Bounded Relational Analysis of Free Data Types

Andriy Dunets, Gerhard Schellhorn, and Wolfgang Reif

Lehrstuhl für Softwaretechnik und Programmiersprachen,
Institut für Informatik,
Universität Augsburg,
86135 Augsburg Germany
{dunets,schellhorn,reif}@informatik.uni-augsburg.de
http://www.informatik.uni-augsburg.de/swt

Abstract. In this paper we report on our first experiences using the relational analysis provided by the Alloy tool with the theorem prover KIV in the context of specifications of freely generated data types. The presented approach aims at improving KIV's performance on first-order theories. In theorem proving practice a significant amount of time is spent on unsuccessful proof attempts. An automatic method that exhibits counter examples for unprovable theorems would offer an extremely valuable support for a proof engineer by saving his time and effort. In practice, such counterexamples tend to be small, so usually there is no need to search for big instances. The paper defines a translation from KIV's recursive definitions to Alloy, discusses its correctness and gives some examples.

Keywords: First-order logic, theorem proving, SAT checking, abstract data types, model checking, verification, formal methods.

1 Introduction

In our work we present an integration of an automatic procedure for finding finite counter examples or witnesses for first-order theories in the theorem prover KIV [4]. KIV supports both functional and state-based approaches to model systems. In this paper, we concern ourselves with the functional approach, which uses hierarchically structured higher-order algebraic specifications. More precisely, we are interested in the automation of its first-order part.

As first-order logic is undecidable we can construct either a decision procedure for decidable fragments or use an automated prover for full logic. Both approaches are useful for provable goals.

Since most of the time in interactive theorem proving is spent to find out why certain goals are not provable, an alternative approach is to try to disprove conjectures and to generate counter examples. Therefore, we were inspired by the automatic analysis method for first-order *relational* logic with *transitive closure* implemented in the Alloy Analyzer [10] and its successful application in the Mondex challenge by Ramananandro [15]. Alloy's algorithm handles the full first-order relational logic with quantifiers and transitive closure [11].

B. Beckert and R. Hähnle (Eds.): TAP 2008, LNCS 4966, pp. 99–115, 2008.
© Springer-Verlag Berlin Heidelberg 2008

Because formal theories in KIV are constructed using structured algebraic specifications, the sought-after automatic procedure involving Alloy Analyzer would represent a relational analysis of algebraic data types. A fundamental work on this topic was done by Kuncak and Jackson [12]. They present a method for the satisfiability checking of first-order formulas which is based on finite model finding, formulate essential properties which should be satisfied by an analyzed data structure and identify a class of the amenable formulas. A reduction from reasoning about infinite structures to reasoning about finite structures was achieved for a minimal theory consisting of selectors only.

In this paper we elaborate the results in [12] by extending the considered language from just selector functions to constructors and recursive functions as they usually occur in first-order theories. We apply these results to universally closed formulas in the KIV theorem prover. As a first step in this direction, the approach presented in this paper is confined to the analysis of recursive definitions over free data types[1], e.g. lists, stacks, binary trees. In our experiments we used Alloy Analyzer version 4.0 [10].

1.1 Related Work

There are different approaches of combining interactive methods with automated ones, which have in common the aim to strengthen interactive proving tools by adding automatic methods. One approach is to use automated theorem provers based on resolution or other calculi as a *tactic* in KIV to prove first-order theorems. A fundamental investigation of a conceptual integration that goes beyond a loose coupling of two proof systems was performed in [1] and some improvements on exploiting the structure of algebraic theories were presented in [16]. In [13] an automation procedure for a theorem prover is described which bridges numerous differences between Isabelle with its higher-order logic and resolution provers Vampire and SPASS (restricted first-order, untyped, clause form). In [7] a proof certification using theorem prover Isabelle/HOL for a decision procedure for the quantifier-free first-order logic in SMT-solver haRVey is described. The theorem prover is used to guarantee soundness of automatically produced proofs, while the automated tool is used as an *oracle*.

Nevertheless automated theorem provers are of limited use, since they do not support induction necessary to reason about algebraic types and recursive definitions. They are also applicable only for provable theorems, while most of the time in interactive theorem proving is spent on unsuccessful proof attempts.

For many applications knowing a counter model to a wrong assumption is as useful as knowing that a conjecture is true itself. This idea is realized in [6], where a proof procedure based on finite model finding techniques is designed for first-order logic. Reversely, [14] presents a so-called small model theorem, which calculates a threshold size for data types. If no counter examples are found at the threshold, the theorem guarantees that increasing the scope still produces no counter examples.

[1] Syntactically different terms built up from the constructors denote different values.

1.2 Outline

We provide some background on the theorem prover KIV and the specification of algebraic data types in Section 2. Section 3 gives a short overview of the Alloy Analyzer tool, the logic it uses and the analysis. Section 4 introduces in generating models for free data types in Alloy. Section 5, which is the central one, provides a detailed insight into an axiomatization of recursive functions for finite models in Alloy. In Section 6 we report on our first experiences from an application in KIV for an example, that has been analyzed earlier using KIV's own counter example generation. This is followed by conclusions and an outlook in Section 7. Throughout this work we use lists as a representative example of free algebraic data types.

2 Theorem Prover KIV

KIV is a tool for formal system development. It provides a strong proof support for all validation and verification tasks and is capable of handling large-scale theories by efficient proof techniques and an ergonomic user interface. Details on KIV can be found in [4,5].

2.1 Specification of Algebraic Data Types

The basic logic underlying the KIV system combines *Higher-Order Logic* (HOL) and *Dynamic Logic* (DL) [8], which allows to reason over imperative programs (partial and total correctness as well as program equivalence are expressible).

In this work we are particularly interested in the FOL part of the KIV system. The reason is, that in almost all proof tasks carried out interactively in KIV, whether in the basic logic or in extensions for temporal logic proofs [2], ASM specifications [19], statecharts [21,3] or Java program proofs [20], eventually a lot of first-order proof obligations arise. These are typically discharged using *simplifier rules*. Most simplifier rules are first-order lemmas which are automatically used for rewriting and other simplifications. In large case studies the number of used rewrite rules is often several thousands, many of them imported from the KIV library of data types.

Defining and proving such simplifier rules is therefore a regular task in interactive verification. Usually, some of these theorems are wrong on the first attempt, so a quick check that identifies unprovable ones is very helpful.

A theory in KIV describes data types, e.g. naturals, lists, arrays or records, which afterwards are used in DL programs. Theories are specified using *structured algebraic specifications*. To specify data structures adequately, in addition to first-order axioms we also need axioms for induction. Unfortunately an induction scheme cannot be specified by a finite set of first-order formulas. As a replacement *generation clauses* are used: s **generated by** $c_1, \ldots, c_n$, where s is a

```
generic data specification
   parameter elem
   list = [] |  . + . ( . .first : elem;  . .rest : list);
   variables
      x, y, z : list;
   order predicates  . < . : list x list;
end generic data specification

enrich list with
   functions
       . + . : list x list -> list;
       rev   : list -> list;
   axioms
      app-nil  : [] + x = x;
      app-cons : (a + x) + y = a + (x + y);
      rev-nil  : rev([]) = [];
      rev-cons : rev(a + x) = rev(x) + (a + []);
end enrich
```

Fig. 1. KIV specification of lists

sort, and $c_1, \ldots, c_n$ are its constructors. The simplest example of a generated sort are natural numbers: *nat* **generated by** $0, +1$.

A basic specification consists of three parts: a description of the signature, the axioms and the principles of induction. Figure 1 shows the specification of lists in KIV. It contains a *basic specification* of the sort *elem* (not shown), a *generic data specification* of lists and finally an *enrichment* of the list data specification by recursive functions *app* and *rev*. Line `list = [] |` ... generates four axioms specifying the free data type *list*. The first axiom is **generated by** clause which declares that the sort is generated by its constructors: `list` **generated by** `[],+`. From freeness the following axioms are generated. *selector* axioms: $(a+x).first = a$, $(a+x).rest = x$, *uniqueness* of constructors: $a+x \neq []$ and *injectivity* of constructors: $a + x = b + x \leftrightarrow a = b \wedge x = y$.

For free data types, axioms for the order predicate $<$, which corresponds to the subterm relation, are automatically included in the theory. In the enrichment we specify two recursive functions *rev* and *app* (overloaded + for *append*). These are defined by *structural recursion* over the first argument of a function.

3 Alloy Analyzer

In this section we introduce a logic which underlies this work. This logic is also used as an intermediate language to which Alloy input is translated and which is also handled by the Alloy algorithm. Although the logic is multi-sorted, for the sake of a better illustration we consider only two sorts: *elem* and *list*. In the

$$
\begin{aligned}
\textbf{sort} \; &::= list \mid elem \\
F \; &::= A \mid \forall x \in sort.\ F \mid \exists x \in sort.\ F \mid F_1 \wedge F_2 \mid \neg F_1 \\
A \; &::= (x_1, \ldots, x_n) \in R^n \mid x_1 = x_2 \\
R^n \; &::= r^n, n \neq 2 \\
R^2 \; &::= \; ^\wedge R_1^2 \mid r^2
\end{aligned}
$$

$$M = (L, E, \gamma), \qquad \alpha : Vars \rightarrow L \cup E$$

$$
\begin{aligned}
[\![\forall \text{ x}\in\text{list. F}]\!]^{M,\alpha} &\equiv \forall\, l \in L.\ [\![\text{F}]\!]^{M,\alpha'},\ \ \alpha' = \alpha[x := l] \\
[\![\exists \text{ x}\in\text{list. F}]\!]^{M,\alpha} &\equiv \exists\, l \in L.\ [\![\text{F}]\!]^{M,\alpha'},\ \ \alpha' = \alpha[x := l] \\
[\![\forall \text{ x}\in\text{elem. F}]\!]^{M,\alpha} &\equiv \forall\, e \in E.\ [\![\text{F}]\!]^{M,\alpha'},\ \ \alpha' = \alpha[x := o] \\
[\![\exists \text{ x}\in\text{elem. F}]\!]^{M,\alpha} &\equiv \exists\, e \in E.\ [\![\text{F}]\!]^{M,\alpha'},\ \ \alpha' = \alpha[x := o] \\
[\![F_1 \wedge F_2]\!]^{M,\alpha} &\equiv [\![F_1]\!]^{M,\alpha} \wedge [\![F_2]\!]^{M,\alpha} \\
[\![\neg F_1]\!]^{M,\alpha} &\equiv \neg[\![F_1]\!]^{M,\alpha}
\end{aligned}
$$

$$
\begin{aligned}
[\![(x_1, \ldots, x_n) \in R^n]\!]^{M,\alpha} &\equiv (\alpha(x_1), \ldots, \alpha(x_n)) \in [\![R^n]\!]^{M,\alpha} \\
[\![^\wedge R^2]\!]^{M,\alpha} &\equiv \{(x_1, x_2) \mid \exists n \geq 1.\ \exists l_1, \ldots, l_n \in L.\ \bigwedge_{i=1}^{n} (l_{i-1}, l_i) \in [\![R^2]\!]^{M,\alpha}\} \\
[\![r^n]\!]^{M,\alpha} &\equiv \gamma(r^n)
\end{aligned}
$$

Fig. 2. Syntax and Semantics for Relational Logic with Transitive Closure [11]

next section we will discuss an axiomatization of free data types in this logic, where we will use the specification of lists as a generic example.

3.1 Logic

The logic used by the Alloy analyzer is a first-order relational logic with transitive closure [11]. Figure 2 shows its syntax and semantics. The input language of Alloy has a very rich syntax, but here we stick only to the most essential part.

We consider two sorts: lists and elements. Formulas F can be constructed using universal as well as existential quantifiers. Atomic formulas A are defined using $\in$ operator on variables $x_1, \ldots, x_n$ for a n-ary relation R^n and using equality operator. Relation-valued expressions R^n are introduced using terminal symbols r^n. In case of binary relations R^2 a transitive closure operator can be applied: $^\wedge R^2$.

Other types of atomic formulas like $R_1^n \subseteq R_2^n$ or $R_1^n = R_2^n$ are provided by the Alloy's syntax which can be derived from the basic ones:

$$
\begin{aligned}
\textbf{setOp} \; &::= \cup \mid \cap \mid \setminus \\
A \; &::= R_1^n \subseteq R_2^n \mid R_1^n = R_2^n \\
R^n \; &::= R_1^n \ \textbf{setOp}\ R_2^n \mid R_1^k . R_2^{k'},\ \ where\ k + k' - 1 = n
\end{aligned}
$$

where

$$\begin{aligned} R_1^n \subseteq R_2^n &\equiv \forall \bar{x}.\ \bar{x} \in R_1^n \rightarrow \bar{x} \in R_2^n \\ R_1^n = R_2^n &\equiv R_1^n \subseteq R_2^n \ \wedge\ R_2^n \subseteq R_1^n \\ (x_1, \ldots, x_{k+k'-1}) \in R_1^k.R_2^{k'} &\equiv \exists y.\ (x_1, \ldots, x_{k-1}, y) \in R_1^k \\ &\qquad \wedge\ (y, x_k, \ldots, x_{k+k'-1}) \in R_2^{k'} \end{aligned}$$

This logic has a standard semantics of multisorted logic. Formulas of the logic are interpreted over structures $M = (L, E, \gamma)$, where L and E represent disjoint domains of both sorts *list* and *elem*. Function γ interprets relational symbols r^n by mapping them on the relations between individual atoms of M, e.g. for a binary relational symbol $first \subseteq list \times elem$ the corresponding mapping would be $\gamma(first) \subseteq L \times E$. Further, a valuation function $\alpha : Vars \rightarrow L \cup E$ assigns values from M to free variables x_i of an evaluated formula. A generic definition of $[\![\varphi]\!]^{M,\alpha}$ for a given structure M and a valuation α is shown in Figure 2. For a given structure M and a formula φ we call M *model* of φ *iff* $[\![\varphi]\!]^{M,\alpha}$ is *true* for *any* valuation α, i.e. $M \models \varphi$. Similarly, we define $M \models \{\varphi_1, \ldots, \varphi_n\}$ *iff* $M \models \varphi_i$ holds for each φ_i.

3.2 Model Finding

Alloy implements a fully automatic analysis for a relational logic and is an efficient model finder. By defining a signature and a set of axioms Φ_{ax} we specify the analyzed system. For a formula φ and a given scope r (upper bound on the size of the domains) Alloy searches for models M satisfying axioms Φ_{ax} but violating the property φ, i.e. $M \models \Phi_{ax} \cup \{\neg\varphi\}$.

We utilize this capability to search for structures $M = (E, L, \gamma)$ which represent finite cutouts from infinite term algebras. We recall, that analyzed formulas are normalized to $Q_1 v_1 :: s_1 \ldots Q_n v_n :: s_n.\ \psi$ where ψ is a quantifier-free formula with free variables $v_1, \ldots, v_n$. In the case of a successful search, Alloy identifies a finite structure M and a valuation α_0 for the specified scope r such that $\neg[\![\psi]\!]^{M,\alpha_0}$. For example, for a universal formula $\forall x, y :: list.\ x = y$ and the scope $r \geq 2$ Alloy would identify M with L containing at least two different atoms l_0, l_1 such that $\neg[\![x = y]\!]^{M,\alpha[x:=l_0,y:=l_1]}$. A detailed demonstration using a more sophisticated example is given in Section 6, where an implementation of interval lists does not satisfy an invariant.

3.3 Translation of KIV Formulas to Relational Form

Since Alloy is based on relational logic, KIV specifications involving functions have to be translated to specifications using relations. Therefore, as a first step we map each function symbol f to the corresponding relation (predicate) F:

$$f : \underline{s} \rightarrow s' \qquad \rightsquigarrow \qquad F : \underline{s} \times s'$$

The basic idea is that the relation F encodes the graph of the function f:

$$[\![f]\!](a_1, \ldots, a_n) = b \Leftrightarrow [\![F]\!](a_1, \ldots, a_1, b) \tag{1}$$

where $[\![f]\!]$ is the semantics of f in a model of the KIV specification and $[\![F]\!]$ is the semantics of F in the corresponding model of the translated specification.

To achieve this we need two axioms for every function, that state that F is the translation result of a total function, namely the `uniqueness` axiom:

$$\forall x_1, \ldots, x_n, y, z.\ F(x_1, \ldots, x_n, y) \wedge F(x_1, \ldots, x_n, z) \rightarrow y = z \tag{2}$$

and the `totality` axiom:

$$\forall x_1, \ldots, x_n.\ \exists y.\ F(x_1, \ldots, x_n, y) \tag{3}$$

We also need to translate the axioms of KIV to axioms over relations. This can be done schematically, the main idea is to introduce auxiliary variables for all intermediate results and to finally replace $f(x_1, \ldots, x_n) = y$ by $F(x_1, \ldots, x_n, y)$. We give a formal definition which assumes that each axiom φ has been normalized to have all quantifiers in front of the formula (prenex normal form):

$$\varphi \equiv Q_1 v_1 :: s_1 \ldots Q_n v_n :: s_n.\ \psi, \tag{4}$$

where ψ is a quantifier-free formula with free variables $v_1, \ldots, v_n$.

The restriction to prenex normal form is not really necessary, but avoids a discussion about a suitable renaming of bound variables and occurrences of terms. As an example, consider a formula φ from the specification of lists in KIV:

$$\varphi \ \equiv\ \forall x, y :: list.\ \ rev(x + y) = rev(y) + rev(x) \tag{5}$$

Its quantifier-free subformula ψ contains the function symbols *rev* and + (for the *append* function). Therefore the translated axiom will use predicates $REV : list \times list$ for *rev* and $APP :\ list \times list \times list$ for +.

To define the translation, we need two sets of terms: the set of "top-level" terms $\mathfrak{T}_{top}$, which consist of all terms t_i that occur in equations $t_i = t_j$ or predicates $P(t_1, \ldots t_n)$ of ψ and which are not just variables. In our example $\mathfrak{T}_{top} \equiv \{rev(x + y), rev(y) + rev(x)\}$. Second we need the set $\mathfrak{T}_{all}$ of all non-variable subterms of terms in $\mathfrak{T}_{top}$. For our example $\mathfrak{T}_{all} \equiv \mathfrak{T}_{top} \cup \{x+y, rev(y), rev(x)\}$.

Based on these two sets the translated formula $\tau(\varphi)$ of an axiom φ is then defined as follows:

Definition 1 (Relational form)
Given a mapping $\vartheta : \mathfrak{T}_{all} \rightarrow Vars$ that generates fresh variables for terms in $\mathfrak{T}_{all}$ and a functional formula φ in KIV of the form given in (4). We construct its relational counterpart $\tau(\varphi)$ for Alloy:

$$\tau(\varphi)\ \equiv Q_1 v_1 :: s_1 \ldots Q_n v_n :: s_n.\ \forall\ \vartheta(\mathfrak{T}_{all}).$$
$$\bigwedge_{f(t_1,\ldots,t_k) \in \mathfrak{T}_{all}} (\vartheta(t_1), \ldots, \vartheta(t_k), \vartheta(f(t_1, \ldots, t_k))) \in F \rightarrow\ \psi[\mathfrak{T}_{top} \diagdown \vartheta(\mathfrak{T}_{top})]$$

where $\psi[\mathfrak{T}_{top} \setminus \vartheta(\mathfrak{T}_{top})]$ *is* ψ *with terms from* $\mathfrak{T}_{top}$ *substituted by corresponding fresh variables* $\vartheta(\mathfrak{T}_{top})$.

To continue with our example above, we compute $\tau(\varphi)$ for (5):

$$\begin{aligned}&\tau(\varphi) \equiv \\ &\quad \forall x, y :: list.\ \forall z_1, z_2, z_3, z_4, z_5 :: list.\ (z_3, z_1) \in REV\ \wedge\ (z_4, z_5, z_2) \in APP \\ &\quad \wedge (x, y, z_3) \in APP\ \wedge\ (y, z_4) \in REV\ \wedge\ (x, z_5) \in REV\ \rightarrow\ z_1 = z_2\end{aligned}$$

It is easy to prove (by induction on the complexity of terms and formulas) that the syntactical transformation τ preserves the meaning of formulas in the following sense: for each model of the original formula φ, the corresponding relational model (where the semantics of F is defined via (1)) satisfies $\tau(\varphi)$. Similarly, for each model of $\tau(\varphi)$, that also satisfies the axioms *Totality* and *Uniqueness*, a model of the original signature can be constructed such that φ and (1) hold. The transformation has linear complexity with respect to the size of a formula.

4 Generating Models of Free Data Types in Alloy

The semantics of free data types is defined on algebraic structures called *term algebras* which represent concrete models of specifications. In term algebras carrier sets are composed of inductively generated terms. Terms are generated using *constructor* operations (functions), e.g. the constant *nil* and the function $cons : elem \times list \rightarrow list$ for lists.

Here we refer to the work of Kuncak and Jackson [12]. We adopt their ideas to generate term algebras in Alloy. We have to specify corresponding structures $M = (E, L, \gamma)$, where E and L represent domains and γ interprets relational symbols r^n over E and L. Again we are using *lists* as a generic example of a free data type.

In Alloy new sorts (types of atoms) are introduced by the keyword **sig**. We specify two new sorts: *elem* and *list*, see Figure 3. Using the keyword **extends** we split the set of atoms of type *list* in two disjunctive subsets: the singleton set *nil* (defined to have exactly one atom, keyword **one**) and the set *cons* which can have an arbitrary number of atoms within specified bounds. Atoms of type *cons* represent results of the constructor function $cons : elem \times list \rightarrow list$ and are always connected over selector relations *first* and *rest* with atoms from which they are constructed. On the right side in Figure 3 the generated metamodel of the signature is shown.

In the next step we specify axioms in Alloy which restrict relations *first* and *rest* to behave properly in M. The following four axioms (SUGA) are necessary, see Figure 4. SUGA axioms generate infinite structures M which contain an isomorphic copy of the term model $M_\infty = (E, L, \gamma)$. Here the language is restricted only to selector functions *first* and *rest*.

```
module list

sig elem {}
sig list {}

one sig nil extends list
sig cons extends list {
   first : elem,
   rest  : list
}
```

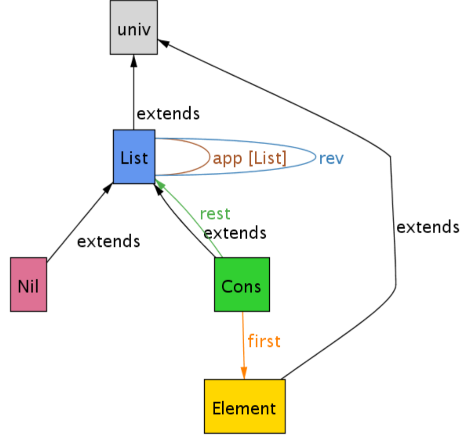

Fig. 3. Metamodel of lists in Alloy

$$
\begin{aligned}
\textit{selectors:}\quad & \forall l : \textit{list}.\ l \neq \textit{nil} \rightarrow \exists!\ l' : \textit{list},\ e : \textit{elem}.\ l.\textit{rest} = l' \ \wedge\ l.\textit{first} = e \\
& \forall l : \textit{list},\ e : \textit{elem}.\ (\textit{nil}, e) \notin \textit{rest}\ \wedge (\textit{nil}, l) \notin \textit{first} \\
\textit{uniqueness:}\quad & \forall l, l' : \textit{list}.\ l.\textit{first} = l'.\textit{first}\ \wedge\ l.\textit{rest} = l'.\textit{rest}\ \rightarrow\ l = l' \\
\textit{generator:}\quad & \forall l' : \textit{list},\ e : \textit{elem}.\ \exists l : \textit{list}.\ l.\textit{first} = e\ \wedge\ l.\textit{rest} = l' \\
\textit{acyclicity:}\quad & \forall l : \textit{list}.\ (l, l) \notin \hat{}\textit{rest}
\end{aligned}
$$

Fig. 4. SUGA axioms

The infiniteness of M_∞ prevents it to be constructed by Alloy. A possible solution to this problem is to omit the *generator* axiom (SUA axioms). This results in producing finite models M_0 which represent specific parts of original infinite structure M_∞, so-called *subterm-closed* models, i.e. models closed under transitive closure of selector relation *rest*, see Figure 5.

[12] establishes a finite satisfiability result by proving that for a specific class of formulas[2] (existential - bounded universal, EBU) satisfiability can be checked on finite models of SUA axioms (axioms without generator). For this purpose a notion of bounded quantification is introduced, see [12]. Roughly, if a witness for an EBU formula is found in a finite model M_0, we can pick the same witness in the infinite model M_∞. So a semi-decision procedure involving Alloy can be constructed that checks satisfiability of these formulas.

Unfortunately, we found that these encouraging results apply to theories only, where just selector functions are present in formulas. In order to use it in practice we have to cope with several difficulties. In the next section we will discuss what we have done to incorporate recursively defined functions in the method and what implications can possibly emerge.

[2] e.g., formula $\forall x :: \textit{list}.\ \exists y :: \textit{list}.\ (x, x, y) \in \textit{APP}$ contains unbounded quantification and has no finite models.

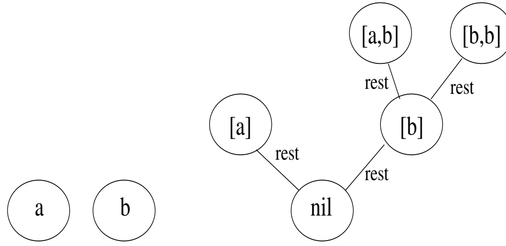

Fig. 5. Finite subterm-closed substructure M_0 of infinite structure M_∞ (2 *elem* and 5 *list* atoms). Relation *first* is omitted for reasons of clearness.

5 Axiomatization of Recursive Functions

We would like to extend our language containing just selector relational symbols {*first,rest*} to the complete language with recursive functions that we use in KIV. Such recursive definitions have the following form:

Definition 2 (KIV axioms for recursion). *To define a function* $f : s_1 \times \cdots \times s_k \to s_1$ *by structural recursion over the first argument, axioms of the following form are used:*

$$\forall u, \underline{v}. \quad \psi_i \to f(c_i(u), \underline{v}) = \Psi_i(f, u, \underline{v})$$

where each c_i *is one of the constructors for the sort* s_1*, each* Ψ_i *is a term that contains invocations of* f *with the first argument* u*. The cases* ψ_i *for one of the constructors* c *form a complete case distinction, i.e. the disjunction of all* ψ_i*, where* $c_i = c$ *is true.*

As a first step we translate the signature. Both the constructors and the recursive definitions have to be defined. For the constructor function *cons* we add a predicate definition in Alloy:

```
pred cons [e: elem, l: list, c: list] { c.first = e and c.rest = l }
```

Recursive functions like *APP* and *REV* are added to the list signature and thus declared as relations between the corresponding sorts:

```
sig list {
   app: list -> lone list,
   rev: lone list
}
```

By the keyword **lone** we tell Alloy that a relation satisfies the uniqueness axiom (2) of the relational translation, i.e. that there is at most one result for append and reverse. The totality axiom (3) would be satisfied too by using the keyword **one** instead of **lone**, but assuming totality of reverse or append prevents

finite models. Therefore we drop this axiom, just like the generator axiom for constructors.

As a second step we have to add appropriate axioms as facts to the Alloy specification that are translated from the axioms of the KIV specification from Figure 1. A simple idea would be to translate axioms directly (using τ) but this yields too weak axioms. Therefore we first combine the axioms for each function into one axiom. The resulting axioms for reverse and append are

$$app(x, y) = z \leftrightarrow x = [] \wedge z = y \vee \exists\, a_0, x_0.\ x = a_0 + x_0 \wedge z = a_0 + app(x_0, y)$$
$$rev(x) = y \leftrightarrow x = [] \wedge y = [] \vee \exists\, a_0, x_0.\ x = a_0 + x_0 \wedge y = rev(x_0) + a_0$$

Although the translation is schematic, it exploits uniqueness and totality. We then translate the resulting axioms to relations. For reverse we get:

$$\begin{aligned} \forall x, y :: list.\ (x, y) \in REV \leftrightarrow\ & (x \in NIL\ \wedge\ y = x\ \vee \\ & \exists a :: elem,\ z, z_1, z_2, z_3 :: list.\ (a, z_1, x) \in CONS \\ & \quad \wedge (z_1, z) \in REV\ \wedge\ z_2 \in NIL\ \wedge\ (a, z_2, z_3) \in CONS \\ & \quad \wedge (z, z_3, y) \in APP) \end{aligned} \tag{6}$$

The translated formula is stronger than what we would get by translating the original axioms: these only would give the implication from right to left of (6). The other direction would have to be inferred using the uniqueness and totality axioms. For the equivalence we do not need uniqueness and totality any more, since it is an instance of the well-founded recursion theorem:

Theorem 1 (Well-founded recursion). *Given a specification that is enriched with a new function g defined by the single axiom*

$$g(\underline{v}) = \Psi(g, \underline{v})$$

where all arguments of recursive calls to g in Ψ are smaller than $\underline{v}$ with respect to a well-founded order $<$. Then for each model M of the original specification the enrichment defines exactly one function g.

A formal proof of this theorem, which views Ψ as a higher-order function, can be found in [9]. For our case g is the relation (= boolean function) F. The theorem implies that just translating the equivalence already fixes exactly one relation F. Since the relational translation, when adding uniqueness and totality gives the relation F that is equal to the graph of f, the translated axiom *alone* must already specify the correct F.

The well-founded recursion theorem is applicable not only for the term models but also for the finite models that Alloy constructs, since the restriction of the well-founded subterm relation to finite models is obviously well-founded again. It is also applicable for the original recursive definitions in KIV.

Together we have: the recursive definitions of KIV extend the term model by a unique function f. The relational transformation also gives a unique extension of

the term model M_∞ by a relation F, which is the graph of f. For a finite subterm-closed model M_0 we get a unique function F_0 using the translated axiom for f too.

The critical question now is: does F_0 in M_0 satisfy the same theorems as F in M_∞? We will give a positive answer below for a class of formulas with universal and bounded existential quantification similar to [12]. The answer has the precondition, that function F, when restricted to the model M_0 (written $F \mid M_0$ in the following) is equal to F_0. In all the examples that we have checked we found that $F \mid M_0 \subseteq F_0$, and it remains as an open question whether this holds in general. In most examples even $F \mid M_0 = F_0$ holds, APP being one positive example. Nevertheless, we found examples, where $F \mid M_0$ is a proper subset of F_0. REV is one instance of the problem:

Example 1. Consider the subterm-closed model M_0 with $L_0 = \{[], [a], [c], [b,c], [b,a], [a,b,c], [c,b,a]\}$. In this model the atoms $[a,b,c]$ and $[c,b,a]$ are not connected by the relation REV_0, even though in the infinite model $REV([a,b,c], [c,b,a])$ holds. The reason is that the intermediate result of reversing $[b,c]$, the list $[c,b]$ (stored as z in axiom (6)) is not in L_0.

The general problem is that subterm-closedness does not guarantee, that the model is closed against chains of results computed by recursive invocations of the defined function. In the example, this chain of results for $[a,b,c]$ is: $rev([])$, $rev([c])$, $rev([b,c])$, $rev([a,b,c])$, since $rev([a,b,c])$ calls $rev([b,c])$ etc.. There is no problem if all results of this chain beyond a certain point are not in the finite model, the problem appears only, if the result of one call (here: $rev([b,c])$ is not in the model, but the result of the next (here: $rev([a,b,c])$) is again in the model. Therefore we have to find a constraint, that rules out such models. We must make sure that with the result of $rev([a,b,c]) = [c,b,a]$ being in the model, the previous result $rev([b,c]) = [c,b]$ is in the model too.

A constraint that guarantees this, is that the model is prefix-closed:

$$\forall y :: list.\ y \in NIL \ \vee \ \exists z_3, z_4, z_5 :: list.\ z_4 \in NIL \wedge (a, z_4, z_5) \in CONS \\ \wedge (z_3, z_5, y) \in APP$$

It seems that this constraint can be derived for surjective functions in general, where we know that any element of the model is a result of the function. The constraint then says that each y (a result of f) must be computable from the results $z_1, \ldots z_n$ of recursive calls. For a recursive definition of F of the form

$$F(\underline{x}, y) \ \leftrightarrow \ \Psi(F(\underline{t_1}, u_1), F(\underline{t_2}, u_2), \ldots, F(\underline{t_n}, u_n), \underline{x}, y)$$

the constraint for constructing a result from the previous call therefore is

$$\forall y, \underline{x}.\ \exists z_1, \ldots z_n.\ \Psi(u_1 = z_1, u_2 = z_2, \ldots, u_n = z_n, \underline{x})$$

The constraint works for reverse and gives the constraint (7) after simplification. The definition of *append* function gives the trivial constraint of subterm-closedness which is fortunately already satisfied by SUA models. For functions

which are not surjective, the constraint would have to quantify *only* over all y in the image of f, but this is not possible, since the only way to characterize the image is again via the recursion. An example which shows the problem is

Example 2. Consider the non-surjective function *palindrome* with axioms

$$pal([\,]) = [\,],\;\; pal(a + x) = a + (pal(x) + (a + [\,]))$$

Obviously not all atoms are results of *pal*. The solution above states that any atom being the result of f can be deconstructed according to the axiomatization of f. But in case of *pal* function for some atoms there are no such deconstruction, Alloy would not be able generate any model. The fundamental problem about this is that we don't know (cannot formulate in the "deconstruction"-axiom) ahead whether some atom is an image of f or not.

Assuming that for all F the equation $F|_{M_0} = F_0$ holds, we can get a similar result as in [12], using the following class of formulas:

Definition 3 (Bounded quantifiers and UBE formulas). *A bounded existential quantifier is of the form $\exists v :: s.\ v < t \rightarrow \psi$, where t is an arbitrary term and $<$ is a subterm order*[3]*. An UBE formula uses universal and bounded existential quantifiers.*

[12] defines EBU formulas, since they are interested in satisfiability, while we define their negation, since we are interested in counterexamples. Their bounded quantification allows $v \in S$ with an arbitrary set S instead of $v < t$, which at first glance looks much more liberal. In fact it is not, since the symbols available to describe a set S are selectors and nothing else. Since selectors can describe subterms of existing terms only, subterm-closedness is then enough to ensure the existence of witnesses. As soon as we allow other functions, the more general form fails to work in the following theorem.

Theorem 2 (Finite refutation). *Let φ be an UBE formula in KIV, $\tau(\varphi)$ its transformation to Alloy and M_∞ the term algebra for the KIV theory translated to use relations. Let M_0 be a finite subterm-closed substructure of M_∞, which also preserves all relations F from M_∞ i.e. $F|_{M_0} = F_0$. Further, let $M_0 \nvDash \tau(\varphi)$. Then $M_\infty \nvDash \tau(\varphi)$.*

The proof of this theorem is exactly like Kuncak's proof by induction over the structure of a formula. It allows to find counter examples for UBE formulas, by incrementally constructing finite models. To be complete, we would have to increase the bound indefinitely, but for practical purposes the search can be stopped as soon as it either finds a counter example or takes too long.

6 Experimental Results

We applied our technique to most representative examples in KIV. As an automatic translator to Alloy input language is not yet implemented, we used manually compiled Alloy models.

[3] Always provided for free data types in KIV.

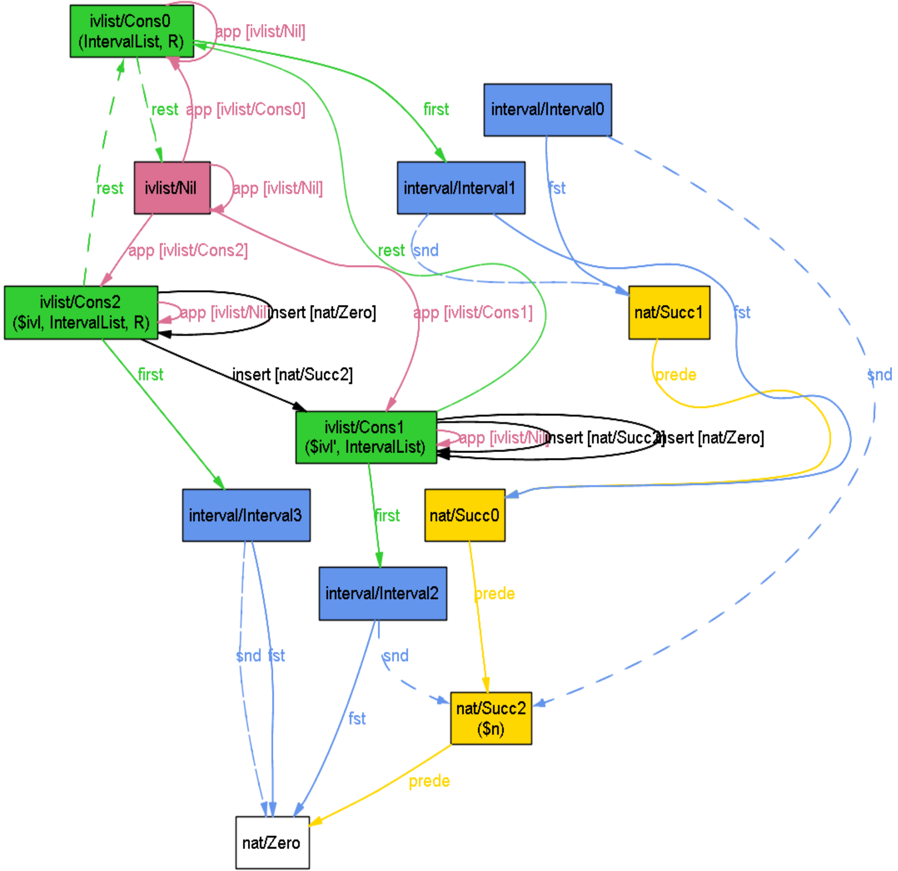

Fig. 6. Counter example generated by Alloy

6.1 Example: Lists of Intervals

As a nice nontrivial example we considered an implementation of sets of natural numbers by intervallists, that was used in [17] to demonstrate algebraic refinement via modules in KIV. The example has also been analyzed previously using KIV's own counter example generation mechanism described in [18]. We first describe the example, the results we got with Alloy and then give a short comparison of the results with KIV.

Sets of natural numbers can be implemented as lists of intervals, where an interval is simply a pair of numbers. For example the set $\{0, 1, 2, 4, 5, 7\}$ can be represented by the list of intervals $[(0, 2), (4, 5), (7, 7)]$ in a unique way. A typical application is the list of free blocks of dynamically allocated memory. A predicate R defines well-formed lists, e.g. $R([(0, 2)]) = true$, $R([(2, 0)]) = false$, $R([(0, 1), (1, 2)]) = false$. Further, an *insert* function is specified, which adds a

number into a list of intervals. A correct specification of *insert* operation must satisfy following invariant:

$$\forall\, ivl_1, ivl_2 \in intervallist,\ n \in nat.\ R(ivl_1)\ \wedge\ ivl_2 = insert(ivl_1, n) \rightarrow R(ivl_2)$$

The original specification contained a bug in the definition of *insert* function: it failed to merge $[_, n]$ and $[n+1, _]$ into one interval when $n+1$ was inserted. As Table 1 shows, using our technique Alloy was able to identify the smallest counter example at the scope[4] of 4 in 2 seconds.

Table 1. Benchmark (berkmin SAT solver, 2.4 GHz Dual Core)

scope (model size)	counter example	clauses	time
1	no	2000	0.1 s
2	no	5000	0.3 s
3	no	13252	0.7 s
4	yes	83302	2 s

The invariant was violated for $ivl_1 = [(0,0),(2,3)]$, $ivl_2 = [(0,1),(2,3)]$ and $n = 1$. This instantiation can be read off from the model generated by Alloy, that is shown in Figure 6. It depicts a finite structure $M_0 = (I_0, L_0, N_0, \gamma)$ with atoms of sorts *interval*, *list* and *nat* together with corresponding relations between them. Alloy labels with marks $\$ivl$, $\$ivl'$, $\$n$ those atoms which violate invariant, i.e. *Cons1*, *Cons2* and *Succ2*. By tracing back constructor relations we rebuild corresponding terms and therewith identify values of ivl_1, ivl_2 and n.

The same example was tried using KIV's counter example generation. This roughly works as follows: first a proof attempt for (7) is done. Using heuristics KIV automatically creates a proof tree in 4 seconds, ending in an open goal. The user then has to analyze this goal and to decide, either that it is unprovable or which proof step to be apply next. In this case, the user will suspect rather soon, that it is unprovable and invoke counter example generation. This proof strategy exploits the fact, that constructing a counter example basically means to instantiate all variables x in the goal by constructor terms. Therefore it does a systematic search by instantiating all variables with all constructor terms and by applying rewrite rules. For the goal at hand the search stops after two seconds with an empty sequent, which is definitely unprovable (in unsuccessful cases the search does not terminate, and the user has to abort manually). KIV will then compute a counter example for the original goal, by examining the proof tree (the effort for doing this is negligible). For our goal the counter example will be $ivl_1 = [(0,1),(1,m)]$ with arbitrary m. The successful application critically depends on heuristics and that suitable rewrite rules have been designed. In summary, the 2 seconds that Alloy needs are a clear improvement compared to the 6 seconds + user analysis of a goal.

[4] Defines maximal number of atoms for each sort. The smallest counter example which is presented here needs at least 4 atoms of the *list* sort, i.e. $[]$, $[(2,3)]$, $[(0,0),\ (2,3)]$ and $[(0,1),(2,3)]$.

7 Conclusion

We have presented an automatic method which can be applied to a wide class of first-order logic formulas. We aim to integrate it into the theorem prover KIV. The method is restricted to universal-bounded existential sentences. The question whether a formula is amenable to the analysis can be answered by a simple syntactic check. This limitation is not a big drawback in our opinion as from our own experience non-UBE formulas are rather rare.

This work was our first experiment with Alloy tool and we achieved very promising results. Naturally, there are open issues. The main open question that remains is: when does the relation F_0 agree with $F\mid_{M_0}$? It seems that in a large number of cases it does, but we have not found a syntactic characterization of this class yet. Another assumption, that is too strong is that all functions are defined recursively. In practice it is also common to specify functions non-recursively using quantified formulas: e.g. a predicate $\in\,:\; elem \times list$ can be specified as $a \in x \leftrightarrow \exists y, z.\, y + a + z = x$.

The translation to Alloy language was done manually and we have to automate it. A new more powerful tool based on Alloy called Kodkod [22] has become available recently. It is implemented as an API rather than as a standalone application and can easily be incorporated as a backend of another tool. We plan to use it for more seamless integration in KIV's graphical user interface and better proof visualization. We also intend to investigate an extension of the method to non-freely generated data types (like arrays or sets).

References

1. Ahrendt, W., Beckert, B., Hähnle, R., Menzel, W., Reif, W., Schellhorn, G., Schmitt, P.: Integrating Automated and Interactive Theorem Proving. In: Bibel, W., Schmitt, P. (eds.) Automated Deduction – A Basis for Applications. Systems and Implementation Techniques, Interactive Theorem Proving, vol. II, Kluwer Academic Publishers, Dordrecht (1998)
2. Balser, M.: Verifying Concurrent Systems with Symbolic Execution. PhD thesis, Universität Augsburg, Fakultät für Informatik (2005)
3. Balser, M., Bäumler, S., Knapp, A., Reif, W., Thums, A.: Interactive verification of UML state machines. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 434–448. Springer, Heidelberg (2004)
4. Balser, M., Reif, W., Schellhorn, G., Stenzel, K.: KIV 3.0 for Provably Correct Systems. In: Hutter, D., Traverso, P. (eds.) FM-Trends 1998. LNCS, vol. 1641, Springer, Heidelberg (1999)
5. Balser, M., Reif, W., Schellhorn, G., Stenzel, K., Thums, A.: Formal system development with KIV. In: Maibaum, T.S.E. (ed.) FASE 2000. LNCS, vol. 1783, pp. 363–366. Springer, Heidelberg (2000)
6. de Nivelle, H., Meng, J.: Geometric resolution: A proof procedure based on finite model search. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 303–317. Springer, Heidelberg (2006)

7. Fontaine, P., Marion, J.-Y., Merz, S., Nieto, L.P., Tiu, A.: Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 167–181. Springer, Heidelberg (2006)
8. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge (2000)
9. Harrison, J.: Inductive definitions: Automation and application. In: TPHOLs, pp. 200–213 (1995)
10. The Alloy Project, `http://alloy.mit.edu`
11. Jackson, D.: Automating first-order relational logic. In: SIGSOFT 2000/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering, pp. 130–139. ACM Press, New York (2000)
12. Kuncak, V., Jackson, D.: Relational analysis of algebraic datatypes. In: Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (2005)
13. Meng, J., Quigley, C., Paulson, L.C.: Automation for interactive proof: First prototype. Inf. Comput. 204(10), 1575–1596 (2006)
14. Momtahan, L.: Towards a small model theorem for data independent systems in alloy. Electr. Notes Theor. Comput. Sci. 128(6), 37–52 (2005)
15. Ramananandro, T.: Mondex, an electronic purse: specification and refinement checks with the Alloy model-finding method. Formal Aspects of Computing 20(1), 21–39 (2008)
16. Reif, W., Schellhorn, G.: Theorem Proving in Large Theories. In: Bibel, W., Schmitt, P. (eds.) Automated Deduction—A Basis for Applications, vol. III, 2, Kluwer Academic Publishers, Dordrecht (1998)
17. Reif, W., Schellhorn, G., Stenzel, K.: Interactive Correctness Proofs for Software Modules Using KIV. In: COMPASS 1995 – Tenth Annual Conference on Computer Assurance, IEEE press, Los Alamitos (1995)
18. Reif, W., Schellhorn, G., Thums, A.: Flaw detection in formal specifications. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, pp. 642–657. Springer, Heidelberg (2001)
19. Schellhorn, G.: Verification of Abstract State Machines. PhD thesis, Universität Ulm, Fakultät für Informatik (1999), `http://www.informatik.uni-augsburg.de/swt/Publications.htm`
20. Stenzel, K.: A formally verified calculus for full Java Card. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 491–505. Springer, Heidelberg (2004)
21. Thums, A., Schellhorn, G., Ortmeier, F., Reif, W.: Interactive verification of statecharts. In: Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (eds.) INT 2004. LNCS, vol. 3147, pp. 355–373. Springer, Heidelberg (2004)
22. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)