emptyset$ or $\mathbb{N}$ results in $\mathbb{N}$. This is illustrated in the following figure:

[15] Here we assume the correctness of KIV-produced proofs, i.e. the correctness of the kernel of the KIV system. For a discussion on how to guarantee correctness of deduction systems see [1, 16].

[16] The reader is invited to try to find them, before continuing.

[17] We hope that no confusion arises from using symbols, as $\emptyset$, on syntactical *and* on semantical level.



The membership predicate is interpreted by $\mathcal{A}_{\text{MODEL}}$ as one might expect:

$$\in_{\mathcal{A}_{\text{MODEL}}} \;:=\; \{\, (n, \mathbb{N}) \mid n \in \mathbb{N} \,\}\,.$$

The reader should convince oneself, that $\mathcal{A}_{\text{MODEL}}$ is in fact a model of the specification MODEL, i.e. that it does not conflict with any of the axioms.

Let $\mathcal{A}$ be a model of the overall requirement specification, which interprets the sub-specification MODEL as $\mathcal{A}_{\text{MODEL}}$. (It can be shown that such a model $\mathcal{A}$ actually exists.) Then, because of the axiom

$$\varphi \text{ is\_valid} \;\leftrightarrow\; \forall\, \mathrm{A}\,.\, \mathrm{A} \models \varphi$$

validity (i.e. the predicate is_valid) is interpreted by $\mathcal{A}$ as follows:

> *A formula is valid if and only if it holds in the empty model (which evaluates all atomic formulas to false) and in the total model (which evaluates all atomic formulas to true).*

Surely, this does not coincide with our intuitive understanding of validity, but this is exactly what the procedure VALIDITY-CHECKER checks.

5 Completing the specification

To summarize the above section, we have located the reason for the unexpected behavior of the verified implementation in the non-monomorphicity of the specification MODEL. So, the gaps in this specification should be filled by adding appropriate axioms. Here two questions arise: firstly, how can we find such appropriate axioms, and secondly, how do we come to know that we can stop adding further axioms, i.e. that the specification in question is already monomorphic. This section is intended to illustrate the difficulty of these questions; a possible answer to them is sketched in the conclusion.

The non-intended interpretation $\mathcal{A}_{\text{MODEL}}$ can be taken as hint which axioms to add to the specification of models. For instance, we may want to exclude interpretations, which contain models for which the membership relation is true for every natural number (e.g. in $\mathcal{A}_{\text{MODEL}}$ the model denoted by $\mathbb{N}$). This can be done by adding the axiom

$$\exists\, \mathrm{n}\,.\, \mathrm{n} \odot \mathrm{A} \neq \mathrm{A}.$$

However, is this enough? The answer is *no*! Again, there exist non-intended interpretations. For example, $\mathcal{B}_{\mathrm{MODEL}}$ with domain

$$\{\emptyset\} \cup \left\{ M \cup \{1\} \;\middle|\; M \subset \mathbb{N} \text{ finite} \right\}$$

In $\mathcal{B}_{\mathrm{MODEL}}$ inserting any natural number n to a set is interpreted as inserting not only n, but in addition the natural number 1. So $\{1\}$ is the only singleton set in $\mathcal{B}_{\mathrm{MODEL}}$. The membership predicate is interpreted by $\mathcal{B}_{\mathrm{MODEL}}$ as usual. Let $\mathcal{B}$ be a model of the overall requirement specification, which interprets the sub-specification MODEL as $\mathcal{B}_{\mathrm{MODEL}}$. Then, validity is interpreted by $\mathcal{B}$ as follows:

> *A formula is valid if and only if it holds in the empty model and in all models which evaluate* S_1 *to true.*

Implementing this interpretation would result in a program that meets the requirement specification, but again claims the validity of $\neg\ (\mathrm{S}_0 \wedge \neg\ \mathrm{S}_1)$. Taking $\mathcal{B}_{\mathrm{MODEL}}$ as hint, we may add a further axiom, e.g. :

$$\mathrm{n} \odot \mathrm{m} \odot \mathrm{A} = \mathrm{m} \odot \mathrm{A} \;\rightarrow\; \mathrm{n} = \mathrm{m} \vee \mathrm{n} \odot \mathrm{A} = \mathrm{A}$$

Eventually, this results in a monomorphic specification of models.[18] The reader is doing right to ask, how we get to this insight. We refer him to the next section.

6 Conclusion and Future Work

We have given an example that illustrates two points: firstly that, even if the requirement specification is constructed very carefully, one is (without taking further measures) not safe from ambiguity. Secondly, ambiguity (even if quite inconspicuous) can have unpleasant, inestimable consequences. The conclusion is, that without taking further measures one has to reckon on unpleasant, inestimable consequences.

What does this mean for application of formal methods in practice? Assume that customer and software developer agree about a formal requirement specification in the sense indicated in the introduction. Then the software developer is always sitting pretty, if he succeeds in verifying that his software meets the specification. However, as illustrated above, the customer can not be sure that this software behaves as he has intended. Therefore, in order to be safe from bad surprises, the customer should insist on a monomorphic requirement specification.[19] This opinion is also taken e.g. in ([10], p. 1049):

[18] Adding only this axiom (and not $\exists$ n . n $\odot$ A $\neq$ A) to the specification MODEL would already result in a monomorphic specification.

[19] This requirement can be weakened (especially in order to facilitate efficient implementations). With this we have in mind, that gaps in requirement specifications must not be completely forbidden, but that it is sufficient to make them explicit.

> "A well-written software requirement specification (SRS) reduces the probability of the customer being disappointed with the final product. The SRS defines the external behavior of the system to be built unambiguously, so there can be no misinterpretation. If there is a disagreement between customer and developer concerning external behavior, it is worked out during the requirements stage, not during acceptance testing, when it is much more costly to correct. Unfortunately many developers prefer to keep the SRS fairly ambiguous in order to provide themselves with more flexibility during design. However, this flexibility significantly increases the customer's risk."

The question arises how to guarantee that a specification is monomorphic. Unfortunately, in general[20] monomorphicity is neither easy to see nor decidable at all. The set of all monomorphic specifications is not even effectively enumerable. However, it is possible to *prove* monomorphicity of a given specification, for example by meta-reasoning [11, 15]. Currently we investigate a method that reduces the task of proving monomorphicity to a task of proving certain properties of programs [18]. This allows one to directly employ well-established techniques known from software verification (as implemented e.g. in the KIV system).

Another important question concerns the construction of monomorphic specifications: how can a given specification be made monomorphic, i.e. how to find appropriate axioms which fill the gaps in it. Maybe, one can get valuable hints for appropriate axioms by trying to prove monomorphicity of the non-monomorphic specification and analyzing the subgoals where the proof attempt gets stuck.[21]

Acknowledgments

I am indebted to Thomas Fuchß, Wolfram Menzel, Wolfgang Reif, Gerhard Schellhorn, and Kurt Stenzel for valuable discussions. Finally, I would like to thank Martin Giese who assist me in carrying out the case-study in the KIV system. Thomas Fuchß and Elmar Habermalz have made helpful comments on an earlier draft of this paper.

[20] If one restricts oneself to freely generated data types enriched by algorithmic specifications, the things get much simpler, since in this case, determinism and totality of these algorithms are sufficient for monomorphicity.

[21] Adding some appropriate axioms will enable us to do some more proof steps, but again we may get stuck, and hope to get valuable hints from analyzing the open subgoals, etc. So, the monomorphicity proof and the specification are completed step by step and hand in hand. This method may be titled as *synthesis of specifications* or *specifying by proving.*

References

[1] D. Basin, F. Giunchiglia, and M. Kaufmann, editors. *Proceedings of the Workshop on Correctness and Metatheoretic Extensibility of Automated Reasoning Systems*, held in conjunction with CADE-12, Nancy, June 1994.

[2] E. Börger. *Computability, Complexity, Logic*, volume 128 of *Studies in Logic and the Foundations of Mathematics.* Elsevier Science Publishers B. V., 1989.

[3] R.S. Boyer and J.S. Moore. *A Computational Logic.* Academic Press, 1979.

[4] C.C. Chang and H.J. Keisler. *Model Theory*, volume 73 of *Studies in Logic and the Foundations of Mathematics.* Elsevier Science Publishers B. V., 1990.

[5] Th. Fuchß, W. Reif, G. Schellhorn, and K. Stenzel. Three selected case studies in verification. In M. Broy and S. Jähnichen, editors, KORSO*, Correct Software by Formal Methods*, Lecture Notes in Computer Science. Springer Verlag, 1995, to appear.

[6] V. Girratana, F. Gimona, and U. Montanari. Observability concepts in abstract data type specification. In *Proc. Internat. Symp. on Mathematical Foundations of Computer Science*, volume 45 of *Lecture Notes in Computer Science*, pages 567–578. Springer Verlag, 1976.

[7] D. Harel. *First Order Dynamic Logic.* Lecture Notes in Computer Science. Springer Verlag, 1979.

[8] M. Heisel, W. Reif, and W. Stephan. A dynamic logic for program verification. In *Logical Foundations of Computer Science*, volume 363 of *Lecture Notes in Computer Science*, pages 134–145. Springer Verlag, 1989.

[9] M. Heisel, W. Reif, and W. Stephan. Tactical theorem proving in program verification. In *Proceedings of the 10th International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pages 117–131. Springer Verlag, 1990.

[10] J. Marcianiak, editor. *Encyclopedia of Software Engineering.* John Wiley & Sons Inc., 1994.

[11] W. Reif. Correctness of full first-order specifications. In *4th Conference on Software Engineering and Knowledge Engineering.* Capri, Italy, IEEE Press, 1992.

[12] W. Reif. Correctness of generic modules. In *Symposium on Logical Foundations of Computer Science. Proceedings*, volume 620 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.

[13] W. Reif. The KIV-system: Systematic construction of verified software. In *Proceedings of the 11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.

[14] W. Reif. Verification of large software systems. In *Foundations of Software Technology and Theoretical Computer Science*, volume 652 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.

[15] W. Reif and A. Schönegge. Meta-level reasoning: Proving monomorphicity of specifications. In Deduktionstreffen 1994. Technical report. Technische Hochschule Darmstadt, Fachbereich Informatik, October 1994.

[16] A. Schönegge. Valid extensions of introspective systems: A foundation for reflective theorem provers. Technical Report 26/94, Universität Karlsruhe, Fakultät für Informatik, 1994.

[17] A. Schönegge. Verifying tactics in the KIV system: The tautology checker example, 1994.

[18] A. Schönegge. Proof obligations for monomorphicity. Technical Report 34/95, Universität Karlsruhe, Fakultät für Informatik, 1995.

[19] N. Shankar. Towards mechanical metamathematics. *Journal of Automated Reasoning*, 1:407–434, 1985.

[20] M. Wirsing. *Algebraic Specification*, volume B of *Handbook of Theoretical Computer Science*, chapter 13, pages 675–788. Elsevier Science Publishers B. V., 1990.

A Verifying the Implementation

In section 3 we have presented only a part of the implementation, namely the procedures (with data structures FORMULA-DATA, BOOL-DATA) implementing the is_valid predicate. In the following is presented, how the other sorts and operations of the requirement specification are implemented. The sorts and operations of the sub-specifications FORMULA and NAT are implemented by identity, i.e. by their corresponding sorts and operations of the data specifications FORMULA-DATA and NAT-DATA. The rest of the requirement specification is implemented as follows:

boolean	**implements**	model
EMPTY-SET	**implements**	$\emptyset$
INSERT	**implements**	$\odot$
MEMBER	**implements**	$\in$
VALIDITY-CHECKER-H	**implements**	$\models$
VALIDITY-CHECKER	**implements**	is_valid

Here VALIDITY-CHECKER and VALIDITY-CHECKER-H are the procedures presented in section 3. It remains to give the (fairly tricky) declarations of the procedures EMPTY-SET, INSERT and MEMBER:

```
EMPTY-SET (var b : boolean)
begin
  b := false
end.

INSERT (n : nat , $b_0$ : boolean ; var b : boolean)
begin
  b := true
end.

MEMBER (n : nat , $b_0$ : boolean ; var b : boolean)
begin
  b := $b_0$
end.
```

Now, what we want to prove here is that this implementation meets the requirement specification given in section 2, which is the case if all procedures terminate and exhibit the behavior specified in the axioms of the requirement specification.[22] This conditions can be formulated in dynamic logic [7, 8],

[22] See [12, 14] for details of the more general approach to correctness of modular software systems, that is taken in the KIV system.

which is done automatically in the KIV system. In the following we list the KIV-generated proof obligations. The first five of them express that the procedures terminate:

(1) termination of procedure EMPTY-SET:

$$\boxed{\big\langle \text{EMPTY-SET}\,(;\text{b})\big\rangle\ \text{true}}$$

(2) termination of procedure INSERT:

$$\boxed{\big\langle \text{INSERT}\,(\text{n}\,,\text{b}_0\,;\text{b})\big\rangle\ \text{true}}$$

(3) termination of procedure MEMBER:

$$\boxed{\big\langle \text{MEMBER}\,(\text{n}\,,\text{b}_0\,;\text{b})\big\rangle\ \text{true}}$$

(4) termination of procedure VALIDITY-CHECKER-H:

$$\boxed{\big\langle \text{VALIDITY-CHECKER-H}\,(\text{b}_0\,,\text{phi}\,;\text{b})\big\rangle\ \text{true}}$$

(5) termination of procedure VALIDITY-CHECKER:

$$\boxed{\big\langle \text{VALIDITY-CHECKER}\,(\text{phi}\,;\text{b})\big\rangle\ \text{true}}$$

The following conditions express that the procedures exhibit the behavior specified in the requirement specification: every axiom of the specifications MODEL and VALIDITY is translated into a proof obligation:

(6) procedures satisfy "$\emptyset \neq \text{n} \odot \text{A}$":

$$\boxed{\neg\ \big\langle \text{EMPTY-SET}\,(;\text{b}_1)\big\rangle\ \big\langle \text{INSERT}\,(\text{n}\,,\text{b}_0\,;\text{b}_2)\big\rangle\ \text{b}_1 = \text{b}_2}$$

(7) procedures satisfy "$\text{n} \odot \text{m} \odot \text{A} = \text{m} \odot \text{n} \odot \text{A}$":

$$\boxed{\begin{array}{l}\big\langle \text{INSERT}\,(\text{n}\,,\text{b}_0\,;\text{b}_1)\big\rangle\ \big\langle \text{INSERT}\,(\text{m}\,,\text{b}_1\,;\text{b}_2)\big\rangle\\ \big\langle \text{INSERT}\,(\text{m}\,,\text{b}_0\,;\text{b}_3)\big\rangle\ \big\langle \text{INSERT}\,(\text{n}\,,\text{b}_3\,;\text{b}_4)\big\rangle\ \text{b}_2 = \text{b}_4\end{array}}$$

(8) procedures satisfy "$\text{n} \odot \text{n} \odot \text{A} = \text{n} \odot \text{A}$":

$$\boxed{\begin{array}{l}\big\langle \text{INSERT}\,(\text{n}\,,\text{b}_0\,;\text{b}_1)\big\rangle\ \big\langle \text{INSERT}\,(\text{n}\,,\text{b}_1\,;\text{b}_2)\big\rangle\\ \big\langle \text{INSERT}\,(\text{n}\,,\text{b}_0\,;\text{b}_3)\big\rangle\ \text{b}_2 = \text{b}_3\end{array}}$$

(9) procedures satisfy "$\text{n} \in \text{A} \leftrightarrow \text{n} \odot \text{A} = \text{A}$":

$$\boxed{\big\langle \text{MEMBER}\,(\text{n}\,,\text{b}_0\,;\text{b})\big\rangle\ \text{b} = \text{true} \leftrightarrow \big\langle \text{INSERT}\,(\text{n}\,,\text{b}_0\,;\text{b})\big\rangle\ \text{b} = \text{b}_0}$$

(10) procedures satisfy "$\text{A} \models \text{S(n)} \leftrightarrow \text{n} \in \text{A}$":

$$\boxed{\begin{array}{l}\big\langle \text{VALIDITY-CHECKER-H}\,(\text{b}_0\,,\text{S(n)}\,;\text{b})\big\rangle\ \text{b} = \text{true} \leftrightarrow\\ \big\langle \text{MEMBER}\,(\text{n}\,,\text{b}_0\,;\text{b})\big\rangle\ \text{b} = \text{true}\end{array}}$$

(11) procedures satisfy "$A \models (\neg' \varphi) \leftrightarrow \neg\, A \models \varphi$":

$$\boxed{\begin{array}{l}\langle \text{VALIDITY-CHECKER-H}\,(b_0\,, \neg'\ \text{phi}\,; b)\rangle\ b = \text{true} \leftrightarrow \\ \neg\ \langle \text{VALIDITY-CHECKER-H}\,(b_0\,, \text{phi}\,; b)\rangle\ b = \text{true}\end{array}}$$

(12) procedures satisfy "$A \models (\varphi \wedge' \psi) \leftrightarrow A \models \varphi \wedge A \models \psi$":

$$\boxed{\begin{array}{l}\langle \text{VALIDITY-CHECKER-H}\,(b_0\,, \text{phi} \wedge' \text{psi}\,; b)\rangle\ b = \text{true} \leftrightarrow \\ \langle \text{VALIDITY-CHECKER-H}\,(b_0\,, \text{phi}\,; b)\rangle\ b = \text{true} \wedge \\ \langle \text{VALIDITY-CHECKER-H}\,(b_0\,, \text{psi}\,; b)\rangle\ b = \text{true}\end{array}}$$

(13) procedures satisfy "φ is_valid $\leftrightarrow \forall\, A\, .\ A \models \varphi$":

$$\boxed{\begin{array}{l}\langle \text{VALIDITY-CHECKER}\,(\text{phi}\,; b)\rangle\ b = \text{true} \leftrightarrow \\ \forall\ b_0\, .\ \langle \text{VALIDITY-CHECKER-H}\,(b_0\,, \text{phi}\,; b)\rangle\ b = \text{true}\end{array}}$$

The last condition expresses that the procedures implementing the constructors of the sort model satisfy the generation principle for the sort model:

(14) procedures satisfy "model **generated by** $\emptyset$, $\odot$":

$$\boxed{\langle \text{GENERATE-MODEL}\,(b\,;\,)\rangle\ \text{true}}$$

Here GENERATE-MODEL is a procedure, which has a terminating run exactly for those input values that can be constructed by finitely many applications of the procedures EMPTY-SET and INSERT. Such a procedure is automatically generated by the KIV system (n := ? and b_1 := ? are indeterministic assignments):

```
GENERATE-MODEL (b : boolean ; )
begin
   var $b_0$ : boolean in
   EMPTY-SET (; $b_0$);
   if $b_0 = b$ then skip
   else var n : nat, $b_1$ : boolean in
         n := ? ; $b_1$ := ?;
         GENERATE-MODEL ($b_1$ ; );
         INSERT (n , $b_1$ ; $b_0$);
         if $b_0 = b$ then skip else abort
end.
```

We briefly sketch the proofs for all the obligations **(1)** − **(14)**. The procedures EMPTY-SET, INSERT, and MEMBER contain no calls of recursive procedures, so they can always be unfolded. This unfolding is enough for proving **(1)**, **(2)**, **(3)**, **(6)**, **(7)**, **(8)**, and **(9)**. The proof of **(4)** works by structural induction[23] on the input formula phi and simple symbolic execution. While

[23] This induction is permitted because there is a corresponding generation principle in the specification FORMULA-DATA.

this symbolic execution exactly these calls of VALIDITY-CHECKER-H are unfolded, which have as input formula a term that is not merely a variable (i.e. which has some structure). **(5)** can be reduced to **(4)**. The proofs for **(10)**, **(11)**, and **(12)** work by symbolic execution (the same way as in the proof for **(4)**). In addition, for **(11)** the (already proven) theorem **(4)** about the termination of VALIDITY-CHECKER-H is useful. Proving **(13)** can be done by unfolding the definition of VALIDITY-CHECKER and employing that a boolean is either true or false. For **(14)** we have to show that for each input value b there is a terminating run of GENERATE-MODEL (b ;). This can be done by executing the procedure call and choosing b_1 = false in the random assignment. So the recursive call does not run in the **else**-case, and termination is trivial.

These proofs have been carried out in the KIV system. All together they took 175 rule applications, where the KIV system found 168 of them (i.e. 96%) on its own.[24] The remaining interactive proof steps are quite natural, especially they have one-to-one equivalences in the above informal proof sketches (e.g. inserting a lemma or choosing b_1 = false in the random assignment while proving **(14)**). No auxiliary lemmas were required.[25] Carrying out this case-study took about 2 hours of interactive work with the KIV system (1.5 hours for setting up the specification and implementation, and about 0.5 hours for the verification).

[24] This degree of automation can even be increased by making more extensive use of specific heuristics.

[25] Thus, this case-study is an extremely atypical application of the KIV system. See [5] for more typical and more impressing case-studies carried out within the KIV system.