Flaw detection in formal specifications

Wolfgang Reif, Gerhard Schellhorn, Andreas Thums

Angaben zur Veröffentlichung / Publication details:

Reif, Wolfgang, Gerhard Schellhorn, and Andreas Thums. 2001. "Flaw detection in formal specifications." In Automated Reasoning: First International Joint Conference, IJCAR 2001, Siena, Italy, June 18-23, 2001, proceedings, edited by Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, 642–57. Berlin: Springer. https://doi.org/10.1007/3-540-45744-5_52.

Flaw Detection in Formal Specifications

Wolfgang Reif, G. Schellhorn, and Andreas Thums

Lehrstuhl für Softwaretechnik und Programmiersprachen,
Universität Augsburg, D-86135 Augsburg
{reif, schellhorn, thums}@informatik.uni-augsburg.de

Abstract. In verification of finite domain models (model checking) counterexamples help the user to identify, why a proof attempt has failed. In this paper we present an approach to construct counterexamples for first-order goals over infinite data types, which are defined by algebraic specifications. The approach avoids the implementation of a new calculus, by integrating counterexample search with the interactive theorem proving strategy. The paper demonstrates, that this integrations requires only a few modifications to the theorem proving strategy.

1 Introduction

It is common knowledge, that most of time in theorem proving is not spent on successful, but on unsuccessful proof attempts. Usually the reason is simply that the theorem under consideration is wrong. For data types with a finite domain, the counterexamples generated by model checkers help the proof engineer to detect the errors. Also tools which generate finite models are available. Unfortunately, there is no such mechanism for data types with an infinite domain. Additionally, proofs for infinite data types usually have to be found interactively and decisions made during the proof attempt may be incorrect.

In this paper we will demonstrate a method which generates counterexamples for infinite data types in algebraic specifications. The method is implemented in the interactive theorem prover KIV [7,2] which supports algebraic specifications in the style of CASL [5] as well as state based specifications (using imperative programs or ASMs [3]). It avoids the need to implement a new calculus by adaption of the existing proof calculus to the construction of counterexamples.

Our paper is organized as follows: Section 2 will first give an informal overview of the method, which is based on reduction of an unprovable goal to *false*. The capabilities of the algorithm will be demonstrated with an example and we will discuss the limitations of the approach.

After introducing some necessary notation (Sect. 3), Sect. 4 describes how conjectures may be falsified by reduction, thereby proving the existence of a counterexample. The backtracing algorithm, that computes the actual variable assignment and the model condition is given in Sect. 5. A number of important implementation issues are discussed in section 6. Some examples and a comparison to related work is given in Sect. 7. The paper concludes with Sect. 8.

2 Overview

To demonstrate the capabilities of the algorithm, let us start with an example:

Example 1
We assume a data type **list** *over a parameter type* **elem**. *Lists are generated by the constructors* nil *for the empty list, and* $e+l$, *which adds an element* e *to a list* l. *The elements are assumed to be totally ordered with* $\leq$. head *and* tail *select* e *resp.* l *from a list* $e + l$, length(l) *computes the length of a list,* append(l_1, l_2) *appends two lists,* member(e, l) *tests, if the element* e *is in* l, *and* sort(l) *states that* l *is (nonstrictly) sorted with respect to the ordering on elements. Then the conjecture*

$$\varphi \equiv \quad sort(l_1) \wedge l_2 = \text{append}(l_1, head(l_3) + nil) \wedge length(l_3) \geq 2 * length(l_1) \\ \wedge\, l_3 \neq nil \wedge member(head(l_1), tail(l_3)) \rightarrow sort(l_2)$$

does not hold in all models of the specification. A counterexample consists of the variable assignment $l_1 := e_1 + nil, l_2 := e_1 + e_2 + nil, l_3 = e_2 + append(l_4, e_1 + l_5)$ *(e_1, e_2 are variables of type* **elem***) and the* model condition $\neg\, e_1 \leq e_2$.

The example has explicitly been chosen to be somewhat contrived for two reasons: First, it should demonstrate several aspects of the counterexample search. Second, we want to stress, that formulas encountered during interactive proof attempts are even *more complex* than the example given above and counterexamples are totally non-obvious. Our algorithm finds the counterexample within 5 seconds, which is faster than we (and we hope the reader) can guess one by inspecting the goal.

In contrast to a proof, where it is tried to reduce the conjecture to true, the counterexample search tries to reduce the conjecture to false. If this is successful, the conjecture is not provable and a counterexample exists. Because a conjecture is usually not false in general, we try to find a variable assignment under which the conjecture can be reduced to false.

The variable assignment is usually built up by incrementally instantiating variables of generated sorts with constructors, resulting in constructor terms. A counterexample for Example 1, which uses constructor terms only, would set l_3 to e_2+e_1+nil. Often this incremental construction can be shortcut by combining it with the application of proof rules. By imposing some restrictions (see Sect. 4) on the ones that may be used, the more abstract counterexample shown above can be derived.

Incrementally instantiating a variable in a conjecture gives several subgoals, one for each constructor. Since each subgoal could lead to a counterexample, we have implemented a heuristic search strategy to choose a case, which (hopefully) quickly leads to a counterexample.

Construction of a variable assignment terminates (at the latest) when all variables of generated sorts have been instantiated by constructor terms. The remaining formula may be *false*, but we may also end up with a *model condition*. Model conditions occur for three reasons: First, as in the example above, we may have parameter sorts, and the model condition may impose a restriction on the

models of the parameter: in the example above $\neg\ e_1 \leq e_2$ ensures that there exist two different elements in **elem**: in models, where **elem** contains just one element, the conjecture would be true. A second reason for model conditions is underspecification: E.g., if *pred(0)*, the predecessor of zero is not specified (over natural numbers), then the counterexample search may finish with a model condition like *pred(0)* $>$ *3*. Finally, the interactive proof strategy may be too weak, to deduce automatically, that a fully instantiated goal is equivalent to false. This case usually arises, when not enough simplifier rules have been provided for the data structure under consideration. Since these would be needed to automate (successful) proofs anyway, we simply restart the counterexample search after adding the required rules.

We have found that the success of counterexample generation depends mainly on the efficiency of the heuristic search strategy that computes variable assignments. If the strategy is successful, we usually end up with trivial model conditions, like the ones shown above. Therefore we have not invested in the automation of the check, whether the model condition is consistent with the specification. Instead we simply ask the user to acknowledge satisfiability.

After the original goal has been reduced to *false* (or a model condition) the variable assignment for the original goal has to be deduced. This is done by a backtracing algorithm, that analyzes the reduction backwards from *false* to the beginning of the counterexample search (see Sect. 5). Because the attempt to compute a counterexample will usually not be made on an initial conjecture, the counterexample will also be traced back further in the original proof attempt.

3 Formal Basics

The basis for our approach are first-order algebraic specifications, consisting of a finite, many-sorted signature with function symbols F, a finite set of axioms (first-order formulas) and a set of constructors $C \subseteq F$ (typically, constructors are given by clauses like *nat* **generated by** *0*,*suc*). A sort is called a *target sort*, if it is generated by some constructors, otherwise it is called a *parameter sort*. A *constructor term* is a term that uses only constructor functions and variables of parameter sorts. If syntactically different constructor terms always represent different elements, the data type is called a *free* or *freely generated* data type. **nat** is a free data type while *int* **generated by** *0*, *suc*, *pred* is not free, because $i = pred(suc(i))$.

We assume loose semantics, i.e. an algebra $\mathcal{A}$ is in the class *Mod(SP)* of models of a specification *SP*, iff the axioms hold in $\mathcal{A}$ and if every element contained in the carrier set of some target sort can be represented as the semantic value of a constructor term, given a suitable valuation for the parameter variables.

In the following, we will use x, y as (metavariables for) variables, f, g as (constructor) function symbols τ, ρ as terms, and φ, ψ, χ as formulas. Free variables $free(\varphi)$ are defined as usual. $gen(\varphi)$ are the free variables of target sorts (also called *target variables*), $param(\varphi)$ are all other free variables. φ^t_x denotes substitution in formulas.

Sequents $\Gamma \vdash \Delta$ consists of two lists $\Gamma = \varphi_1, \ldots, \varphi_n$ and $\Delta = \psi_1, \ldots, \psi_m$ of formulas. $\Gamma \vdash \Delta$ abbreviates the formula $\varphi_1 \wedge \ldots \wedge \varphi_n \rightarrow \psi_1 \vee \ldots \vee \psi_m$.

We will use a sequent calculus for first-order logic (like the one defined in [13]) to derive proofs for conjectures. Proofs in this calculus are trees, in which every node contains a sequent. Starting with a proof tree of one node, which contains the initial goal c, a fixed, predefined set of proof rules of the form $\frac{p_1 \ldots p_n}{c}$ is applied to reduce the goal to simpler subgoals. c is called the *conclusion*, $p_1 \ldots p_n$ are called the *premises* of the rule. By recursive application of further rules to the resulting premises, a proof tree is built up. The conclusion c is proven if all premises in the tree are reduced to the trivially true premise *true*.

Given these preliminaries on specifications and proofs, we can now define the central notion of a *counterexample*:

Definition 1 (counterexample)
Let φ be a formula over a specification SP *with $\underline{x} = \text{gen}(\varphi)$. Then a pair $\mathcal{C} = (\chi,\ \underline{y} = \underline{t})$ is called a counterexample for the formula φ iff $\underline{y}$ is a vector of target variables, $\underline{t}$ is a list of terms such that $y_i \notin t_j$ for $i \leq j$, $\text{gen}(\chi) = \emptyset$ and there is a model $\mathcal{A}$ such that*

$$\mathcal{A} \in \text{Mod}(SP \cup Cl_{\exists}\ \chi) \tag{1}$$

and

$$SP \ \models\ (\chi \wedge \underline{y} = \underline{t}) \rightarrow \neg\ \varphi \tag{2}$$

A counterexample for φ is called a strong *counterexample, if $\underline{y} \subseteq \underline{x}$.*

The definition requests that there is a model $\mathcal{A}$ of the specification *SP*, in which χ is satisfiable (1) and that for every model of the specification either χ does not hold or $\neg\ \varphi$ is true under the variable assignment $\underline{y} = \underline{t}$ (2). The restriction $y_i \notin t_j$ for $i \leq j$ ensures that no cyclic dependencies between variables y_i and terms t_j exists.

We can now prove that with this definition every non-provable formula has a counterexample:

Theorem 1
A formula $\neg\ \varphi$ is satisfiable over SP *iff there exists a (strong) counterexample for φ.*

Proof 1
If a formula $\neg\ \varphi$ is satisfiable, there exists a model $\mathcal{A} \in Mod(SP)$ and a valuation v where $\mathcal{A}, v \models \neg\ \varphi$. Now, let $gen(\varphi) = \underline{x}$. Since the model is generated we have constructor terms $\underline{t}$ and valuations v_i, such that $v_i(x_i) = v_i(t_i)$. By suitably renaming variables in $\underline{t}$ we can find a common valuation v', such that $\mathcal{A}, v' \models \neg\ \varphi$ and $v'(x_i) = v'(t_i)$ for every i. Setting $\chi := \neg\ \varphi^{\underline{t}}_{\underline{x}}$ and $\mathcal{C} := (\chi,\ \underline{x} = \underline{t})$, we have found a counterexample: the substitution theorem of first-order logic implies $\mathcal{A}, v' \models \chi$, and this implies condition (1). Obviously, we also have $SP \ \models\ (\neg\ \varphi^{\underline{t}}_{\underline{x}}\ \wedge \underline{x} = \underline{t}) \rightarrow \neg\ \varphi$, i.e. condition (2) is satisfied too.

Conversely, assume that a formula φ has a counterexample. Choose a model $\mathcal{A} \in Mod(SP)$ and a valuation v such that $\mathcal{A}, v \models \chi$. This is possible because of (1). Then the constraint $y_i \notin t_j$ for $i \leq j$ allows to sequentially modify $v(y_1), \ldots, v(y_n)$, such that for the resulting, modified valuation v' $\mathcal{A}, v' \models \underline{y} = \underline{t}$ holds. Since χ does not contain target variables, we still have $\mathcal{A}, v' \models \chi$. Finally, property (2) implies $\mathcal{A}, v' \models \neg\, \varphi$, i.e. $\neg\, \varphi$ is satisfiable.

The proof shows, that we could restrict the terms assigned to the variables to be constructor terms. We have not done so, to allow more abstract counterexamples as in Example 1.

Finally we would like to note, that for efficiency reasons our algorithm will not construct *strong* counterexamples. Instead it will sometimes assign values to variables, that are only present in intermediate goals. This is not problematic, since it is easy to prove that any counterexample can be transformed to a strong counterexample by iterative application of the following post processing step:

Lemma 1
Let $(\chi, \underline{y} = \underline{t})$ be a counterexample for φ with $x = t \in \underline{y} = \underline{t}$ and $x \notin free(\varphi)$. Let $\underline{y}' = \underline{t}'$ be $\underline{y} = \underline{t}$ with $x = t$ removed. Then $(\chi,\ (\underline{y}' = \underline{t}')^t_x)$ is a counterexample for φ too.

4 Falsifying Conjectures

We found that with a simple restriction sequent calculus is already suitable to do satisfiability reasoning instead of proving theorems. This restriction, called invertibility, will be discussed in the first subsection. A second advantage of using sequent calculus is that we do not have to use a brute-force search for the correct instantiation which is needed for the counterexample. Instead we use an incremental instantiation strategy called structural expansion as discussed in the second subsection. Finally, the last subsection exemplifies, how the combined proof strategy falsifies a conjecture.

4.1 Existence of a Counterexample

To find counterexamples we cannot use every rule of the sequent calculus due to the fact that some of them weaken the conjecture. This means that some rules reduce a provable conjecture (with no counterexample) to one which is no longer provable (and therefore has a counterexample). We call rules which do not weaken conjectures *invertible rules.*

Definition 2 (invertible rule)

A rule $\frac{p_1 \ldots p_n}{c}$ is called invertible, iff for every model $\mathcal{A}$ of the specification:

$$\text{from } \mathcal{A} \models c \text{ follows } \mathcal{A} \models p_i \text{ for } 1 \leq i \leq n \tag{3}$$

This means, that if the conclusion of a rule is valid in every model $\mathcal{A}$ of the specification then every premise p_i is valid in that model. This condition is sufficient to show the existence of a counterexample.

Lemma 2 (existence of a counterexample)
Let T be a proof tree consisting of invertible rules, c the conclusion and $p_1 \ldots p_n$ the premises of T.

If there exists an i, such that $\neg\ p_i$ is satisfiable then there exists a counterexample for the conclusion c.

Proof 2
This lemma is proven by contradiction to the validity of the conclusion c. Assuming c is valid, definition 2 and transitivity imply that all p_i are valid. This contradicts the existence of an i, such that $\neg\ p_i$ is satisfiable.

□

Fortunately, almost every rule of the sequent calculus is invertible, especially all rules that eliminate propositional connectives. The main exception is of course the rule *weakening* which drops formulas from the sequent. This rule must therefore be avoided when searching for a counterexample. The rules *all right* (shown below, x_0 is a new variable) and *exists left* to eliminate quantifiers are also invertible. Two other critical rules are the quantifier instantiation rules *all left* and *exist right*. These rules are invertible only, if we use a version that does not drop the quantifier as shown below for *all left* .

$$\frac{\Gamma \vdash \varphi_x^{x_0}, \Delta}{\Gamma \vdash \forall\ x.\varphi, \Delta}\ \text{all right} \qquad \frac{\forall\ x.\varphi, \varphi_x^t, \Gamma \vdash \Delta}{\forall\ x.\varphi, \Gamma \vdash \Delta}\ \text{all left}$$

4.2 Searching Counterexamples

To compute the variable assignment for a counterexample, we could in principle use a generate and test algorithm, which would enumerate all constructor terms. E.g. for a formula involving two variables *m, n* over natural numbers, we could try to instantiate the variables with all pairs $(suc^i(0),\ suc^j(0))$ of constructor terms. But this would be very inefficient. Consider e.g. a goal of the form φ (m,n) $\wedge$ m $\leq$ n. Then for n = 0 it is totally redundant to consider any instance of m except 0, since the formula obviously has no counterexample for other cases of m.

Therefore a more elaborate search for the existence of a counterexample is needed, which exploits constraints (like m $\leq$ n) present in the conjecture to avoid the enumeration of as many cases as possible. Therefore the variables are instantiated stepwise with respect to the constructors and thereafter proof rules simplify the resulting formula. If, for example, variable n is instantiated with 0 (and $suc(n_1)$), then the conjecture can immediately be simplified to $\varphi(0,\ 0)$ (by the fact m $\leq$ 0 $\leftrightarrow$ m = 0), thereby avoiding any enumeration of instances of m. This method of stepwise instantiation is called *structural expansion.*

Definition 3 (structural expansion)
Let φ be a formula with a free variable x and $sort(x) = s$ generated by $f_1 \dots f_n$. Furthermore let $\underline{x_1} \dots \underline{x_n}$ be proper argument vectors for $f_1 \dots f_n$ that are not free in φ. Then the step from $\varphi(x)$ to

$$(\varphi(f_1(\underline{x}_1)) \vee \dots \vee \varphi(f_n(\underline{x}_n)))$$

is called structural expansion.

Structural expansion spans a search tree and partly instantiates the free variables of the conjecture. Because the structural expansion stepwise enumerates every constructor term – for a variable every possible prefix is generated – the complete search space is generated. This is true because the data types are generated by the constructors, i.e. every value of a variable can be represented by a constructor term. The structural expansion may be included in the sequent calculus through an additional rule called *constructor cut*.

Definition 4 (constructor cut)
The constructor cut rule has the form

$$\frac{(\Gamma \vdash \Delta)_x^{f_1(\underline{x}_1)} \quad \dots \quad (\Gamma \vdash \Delta)_x^{f_n(\underline{x}_n)}}{\Gamma \vdash \Delta} \text{ constructor cut}$$

where $x \in gen(\varphi)$ and $sort(x) = s$ generated by $f_1, \dots f_n$.

To use this rule in the counterexample search it has to be correct and invertible.

Lemma 3 (correctness and invertibility of *constructor cut*)
The rule constructor cut *(see Def. 4) is correct and invertible.*

Because the *constructor cut* rule is itself a weak form of the induction principle, the proof is easy using structural induction.

Adding the *constructor cut* rule to the proof search and avoiding non invertible rules are the only changes needed to adapt the usual sequent calculus to counterexample search.

To have a complete proof strategy in the sense, that every vector of constructor terms $\underline{t}$ is considered, we have to use a *fair* search strategy, that assures, that every branch is considered ultimately as well as every variable is ultimately expanded with the *constructor cut* rule. Then every constructor term instance of every variable will ultimately be tried as a potential counterexample. A simple fair search strategy would be breadth-first search on the branches of the proof tree, combined with expanding one of the variables that has been expanded the fewest times before. A more elaborate strategy is discussed in Sect. 6.2.

4.3 Generating a Counterexample Proof

We will now exemplify the counterexample search.

Example 2
The following conjecture

$$\varphi \equiv \text{sort}(l_1) \wedge \text{sort}(l_2) \rightarrow \text{sort}(\text{append}(l_1, l_2))$$

does not hold in all models of the specification. A counterexample consists of the variable assignment $l_1 := e_1 + nil, l_2 := e_2 + nil$ *and the* model condition $\neg\ e_1 \leq e_2$.

The following proof tree gives an impression how the counterexample generation works.

Example 3

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{e_1 \leq e_2}{\text{sort}(e_2 + nil) \rightarrow \text{sort}(\text{append}(e_1 + nil, e_2 + nil)) \quad \ldots}\ \text{simplify}}{\ldots \quad \text{sort}(e_2 + l_2') \rightarrow \text{sort}(\text{append}(e_1 + nil, e_2 + l_2'))}\ \text{con. cut } l_2'}{\text{sort}(l_2) \rightarrow \text{sort}(\text{append}(e_1 + nil, l_2))}\ \text{con. cut } l_2}{\text{sort}(e_1 + nil) \wedge \text{sort}(l_2) \rightarrow \text{sort}(\text{append}(e_1 + nil, l_2)) \quad \ldots}\ \text{simplify}}{\ldots \quad \text{sort}(e_1 + l_1') \wedge \text{sort}(l_2) \rightarrow \text{sort}(\text{append}(e_1 + l_1', l_2))}\ \text{con. cut } l_1'}{\text{sort}(l_1) \wedge \text{sort}(l_2) \rightarrow \text{sort}(\text{append}(l_1, l_2))}\ \text{con. cut } l_1$$

Starting with the conjecture (conclusion of the proof tree) the rule *constructor cut* stepwise introduces the constructor terms which build up the counterexample. Dots indicate branches of the proof tree, which are not relevant to the outcome. Applications of *constructor cut* are alternated with applications of sequent calculus rules. In our case only the *simplifier* rule is needed, which simplifies goals using rewrite rules: in the example the recursive definitions of *append* and *sort* are used as rewrite rules to simplify e.g. *sort*$(e_1 + nil)$ to true. The search stops at the formula $e_1 \leq e_2$, because there are no more target variables to instantiate and the predicate may not be automatically decided. Under the assumption that $\neg\ e_1 \leq e_2$ is satisfiable – which the user acknowledges to be true – there exists a counterexample for the conclusion.

5 The Counterexample

In section 4 we discussed how to prove the existence of a counterexample. In this section we will discuss how to reconstruct the variable assignment from a failed proof attempt, by tracing an initial variable assignments back through the proof attempt. The first subsection considers the proof generated by the counterexample search which uses invertible rules only whereas the second subsection extends the approach to arbitrary sequent calculus proofs.

5.1 Computing the Counterexample

We will compute the counterexample backwards beginning at a premise ψ. According to Sect. 4, this premise is either *false*, or a satisfiable formula without free target variables. In either case, $\mathcal{C} = (\neg\ \psi, \emptyset)$ is a counter example. The task is now to compute a counterexample for every goal contained in the branch of the considered premise by *backtracing* through it. To adapt the counterexample from φ' to the previous formula φ in the backtrace path the rule which deduces φ' from φ has to be considered. Some rules of the sequent calculus have the property that the variable assignment has to be adjusted. We define rules where the assignment needs no adjustment *strong invertible rules*.

Definition 5 (strong invertible rule)

A rule $\frac{p_1 \ldots p_n}{c}$ *is called strong invertible, iff* $\mathcal{A} \models\ c\ \rightarrow p_1\ \wedge \ldots \wedge p_n$ *holds for every model* $\mathcal{A}$ *of the specification.*

It is easy to prove that this definition implies:

Lemma 4 (backtrace through strong invertible rules)

Let $\frac{p_1 \ldots p_n}{c}$ *be a strong invertible rule and* $\mathcal{C} = (\chi,\ \underline{x} = \underline{t})$ *a counterexample for a premise* p_i*. Then* $\mathcal{C}$ *is also a counterexample for the conclusion* c*.*

In the sequent calculus most rules that are invertible, are also strong invertible. Especially all propositional rules, and the quantifier rules are strong invertible as well as the rule to apply a lemma. The only invertible rules, which are not strong invertible are *insert equation*, *structural induction*, and *constructor cut*.

Example 4 (insert equation)
The insert equation *rule*

$$\frac{\Gamma_x^\tau\ \vdash \Delta_x^\tau}{x = \tau, \Gamma \vdash \Delta}\ \textit{insert equation}$$

with $x \notin vars(\tau)$ *is an invertible rule. Let*

$$\frac{\vdash \text{prime}(4)}{x = 4 \vdash \text{prime}(x)}\ \textit{insert equation}$$

be an instantiation of the rule where the predicate prime(x) *is true iff* x *is a prime number. Because* 4 *is not prime there exists a counterexample* $\mathcal{C} = (true,\ \emptyset)$ *with no conditions for this premise. But for a model* $\mathcal{A}$ *and the variable assignment* $v(x) = 3$ *the antecedent of the conclusion may be refuted, therefore in general* $\mathcal{C}$ *is not a counterexample for the conclusion.*

The example shows, that the counterexample for the premise is not a counterexample for the conclusion. We have to adapt the counterexample by adding the dropped equation: $\mathcal{C}' = (true, x = 4)$. In general the adaption for the three invertible (but not strong invertible) rules are as follows:

Theorem 2 (adaption for *insert equation*)
Let $\mathcal{C} = (\chi,\ \underline{x} = \underline{t})$ *be a counterexample for the premise of the* insert equation *rule. Then* $\mathcal{C}' = (\chi,\ (\underline{x} = \underline{t})^{x_0}_x \ \wedge x = \tau)$ *(where* x_0 *is a new variable) is a counterexample for the conclusion.*

The *structural induction* is similar to the *constructor cut* rule (see Def. 4), but adds an induction hypothesis of the form $\forall\ \underline{y}.(\Gamma \rightarrow \Delta)^{x_i}_x$ for every x_i in $\underline{x}$ ($\underline{y}$ are all free variables of the conclusion $\Gamma \vdash \Delta$ except the induction variable x).

Theorem 3 (adaption for *structural induction* and *constructor cut*)
Let $\mathcal{C} = (\chi,\ \underline{x} = \underline{t})$ *be a counterexample for the premise with* $f_i(\underline{x}_i)$ *of the* structural induction *or the* constructor cut *rule. Then* $\mathcal{C}' = (\chi,\ (\underline{x} = \underline{t})^{x_0}_x) \ \wedge x = f_i(\underline{x}_i))$*, where* x_0 *is a new variable is a counterexample for the conclusion.*

The correctness proofs for this adaption is not too difficult, details are given in [10]. In the following, we will shows the backtrace algorithm applied on Ex. 3.

Example 5
The backtrace begins at the premise. Because the negation of $e_1 \leq e_2$ *is satisfiable the counterexample for the premise is* $\mathcal{C} = (\neg\ e_1 \leq e_2, \emptyset)$*. At the simplifier rule no adaption is necessary, but at the* constructor cut *rule* $(l'_2 = nil)$ *the counterexample is adapted to* $\mathcal{C} = (\neg\ e_1 \leq e_2, l'_2 = nil)$ *and stepwise further to* $\mathcal{C} = (\neg\ e_1 \leq e_2, (l'_2 = nil, l_2 = e_2 + l_2))$ *which also holds for the premise of the* simplification rule, $\mathcal{C} = (\neg\ e_1 \leq e_2, (l'_2 = nil, l_2 = e_2 + l_2, l'_1 = nil))$, *and* $\mathcal{C} = (\neg\ e_1 \leq e_2, (l'_2 = nil, l_2 = e_2 + l_2, l'_1 = nil, l_1 = e_1 + l'_1))$*. Finally, neither* l'_1 *nor* l'_2 *appear in the conclusion, the counterexample may be simplified to the strong counterexample* $\mathcal{C} = (\neg\ e_1 \leq e_2, (l_2 = e_2 + nil, l_1 = e_1 + nil))$ *using lemma 1.*

Because the search for a counterexample applies only invertible rules and the counterexample can be backtraced over these rules, a variable assignment for the free variables of disproved conjectures may be derived.

5.2 Earliest Point of Failure

In interactive proof systems the user does not only look at the conjecture he wants to prove, but also at formulas which will be derived during a proof. Therefore the counterexample search will usually not be started at the conjecture under consideration – why should one try to prove a conjecture assumed to be wrong? – but at some 'fishy' goal within the proof tree. If a counterexample can be found for that goal the users interest is whether some proof decisions were incorrect or

whether nevertheless the conjecture is faulty. Therefore the counterexample has to be backtraced in the original proof attempt tree as well.

Because the counterexample search already uses the (restricted) proof rules, the backtrace algorithm of Sect. 5.1 may also be used for backtracing in the original proof, as long as only invertible rules have been used. For non-invertible rules (like weakening formulas) the best that may be done are two proof attempts: the first tries to show that the counterexample can be propagated through the rule (by proving $(\chi \wedge \underline{x} = \underline{t}) \rightarrow \neg\, \varphi$), the second tries to show that it surely cannot be propagated (by proving $(\chi \wedge \underline{x} = \underline{t}) \rightarrow \varphi$). If the first proof attempt succeeds, backtracing can proceed. If the second attempt succeeds, the earliest point of failure has been found, i.e. application of the rule is at least one of the reasons why the proof attempt failed. In KIV two incomplete proof attempts are done just using the built-in simplifier (these do not cost much time). If both fail, more elaborate proof attempts are optional, but rarely used. Since wrong conjectures are much more common than wrong proof decisions, one usually skips non-invertible rules and tries to compute a counterexample for the conclusion first. Only if this computed counterexample is found not to be an actual counterexample for the conclusion, because it was incorrectly propagated through a non-invertible rule, more elaborate checks are necessary.

6 Implementation

The previous sections have shown the approach for the computation of counterexamples on infinite data types. This approach is tightly integrated in the proof engine of the KIV system. When the user proves a conjecture and gets stuck at some subgoal he might start the counterexample search. This strategy takes the subgoal and tries to reduce it to false with a special set of counterexample heuristics. These heuristics expand a free target variable structurally and simplify the resulting premises. Thereafter the premises are weighted (see Sect. 6.2) and a 'good' premise, i.e. one that hopefully quickly leads to a counterexample, is chosen and structurally expanded again. This process continues as long as one premise has no more free target variables. If this premise is *false* or the user agrees, that the negation is satisfiable, the backtrace algorithm evaluates the variable assignment for the free target variables of the subgoal. Thereafter the counterexample may also be backtraced in the original proof attempt to the earliest point of failure (see Sect. 5.2).

In this section we will first discuss implementation details that are specific to the the sequent calculus and the heuristics to automate proofs used in KIV. Second we will give some heuristics to get an efficient search strategy for the counterexample proofs and finally shortly discuss extensions to Dynamic Logic and to higher-order logic.

6.1 KIV Specific Rules and Heuristics

To simplify formulas with axioms and theorems from the underlying specification KIV uses a *simplifier*. The simplifier does propositional reasoning and

applies theorems from the specification as forward, elimination or conditional rewrite rules. Details on the various types of rules can be found in [9]. Additional heuristics do quantifier instantiations and unfolding of function definitions.

To optimize the proof search for non-free data types, a variant of the structural expansion rule is implemented. The modified rule adds preconditions to each premise. If variable x is expanded to $f(\underline{x})$ in a premise, inequations $f(\underline{x}) \neq x_i$ for each argument $x_i \in \underline{x}$ that has the same sort as x are added. This ensures, that the constructor does not behave like identity on some argument. E.g. when expanding a variable s for a set to $s' \cup \{a\}$, the inequality $s' \cup \{a\} \neq s'$ is added, which ensures that the element a was not already present in s'. This halves the search tree because the case of the identity is omitted.

6.2 Search Strategies for the Counterexample

Instantiating variables using structural expansion has to be done in a fair way: Every premise and every variable of each premise has to be eventually considered. KIV uses an A^* search strategy to implement a fair search. The strategy computes a weight for each open goal and always continues the counterexample search at the branch with the smallest weight. To be fair, it has to be made sure, that there is a $\delta > 0$, such that the weight of goals increases at least by δ, when another rule is applied [11]. This is easily achieved by adding some weight proportional to the branch length. Further criteria rely on the assumption, that less complicated goals lead more quickly to a counterexample.

Therefore the complexity is measured by considering the number of formulas and different variables of the goal, the number of times, the structural expansion was applied, and the complexity of the symbols corresponding to the *specification hierarchy*. KIV supports structured algebraic specifications [9] with the usual operations (union, enrichment, renaming, parameterization and actualization) which form this hierarchy. E.g. the specification of **list** (see Ex. 1) may be an enrichment of a list specification with the definition of the *sort* predicate, which may in turn be a generic specification with **elem** as parameter. Usually axioms, that define operations in specifications higher up in the hierarchy are based on symbols below in the hierarchy (like the definition of *sort* is based on elementary list constructors). Therefore symbols higher up in the hierarchy carry more weight than those lower in the hierarchy.

The expansion of a variable should reduce the complexity of the current formula. Therefore variables get a weight corresponding to the specification hierarchy of the sort, the number of occurrences (since expansion expands all occurrences), and, to get a fair strategy, a negative weight proportional to the number of times the variables have been expanded. Then the variable with the highest weight is chosen.

The exact factors used in the weights were chosen by evaluating a number of experiments. They do a good job in practice.

6.3 Extending the Method

Because the presented method for counterexample search uses a proof calculus, the approach may easily be extended. The KIV system supports program verification of imperative programs in Dynamic Logic [8]. To extend the counterexample search to Dynamic Logic, we only had to check that the program rules (mainly rules for the symbolic execution of programs) are (strong) invertible. We found that almost all are strongly invertible, with only one exception: In Hoare's invariant rule, the introduction of an incorrect program invariant leads to a not invertible rule. Therefore this rule is not used in the counterexample search for programs and backtracing through the proof stops at the invariant rule. The counterexample search for program verification is also implemented and works with good results. The basic strategy is to expand the input variables and then to execute the programs until the program formulas are reduced to pure predicate logic formulas. Then the usual counterexample search can proceed.

Another extension of the proof calculus would be to apply it to higher-order goals (KIV already uses a higher-order logic). The new proof rules (beta reduction and rules for the axioms of choice and extensionality) are not problematic, since they are strongly invertible, but higher-order variables present a problem, which we could not solve yet: since functions are not generated data types, we do not have a simple mechanism available, that can find instances of function variables.

7 Related Work and Examples

A large number of tools has been developed which try to construct finite models [12,15,14], even for first-order specifications [4]. Since our specifications usually do not have finite models at all, these tools are not directly applicable. Nevertheless an interesting alternative to our approach would be to use them by defining suitable abstractions (e.g. by collapsing all data structures with more than a fixed number of constructors into a single element). Model generator tools could also be used to check model conditions over parameter specifications, since these often can be satisfied with finite models.

The idea to incrementally search for constructor terms to generate the search space for counterexamples also appears in [6] and in [1]. The first approach is restricted to the special case of initial specifications over free data types with complete recursive definitions. In this case the satisfiability of $\neg\ \varphi^{t}_{\underline{x}}$ can be decided by unfolding definitions and the inequality between syntactically different constructor terms. The special case allows to prove a completeness result for the counterexample search.

The second approach in [1] is restricted to freely generated data types, and tries to integrate counterexample search with the calculus of a model generation prover.

Both approaches differ from ours in that they try to develop a fully automatic calculus for the counterexample search, while we have developed an approach

that is tightly integrated with the paradigm of interactive theorem proving. Nevertheless our approach is able to solve the problems given in [6] within a few seconds. No residual formula remain, the solutions are found automatically.

Summarizing, we have developed a practically applicable approach to counterexample generation, which can deal with arbitrary data types. Below we will give a small number of examples for graphs. Graphs g are generated from the empty graph $\emptyset$ by inserting nodes a and edges $a \mapsto b$ with g $+_v$ a and g $+_e$ $(a \mapsto b)$. Adding an edge $(a \mapsto b)$ implicitly adds both nodes a and b. The data type is not free, since g $+_v$ a $+_v$ a = g $+_v$ a. A path in a graph will be written $[a_1, \ldots, a_n]$. Here are some conjectures over graphs and the corresponding counterexamples:

1. every graph g is acyclic:
   $\mathcal{C} = (true, (g = \emptyset +_e (a \mapsto a)))$
2. every path x in a graph g is a shortest path (i.e. no path y is shorter):
   $\mathcal{C} = (true, (g = \emptyset +_e (a \mapsto b), x = [a, b], y = [a]))$
3. if g has no edge of the form $(a \mapsto a)$, then it is acyclic:
   $\mathcal{C} = (a \neq b, (g = \emptyset +_e (a \mapsto b) +_e (b \mapsto a)))$
4. two paths x and y in an acyclic graph g with same source and same destination are equal:
   $\mathcal{C} = (a_0 \neq a_1 \wedge a_1 \neq a_2 \wedge a_0 \neq a_2, (x = [a_0, a_1], y = [a_0, a_2, a_1],$
   $g = \emptyset +_e (a_0 \mapsto a_2) +_e (a_2 \mapsto a_1) +_e (a_0 \mapsto a_1)))$

Our approach solves the first two examples automatically, the other two require that the user acknowledges the model condition. The examples are typical for KIV case studies, which often use abstract data types like the one above. They are atypical, since the goals are small and only one data type with only a few axioms is involved. More examples over larger specifications can be found in [10].

8 Conclusion

We have presented an approach for the generation of counterexamples in algebraic specifications. The counterexample search is based on a sequent calculus for first-order logic (and on Dynamic Logic for program verification). The approach extends other approaches limited to free data types, and has been implemented into the specification and verification tool KIV.

The use of the existing proof calculus lead to a relative short implementation time, and allowed to reuse the already existing, well-tested and efficient strategies for theorem proving in the search for a counterexample.

The counterexample mechanism already has been successfully applied in several applications as a supporting feature to aid the proof engineer in detecting flaws in conjectures.

The applications have shown that the satisfiability of model conditions usually is decided easily by the user. Nevertheless an interesting topic for further research would be to further automate this test. Some cases are easy to automate, e.g. terms like *pred(0)* or *head(nil)* could easily be checked to be unspecified by

inspecting the specification of natural numbers or lists. Model conditions for parameters like $\neg\ e_1 \leq e_2$ could be solved either by a decision procedure for total orders or by a model generation tool (finite models often are sufficient for this case). Care has to be taken, that the automation does not waste a lot of unnecessary time, since usually the variable assignment computed is already sufficient to deduce why a conjecture is wrong.

Another topic for further research are restrictions on the search space for non-free data types. Currently only the restriction mentioned at the end of Sect. 6.1 has been implemented, but commutativity constraints like $g +_v\ a +_v\ b = g +_v\ b +_v\ a$ (for graphs) or cancellation laws like $n + 1 - 1 = n$ (for integers) can also be exploited to reduce the search space.

References

1. W. Ahrendt. A basis for model computation in free data types. In *CADE-17 Workshop on Model Computation - Principles, Algorithms, Applications*, 2000.
2. M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in LNCS. Springer, 2000.
3. M. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.
4. D. Jackson. Automating first-order relational logic. In *Proceedings of the ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering (FSE-00)*, volume 25, 6 of *ACM Software Engineering Notes*, pages pp. 130 – 139. ACM press, 2000.
5. P. D. Mosses. CoFI : The common framework initiative for algebraic specification. In H. Ehrig, F. v. Henke, J. Meseguer, and M. Wirsing, editors, *Specification and Semantics*. Dagstuhl-Seminar-Report 151, 1996. http://www.brics.dk/Projects/CoFI.
6. M. Protzen. Disproving conjectures. In D. Kapar, editor, *Automated Deduction – CADE 11*, volume 607 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.
7. W. Reif. The KIV-approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, LNCS 1009. Springer, Berlin, 1995.
8. W. Reif, G. Schellhorn, and K. Stenzel. Interactive Correctness Proofs for Software Modules Using KIV. In *COMPASS'95 – Tenth Annual Conference on Computer Assurance*, Gaithersburg (MD), USA, 1995. IEEE press.
9. W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications*. Kluwer, Dordrecht, 1998.
10. W. Reif, G. Schellhorn, and A. Thums. Fehlersuche in formalen Spezifikationen. Technischer Bericht 2000-06, Fakultät für Informatik, Universität Ulm, Germany, 2000. (in German).
11. S. Russell and P. Norwig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.
12. J. Slaney. FINDER: Finite domain enumerator. *Lecture Notes in Computer Science*, 814:798ff, 1994.

13. V. Sperschneider and G. Antoniou. *Logic: A Foundation for Computer Science*. Addison Wesley, 1991.
14. H. Zhang. SATO: An efficient propositional prover. In *CADE-97*, 1997.
15. J. Zhang and H. Zhang. SEM: A system for enumerating models. In *IJCAI-95*, 1995.