

# Automating Algebraic Specifications of Non-freely Generated 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

**Abstract.** Non-freely generated data types are widely used in case studies carried out in the theorem prover KIV. The most common examples are stores, sets and arrays. We present an automatic method that generates finite counterexamples for wrong conjectures and therewith offers a valuable support for proof engineers saving their time otherwise spent on unsuccessful proof attempts. The approach is based on the finite model finding and uses Alloy Analyzer [1] to generate finite instances of theories in KIV [6]. Most definitions of functions or predicates on infinite structures do not preserve the semantics if a transition to arbitrary finite substructures is made. We propose the constraints which should be satisfied by the finite substructures, identify a class of amenable definitions and present a practical realization using Alloy. The technique is evaluated on the library of basic data types as well as on some examples from case studies in KIV.

**Keywords:** Algebraic specifications, abstract data types, finite models, first-order logic, theorem proving, SAT checking.

## 1 Introduction

The concept of abstract data types is well-established in computer science. Algebraic techniques are integrated in varieties of formal approaches in software development [10,11,20]. This work is aimed at providing automatic techniques for analysis of algebraic specifications of abstract data types. We present an integration of an automatic procedure for finding finite counterexamples or witnesses for first-order theories in the theorem prover KIV [6]. 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. Most of the time in interactive theorem proving is spent to find out why certain goals are not provable. An alternative approach however is to try to disprove conjectures and to generate counterexamples. We were inspired by the automatic analysis method for first-order *relational* logic with *transitive closure* implemented in the Alloy [1] and its successful application in the Mondex challenge by Ramananandro [24]. Alloy's algorithm handles full first-order relational logic with quantifiers and transitive closure [14].

The reason why we are interested in the automation of the FOL part of KIV is that in almost all proof tasks carried out interactively in KIV, whether in the basic logic or in extensions for temporal logic proofs [4], ASM specifications [28], statecharts [5] or Java program proofs [30], 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 basic data types. Defining and proving such simplifier rules is therefore a regular task in interactive verification. Usually, some of these theorems are wrong at the first attempt, so a quick check that identifies unprovable ones is very helpful. Currently, to “test” an algebraic specification the proof engineer has either to perform an informal analysis or try to prove the desired property by starting an interactive proof. Most of the bugs in designs are discovered in the first step while some particularly hard to find ones - in the second. The technique presented in this work is intended to be a complementary quality improvement step which requires minimal additional user effort.

In this paper we extend the results in [15,9]. Kuncak and Jackson [15] presented a general approach of utilizing finite model finding for analysis of term algebras with language restricted to selectors only. [9] was the first step toward extending the language to arbitrary functions which however was restricted to very special class of definitions of recursive functions on freely generated data types only. We extend the application scope of the technique with non-freely generated data types. Furthermore, we define constraints which should be satisfied by the analyzed specification in order to guarantee a sound axiomatization of its operations on finite structures. The main contribution is a generic method for construction of finite instances for first-order specifications of abstract data types. We applied the method to a considerable set of examples to determine the relative size of the application domain. We showed empirically that all specifications of freely generated data types in KIV’s library are amenable to the technique while the generation of instances and checking of theorems with Alloy is accomplished in a reasonable time (from several seconds to few minutes in worst case). In our experiments we used Alloy version 4.0 RC17 [1].

## 1.1 Outline

Section 2 introduces the background notions used throughout the work. In Section 3 using the specification of non-freely generated data type store (abstract memory model) as an example we present an approach for generation of finite models with Alloy. In Section 4 we determine the constraints necessary for a proper axiomatization of functions and predicates on finite structures. Section 5 presents results from the application to the library of basic data types in KIV. This is followed by an overview of the related work in Section 6 and conclusions in Section 7.

## 2 Preliminaries

### 2.1 Specifications of Algebraic Data Types

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. The basic logic underlying the KIV system combines *Higher-Order Logic* (HOL) and *Dynamic Logic* (DL) [12], which allows to reason about partial and total correctness of imperative programs as well as expressing the program equivalence. KIV supports both functional and state-based approaches to specify systems. In this paper, we concern ourselves with the functional approach. A theory in KIV describes data types, e.g. naturals, lists, arrays or records, which afterward are used in DL programs. Theories are specified using hierarchically *structured algebraic specifications*.

#### Definition 1 (Language)

Let  $\Sigma = (S, \mathcal{F}, \mathcal{P})$  be a signature with sorts  $s \in S$  and sets  $\mathcal{F}, \mathcal{P}$  of operations which combine the sets  $\mathcal{F}_{\underline{s} \rightarrow s}$  of function symbols  $f : \underline{s} \rightarrow s$  with the sets  $\mathcal{P}_{\underline{s}}$  of predicate symbols  $p : \underline{s}$ , where  $\underline{s} \in S^*$ .

This definition is accompanied by some auxiliary commonly used constructs:  $X, X_s$  - sets of all variables and variables of the type  $s$  respectively,  $T(\mathcal{F}_0, X_0)$  - set of terms constructed using functions from  $\mathcal{F}_0$  and variables from  $X_0$ ,  $For(\Sigma)$  - set of formulas over  $\Sigma$ . Using a valuation  $\alpha$  (assigns values to free variables) terms and formulas are evaluated over structures called *algebras*:  $\llbracket t \rrbracket^{\mathcal{A}, \alpha}, \llbracket \varphi \rrbracket^{\mathcal{A}, \alpha}$ .

#### Definition 2 (Algebra)

For  $\Sigma$  let  $Alg(\Sigma)$  be a set of structures  $\mathcal{A} = ((A_s)_{s \in S}, (f_{\mathcal{A}})_{f \in \mathcal{F}}, (p_{\mathcal{A}})_{p \in \mathcal{P}})$ , where  $A_s$  denotes a nonempty domain of the sort  $s$  and  $f_{\mathcal{A}}, p_{\mathcal{A}}$  are concrete interpretations of symbols  $f \in \mathcal{F}, p \in \mathcal{P}$  over domains.

To specify data structures adequately, in addition to first-order axioms, axioms for induction are needed. Unfortunately, an induction scheme cannot be specified by a finite set of first-order formulas. As a replacement *generation clauses* are used. Hence, for a signature  $\Sigma$  the corresponding set  $Gen(\Sigma)$  contains term generation clauses: “ $s$  **generated by**  $C_s$ ”. The set of constructors  $C_s = \{c_1, \dots, c_n\} \subseteq \mathcal{F}_{\underline{s} \rightarrow s}$  is assumed to contain at least one constructor  $c_i : \underline{s} \rightarrow s$  with all its argument sorts different from  $s$  (a constant constructor). Generation clauses in  $Gen(\Sigma)$  have the following semantics:

$$\begin{aligned} \mathcal{A} \models s \text{ generated by } C_s &\Leftrightarrow \\ \text{for all } a \in A_s \text{ exists } \alpha, t \in T_s(C_s, X \setminus X_s) &: a = \llbracket t \rrbracket^{\mathcal{A}, \alpha} \end{aligned}$$

The above equivalence ensures that any element in  $A_s$  is a value of some  $s$ -valued constructor term  $t \in T_s(C_s, X \setminus X_s)$  using an appropriate assignment  $\alpha$  to parameter variables  $X \setminus X_s$ , i.e.  $\mathcal{A}_s$  is *term generated*.

**Definition 3 (Specification)**

For a signature  $\Sigma$  a specification  $SP = (\Sigma, Ax, Gen)$  is a triple with finite sets  $Ax \subseteq For(\Sigma)$  and  $Gen \subseteq Gen(\Sigma)$ .

**Definition 4 (Model)**

An algebra  $\mathcal{A}$  is a model of a specification  $SP$ ,  $\mathcal{A} \in Mod(SP)$ :

$$\mathcal{A} \models SP \Leftrightarrow \mathcal{A} \in Alg(\Sigma), \mathcal{A} \models Gen, \mathcal{A} \models Ax$$

A formula  $\varphi \in For(\Sigma)$  is valid:

$$SP \models \varphi \Leftrightarrow \text{for all } \mathcal{A} \in Mod(SP) : \mathcal{A} \models \varphi$$

It is distinguished between free and non-free data types. Freely generated data types have the property that syntactically different terms denote different values, e.g. data type *list* which is generated by constructors *nil* and *cons* and has selectors *first* and *rest*. The corresponding models are called *term algebras* where terms have the above property.

While [9] discusses freely generated data types, in this work we are particularly interested in non-freely generated data types for which different terms can denote the same value, e.g. sets, stores, arrays (unlike for freely generated data types where syntactically different terms denote different values). For example, multiple insertions of the same element in a set have always the same result,  $[\emptyset \ ++ \ a = \emptyset \ ++ \ a \ ++ \ a]^{\mathcal{A}, \alpha}$ . For a non-freely generated data type  $s$  in addition to the generation clause  $Gen_s$  the equivalence relation on  $s$ -valued terms has to be provided.

**Definition 5 ( $\Sigma$ -congruence)**

Let  $\Sigma = (S, \mathcal{F}, \mathcal{P})$  and  $\mathcal{A} \in Alg(\Sigma)$ .  $\Sigma$ -congruence  $R$  is a family of equivalence relations  $(R_s)_{s \in S}$  induced by  $S$  which are compatible with operations  $\mathcal{F} \cup \mathcal{P}$ :

1. for all  $f \in \mathcal{F}$ ,  $f : s_1 \times \dots \times s_n \rightarrow s$ ,  $a_i, b_i \in \mathcal{A}_{s_i}$  holds  
 $\bigwedge_{i=1}^n R_{s_i}(a_i, b_i) \Rightarrow R_s(f_{\mathcal{A}}(a_1, \dots, a_n), f_{\mathcal{A}}(b_1, \dots, b_n))$
2. for all  $p \in \mathcal{P}$ ,  $p : s_1 \times \dots \times s_n$ ,  $a_i, b_i \in \mathcal{A}_{s_i}$  holds  
 $\bigwedge_{i=1}^n R_{s_i}(a_i, b_i) \Rightarrow (p_{\mathcal{A}}(a_1, \dots, a_n) \Leftrightarrow p_{\mathcal{A}}(b_1, \dots, b_n))$

The corresponding models are *quotient algebras*  $\mathcal{A}/R$ .

**Definition 6 (Quotient algebra)**

For a  $\Sigma$ -congruence  $R$  on  $\mathcal{A}$  is  $\mathcal{A}/R = ((Q_s)_{s \in S}, (f^\circ)_{f \in \mathcal{F}}, (p^\circ)_{p \in \mathcal{P}})$  a quotient algebra with:

1.  $Q_s = \{[a] : a \in \mathcal{A}_s\}$  where  $[a] = \{b : R_s(a, b)\}$
2. For operations  $f \in \mathcal{F}$ ,  $p \in \mathcal{P}$  and  $[a_i] \in Q_{s_i}$ :  
 $f^\circ([a_1], \dots, [a_n]) = [f_{\mathcal{A}}(a_1, \dots, a_n)]$   
 $p^\circ([a_1], \dots, [a_n]) \Leftrightarrow [p_{\mathcal{A}}(a_1, \dots, a_n)]$

Because  $R$  is a congruence, all operations are well defined and  $\mathcal{A}/R \in Alg(\Sigma)$ . Furthermore, if  $\mathcal{A}$  is a term generated algebra, then the corresponding quotient

algebra  $\mathcal{A}/_R$  is term generated too [25]. The equivalence relation  $R_s$  (for non-free  $s$ ) on terms is specified with the axiom of *extensionality*, e.g. for *sets* it states:

$$s_1 = s_2 \leftrightarrow (\forall a. a \in s_1 \leftrightarrow a \in s_2)$$

## 2.2 Multisorted Relational Logic in Alloy

The logic used by the Alloy Analyzer is a multisorted first-order relational logic with transitive closure [14]. It extends first-order logic by *relational terms*  $r \in R_{\underline{s}}$  instead of just predicate symbols  $p \in P_{\underline{s}}$ . Thus atomic formulas have form  $x_1 = x_2$  or  $r(x_1, \dots, x_n)$  where  $r$  is  $n$ -ary relational term. Relational terms are either predicate symbols or composition  $r_1.r_2$  or transitive closure  $\wedge r$ , where  $r$  is a binary relation. An Alloy specification  $SP_{Alloy}$  describes a finite multisorted universe  $U = (D_{s_1}, \dots, D_{s_n}, \gamma)$  with finite sets of atoms for each sort and an interpretation  $\gamma$  of relational symbols on domains. First-order formulas in  $SP_{Alloy}$  can be arbitrary quantified. For a specification  $SP_{Alloy}$  and a conjecture  $\varphi$  Alloy can be operated in two basic modes: searching for a *counterexample* ( $SP_{Alloy} \wedge \neg\varphi$ ) or computing a *witness* ( $SP_{Alloy} \wedge \varphi$ ).

We introduce a notion of a *relational specification*, where all functions are replaced by relations (predicates) that behave like corresponding functions. It is constructed from an algebraic specification  $SP = (\Sigma, Ax, Gen)$  using the translation procedure  $\tau$  from functional to relational form previously defined in [9] and the generation principle SUGA (selector, uniqueness, generator, acyclicity axioms for term algebras) described by Kuncak and Jackson [15].

### Definition 7 (Alloy specification)

For an algebraic specification  $SP$  we define a transformation to Alloy language:

$$\tau_{\text{fin}}(SP) = ((S, \tau(\mathcal{F}) \cup \mathcal{P}), \tau(Ax) \cup \text{Unique}(\tau(\mathcal{F})) \cup \text{SUA}(Gen))$$

where  $\tau$  translates all formulas from the functional to the equivalent relational form and maps function symbols to predicates:  $\tau(f) = \{F : f \in \mathcal{F}\}$  while axioms **Unique** require each predicate  $\tau(f)$  to behave like a function:

$$\forall \underline{x}, y_1, y_2. F(\underline{x}, y_1) \wedge F(\underline{x}, y_2) \rightarrow y_1 = y_2$$

SUGA axioms are used to specify term generated algebras (models of free data types, e.g. lists). Dropping of the generator axiom results in SUA axioms which specify finite subterm-closed substructures of infinite term algebras (for any term all subterms are also in the finite structure). Such structures are generated for Alloy specifications  $\tau_{\text{fin}}(SP)$ .

### Definition 8 (Alloy model)

Finite algebra  $\mathcal{M} = ((M_s)_{s \in S}, (p_{\mathcal{M}})_{p \in \mathcal{P} \cup \tau(\mathcal{F})})$  is a model of  $\tau_{\text{fin}}(SP)$ :

$$\mathcal{M} \models \tau_{\text{fin}}(SP) \Leftrightarrow \mathcal{M} \models \text{SUA}(Gen), \mathcal{M} \models \tau(Ax), \mathcal{M} \models \text{Unique}(\tau(\mathcal{F}))$$

In the next section we present an approach for specifying non-free data types in Alloy.

### 3 Example: Store

Non-freely generated data type *store* represents an abstract memory model with *addresses* and *data*. It is used in almost all bigger case studies in KIV where it specifies heaps, file systems, Java VM memory etc. In this section we present an algebraic specification of stores and the corresponding Alloy specification.

#### 3.1 Algebraic Specification

The algebraic specification of stores is parameterized by not generated types *elem* (corresponds to addresses) and *data*, see Figure 1. Data type *store* is generated by constructors  $\emptyset$  (empty store) and  $\cdot [ \cdot , \cdot ]$  (put data in store at some address; allocate the address if necessary). The equivalence relation is specified in axiom of extensionality which uses operations  $\in$  (test an address to be allocated) and  $\cdot [ \cdot ]$  (get data at some address). The basis specification can be enriched with arbitrary operations, e.g. see enrichment with *delete* and *subset* operations. A specification of *naturals* (freely generated) is also included as it will be used later on. Altogether, we have  $SP = (\Sigma, Ax, Gen)$  with

$$\begin{aligned} \Sigma &= (\{store, elem, data\}, \{\emptyset, \cdot [ \cdot , \cdot ], \cdot [ \cdot ], --\}, \{\in, \subseteq\}) \cup \Sigma_{nat} \\ Gen &= \{store \text{ generated by } \emptyset, \cdot [ \cdot , \cdot ]\} \cup Gen_{nat} \\ Ax &= \{ext, in-empty, in-put, at-same, at-other, subset, del-in, del-at\} \cup Ax_{nat} \end{aligned}$$

#### 3.2 Alloy Specification

For freely generated data types it would be enough to take the Alloy specification  $\tau_{fin}(SP)$ , see Definition 7, to generate subterm-closed structures [9]. In the following, we show how to adopt translation procedure  $\tau_{fin}$  for non-freely generated data types.

The corresponding Alloy specification  $\widehat{\tau}_{fin}(SP)$  yielding finite subterm-closed structures  $\mathcal{M}$  is constructed as follows:

$$\begin{aligned} \widehat{\tau}_{fin}(SP) &= (\widehat{\Sigma}, \widehat{Ax} \cup \text{Unique}(\tau(\mathcal{F}) \cup \{\widehat{PUT}, SIZE\}) \cup \text{SUA}(Gen) \cup \mathcal{C}_{store}^k) \\ \widehat{\Sigma} &= \widehat{\tau}_{fin}(\Sigma) \cup \{\widehat{PUT}, BIGGER, SIZE\} \\ \widehat{Ax} &= \{\Phi_{extension}, \Phi_{in}, \Phi_{at}\} \cup \{\Phi_{\widehat{PUT}}, \Phi_{SIZE}\} \cup \{\Phi_{subset}, \Phi_{del}\} [PUT/\widehat{PUT}] \\ \Phi_{\widehat{PUT}} &\equiv \forall st, st_1 : store, a : elem, d : data. \widehat{PUT}(st, a, d, st_1) \leftrightarrow \\ &\quad IN(a, st_1) \wedge AT(st_1, a, d) \wedge \\ &\quad (\forall b : elem. \exists d_1, d_2 : data. a \neq b \rightarrow (IN(b, st_1) \leftrightarrow IN(b, st)) \wedge \\ &\quad (\exists d_1, d_2 : data. AT(st, b, d_1) \wedge AT(st_1, b, d_2) \rightarrow d_1 = d_2)) \end{aligned}$$

where the signature is extended with new symbols  $\widehat{PUT}$ , *SIZE* and *BIGGER*. Further, all occurrences of the symbol  $PUT \equiv \tau(\cdot [ \cdot , \cdot ])$  in axioms  $\Phi_{in}$ ,  $\Phi_{at}$  (relational definitions of  $\in$  and  $\cdot [ \cdot ]$ ) are replaced by the newly introduced symbol  $\widehat{PUT}$ . We have to specify  $\widehat{PUT}$  to behave like the original *put* function in  $\mathcal{A}(\cdot [ \cdot , \cdot ] : store \times elem \times data \rightarrow store)$  using definition  $\Phi_{\widehat{PUT}}$ .

```

generic specification
  parameter elem, data
  target sorts store
  using nat
  constants
     $\emptyset$  : store;      comment : empty
  functions
    . [ . , . ] : store  $\times$  elem  $\times$  data  $\rightarrow$  store;      comment : put
    . [ . ] : store  $\times$  elem  $\rightarrow$  data;      comment : at
  predicates
    .  $\in$  . : elem  $\times$  store;
  variables
    st, st0, st1, st2 : store;
  induction
    store generated by  $\emptyset$ , . [ . , . ];
  axioms
    Ext : st1 = st2  $\leftrightarrow$   $\forall a. (a \in st1 \leftrightarrow a \in st2) \wedge st1[a] = st2[a]$ ;
    In-empty :  $\neg a \in \emptyset$ ;
    In-put :  $a \in st[b, d] \leftrightarrow a = b \vee a \in st$ ;
    At-same :  $(st[a, d][a]) = d$ ;
    At-other :  $a \neq b \rightarrow (st[b, d][a]) = st[a]$ ;
end generic specification

enriched specification
  functions
    . -- . : store  $\times$  elem  $\rightarrow$  store;      comment : delete
  predicates
    .  $\subseteq$  . : store  $\times$  store;
  axioms
    Subset : st1  $\subseteq$  st2  $\leftrightarrow$   $(\forall a. a \in st1 \rightarrow a \in st2 \wedge st1[a] = st2[a])$ ;
    Del-in :  $a \in st--b \leftrightarrow a \neq b \wedge a \in st$ ;
    Del-at :  $a \neq b \rightarrow (st--b)[a] = st[a]$ ;
end enriched specification

```

**Fig. 1.** Basis algebraic specification of the non-freely generated data type store + enrichment with two operations

The size function  $\#_{store} : store \rightarrow nat$  measures the construction complexity of store-valued terms, e.g.  $\#_{store}(\emptyset) = 0$ ,  $\#_{store}(\emptyset[a, d]) = 1$ ,  $a \neq b \rightarrow \#_{store}(\emptyset[a, d][b, d]) = 2$ . The predicate *SIZE* is the relational version of  $\#_{store}$ .

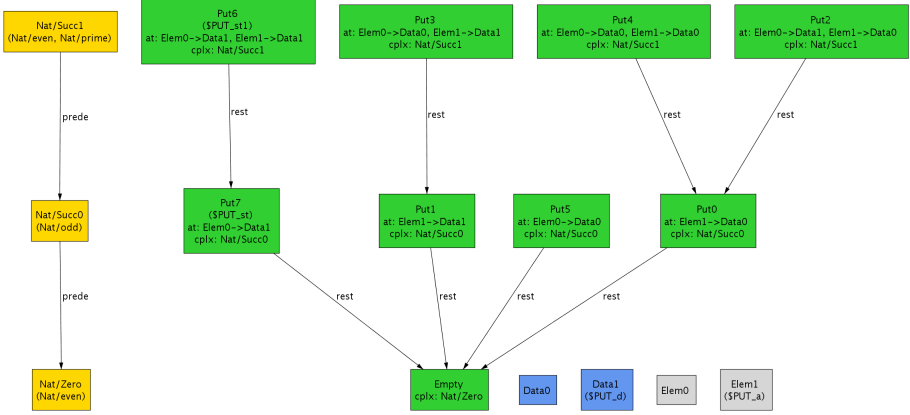


Fig. 2. Finite instance  $\mathcal{M}$  for  $\widehat{\tau}_{\text{fin}}(SP)$  generated by Alloy

The predicate  $BIGGER : store \times elem \times data$  tests whether the insertion of  $(a, d)$  into  $st$  leads to a bigger store (in terms of stored data):

$$\forall st : store, a : elem, d : data. BIGGER(st, a, d) \leftrightarrow a \notin st$$

Finally, we include a closedness constraint  $\mathcal{C}_{store}^k$  which restricts only to those subterm-closed structures which contain *all*  $store$ -valued terms up to the size  $k$ . Closedness constraint  $\mathcal{C}_{store}^k$  restricts to finite SUA-generated structures satisfying the following:

**put-bigger** :  $\forall st, st_1 : store, a : elem, d : data.$

$$PUT(st, a, d, st_1) \rightarrow BIGGER(st, a, d)$$

**generate** :  $\forall st : store, a : elem, d : data.$

$$\#_{store}(st) < k \wedge BIGGER(st, a, d) \rightarrow \exists st_1 : store. \widehat{PUT}(st, a, d, st_1)$$

**bound** :  $\forall st : store. \#_{store}(st) \leq k$

The first axiom restricts the application of the constructor  $PUT$  (responsible for the generation of new  $store$ -atoms) only to cases where it leads to a bigger store. The axiom **generate** is responsible for filling up a model with atoms up to a certain bound  $k$ . Figure 2 is produced by Alloy and shows an instance  $\mathcal{M} = (\mathcal{M}_{store}, \mathcal{M}_{elem}, \mathcal{M}_{data}, \mathcal{M}_{nat}, (p_{\mathcal{M}})_{p \in \mathcal{P}})$  which satisfies the  $\mathcal{C}_{store}^2$  closedness constraint. The domains are composed as follows:

$$\begin{aligned} \mathcal{M}_{store} = \{ \emptyset, \emptyset[a, d_1], \emptyset[a, d_2], \emptyset[b, d_1], \emptyset[b, d_2], \emptyset[a, d_1][b, d_1], \emptyset[a, d_1][b, d_2], \\ \emptyset[a, d_2][b, d_1], \emptyset[a, d_2][b, d_2] \} \end{aligned}$$

$$\mathcal{M}_{elem} = \{a, b\}, \mathcal{M}_{data} = \{d_1, d_2\}, \mathcal{M}_{nat} = \{0, 1, 2\}$$

For a better clearness only essential relations allowing an identification of the values of atoms are shown. Operations  $at$  (data on the given position) and  $cplx$  ( $\#_{store}$ ) are shown as attributes of atoms.

## 4 Model Construction with Alloy

In this section we consider the approach from the previous section from a more general point of view and discuss the constraints which should be satisfied by the analyzed algebraic specifications to guarantee the soundness. As demonstrated in [9] for freely generated data types some definitions of functions do not preserve the semantics under the transition from an infinite structure to some finite substructure. In case of non-freely generated data types it becomes even more complicated since the generation of the carrier set depends on some definitions (axiom of *extensionality*). In this section we discuss constraints which should be satisfied by an algebraic specification  $SP$  in KIV in order the corresponding Alloy specification  $\widehat{\tau}_{\text{fin}}(SP)$  to yield finite structures  $\mathcal{M}$  that are isomorphic to some finite substructures of the algebra  $\mathcal{A}$  (model of  $SP$ ). Roughly speaking, these constraints should guarantee that Alloy does not produce spurious counterexamples. We give a syntactical characterization of the constraints such that in the worst case they can be proven in the theorem prover KIV for a given specification (if not automatically discharged by some efficient heuristics). For this purpose, we enrich  $SP$  with auxiliary operations (in part even higher-order) which will be used for the reasoning about correctness, see Definition 9.

### Definition 9 (Auxiliary operations)

1.  $\#_s : s \rightarrow \text{nat}$  used to measure the construction complexity of  $s$ -valued terms
2.  $\preceq : s_1 \times s_1 \cup s_1 \times s_2 \cup \dots \cup s_k \times s_k$  - a preorder on domains
3.  $\prec : s \times s$  a predicate on terms in  $\mathcal{A}_s$  (for a non-freely generated  $s$ )
4.  $\Upsilon : \text{For}(SP) \times \text{Vars}$  identifies  $\preceq$ -compatible formulas

The preorder  $\preceq$  is used to introduce the  $\preceq$ -closedness of finite substructures of infinite structures which can be semantically characterized as follows (for finite structures  $\mathcal{M}$  and infinite  $\mathcal{A}$ ):

$$\forall d_0 \in \text{dom}(\mathcal{M}), d \in \text{dom}(\mathcal{A}). \quad d \preceq d_0 \rightarrow d \in \text{dom}(\mathcal{M})$$

For example, we can take the *subterm* relation for  $\preceq$ . The carrier set  $A_s/R$  contains quotients  $[t]$ . We can characterize  $[t]$  by their minimal representatives, i.e. terms  $t_m \in [t]$  with minimal size ( $\#_s$ ). Let  $[ ]_m : s \rightarrow s$  be a function calculating  $t_m$ . The predicate  $\prec : s \times s$  is defined such that  $t_1 \prec t_2 \leftrightarrow \#_{\text{store}}[t_1]_m < \#_{\text{store}}[t_2]_m$  (predicate *BIGGER* in the example with stores).  $\prec$  is used to restrict SUA axioms to generate only the terms  $[t]_m$ . Furthermore, we assume  $\prec$  to agree with  $\preceq : \prec \subseteq \preceq$ .

The predicate  $\Upsilon$  is used to define the compatibility of definitions on finite  $\preceq$ -closed substructures:

$$\begin{aligned} \Upsilon(\chi, \underline{x}) &\equiv \text{true}, \quad \text{for quantifier-free } \chi \\ \Upsilon(\exists z. \varphi, \underline{x}) &\equiv (\exists z. \varphi) \rightarrow (\exists z. \varphi \wedge (\bigvee_{u \in \underline{x}} z \preceq u) \wedge \Upsilon(\varphi, \underline{x} \cup z)) \\ \Upsilon(\forall z. \varphi, \underline{x}) &\equiv (\exists z. \neg \varphi) \rightarrow (\exists z. \neg \varphi \wedge (\bigvee_{u \in \underline{x}} z \preceq u) \wedge \Upsilon(\varphi, \underline{x} \cup z)) \end{aligned}$$

Now we can define a class of compatible specifications, i.e. specifications for which the transformation procedure  $\widehat{\tau}_{\text{fin}}$  to Alloy language is sound.

### Definition 10 (Compatibility)

1. *Definition*<sup>1</sup>  $\Phi \equiv \forall \underline{x}, y. f(\underline{x}) = y \leftrightarrow \mathcal{Q}_1 v_1 \dots \mathcal{Q}_n v_n. \chi(\underline{v}, \underline{x}, y)$  of a function  $f : \underline{s} \rightarrow s$  is  $\preceq$ -compatible iff  $\Upsilon(\mathcal{Q}_1 v_1 \dots \mathcal{Q}_n v_n. \chi(\underline{v}, \underline{x}, y), \underline{x} \cup \{y\})$  holds. If  $\chi$  contains recursive calls to  $f$  we additionally require the recursive definition  $\Phi$  to be well-founded [13] and for the associated well-founded order<sup>2</sup>  $\preceq_r$  to satisfy  $\preceq_r \subseteq \preceq$ .
2. *Extensionality axiom*  $\Phi_{\text{ext}} \equiv \forall x_1, x_2. x_1 = x_2 \leftrightarrow \mathcal{Q}_1 v_1 \dots \mathcal{Q}_n v_n. \chi(\underline{v}, x_1, x_2)$  is  $\preceq$ -compatible iff  $\Upsilon(\mathcal{Q}_1 v_1 \dots \mathcal{Q}_n v_n. \chi(\underline{v}, x_1, x_2), \{x_1, x_2\})$  holds
3. *Specification SP* is  $\preceq$ -compatible iff all its definitions and extensionality axioms are  $\preceq$ -compatible.

The following key steps in the process can not be completed automatically currently and require user creativity:

1. specification of the predicate  $\prec$ :  $s \times s$  (indirectly encoded in the predicate *BIGGER* in the example with stores)
2. specification of constructor functions  $\widehat{c}$  for each  $c \in C_s$  for a non-freely generated data type  $s$  (*PUT* and *P $\widehat{U}$ T* in the example in Section 3).
3. verification of  $\preceq$ -compatibility of *SP* (can be partly automated, currently for practical reasons this step is performed informally)

Although, the first two steps are the most crucial and challenging from the user point of view, they are also least labour intensive and require only a thorough understanding of  $\mathcal{A}/R$ . Furthermore, although non-free data types are used in almost every case study, typically only the standard library types (stores, sets, arrays, integers, graphs) are used, i.e. user effort is equal zero as library is already ported to Alloy. The third step can become tedious, since in the worst case proof obligations for  $\preceq$ -compatibility of definitions have to be discharged. The encouraging fact is, that in the considered case studies almost all definitions can be automatically proven  $\preceq$ -compatible using efficient heuristics. These heuristics on their part require a negligible information input from the user concerning input-output  $\preceq$ -relationship of a small number of basis operations.

The choice of the preorder  $\preceq$  has the major impact on the size of the generated structures (performance) and on the scope of amenable definitions ( $\preceq$ -compatibility). The most often used alternative is the *subterm* relation which comes for free from the SUA generation principle, suffices for most practical cases and is also the most efficient choice. As demonstrated in [9] for some definitions subterm-closedness is too weak and additional constraints requiring the finite structures to be filled to a certain degree must be specified. The strongest constraint is to require  $s, k$ -completeness ( $\mathcal{C}_s^k$ ) for some data types  $s$ , see example in Section 3 where we used it.

<sup>1</sup>  $\chi$  quantifier-free with free variables  $\underline{v} \cup \underline{x} \cup \{y\}$ .

<sup>2</sup> A binary relation  $\prec$  on a set  $X$  is said to be well-founded iff every nonempty subset of  $X$  has a  $\prec$ -minimal element:  $\forall S \subseteq X. S \neq \emptyset \Rightarrow \exists m \in S. \forall y \in S. y \not\prec m$ .

**Theorem 1 (Construction).** *For a  $\preceq$ -compatible specification  $SP$  if a finite structure  $\mathcal{M}$  is a model of  $\widehat{\tau}_{\text{fin}}(SP)$ , then  $\mathcal{M}$  is isomorphic to some finite  $\preceq$ -closed substructure  $\mathcal{A}^0$  of  $\mathcal{A}$ , where  $\mathcal{A} \models SP$ .*

Follows from the  $\preceq$ -compatibility of  $SP$  and a similar theorem for freely generated term algebras by Kuncak and Jackson [15]. For the considered data types in the library the reverse direction of the implication ( $\Leftarrow$ ) holds as well.

We refer to the previous results concerning the class of amenable theorems, which is composed of UBE (universal-bounded existential) formulas introduced in [9]. For both subtypes the reasoning on infinite structures can be reduced to reasoning on finite subterm-closed substructures, i.e. for for UBE sentences we have finite refutation. Because the considered  $\preceq$ -closed submodels are subterm-closed anyway (SUA generation), these results apply to them as well.

**Theorem 2 (Finite refutation).** *Let  $\varphi$  be an UBE formula in KIV,  $\tau(\varphi)$  its translation to the relational form,  $SP$  the specification. Then*

$$\text{exists } \mathcal{M} : \mathcal{M} \models \widehat{\tau}_{\text{fin}}(SP), \mathcal{M} \not\models \tau(\varphi) \Rightarrow \text{exists } \mathcal{A} : \mathcal{A} \models SP, \mathcal{A} \not\models \varphi$$

## 5 Results

We applied our technique to algebraic specifications of abstract data types in the theorem prover KIV. The scope of the application comprises KIV's library of the most essential freely and non-freely generated data types as well as some interesting examples from case studies carried out in KIV. For an evaluation we picked the following non-freely generated data types: stores, sets and arrays. The following algebraic specifications were considered:

- $SP_1 = (\{store, elem, data, nat\}, (\mathcal{F}, \mathcal{P}), Gen_{store} \cup Gen_{nat}, Ax)$ ,
- $SP_2 = (\{set, elem, nat\}, (\mathcal{F}, \mathcal{P}), \{\text{set generated by } \emptyset, ++\} \cup Gen_{nat}, Ax)$ ,
- $SP_3 = (\{array, elem, nat\}, (\mathcal{F}, \mathcal{P}), Gen_{array} \cup Gen_{nat}, Ax)$ .

We measure the size of the above specifications by counting defined operations (functions and predicates) and theorems. For  $SP_1$  we get:  $\#\mathcal{F} \cup \mathcal{P} = 12$ ,  $\#ax = 16$ ,  $\#theorems = 84$ , for  $SP_2$  :  $\#\mathcal{F} \cup \mathcal{P} = 16$ ,  $\#ax = 19$ ,  $\#theorems = 341$ , for  $SP_3$  :  $\#\mathcal{F} \cup \mathcal{P} = 12$ ,  $\#ax = 16$ ,  $\#theorems = 14$ . We informally checked the specifications for compatible definitions and theorems amenable to the analysis. In fact all operations in all considered specifications were compatible, i.e. properly definable on finite substructures. There were some theorems not amenable to the analysis, i.e. not belonging to the UBE class ( $\notin UBE$ ). In  $SP_1$  only 3 (of 84) theorems belonged not to the UBE class and in  $SP_2$  - 10 (of 341). In  $SP_3$  all theorems were analyzable. Among those 13 *not amenable* theorems the following distribution occurs: 7 of type  $\varphi \leftrightarrow \exists x. \psi$ , 5 of type  $\varphi \leftrightarrow \forall x. \psi$ , and 1 of type  $\exists x. \forall y. \psi$ .

Finding a bug in the library is not possible as all theorems are proven. Still, for the evaluation purposes we performed the following two steps. First, we picked 40 proven theorems altogether having the high coverage of the intended behavior of the specification and run the tool on them to see whether they produce

**Table 1.** Benchmark: generating an instance (zChaff [34] SAT solver, 2.4 GHz Dual)

| sorts  | bounds                                    | $s, k$ -compl | #ops | #clauses        | #vars           | w-time  |
|--------|---|---------------|------|-----------------|-----------------|---------|
| store, | $ S  \leq 9,  E  = 2,  D  = 2,  N  = 3$   | (store,2)     | 12   | $8 \times 10^4$ | $3 \times 10^4$ | 2 s     |
| elem,  | $ S  \leq 10,  E  = 3,  D  = 3,  N  = 2$  | (store,1)     | 12   | $3 \times 10^5$ | $1 \times 10^5$ | 30 s    |
| data,  | $ S  \leq 16,  E  = 1,  D  = 15,  N  = 3$ | (store,2)     | 12   | $5 \times 10^5$ | $2 \times 10^5$ | 33 s    |
| nat    | $ S  \leq 16,  E  = 2,  D  = 3,  N  = 3$  | (store,2)     | 12   | $3 \times 10^5$ | $1 \times 10^5$ | > 5 min |
| set,   | $ S  \leq 4,  E  = 2,  N  = 3$            | (set,2)       | 16   | $2 \times 10^4$ | $1 \times 10^4$ | 0.3 s   |
| elem,  | $ S  \leq 7,  E  = 3,  N  = 3$            | (set,2)       | 16   | $6 \times 10^4$ | $3 \times 10^4$ | 1 s     |
| nat,   | $ S  \leq 8,  E  = 3,  N  = 4$            | (set,3)       | 16   | $1 \times 10^5$ | $5 \times 10^4$ | 2 s     |
|        | $ S  \leq 11,  E  = 4,  N  = 3$           | (set,2)       | 16   | $2 \times 10^5$ | $8 \times 10^4$ | 30 s    |
|        | $ S  \leq 15,  E  = 4,  N  = 4$           | (set,3)       | 16   | $6 \times 10^5$ | $2 \times 10^5$ | 1 min   |
|        | $ S  \leq 16,  E  = 4,  N  = 5$           | (set,4)       | 16   | $8 \times 10^5$ | $3 \times 10^5$ | 14 min  |
| array, | $ A  \leq 7,  E  = 2,  N  = 3$            | (array,2)     | 5    | $5 \times 10^4$ | $2 \times 10^4$ | 2 s     |
| elem,  | $ A  \leq 14,  E  = 2,  N  = 4$           | (array,2)     | 5    | $2 \times 10^5$ | $1 \times 10^5$ | 1 min   |
| nat    | $ A  \leq 17,  E  = 2,  N  = 4$           | (array,3)     | 5    | $4 \times 10^5$ | $2 \times 10^5$ | > 5 min |

spurious counterexamples, which was not the case. After this kind of *empirical* correctness test we artificially introduced different kinds of anomalies which usually occur in the design process: from simple typos in a specification and/or in theorems to omitting some important cases in a definition of an operation or dropping essential preconditions in theorems. Alloy was able to detect all bugs and obviously was tremendously more efficient as compared to a human checker. It basically considers all possible finite models up to the size  $k$  leaving nothing unregarded within the scope. To measure the dynamics of the resources usage we performed a testing under the following changing parameters: bounds on the number of atoms in the universe (for each sort), level of  $s, k$ -completeness of the finite models and the number of the defined operations in the specification. Further, we distinguish between two kinds of analysis: generation of an instance for a given specification (*run* in Alloy notation), see Table 1, and checking a theorem for a specification (*check* in Alloy notation). For all wrong theorems it was enough to consider relatively small finite models, e.g. for stores we limited  $\mathcal{M}$  to  $|S| \leq 9, |E| = 2, |D| = 2, |N| = 3$  and required the *store*, 2-completeness ( $\mathcal{C}_{store}^2$ ). Beside the number of atoms in the universe, which has a major impact on the size of the SAT instance, the arity of the relations included in the signature of Alloy's specifications significantly affects the time to generate the SAT instance (even for moderate instance sizes). For example, in an unsorted universe with only 4 atoms a simple inclusion of a 5-ary relation results in the SAT instance generation time of 3 minutes, although instance size is just  $10^5$  clauses and it is checked in 55 ms. Consequently, for the example with *arrays* we were not able to analyze conjectures involving function  $fill : array \times elem \times nat \times nat \rightarrow array$ , which is a 5-ary relation. Figure 3 shows a finite instance of  $SP_2$  where most of 16 operations are hidden to get a clear picture of the universe. The instance  $\mathcal{M}$  satisfies the highest completeness possible  $\mathcal{C}_{set}^4$  (for  $|E| = 4$  we get the whole power set  $\mathcal{P}(E)$ ).

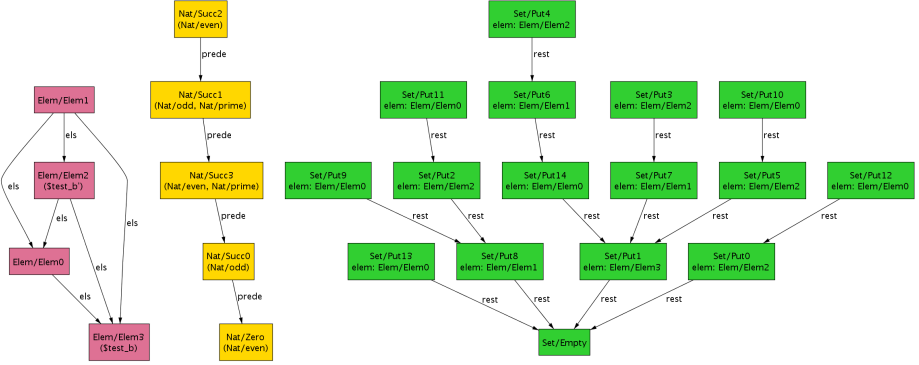


Fig. 3. Finite instance  $\mathcal{M}$  for  $SP_2$  (16 operations hidden for clarity)

## 6 Related Work

There exist various finite model generators: McCune’s tool MACE [16] (translation to propositional logic), Kodkod [32] (used in the most recent version of Alloy), Paradox [8], SEM [35] (model search directly on a set of first-order clauses), FINDER [29], to name only a few. Combination of theorem proving and model finding techniques have been done in a number of case studies. They have in common rather symmetrical objectives: strengthen interactive proving tools by adding automatic methods on one hand and extending automatic tools with interactive formal reasoning on the other, e.g. Alloy has no support for theorem proving. Both reasoning techniques are increasingly seen as complementary activities. One approach is to use automated theorem provers based on resolution or other calculi as a *tactic* in KIV to prove first-order theorems which was investigated in [3] and in [26] with some improvements on exploiting the structure of algebraic theories. McCune’s automated theorem prover Prover9 [18] (successor of Otter [17]) employs the MACE generator for search of countermodels. Reversely, the Paradox model finder has been augmented by an automated first-order theorem prover Equinox [7].

However, automated theorem provers are of limited use, since they do not support induction necessary to reason about algebraic types and recursive definitions. Weber developed a model generator [33] for higher-order logic (HOL) and integrated it in Isabelle [22]. In [19] 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). There exist similar approaches to integrate model checking tools in theorem provers for more efficient identification of reasons why proofs fail, e.g. Pike’s [23] integration of SPIN model checker in PVS [21]. Improving quality of specifications by flaw detection plays a crucial role in practice and always remains an up-to-date issue. Earlier works by Thums et al. [31,27] and Ahrendt [2] considered an error detection in *loose* specifications of abstract data types.

## 7 Conclusion

We have presented a method that is aimed at refutation of unprovable first-order conjectures which in numbers arise in the interactive theorem proving practice. It is also meant to be a complementary quality improvement step in the algebraic specification development process improving efficiency of proof engineers. A big advantage of this approach is its ability to generate finite instances (nicely visualized with Alloy) for counterexamples or witnesses. They can serve as an important source of information to designers and proof engineers, helping in getting better understanding of why their assumptions on models are wrong.

## References

1. The Alloy Project, <http://alloy.mit.edu>
2. Ahrendt, W.: Deductive search for errors in free data type specifications using model generation. In: Voronkov, A. (ed.) CADE 2002. LNCS (LNAI), vol. 2392. Springer, Heidelberg (2002)
3. 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. Kluwer Academic Publishers, Dordrecht (1998)
4. Balsler, M.: Verifying Concurrent Systems with Symbolic Execution. PhD thesis, Universität Augsburg, Fakultät für Informatik (2005)
5. Balsler, M., Bäuml, 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)
6. Balsler, 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. Springer, Heidelberg (2000)
7. Claessen, K.: Equinox, a new theorem prover for full first-order logic with equality. Dagstuhl Seminar 05431 on Deduction and Applications (October 2005)
8. Claessen, K., Srensson, N.: New techniques that improve mace-style model finding. In: Proc. of Workshop on Model Computation (MODEL) (2003)
9. Dunets, A., Schellhorn, G., Reif, W.: Bounded Relational Analysis of Free Data Types. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 99–115. Springer, Heidelberg (2008)
10. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification. Springer, Heidelberg (1985)
11. Ehrig, H., Mahr, B.: Algebraic techniques in software development: A review of progress up to the mid nineties. In: Current Trends in Theoretical Computer Science, pp. 134–152 (2001)
12. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge (2000)
13. Harrison, J.: Inductive definitions: Automation and application. In: TPHOLs, pp. 200–213 (1995)
14. Jackson, D.: Automating first-order relational logic. In: Proceedings of the 8th ACM SIGSOFT Symposium, pp. 130–139. ACM Press, New York (2000)
15. Kuncak, V., Jackson, D.: Relational analysis of algebraic datatypes. In: Proceedings of the 13th ACM SIGSOFT Symposium (2005)

16. McCune, W.: Mace4 reference manual and guide (2003)
17. McCune, W.: Otter 3.3 reference manual (2003)
18. McCune, W.: Prover9 manual (April 2008)
19. Meng, J., Quigley, C., Paulson, L.C.: Automation for interactive proof: First prototype. *Inf. Comput.* 204(10), 1575–1596 (2006)
20. Mosses, P.D.: CASL Reference Manual, The Complete Documentation of the Common Algebraic Specification Language. LNCS, vol. 2960. Springer, Heidelberg (2004)
21. Owre, S., Rushby, J.M., Shankar, N.: PVS: A Prototype Verification System. In: Kapur, D. (ed.) CADE 1992. LNCS (LNAI), vol. 607, pp. 748–752. Springer, Heidelberg (1992)
22. Paulson, L.C.: Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow). LNCS, vol. 828. Springer, Heidelberg (1994)
23. Pike, L., Miner, P., Torres-Pomales, W.: Diagnosing a failed proof in fault-tolerance: A disproving challenge problem. In: DISPROVING 2006 Participants Proceedings, pp. 24–33 (2006)
24. 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)
25. Reif, W.: Korrektheit von Spezifikationen und generischen Moduln. PhD thesis, Universität Karlsruhe, Germany (1991) (in German)
26. 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)
27. 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)
28. Schellhorn, G.: Verification of Abstract State Machines. PhD thesis, Universität Ulm, Fakultät für Informatik (1999), [www.informatik.uni-augsburg.de/swt/Publications.htm](http://www.informatik.uni-augsburg.de/swt/Publications.htm)
29. Slaney, J.K.: Finder: Finite domain enumerator - system description. In: CADE, pp. 798–801 (1994)
30. 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)
31. Thums, A.: Fehlersuche in Formalen Spezifikationen. Master's thesis, Universität Ulm, Germany (1998) (in German)
32. 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)
33. Weber, T.: SAT-based Finite Model Generation for Higher-Order Logic. PhD thesis, Institut für Informatik, Technische Universität München, Germany (April 2008)
34. zChaff SAT solver, <http://www.princeton.edu/chaff/zchaff.html>
35. Zhang, J., Zhang, H.: Sem: a system for enumerating models. In: IJCAI, pp. 298–303 (1995)