Combining Theorem Proving and Model Checking — A Case Study*

Dennis Dams[1], Dieter Hutter[2], and Natalia Sidorova[3]

[1] Dept. of Electrical Eng., Eindhoven University of Technology,
PO Box 513, 5600 MB Eindhoven, The Netherlands
d.dams@tue.nl

[2] German Research Center for Artificial Intelligence,
Stuhlsatzenhausweg 3, D-66123, Saarbruecken, Germany
hutter@dfki.de

[3] Dept. of Math. and Computer Science, Eindhoven University of Technology,
PO Box 513, 5600 MB Eindhoven, The Netherlands
n.sidorova@tue.nl

Abstract. We report on a case study on the verification of the Bounded Retransmission Protocol in which the inductive theorem prover INKA was used to justify data abstractions. These abstractions arose in building a finite-state verification model, to be submitted to a model checker. Our initial experiments led us to equip INKA with new heuristics, after which the proofs went through without user interaction. We discuss the idea behind these heuristics and argue why we expect them to work as well in other cases of data abstractions.

1 Introduction

Automatic verification is a rapidly growing field with many industrial applications. One of the most popular and successful verification techniques is model checking. It is well-known that this technique is only applicable to relatively small finite-state systems. Industrial-size specifications/models (whose state space is often even infinite) can not be model checked in a direct way — a verification model of a system is model checked instead. The limitation on the state-space size implies that a verification model should be abstract enough. However, the more abstract the verification model is, the harder it is to relate it to the (specification or implementation of the) original,"real" system. Indeed, as argued in [12], inexperienced users are often inclined to specify details in their models that are redundant for verification purposes. The reason for this is that such details are also present in the real system, and by incorporating them into the verification model, the user's confidence in the faithfulness of the model with respect to the real system is increased. One could say that there exists a trade-off between the effort (by a human user) to relate the model to the real system and

* This research has been supported by the VIRES project (Verifying Industrial Reactive Systems, Esprit Long Term Research Project #23498).

the effort (by a model checker) to automatically verify the model. Introducing property-preserving abstractions is a way both to allow a human user to make a model detailed and to produce a verification model that is "model-checkable". Abstraction techniques extend the applicability of model checking in this sense.

Abstraction, intuitively, means replacing one semantical model by an abstract, in general, simpler one. In addition to the requirement that an abstract (verification) model should have a smaller state space than the concrete (implementation) one, the abstraction needs to be *safe*, which means that every property checked to be true on the abstract model, holds for the concrete one as well. This allows the transfer of positive verification results from the abstract model to the concrete one.

There are several approaches to constructing abstractions. One is to use abstractions which are safe by (automatic) construction [2, 9]. Here, we assume that the abstraction is specified by the user. A similar approach is taken in the Bandera tool set [1]. This way, the particular insight of the system's designer may be exploited in constructing an appropriate abstraction, thus opening the road for certain "clever" abstractions that are less likely to be constructed automatically. The danger, however, is that such a hand-crafted abstraction is not necessarily safe with respect to the property to be verified. The user is left with a considerable task to argue the faithfulness of the abstraction. Such an argument is in most practical verification cases an informal one. Given the observation that the probability of errors for such a task increases with the problem size, the need for a formal and automated approach to the justification of abstractions is clearly apparent.

Our goal is to provide a mechanism for establishing the correctness of user-constructed abstractions. Here, we investigate the use of *theorem proving* for this purpose. To evaluate the applicability of such an approach, we consider a model of the Philips' Bounded Retransmission Protocol (BRP) [7]. BRP is a variant of the Alternating-Bit Protocol, where only a bounded number of retransmissions of packets is allowed and time-outs are used to detect packet loss. In the concrete system, packets are lists of natural numbers. That means that the system is infinite. Therefore, we apply an abstraction in order to arrive at a finite-state verification model. The whole variety of lists is represented there by 6 abstract lists. Though the abstraction we use seems to be simple, defining even relatively simple functions on abstract lists is error-prone because of the considerable number of cases that should be taken into account. We took an attempt to prove its correctness with the INKA (v 4.1) theorem prover [14]. INKA is a (full) first-order theorem prover with induction based on the explicit induction paradigm. It provides also definition principles for generated datatypes, which are encoded as sorts of the underlying order-sorted calculus, and algorithmic function specifications, which are used to synthesize new induction schemes (cf. [19]). Throughout this paper, we will use the INKA input syntax to describe the specification of our running example.

It turns out that the arising proof obligations are challenging examples for theorem proving in general. INKA needed to be tuned before it could automatically deal with the safety proofs; several heuristics of INKA strategies were improved for that purpose. We believe that some of our conclusions about the theorem-proving strategies drawn from the BRP case study do not depend on the particularities of BRP, and thus are valid for a wider range of safety proofs.

The paper is organized as follows. In Section 2, we describe a method for applying data abstractions in practice. Section 3 is devoted to the case study on the Philips' Bounded Retransmission Protocol. The process of tuning the theorem prover INKA to deal with safety proofs is described in Section 4. Finally, we conclude in Section 5 with evaluating the perspectives of the described approach to safety proofs.

2 Applying Data Abstractions

Currently, model checkers provide some facilities for (automatic) reducing a state space, like partial-order reduction techniques. These techniques deal mainly with the control flow of a model. On the contrary, data (values stored and transmitted in a system), whose domain is often infinite or very large, are not handled by them; it is a task of a user to present data in a verification model in a finite form of reasonable size. Depending on a property to verify, the actual values of data may sometimes be ignored or replaced by some abstract values. Abstract data types and encapsulation techniques, whose usage is growing, facilitate introducing data abstractions (by a user). In an abstract model, the operations on data are mimicked by new ones on the abstract data. Besides decreasing the state space of the system, the main requirement for an abstraction is that the abstract system behaviour should correctly reflect the behaviour of the original system with respect to a verification task in the sense that (1) an abstraction should capture all essential points in the system behaviour, i.e., be not "too abstract", and (2) an abstraction should be safe.

The concept of safe abstraction is well-developed within the *Abstract Interpretation* framework [5]. The requirement that Abstract Interpretation puts on the relation between the concrete model and its safe abstraction can be formalized as a requirement on the relation between the data operations of the concrete system and their abstract counterparts, as follows. Every value of the concrete state space is mapped by the *abstraction function* α into an abstract value which, intuitively, "describes" the concrete value. As an example consider the abstraction of integers into their signs in which -3 is mapped by α into **neg**. For every operation (function) f_{conc} on the concrete state space, an abstraction f_{abs} needs to be defined which "mimics" f_{conc}. For example, if the concrete operation is the squaring of an integer, then its abstraction is the function over $\{\mathbf{neg}, \mathbf{pos}\}$ that maps every input to **pos**. In general, the abstraction can be nondeterministic. For example, addition $(+)$ over the integers is abstracted into an operation $(+_{abs})$ such that $\mathbf{pos} +_{abs} \mathbf{neg}$ may yield **pos** or **neg** nondeterministically. This is formally captured by letting f_{abs} be a function into the powerset over the domain of abstract values. The requirement of mimicking is then formally phrased as:

$$\forall x : \alpha(f_{conc}(x)) \in f_{abs}(\alpha(x))$$

In the following we call this the *safety statement*. A safe abstract system is, intuitively, a system whose behaviour (the set of all transitions) is a superset of the concrete system behaviour.

Working within the *Abstract Interpretation* framework guarantees the preservation (in the direction from the abstract to the concrete model) of the truth of all formulae of $\Box L_\mu$ (i.e., all formulae of the μ-calculus without negation and containing only the $\Box$ operator) [18].

Usually, safety statements are not trivial to prove by hand; therefore, designers tend to rely on their intuition and common sense instead of performing costly proofs. We approach this problem with *automatic theorem-proving*. The concrete and abstract definitions of data types and functions together with the abstraction function form the input for a theorem prover (Fig.1). The safety statements are proof obligations. If the theorem prover succeeds in proving the obligations (preferably completely automatically, or with little human interaction) then the abstraction is safe and it can be applied to the model. If not, some information (like a counter-example) is given to indicate to a designer what causes the error. Then, the abstraction is corrected and safety of this new abstraction is checked.

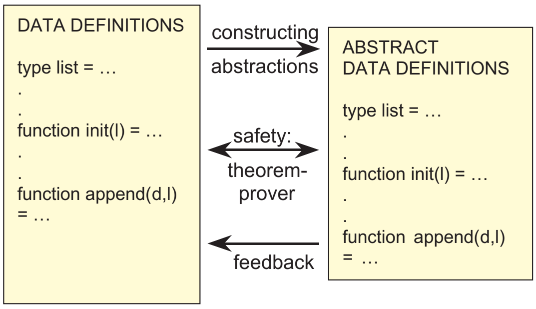

Fig. 1. Scheme of proving safety statements

When a safe data abstraction is obtained, *verification* of the model against a property can start. First, the abstract data type and function definitions are inserted into the system specification substituting the concrete definitions of the original model. Next, the abstract model together with the property to be verified are run through a verifier (Fig. 2). If the answer is positive, i.e., the property holds for the abstract model, it holds for the concrete model as well and the verification of this property is completed. If not, the model checker gives a trace showing a property violation. Now, the designer should check whether this trace represents some real behaviour of the concrete system or whether it is a behaviour that is added with the abstraction. In the first case, the conclusion of an error in the model or in the property can be drawn; after correcting it, the model checker can be run again (with the same data abstraction). In the second case, the abstraction should be refined (also proving the safety statements for the new abstraction) and the refined model is then model-checked.

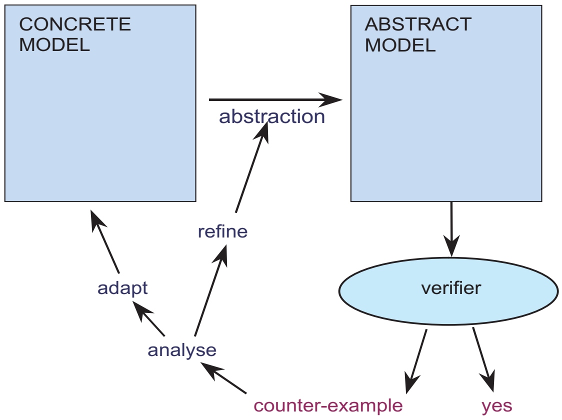

Fig. 2. Scheme of verification

The main problem to be solved in order to bring the described approach into practice is how to simplify the process of proving safety statements while hiding from the user as many technical details as possible. We propose to adapt a theorem prover to the special kind of proofs that one needs to provide for safety obligations. Inductive theorem provers are very appropriate for this purpose. Indeed, data types and functions on data are naturally defined using a recursive style. Hence, one needs induction in proofs dealing with them. In our experiments described further, we used the automatic inductive theorem-prover INKA [14].

3 Abstracting the Bounded Retransmission Protocol

To evaluate the practical applicability of our approach we applied it to a case study, which was the Philips' Bounded Retransmission Protocol (BRP) [7]. BRP is a variant of the Alternating-Bit Protocol, where only a bounded number of retransmissions of packets is allowed and time-outs are used to detect packet loss.

3.1 Specification Language

As mentioned above, we use the INKA input language to describe the specification of concrete and abstract data types and operations. Finitely generated data types are specified by the declaration of their constructor functions. Special instances of this definition principle are freely generated data types satisfying the unique factorization property, i.e. there is a *unique* syntactical representation (constructor term) for each denoted object. For example,

```
structure 0, s(p:nat):nat
```

specifies the set of natural numbers based on the constructor functions 0 and s together with its selector function p. An enumeration data type is a freely generated data type with finite domain which gives rise to an implicit definition of a well-founded ordering on its denoted objects:

```
enum structure t, f:bool
```

INKA provides also a definition principle to specify functions in a constructive way. Each so-called *algorithm* is a set of conditional equations (equivalences). The sum of two numbers can be specified as follows.

```
function plus(x:nat, y:nat):nat =
 if x = 0 then y
 if x = s(p(x)) then s(plus(p(x), y))
```

The idea is that these sets should denote confluent and Noetherian rewrite systems in order to evaluate each ground term into a constructor ground term. Using the specification of `plus`, for instance, we are able to calculate each individual sum of two constructor ground terms. However in order to guarantee this property in general, the "termination" of the specified algorithms have to be proved which is done by using a generalizion of Walther's approach [19]. Once INKA proved an algorithm of f to be terminating, its recursion ordering gives rise to a new induction scheme. Roughly speaking, each base case of f forms a base case of the inductive formula while each recursively defined case forms an induction step in which each recursive call in the definition gives rise to an induction hypothesis. For instance, proving that `plus` terminates because of its first argument, gives rise to the following induction scheme

$$\frac{x = 0 \rightarrow \Phi(X) \wedge \forall x : nat\ (X = s(p(X)) \rightarrow (\Phi(p(X)) \rightarrow \Phi(s(p(X)))))}{\forall x : nat\ \Phi(X)} \tag{1}$$

3.2 Data Types and Operations in BRP

In the concrete system, packets are lists of natural numbers. That means that the system is infinite. Therefore, in order to arrive at a finite verification model, a data abstraction should be introduced. The operations on lists used in the model are removing the head of a list, appending an element to a list, deciding whether one list is a prefix of another one, etc. Definitions of the data type and an example of a data operation in the concrete model are given below. (Note that both the data type and the operation are defined recursively.)

```
structure nil, cons(car:nat, cdr:list):list;

function tcons(p:nat, l:list):list =
 % tcons appends a natural p at the tail of list l.
 if l = nil then cons(p, nil)
 otherwise cons(car(l), tcons(p, cdr(l)));
```

3.3 Introducing a Data Abstraction

An essential property of the protocol is the following: "If the sender is transmitting a list l ("the input list") and the first element of l arrives at least once at the receiver, then the sequence of elements that the receiving client gets ("the output list"), forms a prefix of l".

It can be shown that this property holds if in a list of non-repeating naturals the following properties hold:

1. for any two values e_1 and e_2 on positions i and j resp. in the input list, with $i < j$, either e_2 does not occur in the output list, or e_1 and e_2 occur in the output list on positions i', j' resp., with $i' < j'$.
2. For any two values e_1 and e_2 on positions i' and j' resp. in the output list, with $i' < j'$, e_1 and e_2 occur on positions i and j resp. in the input list with $i < j$.

That gives the following idea of an abstraction to apply: one should distinguish two natural numbers $p1, p2$, which are abstracted into `e1`, `e2` respectively, while all the other naturals are non-distinguishable and they are abstracted into an abstract element `ne`. Thus we define

```
enum structure e1, e2, ne : anat;
```

An arbitrary list of naturals is represented by its abstracted head element, an abstract representation of the whole list, where the last one is of the form `epsl`, `e1l`, `e2l`, `e1e2l`, `e2e1l`, `error`, and, furthermore, the information whether it is an empty or a one-element list.

Here, `error` is an abstraction for an "incorrect" list, i.e. a list with duplicated elements (the assumption for the sender side is that only lists of non-repeating naturals are sent, but one need to assure that no repetition takes place at the receiver side as well); `epsl` represents correct lists where no `e1` and no `e2` occur, `e1l` (`e2l`) is a representation of correct lists where only `e1` (`e2` resp.) occurs; `e1e2l`, `e2e1l` represent lists where both `e1` and `e2` occur, in the corresponding orders. The description of abstract data types `atail` (abstract representation of the list content) and `alist` (abstract list) is as follows:

```
enum structure epsl, e1l, e2l, e1e2l, e2e1l, error : atail;

structure quad(is.cons:bool, is.one:bool, hd:anat, tl:atail) : alist;
```

Given these definitions of the data type abstractions, we are now able to define the abstraction functions `a_nat`, mapping numbers to their abstractions, and `a_list`, mapping a concrete list to its abstract representation, using auxiliary functions `a_cons` and `a_tail`. Note that not a recursion but an intricate case analysis is used there (compare to the concrete case).

```
function a_nat(a:nat) : anat =
 if a = p1 then e1
 otherwise {if a = p2 then e2
            otherwise ne} ;

function a_cons(x:anat, y:atail) : atail =
 if x = ne then y
 if x = e1 and y = epsl then e1l
 if x = e1 and y = e2l then e1e2l
 if x = e2 and y = epsl then e2l
 if x = e2 and y = e1l then e2e1l
 otherwise error;

function a_tail(l:list) : atail =
 if l = nil then epsl
 otherwise a_cons(a_nat(car(l)), a_tail(cdr(l)));

function consP(l:list) : bool =
 if l = nil then t
 otherwise f;

function oneP(l:list) : bool =
 if l = nil then f
 otherwise {if cdr(l) = nil then t
            otherwise f};

function a_list(l:list) : alist =
 quad(consP(l), oneP(l), a_nat(car(l)), a_tail(l));
```

Let us return to the function `tcons` which attaches an element at the tail of a list. In the following we define its corresponding abstract function `a_tcons` with the help of the auxiliary functions `a_tcons1` and `aux`.

```
function a_tcons1(e:anat, tl:atail):atail =
 if e = ne then tl
 if e = e1 and tl = epsl then e1l
 if e = e1 and tl = e2l then e2e1l
 if e = e2 and tl = epsl then e2l
 if e = e2 and tl = e1l then e1e2l
 otherwise error;

function aux(e: anat, l:alist):anat =
 if consP(l) = ff then e
 otherwise hd(l);
```

```
function a_tcons(e: aelem, l:alist):alist =
 quad(tt, not(consP(l)), aux(e, l), a_tcons1(e, tl(l)));
```

Now, with the concrete and abstract data types and operations defined, the task is to prove that the abstraction is safe. The process of proving is the subject of the next section.

4 Proving the Safety of Abstractions

To prove the safety of an abstraction α we have to guarantee that for each function f_{conc} in the concrete space there is an abstract function f_{abs} which mimics f_{conc}, i.e.

$$\alpha(f_{conc}(\overline{X})) \in f_{abs}(\alpha(\overline{X})) \tag{2}$$

In our running example, we will prove that `a_tcons` on abstract lists mimics `tcons` on concrete lists. In this case the abstraction is deterministic, i.e. `a_list` maps concrete lists into abstract lists which allows us to simplify and instantiate the condition (2) to

$$\texttt{a\_list}(\texttt{tcons}(n, l)) = \texttt{a\_tcons}(\texttt{a\_nat}(n), \texttt{a\_list}(l)). \tag{3}$$

In order to verify the condition (3) we have to prove properties of lists which gives rise to inductive proofs. Therefore we tackled this and similar proof obligations using the inductive theorem prover INKA. The abstraction we used seems to be rather simple, and one could say that no formal proof is needed here. However, on the other hand, case analyses arising in the definitions on the abstract side are sometimes rather complicated. That leads to errors in function definitions. Proving safety with INKA, we discovered several errors of that type.

Nevertheless, the verification of BRP was not our prime goal. First of all, the intention was to find out what kind of difficulties one would have in using an inductive theorem prover for safety proofs. The BRP example showed quite a lot of them. When starting our study, INKA was not able to prove most of the non-trivial examples completely automatically — in most cases appropriate induction schemes or case analyses had to be provided by the user. Examining the failures of INKA, we improved various heuristics with respect to computing induction schemes and generalization which enable INKA now to prove most of the examples without any user interaction.

In this section we illustrate the improvements of INKA with the help of our running example. Although these improvements were triggered by the arising proof obligations, we like to emphasize that they do not incorporate specific application knowledge but are general heuristics which are also useful in other domains and are now part of the general system.

4.1 Proof Search

In INKA the proof search is organized in various phases. In a first phase simplification rules are applied to the given problem. These rules are automatically generated by the given definitions of functions and predicates and allow the system to unfold definitions as long as the conditions of the individual cases are satisfied. Tackling the proof obligation (3), INKA unfolds, for instance, the (non-recursive) definitions of `a_list` and `a_tcons` which results in the following formula.

$$\begin{aligned} &\mathtt{a\_tail}(\mathtt{tcons}(n,l)) = \mathtt{a\_tcons}(\mathtt{a\_nat}(n), \mathtt{a\_tail}(l)) \\ \wedge\ &\mathtt{a\_nat}(\mathtt{car}(\mathtt{tcons}(n,l))) \\ &= \mathtt{aux}(\mathtt{a\_nat}(n), \mathtt{quad}(\mathtt{consP}(l), \mathtt{oneP}(l), \mathtt{a\_nat}(\mathtt{car}(l)), \mathtt{a\_tail}(l))) \\ \wedge\ &\mathtt{oneP}(\mathtt{tcons}(n,l)) = \mathtt{not}(\mathtt{consP}(l)) \wedge \mathtt{consP}(\mathtt{tcons}(n,l)) = t \end{aligned} \tag{4}$$

In a next step INKA applies first-order theorem proving techniques in order to prove (4) without induction. As these heuristics fail, INKA backtracks to (4) and speculates about an appropriate induction scheme. The presence of an induction rule constitutes an infinite branching point as in principle we may have to speculate an arbitrary lemma which has to be proven by induction in order to push our proof forward. To overcome this problem, INKA selects the simplified problem (e.g. (4)) as such a lemma and formulates an appropriate induction formula according to the recursion orderings of the occurring function symbols. In the original version of INKA, the recursion orderings of `tcons` and `a_tail` suggested the use of a structural induction to prove (4). However, INKA failed to prove the step case. The failure was caused by the presence of a function call $\mathtt{oneP}(l)$. Since `oneP` is non-recursively defined, unfolding its definition inside the induction conclusion of the step case resulted in a failure to match the corresponding induction hypothesis.

4.2 Induction vs. Case Analysis

To overcome such situations we improved INKA's heuristics to select appropriate induction schemes by considering also pure case analyses. While different occurrences of recursively defined functions suggest induction schemes, now also occurrences of non-recursively defined functions suggest case analyses. Thus analyzing a problem like (4), INKA obtains in general a set of different suggested induction schemes and case analyses. Based on this computed set, a scheme has to be computed in order either to formulate an induction formula or to perform a case analysis. Heuristics are used to assess the suggested schemes according to the probability that their use will push the proof further.

In case of induction schemes this is done by analyzing the abstract proof of the induction step. To describe these heuristics in more detail, let us first illustrate how INKA tries to prove induction steps. INKA incorporates *difference reduction techniques* [16] generalizing the idea of rippling [4]. Syntactical differences between induction hypothesis and induction conclusion are used to guide the manipulation of the formula. Highlighting differences by hatching them, the problem of such an induction step has the following form

$$l = \mathtt{cons}(\mathtt{car}(l), \mathtt{cdr}(l)) \rightarrow (\Phi(\mathtt{cdr}(l)) \rightarrow \Phi(\mathtt{cons}(\mathtt{car}(l), \mathtt{cdr}(l)))) \tag{5}$$

Inside the conclusion the non-hatched parts, the so-called *skeleton*, are the common parts of hypothesis and conclusion while hatched ones denote the differences or so-called *wave-fronts* of both. To enable the application of the hypothesis we have to move (i.e. "ripple") these wave fronts outside until we obtain the following formula

$$l = \mathtt{cons}(\mathtt{car}(l), \mathtt{cdr}(l)) \rightarrow (\Phi(\mathtt{cdr}(l)) \rightarrow \Psi(\Phi(\mathtt{cdr}(l)))). \tag{6}$$

This is done by using specific rewrite rules which allow us to monitor how the differences between hypothesis and conclusion will change when applying such a rule. Therefore the differences of left- and right-hand side of a rule are hatched, like for instance in

$$\mathtt{tcons}(X, \mathtt{cons}(Y, Z)) = \mathtt{cons}(Y, \mathtt{tcons}(X, Z)) \tag{7}$$

Matching hatched parts of the rewrite rule only with hatched parts of the conclusion we can easily determine how the differences of conclusion and hypothesis will change when applying this rule. In particular, we are interested at which positions — relative to the skeleton — wave-fronts occur. We assume to make progress in the proof search if we succeed to move wave-fronts outside.

Rippling allows for a hierarchical proof planning if we abstract from the concrete shape of wave fronts and consider only their position relative to the skeleton (cf. [15, 17] for details). To prove an induction step we have to *move* the wave-fronts located at the occurrences of the induction variables towards the top level of the conclusion[1]. If we abstract from the concrete form of the differences and denote occurrences of differences only by a $\bullet$ then we obtain the following "abstract proof":

$$\Phi(f_1(\ldots f_n(\bullet(x)))) \rightarrow \Phi(f_1(\ldots \bullet(f_n(x)))) \rightarrow \ldots \rightarrow \bullet(\Phi(f_1(\ldots f_n(x)))). \tag{8}$$

The abstracted rewrite rules, used in this proof, have the following form

$$f_i(\bullet(x)) = \bullet(f_i(x)) \tag{9}$$

In order to determine the efficiency of an induction scheme, INKA performs the proof of the corresponding induction steps on this abstract level. The analysis of how wave fronts can be moved around within the conclusion reveals information about appropriate induction variables and thus, about appropriate combinations of available induction schemes. In our example, the recursive definitions of `tcons`

[1] There are also other schemes how to prove induction steps in the presence of universally or existentially quantified, non-induction variables (see [15] for details)

and `a_tail` suggest possible (structural) induction schemes on l. However, the abstract proof of the corresponding induction step fails because there is no abstract wave rule of the form $\texttt{oneP}(\boxed{\bullet}(l)) = \boxed{\bullet}(\texttt{oneP}(l))$ inside the axiomatization.

Analyzing the occurrences of the non-recursively defined functions `oneP` and `consP` in (3) suggest case analyses on l. The case analysis of `oneP` subsumes the one of `consP`, i.e. each case of the definition of `oneP` is covered by exactly one case of the definition of `consP`. Since these occurrences of `oneP` and `consP` also block the rippling process, INKA selects a case analysis on $\texttt{oneP}(l)$ instead of an induction on l. The rational behind this decision is that after performing the case analysis and simplifying the resulting formula, these symbol occurrences will disappear since the system is now able to unfold the definitions of `oneP` and `consP`. We obtain three cases considering $l = nil$, $l = \texttt{cons}(\texttt{car}(l), nil)$ and $l = \texttt{cons}(\texttt{car}(l), \texttt{cdr}(l)) \wedge \texttt{cdr}(l) = \texttt{cons}(\texttt{car}(\texttt{cdr}(l)), \texttt{cdr}(\texttt{cdr}(l)))$. While the first two cases can be easily proved without induction, induction is needed in order to prove the third case. In general, INKA performs explicit case analyses for various reasons:

- to unblock the rippling process as described in our example,
- to remove occurrences (by unfolding their definitions) of functions which are maximal with respect to the definition ordering (i.e. f is less than g if the algorithm of g relies on the algorithm of f), or
- to identify different calls of the same function by applying its (recursive) definition (e.g. when proving $\forall x, n\!:\!\texttt{nat}\ \forall y\!:\!\texttt{list}\ x \in y \rightarrow x \in \texttt{cons}(n, y)$).

4.3 Generalization

As mentioned above, proving theorems by induction usually requires the speculation of intermediate lemmata. To guide this speculation, generalization techniques (see e.g. [3] for an introduction) are used to reformulate (and simplify) problems before applying an induction scheme. Instead of proving a formula like $\forall x, y : S\ \Phi(x, f(y))$ we may also prove a generalized formula like $\forall x : S\ \forall z : S'\ \Phi(x, z)$ by induction.

INKA uses generalization techniques, for instance, to replace selector terms like $\texttt{car}(l)$ or $\texttt{cdr}(l)$ by new variables provided the property $l = \texttt{cons}(\texttt{car}(l), \texttt{cdr}(l))$ can be derived from the given formula. Thus, a formula $\forall l\!:\!\texttt{list}\ \Phi(\texttt{car}(l), \texttt{cdr}(l), l)$ is generalized to $\forall m\!:\!\texttt{nat}\ \forall n\!:\!\texttt{list}\ \Phi(m, n, \texttt{cons}(m, n))$. The reason for this heuristics is that in this case the resulting "generalized" formula is equivalent to the original one and we do not have to backtrack this generalization. This idea is formalized by a generalized notion of *covering* (cf. [10]):

A term $t(\overline{X})$ of a sort S is covering *wrt. S if $\forall y\!:\!S\ \exists \overline{X}\!:\!\overline{S}\quad t(\overline{X}) = y$ holds in the given theory.*

Obviously, both $\texttt{car}(l)$ and $\texttt{cdr}(l)$ are covering wrt. `nat` and `list` respectively. Now, given a formula Φ which contains covering terms $t_1(\overline{X}), \ldots, t_n(\overline{X})$ and no variables in $\overline{X}$ occur elsewhere in Φ, it is safe to generalize Φ by replacing each occurrence $t_i(\overline{X})$ by a corresponding fresh ($\forall$-quantified) variable z_i. Applying this technique to the simplified third case of our problem results in the formula:

$$\begin{aligned} &\mathtt{a\_cons}(\mathtt{a\_nat}(i), \mathtt{a\_cons}(\mathtt{a\_nat}(j), \mathtt{a\_tail}(\mathtt{tcons}(n, k)))) \\ &= \mathtt{a\_tcons1}(\mathtt{a\_nat}(n), \mathtt{a\_cons}(\mathtt{a\_nat}(i), \mathtt{a\_cons}(\mathtt{a\_nat}(j), \mathtt{a\_tail}(k)))) \end{aligned} \tag{10}$$

To improve the behavior of INKA, we incorporated heuristics to detect covering terms of the form $f(\overline{X})$ with f being a function with finite range. Using these techniques, INKA automatically recognizes that a term like $\mathtt{a\_nat}(i)$ is covering and can be safely generalized. As a consequence, formula (10) is generalized to[2]:

$$\begin{aligned} &\mathtt{a\_cons}(x, \mathtt{a\_cons}(y, \mathtt{a\_tail}(\mathtt{tcons}(n, k)))) \\ &= \mathtt{a\_tcons1}(\mathtt{a\_nat}(n), \mathtt{a\_cons}(x, \mathtt{a\_cons}(y, \mathtt{a\_tail}(k)))) \end{aligned} \tag{11}$$

Applying structural induction on k, suggested by the definition of `tcons`, we obtain the following induction hypothesis

$$\begin{aligned} &\mathtt{a\_cons}(X, \mathtt{a\_cons}(Y, \mathtt{a\_tail}(\mathtt{tcons}(N, \mathtt{cdr}(k))))) \\ &= \mathtt{a\_tcons1}(\mathtt{a\_nat}(N), \mathtt{a\_cons}(X, \mathtt{a\_cons}(Y, \mathtt{a\_tail}(\mathtt{cdr}(k))))) \end{aligned}$$

which corresponds to the corresponding induction conclusion

$$\begin{aligned} &\mathtt{a\_cons}(x, \mathtt{a\_cons}(y, \mathtt{a\_tail}(\mathtt{tcons}(n, \mathtt{cons}(\mathtt{car}(k), \mathtt{cdr}(k)))))) \\ &= \mathtt{a\_tcons1}(\mathtt{a\_nat}(n), \mathtt{a\_cons}(x, \mathtt{a\_cons}(y, \mathtt{a\_tail}(\mathtt{cons}(\mathtt{car}(k), \mathtt{cdr}(k)))))). \end{aligned}$$

Rewriting this formula with the help of wave-rules from the definition of `a_tail` and `tcons` results in

$$\begin{aligned} &\mathtt{a\_cons}(x, \mathtt{a\_cons}(y, \mathtt{a\_cons}(\mathtt{a\_nat}(\mathtt{car}(k)), \mathtt{a\_tail}(\mathtt{tcons}(n, \mathtt{cdr}(k)))))) \\ &= \mathtt{a\_tcons1}(\mathtt{a\_nat}(n), \mathtt{a\_cons}(x, \mathtt{a\_cons}(y, \mathtt{a\_cons}(\mathtt{a\_nat}(\mathtt{car}(k)), \mathtt{a\_tail}(\mathtt{cdr}(k)))))) \end{aligned}$$

which allows us to apply the induction hypothesis on its left-hand side using a substitution $X \leftarrow y, Y \leftarrow \mathtt{a\_nat}(\mathtt{car}(k)), N \leftarrow n$.

$$\begin{aligned} &\mathtt{a\_cons}(x, \mathtt{a\_tcons1}(\mathtt{a\_nat}(n), \mathtt{a\_cons}(y, \mathtt{a\_cons}(\mathtt{a\_nat}(\mathtt{car}(k)), \mathtt{a\_tail}(\mathtt{cdr}(k)))))) \\ &= \mathtt{a\_tcons1}(\mathtt{a\_nat}(n), \mathtt{a\_cons}(x, \mathtt{a\_cons}(y, \mathtt{a\_cons}(\mathtt{a\_nat}(\mathtt{car}(k)), \mathtt{a\_tail}(\mathtt{cdr}(k)))))) \end{aligned}$$

Notice however, that we can not apply the induction hypothesis on the right-hand side of the conclusion because we were not able to ripple out its wave front $\mathtt{a\_cons}(x, \ldots)$. This gives rise to the speculation of a lemma which will allow us to deal with this blocked wave front. To formulate this lemma INKA generalizes the skeleton inside the blocked wave front $\mathtt{a\_cons}(y, \mathtt{a\_cons}(\mathtt{a\_nat}(\mathtt{car}(k)), \mathtt{a\_tail}(\mathtt{cdr}(k))))$ to a fresh variable u. Since $\mathtt{a\_nat}(n)$ is covering and n does not occur elsewhere after this generalization, also $\mathtt{a\_nat}(n)$ is replaced by a new fresh variable v which finally results in the speculated lemma

$$\mathtt{a\_cons}(x, \mathtt{a\_tcons1}(v, u)) = \mathtt{a\_tcons1}(v, \mathtt{a\_cons}(x, u)). \tag{12}$$

This lemma is easily proven by INKA using a case analysis suggested by `a_cons` and `a_tcons1`.

[2] Notice, that although $\mathtt{a\_nat}(k)$ is covering, its generalization is impossible because k occurs also inside $\mathtt{tcons}(k, z)$.

4.4 Nested Induction and Case Analysis

The approach we sketched above allows for an arbitrary nesting of applications of induction or case analysis. In order to prove our example, INKA started with a case analysis, then applied induction in these cases and finally formulated a lemma in the step case which was proven with the help of a case analysis. Obviously we need some tie-breaking rules to avoid infinite looping if INKA is not able to prove all arising subproblems. Therefore we use the (partial) definition ordering $<$ on function symbols mentioned above. INKA uses an extension of the corresponding multi-set ordering of $<$ to compare two arbitrary formulas. A nested induction is only allowed if the formula under consideration is less than the formula causing the previous application of the induction rule.

Summing up, the case study on verifying abstractions revealed that most of the problems are caused by the fact that these proofs require a sophisticated nesting of induction and case analysis. While induction is needed to reason about recursively defined functions, case analyses are used to unfold the definition of non-recursive functions. With the help of the improved heuristics, INKA is able to prove our example and other similar problems without any user interaction and without any lemmata speculated by the user. The overall time necessary to verify our running example was about 10 seconds on an Intel machine (PII, 266Mhz).

5 Conclusions and Future Work

We have presented a case study demonstrating the use of the inductive theorem prover INKA in order to verify the correctness of data abstractions. Such abstractions arise in the practice of model checking, when building a verification model of a system that is too large to be directly submitted to the checker. While model checkers are typically successful in handling complexity related to control and interaction, data types need to be replaced by abstract versions in order to avoid the state explosion. This abstraction step is usually an informal activity. Practical experience however has shown that it is error-prone, although (or maybe: because) in many cases it is simple.

In the methodology that we advocate, the justification of data abstractions is subjected to the scrutiny of formal methods. In order to check the safety of a particular data abstraction, the concrete data type, its abstracted version, and the safety statement are specified in the INKA input format, and INKA's inductive powers are then invoked to try and dismiss the proof obligation and any generated sublemmata. This way, the specification of abstractions appears as a programming activity, and the refinement of abstractions in case they are too coarse is a matter of debugging. We see this as a methodological advantage which could endorse the integration of this approach into model checkers.

Data abstractions as used in checking the Bounded Retransmission Protocol have been verified this way. Our initial experiments required manual interaction, but also indicated at which points INKA's strategies fell short. In particular, the alternation of case distinction and induction was problematic — not only in the case of the BRP but for safety proofs of data abstractions more generally, we believe. Consequently, INKA was equipped with a number of improved heuristics designed specifically to deal with such situations. As a result, all safety proofs went through automatically.

Currently, we are applying the same strategy to prove safety of data-abstractions for different examples, and initial results show that the tuned version of INKA also performs well for those. Data types and abstractions that are typical for certain classes of systems (like queues and "order-preservation" abstractions, as used in this paper, for communication protocols), once proven correct, may be stored so as to develop libraries of reusable abstractions.

The specifications of our case study are available on
`http://www.ics.ele.tue.nl/d̃ennis/SafetyProving/`.

Acknowledgments

Several discussions with Kai Baukus, Leszek Holenderski, Michael Siegel, Karsten Stahl, and Martin Steffen have helped us in the early stages of this work.

References

1. The Bandera project. See `http://www.cis.ksu.edu/santos/bandera/`.
2. S. Bensalem, Y Lakhnech and S. Owre, *InVeSt : A Tool for the Verification of Invariants*, In Conference on Computer Aided Verification CAV'98, LNCS 1427, Springer Verlag, 1998.
3. R.S. Boyer, J S. Moore, *A Computational Logic*, Academic Press, New York, 1979
4. A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill, *Rippling: a heuristic for guiding inductive proofs*, Artificial Intelligence, North Holland, 62:185–253, 1993.
5. P. Cousot, R. Cousot, *Abstract Interpretaion: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*, In 4th POPL, Los Angeles, CA, ACM, January 1977.
6. D. Dams, *Abstract Interpretation and Partition Refinement for Model Checking*, PhD Thesis, Eindhoven University of Technology, 1996.
7. D. Dams, R. Gerth, *The Bounded Retransmission Protocol Revisited (extended abstract)*, In Faron Moller, editor, Second International Workshop on Verification of Infinite State Systems (Infinity'97), volume 9 of Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers B.V. (North-Holland), 1997.
8. S. Graf, *Verification of a Distributed Cache Memory by Using Abstractions*, In Workshop on Computer-Aided Verification, CAV'94, Stanford, Springer Verlag, LNCS 818, 1994.
9. S. Graf, H. Saidi, *Construction of abstract state graphs with PVS*, In Conference on Computer Aided Verification CAV'97, Springer Verlag, LNCS 1254, 1997.
10. B. Gramlich, *Unicom: a refined completion based inductive theorem prover.*, In M. Stickl (ed.), 10th Int. Conf. on Automated Deduction (CADE 90), Springer Verlag, LNAI 449, 1990.
11. G. J. Holzmann, *Design and Validation of Communication Protocols*, Prentice Hall, 1991. Also: `http://netlib.bell-labs.com/netlib/spin/whatispin.html`

12. G. J. Holzmann, *Designing Executable Abstractions. Keynote address*, Proc. Formal Methods in Software Practice, March 1998.
13. G. J. Holzmann, M. H. Smith *Software Model Checking - Extracting Verification Models from Source Code*, In Formal Methods for Protocol Engineering and Distributed Systems, Kluwer Academic Publ., October 1999.
14. D. Hutter, C. Sengler, *INKA - The Next Generation*, In M. McRobbie J. Slaney (ed), 13th International Conference on Automated Deduction (CADE 96), Springer Verlag, LNAI 1104, 1996.
15. D. Hutter, *Synthesizing Induction Orderings for Existence Proofs*, In A. Bundy (ed), 12th International Conference on Automated Deduction (CADE 94), Springer Verlag, LNAI 814, 1994
16. D. Hutter, *Coloring terms to control equational reasoning*, In Journal of Automated Reasoning, Vol. 18, pp. 399-442, 1997.
17. D. Hutter, *Annotated Reasoning*, to appear in: Annals of Mathematics and Artificial Intelligence (AMAI). Special issue on Strategies in Automated Deduction, 2000
18. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, S. Bensalem, *Property Preserving Abstractions for the Verification of Concurrent Systems*, In Formal Methods in System Design, Kluwer Academic Publ., 6, 1-36, 1995.
19. C. Walther: On Proving the Termination of Algorithms by Machine. *Artificial Intelligence*, 71(1):101-157, 1994.