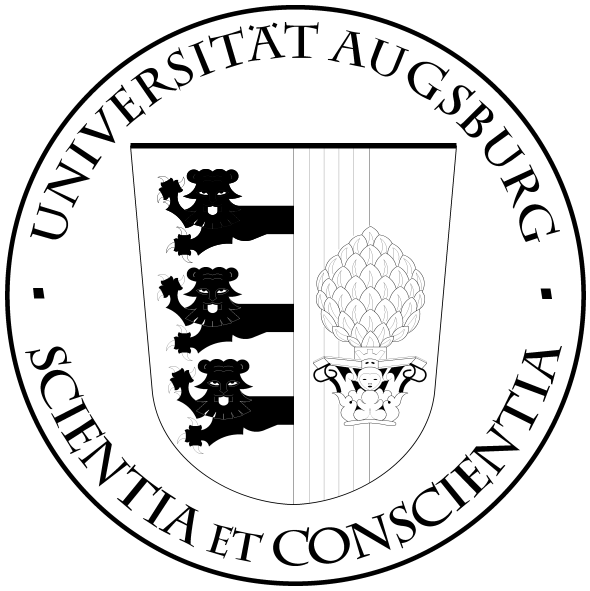
Automatische Fehlersuche in Algebraischen Spezifikationen

Dissertation zur Erlangung des Doktorgrades Dr. rer. nat.
der Fakultät für Angewandte Informatik
der Universität Augsburg
im Jahr 2009 von

Andriy Dunets



Amtierender Dekan:	Prof. Dr. Wolfgang Reif
Gutachter:	Prof. Dr. Wolfgang Reif
	Prof. Dr. Bernhard Bauer
Tag der mündlichen Prüfung:	11. Februar, 2010
Prüfer	Prof. Dr. Wolfgang Reif
	Prof. Dr. Bernhard Bauer
	Prof. Dr. Bernhard Möller

Danksagung

Herzlich danke ich

- meinem Doktorvater **Prof. Wolfgang Reif** für sein Vertrauen, für die gegebene Möglichkeit diese Arbeit zu verwirklichen, sein umfassendes Wissen sowie seine fachlichen Anleitungen und Kritik
- **Dr. Gerhard Schellhorn** für die zahlreichen fruchtbaren Diskussionen und seine fachliche Kompetenz
- **Simon Bäumler, Dr. Frank Ortmeier** und **Dr. Michael Balser** für die fachliche und menschliche Unterstützung vom Anfang an
- allen Partnern aus dem EU-Projekt **Protocure**
- meinen Kollegen **Andreas Angerer, Matthias Güdemann, Dr. Holger Grandy, Dr. Dominik Haneberg, Alwin Hoffmann, Florian Nafz, Dr. Jonathan Schmitt, Jan-Philipp Steghöfer, Nina Moebius, Hella Seebach, Dr. Kurt Stenzel** und **Michael Vistein** für ihre freundschaftliche Arbeitsatmosphäre
- **Lili**, für Deine Liebe, Verständnis und Ermutigung, Du hast mir immer den richtigen Weg gedeutet
- Meinem guten Großvater der meine Begeisterung für die Mathematik und Informatik geweckt hat
- Meiner klugen Mutter die mich stets liebevoll gefördert hat.

Andriy Dunets

Kurzzusammenfassung

Abstrakte Datentypen eignen sich sehr gut für die High-Level Spezifikation von Software. Die algebraischen Spezifikationen von abstrakten Datentypen bringen durch die Strukturierung gute Überschaubarkeit des Entwurfs und erlauben die Ausdrucksstärke der Logik der ersten Stufe. Wir haben einen Ansatz zur Fehlersuche mittels endlichen Modellsuche entworfen und auf mehreren Fallstudien evaluiert. Die entwickelte Methodik ermöglicht automatische Fehlersuche in den algebraischen Spezifikationen der freien und nicht-freien Datentypen. Bei Axiomatisierungen der Operationen kann die resultierende Semantik auf unendlichen Strukturen sich von der Semantik auf endlichen Teilstrukturen unterscheiden. Wir haben formalen Kriterien aufgestellt, die die Erhaltung der Semantik auf endlichen Teilmodellen und damit die Korrektheit des Ansatzes garantieren. Die Technik wurde in Java implementiert und in das KIV System eingebaut. Die Implementierung verwendet den Modellgenerator und Constraint Solver Kodkod.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einführung

1.1 Motivation

Formale Methoden spielen eine immer wichtigere Rolle beim Entwurf und der Realisierung von Softwaresystemen. Mit steigender Komplexität der zu entwerfenden Systeme steigt auch die Wahrscheinlichkeit Fehler in Designs zu produzieren. Testen ist eine alte Technik zur Entdeckung der Problemstellen in Systemrealisierungen, die sich schon immer bewährt hat. Leider ist es vor allem bei größeren Systemen schwierig mit dieser Technik eine hohe Fehlerentdeckungsrate zu erzielen. Dieses Thema ist der Gegenstand der aktuellen Forschung. Eine alternative Methode die Qualität der Systemdesigns sicherzustellen ist die Verifikation, mit der die totale Fehlerfreiheit der Spezifikationen gezeigt werden kann. Anderseits, ist die Verifikation eine nicht-produktive und potentiell sehr kostenlastige Tätigkeit, die von den meisten Softwareingenieuren als unattraktiv betrachtet wird. Die Verifikationswerkzeuge müssen möglichst automatisch sein, damit sie schließlich von Entwicklungsingenieuren auch benutzt werden. Der Markt für solche Werkzeuge wird durch die Kunden bestimmt, die Angst vor dem Aufwand für den Einsatz von mathematischen Methoden haben aber auch meistens schlecht informiert sind. Damit ist es für die integrierten Methoden zur Fehlersuche wichtig, ihre Effizienz deutlich zu machen und gleichzeitig die Komplexität zu verbergen. Als Folge kann damit der Angstfaktor gemindert werden.

Bei der Fehlersuche während der Systementwicklung verschieben sich die Schwerpunkte von der Suche nach technischen Fehlern, wie z.B. Speicherzugriffsfehler, in die Richtung der anwendungsspezifischen Fehlern, wie z.B. die Verletzung der Sicherheitseigenschaften. Dies ist die Folge der Modellgetriebenen Entwicklung, bei der mittels der ausdrucksstarken Spezifikationssprachen die anwendungsspezifischen Aspekte des entwickelnden Systems aufgefasst werden und die technischen Aspekte weggelassen werden. Anschließend, generieren die verifizierten Compiler automatisch aus solchen Spezifikationen den Code. Die Fehlerfreiheit der Spezifikation garantiert automatisch die Fehlerfreiheit des Codes. Die zur Zeit auf dem Markt verfügbaren automatischen Verfikationstools, auch als Model Checker bekannt, unterstützen meistens nur in ihrer Ausdrucksstärke sehr beschränkte automatenbasierten Sprachen. Daher sind sie für solche high-level

Spezifikationen eher ungeeignet.

In dieser Arbeit wird eine Technik zur automatischen Fehlersuche in den algebraischen Spezifikationen entwickelt. Eine Realisierung wurde basierend auf einem in MIT entwickelten Modellgenerator Alloy [22] und Constraint Solver Kodkod [54] durchgeführt sowie anhand mehrerer Fallstudien evaluiert. Dabei wurden verschiedene praktischen und theoretischen Problemstellungen untersucht und gelöst.

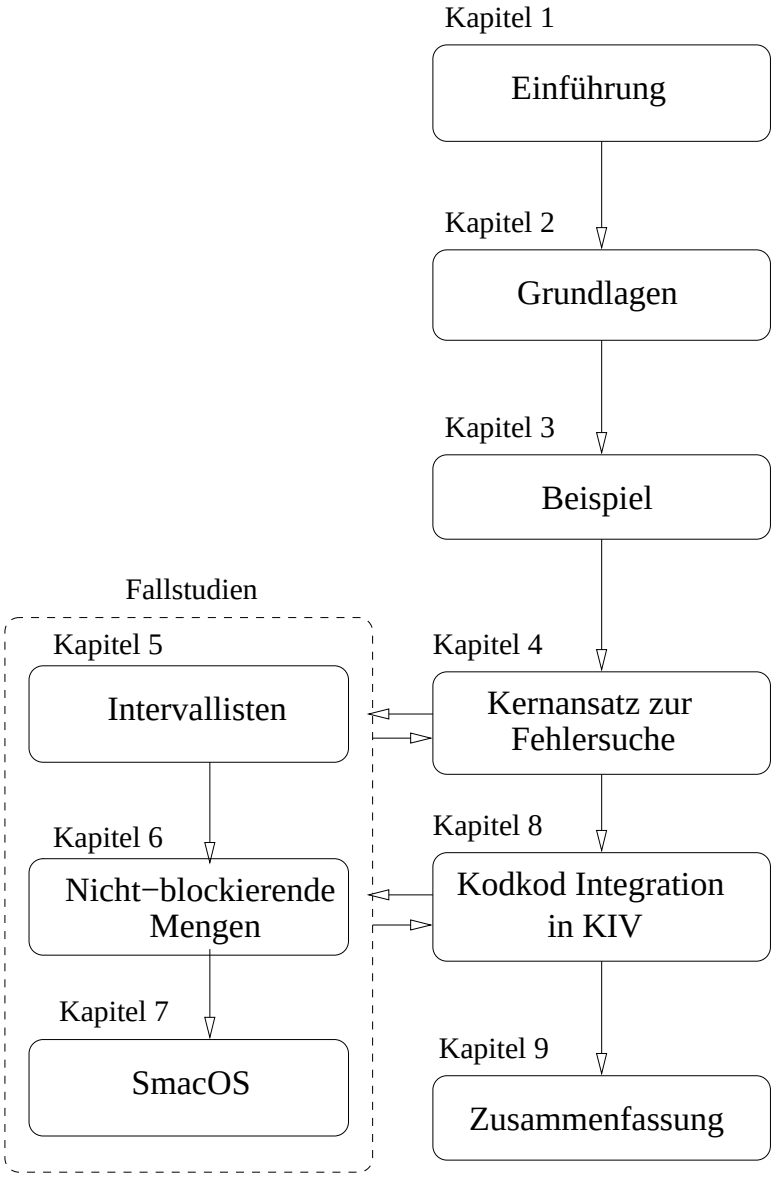


Abbildung 1.1: Zusammenhänge zwischen den einzelnen Kapiteln dieser Arbeit.

1.2 Überblick über die Arbeit

Die Arbeit besteht aus den folgenden drei Hauptteilen: die Kernmethodik zur Analyse und Fehlersuche in algebraischen Spezifikationen, die Fallstudien, Integration von Kodkod in das KIV System. Die Abbildung 1.1 visualisiert die Aufteilung der Arbeit.

Der Ansatz dieser Arbeit wurde speziell für das Spezifikations- und Beweissystem KIV [15] entwickelt und in das System integriert.

Kapitel 2 umfasst die theoretischen Grundlagen sowie eine Beschreibung der Werkzeuge, die in dieser Arbeit verwendet wurden, insbesondere wird der weltweit bekannte und in zahlreichen Projekten eingesetzte relationale Modellgenerator Alloy präsentiert.

Anhand eines einfachen Beispiels der Spezifikation von Listen wird im Kapitel 3 anschaulich die Motivation für diese Arbeit erläutert.

Der Kern der Arbeit ist im Kapitel 4 zu finden. Es wird der Ansatz der endlichen Modellsuche, die damit verbundenen Komplikationen und die Lösungen vorgestellt. Ferner werden die Klassen der analysierbaren Spezifikation und der Theoreme formal definiert und die Korrektheit des Ansatzes bewiesen. Die Hauptschwierigkeit war ein formales Kriterium zu entdecken, der die Korrektheit der endlichen Modellsuche garantiert.

Schließlich werden in Kapiteln 5 und 6 die Ergebnisse der Anwendung dieser Technik auf zwei überschaubare Fallstudien präsentiert. Dabei wurde der Modellgenerator Alloy verwendet.

Es wurde eine weitere Fallstudie realisischer Größe für die Evaluierung hinzugezogen. Es handelt sich um die formale Spezifikation des Sicherheitsmodells eines multiapplikativen Smartcardbetriebssystems “SmacOS”. Die Fallstudie “SmacOS” und die Ergebnisse aus der Anwendung der Technik sind im Kapitel 7 enthalten.

Kapitel 8 beschreibt die Realisierung der Technik in Java mit Kodkod API [54] und die Integration in das KIV System. Kodkod ist ein Nachfolger von Alloy, der ebenfalls im Laboratorium für künstliche Intelligenz an MIT entwickelt wurde. Ein Vorteil von Kodkod ist die Unterstützung für *partial instance* (Teillösungen), die wo die Modellberechnung in Teilschritte zerlegt wird. Wir berichten über erste Erfahrungen mit dem inkrementellen Ansatz, die eine zukünftige Realisierung motivieren.

Kapitel 9 fasst die erreichten Resultate und die gemachten Erfahrungen zusammen und zeigt einen Ausblick.

1.3 Zusammenfassung der Ergebnisse

Die wichtigsten Ergebnisse dieser Dissertation werden in folgenden Punkten zusammengefasst:

- **Entwurf eines Ansatzes zur Fehlersuche in algebraischen Spezifikationen mittels der endlichen Modellsuche**

Es wird ein generischer Ansatz zur Anwendung der endlichen Modellsuche auf die algebraischen Spezifikationen von abstrakten Datentypen in KIV beschrieben. Die meisten Modellgeneratoren sind *stand-alone* Werkzeuge, die nicht in die Theorembeweiser integriert sind. Der Ansatz unterstützt sowohl die freien als auch die nicht-freien Datentypen und definiert formale Kriterien für eine korrekte Fehlersuche.

- **Java Implementierung und Integration der Technik in das KIV System**
  Die entwickelten Techniken wurden unter Verwendung von Kodkod API in der Programmiersprache Java implementiert. Das Ergebnis dieser Arbeit ist eine nahtlose Integration von dem Modellgenerator Kodkod in das KIV System.

- **Auswertung auf größeren Fallstudien**
  Ein Nachweis der Anwendbarkeit des Ansatzes auf die Systemdesigns realistischer Größe wird durch die Evaluation auf zwei größeren Fallstudien gemacht.

- **Testen der algebraischen Spezifikationen**
  Bisher war es nicht möglich eine algebraische Spezifikation zu testen. Die einzige Alternative war es einen formalen interaktiven Beweis einer Eigenschaft zu machen. Nun ist es mit unserem Ansatz möglich ähnlich wie bei den Programmen einen Testablauf (ein endliches Teilmodell) zu berechnen und auch auf die Einhaltung von gewünschten Eigenschaften automatisch zu überprüfen.

- **Erste Versuche mit der inkrementellen Modellgenerierung für eine bessere Skalierbarkeit**
  Die in größeren Fallstudien gemachten Erfahrungen haben gezeigt, daß eine Erweiterung der Technik zwecks besseren Skalierbarkeit sehr hilfreich wäre. Es wurden erste Experimente mit der inkrementellen Technik zur Modellgenerierung gemacht. Die Ergebnisse sind vielversprechend und motivieren eine Realisierung in der Zukunft.

Kapitel 2

Grundlagen

In diesem Kapitel werden die Grundlagen der Arbeit beschrieben. Es wird ein kurzer Überblick über das KIV System und die darin verwendeten Formalismen gegeben. Anschließend wird der am MIT entwickelte und in Rahmen dieser Arbeit eingesetztes Werkzeug Alloy Analyzer vorgestellt.

2.1 KIV System

KIV [15, 36, 38, 5] ist ein Werkzeug zur formalen Entwicklung von Software-basierten Systemen. Es unterstützt die Entwickler beim Entwurf, Validierung und Verifikation des Systemdesigns. Entwurf eines zukünftigen Systems wird mit den strukturierten algebraischen Spezifikationen repräsentiert. Eine gute Einführung in unterschiedliche Aspekte der algebraischen Spezifikationen bietet das Buch von Loeckx et al. [27]. Dank der ergonomischen Benutzerschnittstelle [16] und den effizienten Beweistechniken lassen sich mit KIV auch sehr große Systementwürfe spezifizieren. KIV hat sich bereits in zahlreichen großen Fallstudien [32, 14, 43, 45, 44] als sehr erfolgreich erwiesen. Dabei wird das System kontinuierlich verbessert und erweitert. Einige dieser Fallstudien sind über die Internetseite des Lehrstuhls [25] verfügbar.

KIV unterstützt sowohl funktionales als auch zustandsbasiertes Vorgehen bei der Spezifikation. Die grundlegende Logik kombiniert Higher Order Logik (HOL) und Dynamische Logik (DL) [17]. Es wurden aber auch fortgeschrittene formale Spezifikationssprachen in das System integriert: Statecharts [6], Abstract State Machines (ASM) [42], Java Kalkül [50, 51] und Temporallogik [4].

2.2 Algebraische Spezifikationen in KIV: Syntax und Semantik

Die Spezifikationshierarchien werden mittels der Operationen *enrichment* (Erweiterung um zusätzliche Operationen und ihre Axiomatisierung), *union*, *parameterization* und *actualization* (Instantiierung der parametrisierten Spezifikati-

on) aufgebaut. Datentypen lassen sich mit speziellen *data specifications* spezifizieren.

Definition 1 *[Signatur]*
Eine Signatur $\Sigma = (S, \mathcal{F}, \mathcal{P})$ enthält eine Menge von Sorten S, und Mengen von Operationen $\mathcal{F}$ und $\mathcal{P}$, wobei $\mathcal{F}$ vereint die Mengen $\mathcal{F}_{\underline{s}\to s}$ der Funktionssymbolen $f : s_1 \times \ldots s_n \to s$ und $\mathcal{P}$ - die Mengen $\mathcal{P}_{\underline{s}}$ der Prädikatensymbolen $p : s_1 \times \ldots \times s_n$.

Eine abzählbar unendliche Menge X von Variablen vereint die Mengen X_s von Variablen der Sorte s. Mit den Funktionssymbolen und Variablen aus den Mengen $\mathcal{F}_0 \subseteq \mathcal{F}$ und $X_0 \subseteq X$ lassen sich die Terme der Menge $T(\mathcal{F}_0, X_0)$ konstruieren. Darauf aufbauend ist die Menge $For(\Sigma)$ der Formeln über die Signatur Σ definiert.

Um die Datenstrukturen adäquat zu spezifizieren werden die Induktionsaxiome benötigt. Da die Induktion in der Logik der ersten Stufe nicht ausdrückbar ist, werden neben den First-Order Formeln die *Generiertheitsklauseln* benutzt.

Definition 2 *[Generiertheitsklauseln]*
Für eine Signatur Σ enthält die Menge $Gen(\Sigma)$ die Generiertheitsklauseln "s **generated by** *C_s". Solche Klausel legt fest, daß die Trägermenge der Sorte s mit den Konstruktoren aus C_s generiert wird. Die Menge der Konstruktoren $C_s = \{c_1, \ldots, c_n\} \subseteq \mathcal{F}_{\underline{s}\to s}$ muß zusätzlich mindestens eine Konstruktorfunktion $c_i : \underline{s} \to s$ enthalten, bei der alle Argumentensorten sich von der Sorte s unterscheiden ($\underline{s} \in (S \backslash \{s\})^*$).*

Mit diesen Konstrukten läßt sich der Begriff einer *Spezifikation* definieren.

Definition 3 *[Spezifikation]*
Eine nicht parametrisierte Spezifikation $SP = (\Sigma, Ax, Gen)$ ist ein Tripel bestehend aus einer Signatur Σ, einer endlichen Menge $Ax \subset For(\Sigma)$ der Axiome und der Menge $Gen \subset Gen(\Sigma)$ der Generiertheitsklauseln für jede generierte Sorte.

Die *Modelle* einer Spezifikation sind mathematische Strukturen, die auch als *Algebren* bezeichnet werden.

Definition 4 *[Σ-Algebra] Für die Signatur Σ die Menge der Σ-Algebren $Alg(\Sigma)$ enthält Strukturen $\mathcal{A} = ((\mathcal{A}_s)_{s\in S}, (f_\mathcal{A})_{f\in\mathcal{F}}, (p_\mathcal{A})_{p\in\mathcal{P}})$, wobei $\mathcal{A}_s$ bezeichnet eine nicht leere Trägermenge für die Sorte s und die Operationen $f_\mathcal{A}$, $p_\mathcal{A}$ sind die konkrete Interpretationen der Operationssymbolen f, p auf den Domänen $\mathcal{A}_s$.*

Auf einer Algebra $\mathcal{A}$ läßt sich die Semantik einer Generiertheitsklausel definieren:

$$\mathcal{A} \models s \textbf{ generated by } C_s \;\Leftrightarrow$$
$$\texttt{for all } a \in \mathcal{A}_s \texttt{ exists } \alpha,\ t \in T_s(C_s, X \backslash X_s)\ :\ a = \ [\![t]\!]^{\mathcal{A},\alpha}$$

Damit wird garantiert, daß in der Algebra $\mathcal{A}$ die Trägermenge $\mathcal{A}_s$ genau aus den s-wertigen Konstruktortermen $t \in T_s(C_s, X \backslash X_s)$ besteht, wobei die Variablen aus $X \backslash X_s$ mittels der Evaluierungsfunktion α entsprechend belegt werden.

Die Semantik einer Spezifikation wird durch die Menge aller Modelle, die diese Spezifikation erfüllen, definiert.

Definition 5 *[Modell]*
Eine Algebra $\mathcal{A}$ ist ein Modell der Spezifikation $SP = (\Sigma, Ax, Gen)$, $\mathcal{A} \in Mod(SP)$, wenn

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

Eine Formel $\varphi \in For(\Sigma)$ ist gültig für eine Spezifikation SP, wenn

$$SP \models \varphi \;\Leftrightarrow\; \text{für alle } \mathcal{A} \in Mod(SP) \;:\; \mathcal{A} \models \varphi$$

2.3 Alloy Analyzer und Mehrsortige Relationale Logik

Alloy Analyzer [22] ist ein Werkzeug zur Automatisierung der mehrsortigen Relationalen Logik der ersten Stufe mit dem transitiven Abschluß [23]. Zusätzlich zu den Prädikaten $p \in \mathcal{P}_{\underline{s}}$ enthält die Logik die *Relationalen Terme* $r \in \mathcal{R}_{\underline{s}}$. Die atomaren Formeln haben die folgende Form:

$$\begin{aligned} A &\equiv x_1 = x_2 \mid RT(x_1, \ldots, x_n) \\ RT &\equiv P \mid RT_1.RT_2 \mid {}^{\wedge}P \end{aligned}$$

wobei RT ein n-stelliger relationaler Term ist. Ein relationaler Term RT ist entweder ein Prädikatensymbol P (eine n-stellige Relation) oder die Komposition $RT_1.RT_2$ von zwei relationalen Termen oder ein transitiver Abschluß ${}^{\wedge}P$ einer binären Relation P. Die Formeln in Alloy können beliebig quantifiziert werden.

Eine Alloy Spezifikation SP_{Alloy} beschreibt ein endliches mehrsortiges Universum $U = (D_{s_1}, \ldots, D_{s_n}, \gamma)$ mit endlichen Mengen der Atome D_{s_i} für jede Sorte s_i und der Abbildung γ, die die relationalen Symbole $p \in \mathcal{P}$ auf den Domänen D_{s_i} interpretiert.

Für die Spezifikation SP_{Alloy} und eine Formel (ein Theorem) φ kann Alloy in zwei grundlegenden Betriebsarten benutzt werden. Es kann die Suche nach einem Gegenbeispiel gestartet werden (ein konkretes Universum, das $SP_{Alloy \wedge \neg\varphi}$ erfüllt). Anderseits, kann Alloy auch einen Zeugen berechnen, der $SP_{Alloy} \wedge \varphi$ erfüllt.

Ein einfaches Beispiel [3] illustriert die Stärke des Modellgenerators Alloy bei der Suche nach speziellen Lösungen, die den menschlichen Fähigkeiten weit überlegen ist.

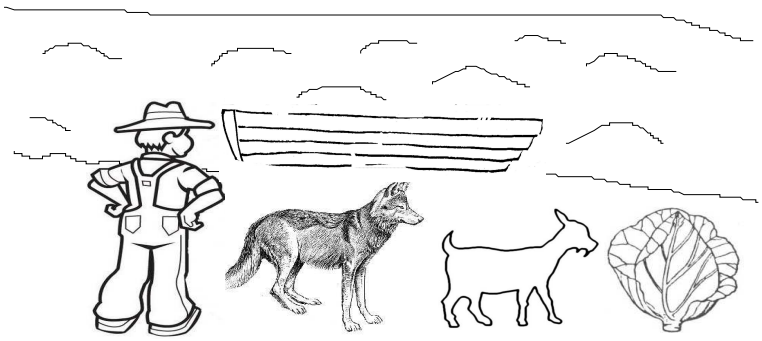


Abbildung 2.1: Ein Beispiel, der die Stärken von Alloy zeigt.

Beispiel.

Betrachten wir das bekannte Problem, das einen Farmer beschäftigt, der mit seinen Gütern auf die andere Seite des Flusses will. Er steht mit einem Wolf, einem Schaf sowie einem Kohl auf einer Seite eines breiten Flusses. Ferner, steht ein Boot zur Verfügung. Allerdings, kann der Farmer mit dem Boot nur eines der drei Güter zur gleichen Zeit transportieren. Zusätzlich, darf er nicht zur gleichen Zeit Wolf und Ziege bzw. Ziege und Kohl alleine auf einem Ufer lassen, da ansonsten das Erste das Zweite frisst. Wie er ohne Verluste auf die andere Seite kommt ist keine triviale Frage.

Da Alloy die relationale Logik der ersten Stufe benutzt sowie keine syntaktischen Einschränkungen auf die Quantifizierung macht, lassen sich solche Probleme elegant und unkompliziert spezifizieren:

```
module farmer

// import: verlinkte Liste von States mit next, first und last
open util/ordering[State] as ord

// signature
abstract sig Object { eats: set Object }
one sig Farmer, Wolf, Ziege, Kohl extends Object {}

sig State {
   near: set Object,
   far: set Object,
   next: State
}

// axioms
fact eating { eats = Wolf->Ziege + Ziege->Kohl }
fact initialState {
   let s0 = ord/first | s0.near = Object && no s0.far
}
fact stateTransition {
  all s: State, s': ord/next[s] {
    Farmer in s.near =>
      crossRiver[s.near, s'.near, s.far, s'.far] else
      crossRiver[s.far, s'.far, s.near, s'.near]
  }
}
```

```
fact Next {
  all s: State, s': ord/next[s] | s.next = s'
}

// predicates
pred crossRiver [from, from', to, to': set Object] {
  // Farmer nimmt nichts mit
  ( from' = from - Farmer &&
    to' = to - to.eats + Farmer ) ||
  // oder Farmer nimmt etwas mit
  (some item: from - Farmer {
    from' = from - Farmer - item
    to' = to - to.eats + Farmer + item
  })
}
pred solvePuzzle {
     ord/last.far = Object
}


// suche nach einem Modell, daß die oberen Constraints erfuellt
run solvePuzzle for 8 State
```

Die Signatur einer Alloy Spezifikation enthält die Deklarationen eines mehrsortigen Universums und der Relationen auf den Atomen (Schlüsselwort **sig**). Ferner, werden die Axiome, die das Verhalten der Relationen einschränken, in die Spezifikation aufgenommen (Schlüsselwort **fact**). Für eine bessere Strukturierung der verwendeten Formeln können auch die Prädikate definiert werden (Schlüsselwort **pred**).

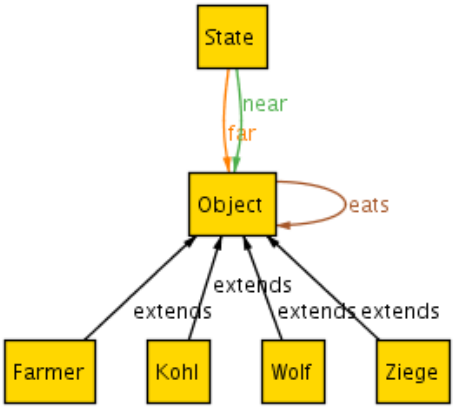


Abbildung 2.2: Metamodell der Signatur.

Nun kann Alloy aufgefordert werden ein Modell (ein endliches Universum) zu berechnen, das die Deklarationen und die Axiome erfüllt. Dies wird mit der Anweisung

$$\textbf{run} < Formel > \textbf{for} < Scope >$$

gemacht. Die hier verwendete Formel schränkt die Suche weiter ein, d.h. das berechnete Modell muss auch diese Formel erfüllen. Es müssen auch die Schranken für die Größe des gesuchten Universums definiert werden.

In unserem Beispiel haben wir zwei Sorten von Atomen: *Object* und *State*. Die Sorte *Object* ist in Untersorten unterteilt: *Farmer, Wolf, Kohl, Ziege*. Wir verlangen, daß diese Untersorten jeweils aus einem Atom bestehen (Schlüsselwort **one**). Ferner, werden Relationen $eats : Object \times Object$, $far : State \times \mathcal{P}(Object)$ und $near : State \times \mathcal{P}(Object)$ deklariert. Die Relationen *far* und *near* charakterisieren einen Zustand, d.h. sie ordnen einem Zustand die Mengen von Objekten, die auf der jeweiligen Seite des Flusses stationiert sind. Mit Alloy läßt sich auch ein Metamodell generieren, das die Deklaration des Universums übersichtlich visualisiert, siehe Abbildung 2.2. Die Zustandsmenge wird als eine geordnete Liste deklariert (*first,next,last*). Dies geschieht durch die Importierung des Moduls *ordering*, das mit der Sorte *State* parametrisiert wird. Die Lösungsidee zum Rätsel ist die Entscheidungsschritte des Farmers als ein Zustandsübergangssystem zu Spezifizieren, wobei im initialen Zustand alle Objekte auf einer Seite des Flusses sind und im finalen Zustand - auf der anderen. Diese Zusammenhänge werden in den Axiomen *initialState, stateTransition* und *solvePuzzle* reflektiert. Die "Spielregeln" werden im Prädikat *crossRiver* beschrieben. Das Prädikat *solvePuzzle* verlangt, daß im letzten Zustand alle Objekte auf der anderen Seite des Flusses sind.

Mit dem Befehl

```
run solvePuzzle for 8 State
```

wird das Modell generiert, das die gesuchte Entscheidungsabfolge enthält. Die Abbildung 2.3 zeigt das von Alloy berechnetes Modell mit Zuständen und den Objekten, sowie Zuordnungen der Objekte zu den Zuständen. Der initiale Zustand und der finale Zustand lassen sich leicht anhand der *near* und *far* Relationen erkennen. Der Zustand *State0* ist der initiale Zustand, weil hier alle Objekte mit *near* zugeordnet sind, d.h. sind auf der einen Seite das Flusses. Analog ist der Zustand *State7* der finale Zustand, siehe - *far* Relation. Die Lösung des Rätsels ist eine Entscheidungsabfolge, die aus der Folge von Zuständen (*next* Relation) ablesbar ist.

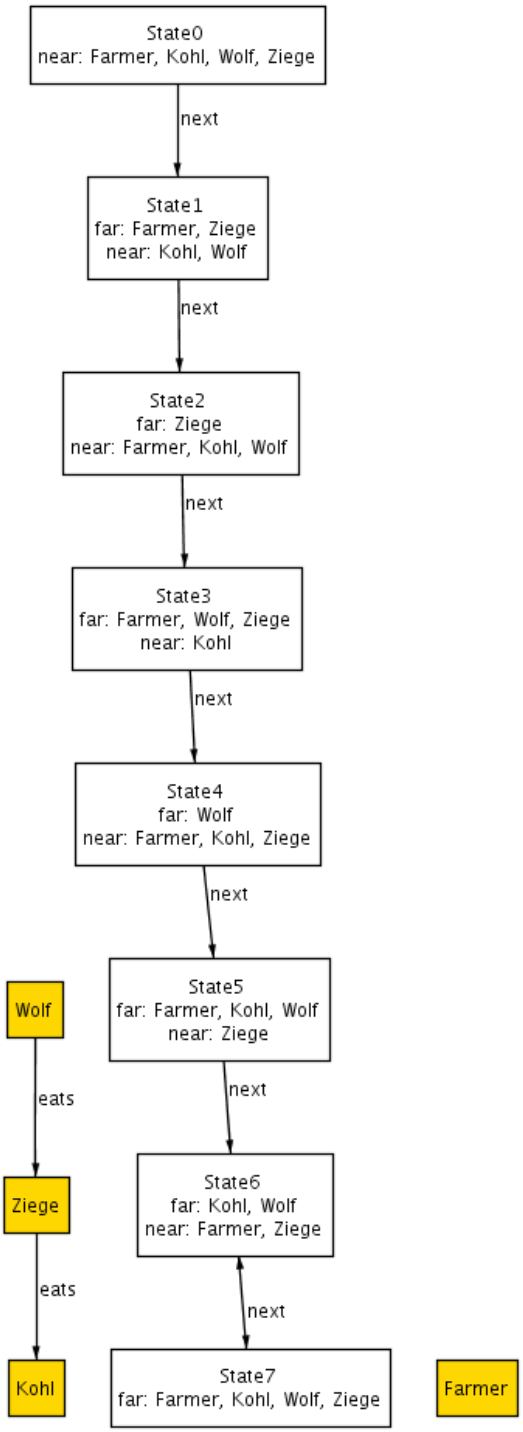


Abbildung 2.3: Von Alloy berechnetes Modell, das die Eigenschaften einer Lösung besitzt.

Kapitel 3

Beispiel: Listen von Intervallen

Der initiale Impuls für viele Theorien kommt bekanntlich aus der Praxis. Es wurde beobachtet, daß bei der Spezifikation und der anschließenden deduktiven Analyse von Softwaredesigns die gemachten Fehler erst später und relativ ineffizient von Beweisingenieuren entdeckt werden. Als Folge der fehlgeschlagenen deduktiven Beweisführungen ist erheblicher Zeit- und Kraftverlust und insgesamt der ineffiziente Einsatz menschlicher Fähigkeiten zu beklagen. Im wesentlichen sind es zwei Arten von Fehlern: Fehler in der Spezifikation und die falsch formulierten Lemmas, die in späteren Beweisen von wichtigen Eigenschaften verwendet werden. Solche Lemmas werden in größeren Fallstudien zu Hunderten formuliert, öfters ohne die notwendigen Sorgfalt und einen formalen Beweis. Ein automatischer Assistent, der dem Beweisingeniuer bei seiner Tätigkeit hilft einen großen Teil der Fehler zu entdecken, ist die Grundmotivation dieser Arbeit. Die gewonnene Zeit, die dem Menschen erspart wird, kann anschließend in viel kreativere Tätigkeiten investiert werden.

3.1 Listen von Intervallen

Als erste größere Fallstudie wurde die Spezifikation der Mengen von natürlichen Zahlen, die als Listen von Intervallen repräsentiert werden, untersucht. Diese Fallstudie wurde ausgewählt nicht nur weil sie interessant aber trotzdem eher klein und überschaubar ist, sondern auch weil sie bereits in einer früheren Arbeit über die Fehlersuche in den Spezifikationen als Referenzbeispiel benutzt wurde. Damit wollen wir auch einen Vergleich zu der bisherigen Methode machen, die von der Idee (Widerlegung durch Gegenbeispiele) ähnlich, aber von der Realisierung (heuristische Termgenerierung durch Rewriting Regeln) komplett anders ist.

Die Mengen von natürlichen Zahlen können durch die Listen von Intervallen eindeutig repräsentiert werden. Dabei werden die in der Menge enthaltenen natürlichen Zahlen zuerst aufsteigend geordnet und dann die zusammenhängende Sequenzen als Intervalle repräsentiert. z.B. wird die Menge $\{0, 1, 2, 4, 5, 7\}$ als

die Liste $[(0,2),(4,5),(7,7)]$ eindeutig repräsentiert. Eine praktische Anwendung für diese Datenstrukturen ist die Speicherung der freien Blöcke des dynamisch allozierten Speichers.

Folgende Datentypen werden in der KIV Spezifikation definiert:

- *Interval* (Paar von natürlichen Zahlen)
- *List*[*Interval*] (Parameter *Elem* instanziert mit *Interval*)
- *Nat* (natürlichen Zahlen)

wobei alle diese Datentypen in der KIV Bibliothek spezifiziert sind und einfach übernommen werden können. Da die Bibliothek bereits auf Alloy portiert ist, entsteht hier kein Aufwand für den Benutzer.

Die Eindeutigkeit der Repräsentation ergibt sich nur unter der Voraussetzung, daß die Listen von Intervallen gewisse Eigenschaften haben. Das Prädikat $R : ivlist$ definiert die Menge der wohlgeformten Listen von Intervallen. Für die Liste muss gelten: keine zusammenhängenden Intervalle, wohlgeformte Intervalle, d.h. für (k_1, k_2) sollte $k_1 \leq k_2$. Z.B. $R([(0,2)]) = true$, $R([(2,0)]) = false$, $R([(0,1),(1,2)]) = false$. Diese Eigenschaften werden mit folgenden zwei Axiomen spezifiziert:

$$R([\,]);$$
$$R((m,n)+x) \leftrightarrow m \leq n \wedge n+1 < x.first.fst \wedge R(x);$$

Dabei wurden folgende Operationen in Infixform benutzt

$$\begin{array}{rll} + : & interval \times list \rightarrow list & (\textit{Vorne Einfügen eines Intervalls in die Liste}) \\ .first : & list \rightarrow interval & (\textit{Erstes Intervall in der Liste}) \\ .fst : & interval \rightarrow nat & (\textit{Untere Schranke im Intervall}) \end{array}$$

Nun wird auf der gegebenen Datenstruktur eine zentrale Operation $insert : ivlist \times nat \rightarrow ivlist$ definiert, die eine Zahl in eine Liste von Intervallen *korrekt* einfügt, d.h. so, daß die Wohlgeformtheit der Liste erhalten bleibt. Folgende Formel ψ_{INV} beschreibt die Erhaltung der Invariante durch *insert*:

$$\psi_{INV} \equiv \forall\, ivl_1, ivl_2 \in intervallist,\ n \in nat.\ R(ivl_1)\ \wedge\ ivl_2 = insert(ivl_1, n) \rightarrow R(ivl_2)$$

Die Operation *insert* ist rekursiv spezifiziert. Sie läuft die Liste durch und sucht nach der passenden Stelle, wo die Zahl n eingefügt werden soll. Die Definition ist eine große Fallunterscheidung über das Intervall (k_1, k_2), das im Rekursionsschritt behandelt wird: entweder ist es das falsche Intervall $(k_1 > n \vee k_2 < n)$ und die Liste wird weiter rekursiv abgearbeitet oder die Zahl ist bereits im

Intervall vorhanden ($k_1 < n \wedge k_2 > n$) oder das Intervall muss angepasst werden ($k_1 = n + 1 \vee k_2 = n - 1$) oder ein neues Intervall muss eingefügt werden (ansonsten entstehen zusammenhängende Intervalle). Die ursprüngliche Axiomatisierung von *insert* ist:

(1) $insert([\,], k) = [(k, k)]$

(2) $k < m \wedge k \neq m - 1 \rightarrow insert((m, n) + x, k) = (k, k) + (m, n) + x$

(3) $k < m \wedge k = m - 1 \rightarrow insert((m, n) + x, k) = (k, n) + x$

(4) $k \geq m \wedge k \leq n \rightarrow insert((m, n) + x, k) = (m, n) + x$

(5) $k \geq m \wedge k = n + 1 \rightarrow insert((m, n) + x, k) = (m, k) + x$

(6) $k \geq m \wedge k > n + 1 \rightarrow insert((m, n) + x, k) = (m, n) + insert(x, k)$

Sie ist fehlerhaft, d.h. das Theorem ψ_{INV} ist unbeweisbar. Der Fehler liegt in der nicht sorgfältigen Behandlung eines Randfalls. Es ist möglich, daß nach dem Einfügen einer Zahl zwei zusammenhängende Intervalle entstehen und damit die Wohlgeformtheit verletzt wird. Bei der oberen Axiomatisierung von *insert* gilt z.B.:

$$insert(1, [(0, 0), (2, 2)]) = [(0, 1), (2, 2)]$$

Das richtige Ergebnis wäre die Liste $[(0, 2)]$ (Verschmelzung von Intervallen $(0, 1)$ und $(2, 2)$). Eine detailierte Beschreibung der Fehlersuche mit Alloy ist im Kapitel 5 zu finden. In der richtigen Axiomatisierung wird der Fall $k \geq m \wedge k = n + 1 \wedge x = (m_1, n_1) + y \wedge m_1 = k + 1$ extra behandelt indem die betroffenen Intervalle verschmolzen werden.

Kapitel 4

Fehlersuche durch Konstruktion der Gegenbeispiele

Dieses Kapitel beschreibt den Ansatz zur automatisierten Analyse und Fehlersuche in Algebraischen Spezifikation. Dieser Ansatz liegt im Kern der Arbeit und kommt in den späteren Kapiteln zum Einsatz. Es werden beide Arten von in der Praxis verwendeten Datentypen behandelt: freie Datentypen und nicht frei generierten Datentypen.

4.1 Grundansatz

Bei der Erstellung der Algebraischen Spezifikationen ähnlich wie z.B. beim Programmieren wird der Entwickler ständig mit den dabei entstehenden Fehlern konfrontiert. Die Fehler zu lokalisieren ist insbesondere bei den komplexen Systemen eine sehr anspruchsvolle Aufgabe. Es werden dabei standardgemäß solche Techniken wie Testen und Debuggen verwendet. Insbesondere Debuggen ist für den Menschen extrem Zeit und Kraft aufwendig, da hier das System sorgfältig Schritt für Schritt ausgeführt werden muss.

Im Fall von Algebraischen Spezifikationen ist die Problematik von der Idee her ähnlich, technisch aber ganz anders. Um sicherzustellen, daß das entworfene Design auch die gewünschten Eigenschaften besitzt, werden diese Eigenschaften als Theoreme formuliert, z.B. über die Korrektheit und über die Funktionalität. Anschließend wird versucht diese Theoreme zu beweisen. Die fehlgeschlagenen Beweise führen meistens nach der sorgfältigen Analyse des Beweisbaumes zu Fehlern in der Spezifikation. Die typischen Fehler sind, z.B., Vergessen gewisser Randfälle bei der Definition der Operation, aber auch ein falsches Verständnis des Systems aufgrund von dessen Komplexität und daraus folgende unpräzise Spezifikation von angestrebten Systemeigenschaften.

Daraus ergibt sich die Grundmotivation für den Einsatz eines Modellgenerators wie Alloy. Automatische Generierung der Modelle für eine KIV Spezifikation und

Testen der gewünschten Eigenschaften auf diesen Modellen würde den bisherigen Prozeß zur Fehlerlokalisierung leichter und effizienter machen. Allerdings sollte die entworfene Technik gewisse Grundeigenschaften besitzen, damit sie auch in der Praxis eingesetzt werden kann. Für eine KIV Spezifikation SP_{KIV}, die entsprechende Alloy Spezifikation SP_{Alloy} und ein Theorem (Eigenschaft) φ sollten folgende Eigenschaften gelten:

$$\textit{Korrektheit}: \qquad Wenn \;\; SP_{KIV} \models \varphi, \;\; dann \;\; SP_{Alloy} \models \varphi$$

$$\textit{Vollständigkeit}: \qquad Wenn \;\; SP_{KIV} \not\models \varphi, \;\; dann \;\; SP_{Alloy} \not\models \varphi$$

Die essentielle Anforderung bei der Korrektheit garantiert, daß keine falschen Gegenbeispiele berechnet werden. Falsche Alarme würden dem Entwickler Zeit und Aufwand kosten, und damit schließlich zum Verlust des Vertrauens in die Technik führen. In späteren Abschnitten wird die Korrektheit detailiert besprochen und für den vorgestellten Ansatz bewiesen.

Die zweite Anforderung der Vollständigkeit ist für die praktische Nutzbarkeit zwar wichtig dafür aber weniger kritisch als die Korrektheit. Es wird erwartet, daß Alloy in der Lage ist für die ungültigen Eigenschaften ($SP_{KIV} \not\models \varphi$) ein Gegenbeispiel zu produzieren.

Um diese abstrakte Idee zu realisieren, müssen wir uns mit der Semantik einer Algebraischen Spezifikation beschäftigen. Diese wird durch die Menge der Modelle (Algebren) definiert, die diese Spezifikation erfüllen. Laut der Definition 5 ist ein Theorem φ für eine Spezifikation SP gültig, wenn es für alle ihre Modelle $\mathcal{A}$ ($SP \models \mathcal{A}$) gültig ist ($\mathcal{A} \models \varphi$). Damit kann das Testen eines Theorems mit Alloy durch Spezifizieren der endlichen Teilstücke $\mathcal{M}$ der unendlichen Strukturen $\mathcal{A}$ in Alloy Sprache realisiert werden, siehe Abb. 4.1.

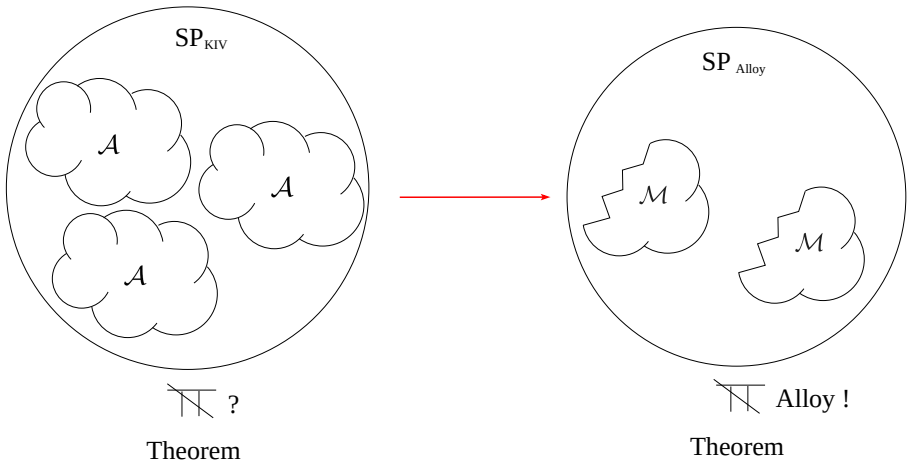


Abbildung 4.1: Grundidee für Einsatz der endlichen Modellsuche zur Fehlerlokalisierung.

Dieses Vorgehen setzt voraus, daß die betrachteten Theoreme der Analyse mit

endlichen Modellen zugänglich sind, d.h. sie müssen folgende Eigenschaft besitzen:

$$\textit{endliche Widerlegbarkeit}: \quad \textit{für } \mathcal{M} \subseteq \mathcal{A} \;\; \mathcal{M} \not\models \varphi \;\Rightarrow\; \mathcal{A} \not\models \varphi$$

Damit lässt sich folgern, daß wenn eine endliche Substruktur $\mathcal{M}$ von $\mathcal{A}$ ein Gegenbeispiel für φ ist, dann ist auch $\mathcal{A}$ gleichzeitig ein Gegenbeispiel für φ. Der Abschnitt 4.5 beschreibt die Klasse der Formeln, die diese Eigenschaften besitzen. Da die Formeln in KIV funktional sind und die Alloy Sprache relational ist, beschreiben wir die Transformation zu der relationalen Form im nächsten Abschnitt. Die Abschnitte 4.3 und 4.4 präsentieren detailiert den Ansatz zur Generierung der endlichen Teilmodelle mit Alloy für die KIV Spezifikationen der abstrakten Datentypen.

4.2 Transformation zu der Relationalen Form

Auf dem Weg zu der relationalen Form wird zuerst jedes Funktionssymbol f auf die entsprechende Relation (Prädikat) F abgebildet:

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

Die grundlegende Idee, die zu realisieren ist, den Graphen der Funktion f auf die Relation F abzubilden:

$$[\![f]\!](a_1, \ldots, a_n) = b \Leftrightarrow\; [\![F]\!](a_1, \ldots, a_n, b) \tag{4.1}$$

wobei $[\![f]\!]$ die Semantik von f in einem Modell der KIV Spezifikation ist und $[\![F]\!]$ die Semantik von F in der entsprechenden Alloy Spezifikation ist. Um dies zu erreichen werden zwei Axiome benötigt, die erzwingen, daß die Relation sich wie eine Funktion verhält. Es sind das Axiom der *Eindeutigkeit*:

$$\forall x_1, \ldots, x_n, y, z.\; F(x_1, \ldots, x_n, y) \wedge F(x_1, \ldots, x_n, z) \to y = z$$

und das Axiom der *Totalität*:

$$\forall x_1, \ldots, x_n.\; \exists y.\; F(x_1, \ldots, x_n, y)$$

Weiter müssen auch die KIV Axiome, die die Operationen spezifizieren, in die relationale Form gebracht werden. Dies kann schematisch erreicht werden. Dabei wird in der Formel für jeden Ausdruck $f(x_1, \ldots, x_n)$ eine Hilfsvariable z eingeführt und anschließend die funktionale Gleichung $f(x_1, \ldots, x_n) = z$ durch den relationalen Ausdruck $F(x_1, \ldots, x_n, z)$ ersetzt. Wir geben eine formale Beschreibung des Schemas. Dabei nehmen wir an, daß die Formeln in KIV normalisiert sind: alle Quantoren sind nach vorne gebracht (Prenex Normal Form):

$$\varphi \equiv Q_1 v_1. \ \ldots \ Q_n v_n. \ \psi$$

wo die quantorenfreie Formel ψ die freien Variablen $v_1, \ldots, v_n$ hat. Die Restriktion zu der Prenex Normal Form vermeidet die Diskussion über eine passende Umbenennung der gebundenen Variablen. Betrachten wir als Beispiel eine Formel aus der Spezifikation der Listen:

$$\varphi \equiv \forall x, y. \ rev(app(x, y)) = app(rev(y), rev(x))$$

Der quantorenfreie Teil der Formel enthält die Funktionssymbole *app* und *rev* (Funktionen *append* und *reverse*). In der übersetzten Formel werden an der Stelle die Relationen $APP : list \times list \times list$ und $REV : list \times list$ benutzt. Um die Übersetzung zu definieren betrachten wir zwei Mengen von Termen: die Menge aller Terme $\mathcal{T}_{all}$ und die Menge der Top-Level Terme $\mathcal{T}_{top}$. Die Menge $\mathcal{T}_{top}$ enthält die Terme, die in der Formel in Gleichungen $t_i = t_j$ oder Prädikaten $P(t_1, \ldots, t_n)$ vorkommen und keine Variablen sind. In unserem Beispiel sind es die Mengen

$$\begin{aligned} \mathcal{T}_{top} &= \{rev(app(x, y)), app(rev(y), rev(x))\} \\ \mathcal{T}_{all} &= \mathcal{T}_{top} \cup \{rev(x), rev(y), app(x, y)\} \end{aligned}$$

Mit $\mathcal{T}_{top}$ und $\mathcal{T}_{all}$ ist die Übersetzung der Formel φ von der funktionalen Form in die relationale Form $\tau(\varphi)$ wie folgt definiert:

Definition 6 *[Relationale Form]*
Gegeben eine Abbildung $\theta : \mathcal{T}_{all} \to Vars$, die Terme in $\mathcal{T}_{all}$ auf neue Variablen abbildet. Sei φ eine PNF normalisierte Formel in der funktionalen Form:

$$\varphi \equiv Q_1 v_1 :: s_1 \ldots Q_n v_n :: s_n. \ \psi$$

Die entsprechende Formel in der relationalen Form wird wie folgt aufgebaut:

$$\begin{aligned} \tau(\varphi) \quad &\equiv Q_1 v_1 :: s_1 \ldots Q_n v_n :: s_n. \forall \vartheta(\mathfrak{T}_{all}). \\ &\bigwedge_{f(t_1,\ldots,t_k) \in \mathfrak{T}_{all}} F(\vartheta(t_1), \ldots, \vartheta(t_k), \vartheta(f(t_1, \ldots, t_k))) \to \psi[\mathfrak{T}_{top} \setminus \vartheta(\mathfrak{T}_{top})] \end{aligned}$$

Mit der Induktion über den Aufbau der Terme kann es bewiesen werden, daß die Transformation τ die Bedeutung der Formeln behält. D.h. für jedes Modell der Formel φ das entsprechende (isomorphe) relationale Modell die Formel $\tau(\varphi)$ erfüllt. Ähnlich, für jedes Modell von $\tau(\varphi)$, daß auch die Axiome der *Totalität* und *Eindeutigkeit* erfüllt, existiert ein Modell der ursprünglichen Signatur, das φ und (4.1) erfüllt. Die Transformation τ hat lineare Komplexität bezüglich der Formelgröße.

4.3 Freie Datentypen

Das Konzept der abstrakten Datentypen ist bekannt und hat sich gut bewährt. In diesem Abschnitt betrachten wir die *freien* Datentypen - der am häufigsten benutzte Subtyp. Die freien Datentypen haben die Eigenschaft, daß syntaktisch verschiedene Terme auch unterschiedliche Werte des Datentyps darstellen. Ein häufig auftretendes Beispiel sind die Listen. Datentyp *List* wird mit den Konstruktorfunktionen generiert:

$$\begin{aligned} nil: &\quad list \\ cons: &\quad elem \times list \rightarrow list \end{aligned}$$

Diesen entsprechen die Selektorfunktionen[1]:

$$\begin{aligned} .first: &\quad list \rightarrow elem \\ .rest: &\quad list \rightarrow list \end{aligned}$$

Die Generiertheitsklauseln für die freie Datentypen im KIV werden in einer kompakten Form geschrieben, die neben den Konstruktorfunktionen implizit auch die Selektorfunktionen definiert. Es werden implizit auch automatisch die Zusatzaxiome generiert, die die Freiheit des Datentyps spezifizieren. Betrachten wir den Datentyp *list* als Beispiel. Die entsprechende Generiertheitsklausel in KIV ist:

`list = [ ] | . + . (. .first : elem; . .rest : list)`

Diese Definition deklariert implizit, daß der Datentyp *list* durch die Konstruktorfunktionen $[\,] : list$ (Konstante *nil*, leere Liste) und $. + . : elem \times list \rightarrow list$ (*cons*) generiert wird, d.h. "*list* **generated by** [ ], +". Es werden auch die Selektorfunktionen $.first : list \rightarrow elem$ und $.rest : list \rightarrow list$ definiert und mit den automatisch generierten Axiomen spezifiziert. Diese Axiome erzwingen, daß die Konstruktoren disjunkt, injektiv und total sind:

$$\begin{aligned} \texttt{selectors}: &\quad (a+x).first = a, \quad (a+x).rest = x \\ \texttt{disjoint}: &\quad [\,] \neq a + x \\ \texttt{injectivity}: &\quad a + x = a_0 + x_0 \leftrightarrow a = a_0 \wedge x = x_0 \\ \texttt{case}: &\quad x = [\,] \ \vee \ x = x.first + x.rest \end{aligned}$$

4.3.1 Subtermabgeschlossene Modelle

Die Semantik der freien Datentypen ist auf den *Termalgebren* definiert. Die Trägermengen sind frei generiert, d.h. sie bestehen aus Termen, die induktiv mit

[1] in Postfixform

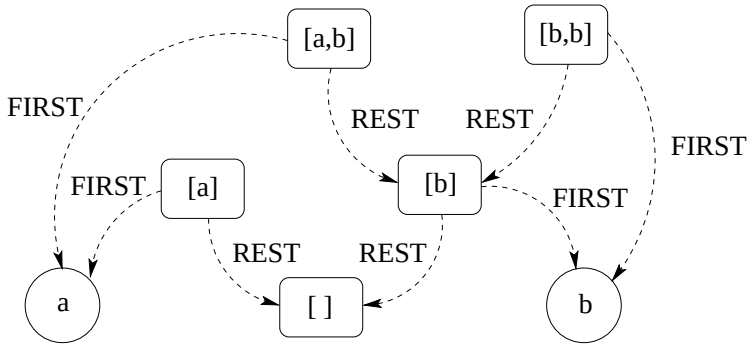


Abbildung 4.2: Endliches subtermabgeschlossenes Teilmodell mit 2 *elem* und 5 *list* Atomen.

den Konstruktoren generiert sind. Der Grundansatz zur Generierung der endlichen Teilmodelle basiert auf der Arbeit von Kuncak und Jackson [26] (Entwickler von Alloy am MIT). Man betrachtet die endlichen Teilstücke der unendlichen Termalgebren, die unter der *Subtermrelation* abgeschlossen sind. Z.B. für die Listen ist die Subtermrelation als die transitive Hülle der *list*-wertigen Selektoren definiert. Die Abb. 4.2 zeigt ein endliches subtermabgeschlossenes Teilmodell der Listen.

Die Relationen *FIRST* und *REST* simulieren die entsprechenden Selektorfunktionen. Mit ihren Hilfe kann die Bedeutung (auf dem Bild als Beschriftung der Atome zu sehen) einzelner Atome entschlüsselt werden. Solche Teilstrukturen der unendlichen Algebren lassen sich mit *SUA* Axiomen generieren, die das "Verhalten" der Selektorrelation spezifizieren. Z.B. für die Listen spezifizieren diese Axiome die Selektorrelationen *FIRST* und *REST*, siehe Abb. 4.3. Dabei wird Alloy Syntax für den transitiven Abschluss verwendet: Operator $^\wedge : \mathit{Relation} \rightarrow \mathit{Relation}$. Ferner, da wir die relationale Sprache benutzen, wird an der Stelle der Konstruktorkonstante *nil* die entartete Relation (als eine Singletonmenge repräsentiert) *NIL* verwendet.

$$
\begin{array}{ll}
\textit{Selectors:} & \forall l : \mathit{list}.\ \neg NIL(l) \rightarrow \\
 & \qquad \exists!\ l' : \mathit{list},\ e : \mathit{elem}.\ REST(l, l')\ \wedge\ FIRST(l, e) \\
 & \forall l : \mathit{list},\ e : \mathit{elem}.\ \neg REST(nil, e)\ \wedge \neg FIRST(nil, l) \\
\textit{Uniqueness:} & \forall l, l_0, l', l'_0 : \mathit{list},\ e, e' : \mathit{elem}.\ FIRST(l, e) \wedge FIRST(l', e') \wedge \\
 & \qquad REST(l, l_0) \wedge REST(l', l'_0) \wedge e = e' \wedge l_0 = l'_0 \rightarrow l = l' \\
\textit{Acyclicity:} & \forall l : \mathit{list}.\ (l, l) \notin\ ^\wedge REST
\end{array}
$$

Abbildung 4.3: SUA Axiome

Mit *SUA* Axiomen lassen sich für die KIV Spezifikationen der freien Datentypen die subtermabgeschlossenen Teilmodelle in Alloy spezifizieren. Auf dieser Basis definieren wir die Transformation der KIV Spezifikationen der freien Datentypen nach Alloy.

Definition 7 *[Alloy Spezifikation - Freie Datentypen]*
Für die algebraische Spezifikation SP_{free} der freien Datentypen wird die entsprechende Alloy Spezifikation $\tau_{\texttt{fin}}(SP_{free})$ wie folgt definiert:

$$\tau_{\texttt{fin}}(SP_{free}) = ((S, \emptyset, \tau(\mathcal{F}) \cup \mathcal{P}), \tau(Ax) \cup \texttt{Func}(\tau(\mathcal{F})) \cup \texttt{SUA}(Gen))$$

In $\tau_{\texttt{fin}}(SP_{free})$ werden alle Funktionssymbole $f \in \mathcal{F}$ sind durch die Prädikatsymbole $\tau(f)$ ersetzt. Die Axiome Ax werden in die relationale Form übersetzt: $\tau(Ax)$. Die zusätzlichen Axiome $\texttt{Func}(\tau(\mathcal{F}))$ erzwingen, daß jede Relation $\tau(f) \equiv F$ die Eigenschaften einer partiellen Funktion besitzt:

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

Die Abbildung 4.4 zeigt die resultierende Alloy Spezifikation für die Generierung der endlichen Teilmodelle der Listen (zuerst ohne weitere Funktionen außer Selektoren). Das Modul *list* wird mit der Sorte *Element* parametrisiert, d.h. später kann die Spezifikation mit einer beliebigen anderen Sorte aktualisiert werden. Die Deklaration des Universums besteht aus zwei Sorten: *List* und *Element*. Die Listenatome sind in zwei Untersorten unterteilt: Singleton *Nil* und *Cons* (Schlüsselwort **extends**).

```
module list[Element]

// signature
sig List {}

one sig Nil extends List {}

sig Cons extends List {
   first:  Element,
   rest:  List
}


// SUA
fact Uniqueness {
   all l,l': Cons |
      l.first = l'.first and l.rest = l'.rest => l = l'
}

fact Acyclic {
   no l: Cons | l in l.^rest
}

pred testRun() {}
```

Abbildung 4.4: Alloy Spezifikation für die Generierung der subtermabgeschlossenen Teilmodelle der Listen

Auf dem dazugehörigen Metamodell ist die Sortenhierarchie sowie die deklarierten Relation auf den Sorten zu sehen: *first, rest*. Die beiden Axiome *Uniqueness* und *Acyclic* zusammen mit der Deklaration des Universums reflektieren das

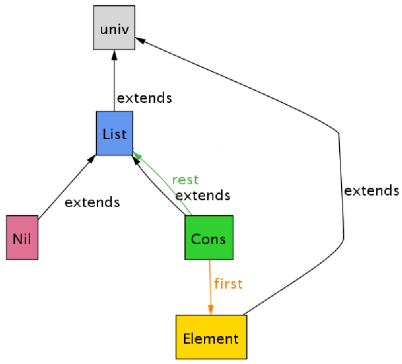


Abbildung 4.5: Das Metamodell für die Alloy Spezifikation der Listen.

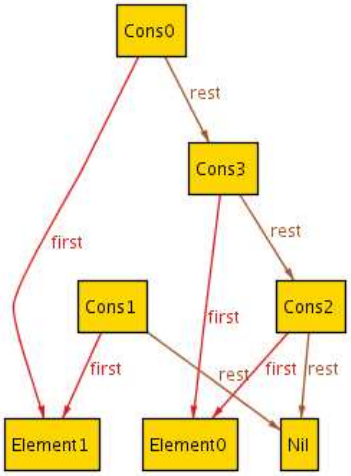


Abbildung 4.6: Das mit Alloy generiertes subtermabgeschlossenes Teilmodell mit 5 Listen und zwei Elementen.

Prinzip der SUA-Generierung. Das *Uniqueness* Axiom fordert, daß nicht zwei Atome strukturell gleichen Listen repräsentieren. Das Axiom *Acyclic* verhindert die Modelle mit Listen, die zyklisch bezüglich der Relation *rest* sind.

Das Teilmodell der Listen auf der Abbildung 4.2 kann nun mit dem folgenden Alloy Befehl generiert werden:

run `testRun` **for exactly** `5 List`, **exactly** `2 Element`

Die Abbildung 4.6 zeigt das für die Alloy Spezifikation aus der Abbildung 4.4 generierte Teilmodell, das gegen die Subtermrelation abgeschlossen ist. Um eine volle Übereinstimmung mit der Semantik der Listen in KIV zu erzielen muss noch ein wichtiger Aspekt bezüglich der unterspezifizierten Operationen berücksichtigt werden. In der relationalen Semantik von Alloy würde der relationale Ausdruck *Nil.rest* laut SUA-Axiomatisierung als leere Menge interpretiert. Auf

der anderen Seite in KIV ist die Semantik vom Term `[].rest` unterspezifiziert, d.h. es gibt unendlich viele Algebren $\mathcal{A} \in Mod(SP)$, in denen `[].rest` als irgendeine zufällige Liste interpretiert wird. Diese semantische Abweichung wird korrigiert indem die Selektorfunktionen in der Alloy Signatur der Sorte *List* definiert werden und nicht in der Sorte *Cons*:

```
module list[Element]

// signature
sig List {
   // cons selectors
   first:  Element,
   rest:  List
}

one sig Nil extends List {}
sig Cons extends List {}

// SUA
fact Uniqueness {
   all l,l': Cons |
      l.first = l'.first and l.rest = l'.rest => l = l'
}

fact Acyclic {
   no l: Cons | l in l.^rest
}

fact Domain {
   List = Nil + Cons
}


pred testRun() {}
```

Damit wäre $Nil.rest$ unspezifiziert genauso wie in KIV. Es hat sich in der Praxis gezeigt, daß viele interessante Fehler gerade aus der unvollständigen Axiomatisierung der Operationen hervorgehen, wo die Randfälle, wie z.B. die leere Liste, nicht richtig behandelt werden.

4.3.2 Axiomatisierung der Rekursiven Funktionen

Um den Ansatz von Kuncak und Jackson in KIV einsetzen zu können müssen wir die Haupthürde überwinden: die Einschränkung der Sprache nur auf die Selektoren. Das Vorgehen und die Komplikationen, die dabei entstehen, lassen sich am besten an einem Beispiel demonstrieren.

Betrachten wir die Listen als ein repräsentatives Beispiel für einen freien Datentyp und zwei rekursive Funktionen, die auf Listen arbeiten: *append* (Konkatenation von Listen) und *reverse* (Umdrehen einer Liste). Die Abbildung 4.7 zeigt die KIV Spezifikation für das Beispiel.

In der Definition 8 wird die allgemeine Form einer rekursiven Definition in KIV aufgefasst. Hier sind c_i die Konstruktorfunktionen für die Sorte s_1. Jedes Ψ_i ist ein Term, in dem die Aufrufe von f als erstes Argument u haben. Die Fallunter-

```
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
```

Abbildung 4.7: Algebraische Spezifikation der Listen in KIV.

scheidungen mittels der Formeln ψ_i bilden eine vollständige Fallunterscheidung für einen der Konstruktoren c, d.h. ihre Disjunktion ist äquivalent zu *true*.

Definition 8 *[KIV Axiome für die Rekursion]*
Eine rekursive Definition der Funktion $f : s_1 \times \ldots s_k \rightarrow s$ benutzt die strukturelle Induktion im ersten Argument. Die Spezifikation erfolgt mit Axiomen:

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

Im ersten Schritt auf dem Weg zur Axiomatisierung übersetzen wir die Signatur. Die beiden Operationen *app* und *rev* werden als Relationen in die Signatur der Sorte *list* aufgenommen:

```
sig list {
   APP: list -> lone list,
   REV: lone list
}
```

Mit dem Stichwort **lone** teilen wir Alloy mit, daß die Relation das Axiom der *Eindeutigkeit* erfüllt. Bei einer bestimmten Eingabe gibt es also maximal nur ein Ergebnis für die Operationen *APP* und *REV*.

Das Axiom der *Totalität* wäre auch spezifiziert, hätte man den Stichwort **one** benutzt. Anderseits, würde das zu unendlichen Modellen führen, was die Alloy Analyse unmöglich machen würde. Wir müssen dieses Axiom weglassen. Leider kann das eventuell zu unerwünschten Effekten bei der Modellkonstruktion führen, wie es später anhand eines Beispiels demonstriert wird.

Im zweiten Schritt werden die KIV Axiome für *app* und *rev* mit der Transformationsfunktion τ in die relationale Form gebracht und der Alloy Spezifikation hinzugefügt.

Dafür kombinieren wir jeweils die beiden funktionalen Axiome aus der KIV Spezifikation zu einer einzigen Formel:

$$\begin{aligned} app(x_1, x_2) = y \leftrightarrow \quad & x_1 = [] \wedge y = x_2 \vee \\ & \exists\, a_0, z_0.\ x_1 = cons(a_0, z_0) \wedge y = cons(a_0, app(z_0, x_2)) \\ rev(x) = y \leftrightarrow \quad & x = [] \wedge y = [] \vee \\ & \exists\, a_0, z_0.\ x = cons(a_0, z_0) \wedge y = app(rev(z_0), cons(a_0, [])) \end{aligned}$$

Obwohl die Übersetzung schematisch ist, benutzt es die Axiome der *Eindeutigkeit* und der *Totalität*. Für die Funktion *reverse* erhalten wir die folgende relationale Definition, die in die Alloy Spezifikation aufgenommen wird:

$$\begin{aligned} \forall x, y :: list.\ (x, y) \in REV \leftrightarrow \\ (x \in NIL \ \wedge \ y = NIL \vee \\ \exists a_0 :: elem,\ z_0, z_1, z_2, z_3 :: list.\ (a_0, z_0, x) \in CONS \\ \wedge\, (z_0, z_1) \in REV \ \wedge \ z_2 \in NIL \ \wedge \ (a, z_2, z_3) \in CONS \\ \wedge\, (z_1, z_3, y) \in APP) \end{aligned}$$

Die übersetzte Formel ist stärker als die ursprüngliche Formel, die nur die Implikationsrichtung von rechts nach links haben würde. Die andere Richtung könnte dann aus den beiden Axiome für die *Totalität* und die *Eindeutigkeit* gefolgert werden. Auf der anderen Seite, haben wir das Axiom für die *Totalität* fallen gelassen. Da wir jetzt die Äquivalenz haben, werden für diese Definition die beiden Axiome nicht mehr benötigt. Dies gilt, weil es sich um eine Instanz des Theorems über die wohlfundierte Rekursion handelt.

Definition 9 *[Wohlfundierte Ordnung]*
Eine binäre Relation auf der Menge X ist wohlfundiert genau dann wenn jede nichtleere Teilmenge von X ein $\prec$-minimales Element enthält:

$$\forall S \subseteq X.\ S \neq \emptyset \Rightarrow \exists m \in S.\ \forall y \in S.\ y \not\prec m$$

Theorem 1 *[Wohlfundierte Rekursion]*
Gegeben eine Spezifikation SP und ihre Erweiterung um eine Funktion g mit der Definition Ψ:

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

Ferner, seien alle Argumente der rekursiven Aufrufe von g in Ψ kleiner als $\underline{v}$ bezüglich einer wohlfundierten Ordnung $\prec$.

Dann gilt: für jedes Modell $\mathcal{A}$ der ursprünglichen Spezifikation gibt es genau eine Erweiterung $\mathcal{A} + g_{\mathcal{A}}$, die Modell von $g(\underline{v}) = \Psi(g, \underline{v})$ ist.

Ein formaler Beweis des Theorems, der Ψ als eine Higher Order Funktion betrachtet, wird von Harrison [19] präsentiert. In unserem Fall, ist g die Relation (= boolesche Funktion) $F_{\mathcal{A}}$. Das Theorem garantiert, daß die übersetzte relationale Definition eine eindeutige Relation $F_{\mathcal{A}}$ fixiert. Damit ist die Übersetzung in die relationale Form semantikerhaltend: die Relation $F_{\mathcal{A}}$ bildet genau den Graphen der Funktion f ab.

Das Theorem ist nicht nur auf die Termmodelle anwendbar, sondern auch auf die endlichen Strukturen, die von Alloy generiert werden. Die Subtermrelation ist trivialerweise wohlfundiert, damit ist sie auch auf endlichen Teilmodellen wohlfundiert.

Insgesamt haben wir:

- die rekursiven Definitionen in KIV erweitern das Modell um eine eindeutige Funktion,
- die relationale Transformation dieser Definitionen erweitert das relationale Modell auch um eine eindeutige Funktion $F_{\mathcal{A}}$, die den Graphen der Funktion f repräsentiert,
- das gleiche Ergebnis gilt auch für eine endliche subtermabgeschlossene Teilstruktur $\mathcal{M}$ und die Relation $F_{\mathcal{M}}$.

Jetzt ist die kritische Frage ob für die Funktion $F_{\mathcal{M}}$ in $\mathcal{M}$ die gleichen Theoreme gelten, wie für $F_{\mathcal{A}}$ in $\mathcal{A}$. Die positive Antwort wird für die Klasse von Formeln mit universellen und beschränkten existentiellen Quantoren gegeben. Eine formale Definition der Klasse folgt in einem späteren Abschnitt, wo wir uns mit dieser Fragestellung befassen.

Eine wichtige Vorbedingung für diese positive Antwort ist, daß die Funktion $F_{\mathcal{A}}$, wenn eingeschränkt auf die Domäne von $\mathcal{M}$ (geschrieben $F_{\mathcal{A}}|\mathcal{M}$) äquivalent zu $F_{\mathcal{M}}$ ist. In den betrachteten Beispielen haben wir festgestellt, daß $F_{\mathcal{A}}|\mathcal{M} \subseteq F_{\mathcal{M}}$ gilt. In den meisten Beispielen haben wir sogar $F_{\mathcal{A}}|\mathcal{M} = F_{\mathcal{M}}$. Dies gilt ,z.B., für die APP Funktion. Es bleibt zuerst offen ob die Inklusion auch im allgemeinen gilt. Wir haben aber auch Beispiele, wo $F_{\mathcal{A}}|\mathcal{M}$ eine echte Teilmenge von $F_{\mathcal{M}}$ ist. Die Funktion REV ist ein Beispiel dafür.

Beispiel ($F_{\mathcal{A}}|\mathcal{M} \subsetneq F_{\mathcal{M}}$).
Betrachte ein subtermabgeschlossenes Teilmodell $\mathcal{M}$ der Listen auf der Abbildung 4.8 mit der Domäne:

$$M_{list} = \{[], [a], [c], [b, c], [b, a], [a, b, c], [c, b, a]\}$$
$$M_{elem} = \{a, b, c\}$$

In dieser Struktur sind die Atome $[a, b, c]$ und $[c, b, a]$ nicht mit der Relation REV verbunden. Im unendlichen Modell $\mathcal{A}$ ist dies aber der Fall. Der Grund für diese Diskrepanz ist, daß das endliche Modell das Atom $[c, b]$ nicht enthält. Dies spiegelt sich in der Definition von REV: die rechte Seite der Äquivalenzformel in der Definition hat eine andere Semantik im endlichen Teilmodell als im unendlichen Modell, wenn der Fall von $[a, b, c]$ und $[c, b, a]$ kommt.

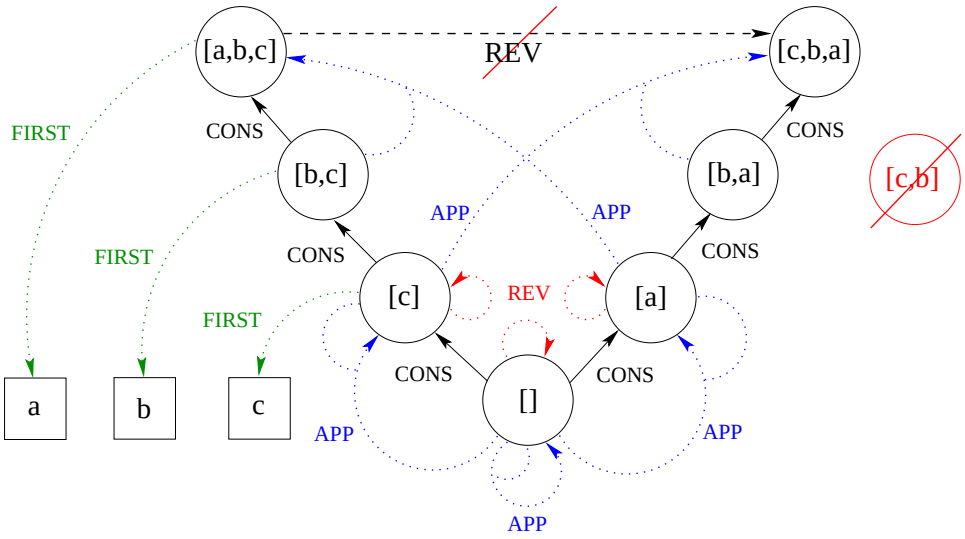


Abbildung 4.8: Endliches subtermabgeschlossenes Teilmodell $\mathcal{M}$ von $\mathcal{A}$, daß den Fall $F_{\mathcal{A}}|\mathcal{M} \neq F_{\mathcal{M}}$ für $F_{\mathcal{M}} \equiv REV_{\mathcal{M}}$ demonstriert.

Es gibt ein grundsätzliches Problem. Die Subtermabgeschlossenheit garantiert nicht, daß das Teilmodell gegen alle Ergebnisketten der rekursiven Aufrufe abgeschlossen ist. In diesem Beispiel ist die Ergebniskette der REV Aufrufe für $[a, b, c]$:

$$rev([a, b, c]) \rightarrow rev([b, c]) \rightarrow rev([c]) \rightarrow rev([])$$

da $rev([a, b, c])$ $rev([b, c])$ aufruft usw. Das Ergebnis $rev([b, c]) = [c, b]$ ist nicht im endlichen Modell. Damit ist die Kette für das Ergebnis $rev([a, b, c])$ unterbrochen. Eine mögliche Zusatzbedingung, die die obengenannte Abgeschlossenheit garantieren würde, wäre:

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

Damit hätte man die Prefixabgeschlossenheit des Teilmodells (Prefix auf Listen) und es würde gelten $REV_{\mathcal{A}}|\mathcal{M} = REV_{\mathcal{M}}$. Im Allgemeinen, könnte man für die surjektiven Funktionen diesen Constraint systematisch erstellen, da jedes Element im Modell ein Ergebnis der Funktion ist. Die Zusatzbedingung verlangt, daß jedes y (ein Ergebnis von f) aus den Ergebnissen $z_1, \ldots, z_n$ der rekursiven Aufrufe berechenbar sein muss.

Sei eine rekursiv definierte Funktion F mit der Definition:

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

Dann ist die Zusatzbedingung für die Konstruktion des Ergebnisses aus den rekursiven Aufrufen:

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

Für die Definition der *append* Funktion wäre solche Zusatzbedingung die Subtermabgeschlossenheit, die bereits von den *SUA* Modellen erfüllt ist. Für die Funktionen, die nicht surjektiv sind, wäre die Zusatzbedingung nur über die y zu quantifizieren, die im Abbild von f liegen. Dies ist aber nicht möglich, da der einzige Weg den Bild von f zu charakterisieren wieder nur über die Rekursion geht. Ein Beispiel, das das Problem illustriert, ist die Definition der Funktion *palindrome*.

Beispiel $(F_{\mathcal{A}}|\mathcal{M} \neq F_{\mathcal{M}})$.
Betrachte die nicht surjektive Funktion *palindrom* mit den Axiomen

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

Offensichtlich sind nicht alle Atome Ergebnisse der Funktion *pal*. Da wir diese Atome auf eine nicht rekursive Weise nicht beschreiben können, würde die Zusatzbedingung nicht funktionieren, d.h. Alloy würde kein Modell konstruieren können.

4.3.3 Kompatibilität der Spezifikationen

Wir brauchen Kriterien für die Operationsdefinitionen, mit denen die Konstruktion von falschen Gegenbeispielen (falsche Alarme) sich vermeiden lässt. Eine Definition muss überprüft werden, ob sie sich semantikerhaltend auf die endlichen abgeschlossenen Teilstrukturen übertragen lässt. Diese Aufgabe ist implizit an die Frage gebunden, welche Klasse von Teilstrukturen für den Ansatz der endlichen Modellsuche am besten geeignet ist.

Gegeben eine algebraische Spezifikation $SP = (\Sigma, Ax, Gen)$ erster Stufe in KIV und eine Erweiterung dieser Spezifikation um eine Operation f mit den Axiomen $\varphi_!, \ldots, \varphi_n$:

$$SP' = ((S, \mathcal{F} \cup \{f\}), Ax \cup \{\varphi_1, \ldots, \varphi_n\}, Gen)$$

Die entsprechende Alloy Spezifikation $\tau_{\mathtt{fin}}(SP')$ enthält eine Relation F für die Funktion f. Die Axiome $\varphi_1, \ldots, \varphi_n$ werden normalisiert und mittels der Transformationsfunktion τ in die relationale Form gebracht:

$$\forall \underline{x}, y.\ F(\underline{x}, y) \ \leftrightarrow\ \Psi(\underline{x}, y)$$

wobei $\Psi(\underline{x}, y) \equiv Qv_1.\ \ldots Qv_k.\ \chi(\underline{v}, \underline{x}, y)$, χ quantorenfrei ist und $Free(\chi) = \{\underline{v}, \underline{x}, y\}$.

Wir unterscheiden zwischen den rekursiven und nicht rekursiven Operationen. Im Fall von nicht rekursiven Definitionen enthält die Formel χ nur Relationssymbole, die sich von F unterscheiden. Anderenfalls, ist die Definition rekursiv.

Zusätzlich verlangen wir, daß χ nur die Relationssymbole benutzt, die bereits vorher definiert wurden, d.h. die Fälle von gegenseitig abhängigen Definitionen werden nicht behandelt.

Folgendes Beispiel zeigt eine rekursive und eine nicht rekursive Definition in KIV (in der funktionalen Form):

$$\begin{aligned} \texttt{reverse}: &\quad rev([]) = [],\ rev(a + x) = rev(x) + (a + []) \\ \texttt{disjoint}: &\quad disj(x, y) \leftrightarrow (\forall a.\ \neg\ (a \in x\ \wedge\ a \in y)) \\ \texttt{in}: &\quad a \in x \leftrightarrow \exists y, z, a.\ x = y + a + z \end{aligned}$$

Das Funktionssymbol + ist zweimal überladen. Einmal bezeichnet es die Funktion $cons : elem \times list \rightarrow list$ und einmal - $append : list \times list \rightarrow list$. Die rekursiven Definitionen in KIV benutzen die strukturelle Induktion über einen der Argumente in der definierten Operation. Die strukturelle Induktion garantiert trivialerweise die Wohlfundiertheit der Rekursion, die für die Korrektheit des Ansatzes von Bedeutung ist.

Die folgenden Fragen sind gegenseitig abhängig:

- Welche Klasse der Definitionen ist mit dem Ansatz kompatibel?
- Welche Klasse von endlichen abgeschlossenen Teilstrukturen ist am besten geeignet?

Es wird die Eigenschaft der Semantikerhaltung beim Übergang von unendlichen Termalgebren $\mathcal{A}$ zu deren endlichen Teilstrukturen $\mathcal{M}$ angestrebt. Im Folgenden, bezeichnen wir die Definitionen als *kompatibel*, falls sie diese Eigenschaft nicht verletzen. Daraufaufbauend, werden die Begriffe *kompatible Erweiterung*, *kompatible Spezifikation* eingeführt.

Um die obenerwähnte gegenseitige Abhängigkeit aufzulösen, abstrahieren wir von der geeigneten Klasse der endlichen Teilstrukturen. Sei $\preceq$: $dom(\mathcal{A}) \times dom(\mathcal{A})$ eine Präordnung (die Antisymmetrie muss nicht gelten) auf der Domäne der unendlichen Termmodelle $dom(\mathcal{A}) := \bigcup_{s \in S} \mathcal{A}_s$. Diese Präordnung induziert die Klasse der endlichen Teilmodelle $\mathcal{M}$ von unendlichen Termmodellen $\mathcal{A}$, die gegen $\preceq$ abgeschlossen sind. Eine semantische Charakterisierung von $\preceq$ gibt die Definition 10.

Definition 10 *[Abgeschlossene endliche Teilstrukturen]*
$\mathcal{M}$ ist $\preceq$-abgeschlossene Teilstruktur von $\mathcal{A}$ genau dann wenn

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

Wir lassen es an der Stelle offen, wie die $\preceq$-Abgeschlossenheit sich in Alloy spezifizieren lässt. Offensichtlich gibt es eine Reihe von verschiedenen Alternativen, die je nach Anwendungsfall eingesetzt werden. In der Praxis hat sich gezeigt, daß meistens die *Subtermabgeschlossenheit* (d.h. $\preceq\ \equiv$ Subtermrelation) die beste Wahl ist.

Sei $\mathcal{C}_{\preceq}$ die Randbedingung in der Alloy Spezifikation, die die $\preceq$-Abgeschlossenheit der endlichen Modelle bewirkt. Es bleibt nun die Operationsdefinitionen zu identifizieren, die auf $\preceq$-abgeschlossenen endlichen Teilstrukturen die ursprüngliche Semantik der Operationen auf den unredlichen Strukturen erhalten:

$$[\![F(\underline{x})]\!]^{\mathcal{A},\alpha} = [\![F(\underline{x})]\!]^{\mathcal{M},\alpha}$$

Die obere Gleichheit muss für alle definierten Operationen F sowie beliebigen Belegungen $\alpha : X \to dom(\mathcal{M})$ der freien Variablen $\underline{x}$ gelten.

Definition 11 *[Kompatibilität]*
Eine Definition $\Phi \equiv \forall \underline{x}, y.\ F(\underline{x}, y) \leftrightarrow \mathcal{Q}v_1.\ \ldots \mathcal{Q}v_k.\ \chi(\underline{v}, \underline{x}, y)$ *ist kompatibel (bezüglich einer festen Präordnung* $\preceq$*) genau dann wenn*

$$\mathcal{A} \models \Upsilon(\mathcal{Q}v_1.\ \ldots \mathcal{Q}v_k.\ \chi(\underline{v}, \underline{x}, y), \underline{x} \cup \{y\})$$

Das Prädikat $\Upsilon : For(\Sigma) \times \mathcal{P}(X)$ *testet die Kompatibilität einer Definition und ist mit der strukturellen Induktion über die betrachtete Formel definiert:*

$$\begin{aligned}
\Upsilon(\chi, \underline{v}) &\quad \Leftrightarrow true, \quad \text{für quantorenfreie } \chi \\
\Upsilon(\exists z.\ \varphi, \underline{v}) &\quad \Leftrightarrow (\exists z.\ \varphi) \to (\exists z.\ \varphi \wedge (\bigvee_{u \in \underline{v}} z \preceq u) \wedge \Upsilon(\varphi, \underline{v} \cup \{z\})) \\
\Upsilon(\forall z.\ \varphi, \underline{v}) &\quad \Leftrightarrow (\exists z.\ \neg\varphi) \to (\exists z.\ \neg\varphi \wedge (\bigvee_{u \in \underline{v}} z \preceq u) \wedge \Upsilon(\varphi, \underline{v} \cup \{z\}))
\end{aligned}$$

Spezifikation SP ist kompatibel genau dann wenn alle enthaltenen Definitionen kompatibel sind.

Sollte die Formel Ψ in der oberen Definition rekursive Aufrufe von F enthalten, verlangen wir zusätzlich, daß die rekursive Definition wohlfundiert ist sowie die entsprechende wohlfundierte Ordnung $\prec$ mit der Präordnung $\preceq$ sich verträgt: $\prec \subseteq \preceq$.

Intuitiv betrachtet, erzwingt die Kompatibilität einer Definition, daß die semantische Auswertung der Quantoren auf der rechten Seite der Äquivalenz $F(\underline{x}, y) \leftrightarrow \Psi(\underline{x}, y)$ nicht durch die Endlichkeit der Domäne von $\mathcal{M}$ beeinträchtigt wird. Z.B., wenn der existentielle Quantor $\exists v.$ im unendlichen Modell als *true* ausgewertet wird, dann garantiert die Kompatibilität, daß der entsprechende Zeuge im endlichen Modell immer vorhanden ist. Ähnlich, wenn der Allquantor $\forall v.$ als *false* ausgewertet wird, sollte das Atom, das eine Gegenbeispielbelegung von v darstellt, auch unbedingt im endlichen Modell enthalten sein.

4.3.4 Korrektheit

Im vorherigen Abschnitt haben wir die Kompatibilität der algebraischen Spezifikationen der ersten Stufe (für frei generierten Datentypen) eingeführt. Mit der Einführung der abstrakten Präordnung $\preceq$: $dom(\mathcal{A}) \times dom(\mathcal{A})$ auf Domänen

wurde das Konzept der Abgeschlossenheit der endlichen Strukturen abstrakt aufgefasst und damit auch insgesamt von der technischen Realisierung abstrahiert. In Kapiteln 5, 6 und 7 wird die praktische Realisierbarkeit des Konzepts untersucht, sowie die technische Seite des Ansatzes ausführlich beschrieben. Insbesondere werden folgenden Fragen behandelt:

- Wie wird $\preceq$ spezifiziert?
- Wie werden kompatible Definitionen effizient identifiziert?

Auf der Grundlage der Präordnung $\preceq$, endlichen abgeschlossenen Teilmodellen und kompatiblen Definitionen kann die Korrektheit des Ansatzes untersucht werden. Unter der Korrektheit wird intuitiv verstanden: Alloy berechnet keine "falschen" Modelle. D.h., jedes von Alloy generiertes endliches Teilmodell $\mathcal{M}$ ist isomorph zu einer Teilstruktur des unendlichen Termmodells $\mathcal{A}$ der KIV Spezifikation. Die folgenden Theoreme fassen es zusammen.

Theorem 2 *[Korrektheit: Freie Datentypen]*
Seien $SP = (\Sigma, Ax, Gen)$ eine algebraische Spezifikation der ersten Stufe mit frei generierten Datentypen, deren Signatur nur Selektorfunktionen enthält. Sei $\tau_{\texttt{fin}}(SP)$ die entsprechende Alloy Spezifikation zur Berechnung von endlichen abgeschlossenen Teilmodellen.

Dann gilt:

$$\mathcal{M} \models \tau_{\texttt{fin}}(SP) \quad \Leftrightarrow \quad \begin{array}{l} \text{es gibt } \mathcal{A}, \mathcal{A}^0 \ : \ \mathcal{A} \models SP,\ \mathcal{A}^0 \subseteq \mathcal{A},\ \mathcal{M} \simeq \mathcal{A}^0, \\ \mathcal{M} \text{ ist abgeschlossen} \end{array}$$

Proof.
Das Theorem wurde von Kuncak et al. [26] bewiesen. □

Theorem 3 *[Korrektheit: Erweiterung I]*
Sei SP eine algebraische Spezifikation der freien Datentypen, für die gilt

$$\mathcal{M} \models \tau_{\texttt{fin}}(SP) \quad \Leftrightarrow \quad \begin{array}{l} \text{es gibt } \mathcal{A}, \mathcal{A}^0 \ : \ \mathcal{A} \models SP,\ \mathcal{A}^0 \subseteq \mathcal{A},\ \mathcal{M} \simeq \mathcal{A}^0, \\ \mathcal{M} \text{ ist abgeschlossen} \end{array}$$

Sei SP' eine Erweiterung von SP um die Operation f mit einer kompatiblen Definition Φ: $SP' = SP + (f, \Phi)$. Φ ist bereits mit τ in die relationale Form transformiert:

$$\Phi \equiv \forall \underline{x}, y.\ F(\underline{x}, y) \ \leftrightarrow \ \Psi(\underline{x}, y)$$

Dann gilt:

$$\mathcal{M} \models \tau_{\mathtt{fin}}(SP') \quad \Leftrightarrow \quad \begin{array}{l} \textit{es gibt } \mathcal{A}, \mathcal{A}^0 \ : \ \mathcal{A} \models SP', \ \mathcal{A}^0 \subseteq \mathcal{A}, \ \mathcal{M} \simeq \mathcal{A}^0, \\ \mathcal{M} \textit{ ist abgeschlossen} \end{array}$$

Proof.
Wir beweisen beide Richtungen der Äquivalenz.

($\Leftarrow$) :
Angenommen, die endliche Struktur $\mathcal{M}$ ist isomorph zu einer endlichen $\preceq$-abgeschlossenen Teilstruktur $\mathcal{A}^0$ von $\mathcal{A}$. Dann erfüllt $\mathcal{M}$ die gleichen Formeln wie $\mathcal{A}^0$. Dann reicht es aus zu zeigen, daß $\mathcal{A}^0 \models \tau_{\mathtt{fin}}(SP')$ gilt.

Weil gilt

- $\mathcal{A} \models SP'$
- `SUA` ist eine Teilmenge von `SUGA`
- die Definitionen $Ax \cup \{\Phi\}$ sind kompatibel bezüglich der Präordnung $\preceq$
- $\mathcal{A}^0$ ist $\preceq$-abgeschlossen

bekommen wir:

- $\mathcal{A}^0 \models \mathtt{SUA}(Gen)$
- $\mathcal{A}^0 \models \tau(Ax) \cup \{\Phi\}$
- $\mathcal{A}^0 \models \mathtt{Func}(\tau(f))$
- $\mathcal{A}^0 \models \tau_{\mathtt{fin}}(SP')$

($\Rightarrow$) :
Wir erweitern die Spezifikation SP um eine Operation f und deren Definition in der relationalen Form Φ. Wir zeigen die Behauptung des Theorems für die erweiterte Spezifikation SP'.

Sei $\mathcal{M}$ eine endliche Struktur mit $\mathcal{M} \models \tau_{\mathtt{fin}}(SP')$. Es muss eine Algebra $\mathcal{A}$ gefunden werden, die Modell von SP' ist und eine $\preceq$-abgeschlossene Teilstruktur $\mathcal{A}^0$ enthält, die zu $\mathcal{M}$ isomorph ist.

Aus $\mathcal{M} \models \tau_{\mathtt{fin}}(SP')$ folgt $\mathcal{M} \models \tau_{\mathtt{fin}}(SP)$ (wenn wir die Operation F in $\mathcal{M}$ maskieren). Nun benutzen wir die Annahme an SP:

$$\mathcal{M} \models \tau_{\mathtt{fin}}(SP) \quad \Leftrightarrow \quad \begin{array}{l} \textit{es gibt } \mathcal{A}, \mathcal{A}^0 \ : \ \mathcal{A} \models SP, \ \mathcal{A}^0 \subseteq \mathcal{A}, \ \mathcal{M} \simeq \mathcal{A}^0, \\ \mathcal{M} \textit{ ist abgeschlossen} \end{array}$$

Seien $\mathcal{B}$ und $\mathcal{B}^0$ Strukturen mit $\mathcal{B} \models SP$, $\mathcal{B}^0 \subseteq \mathcal{B}$, $\mathcal{M}$ ist $\preceq$-abgeschlossen in $\mathcal{B}$ und $\mathcal{B}^0 \simeq_F \mathcal{M}$ (*modulo* F, weil F in $\mathcal{B}$ nicht modelliert ist). Wir konstruieren $\mathcal{A}, \mathcal{A}^0$ aus $\mathcal{B}, \mathcal{B}^0$ und zeigen, daß diese Konstruktionen die gewünschten Eigenschaften bezüglich SP' aufweisen.

Sei $\mathcal{A}$ die Erweiterung von $\mathcal{B}$, so daß $[\![\Phi]\!]^{\mathcal{A}}$ gilt. Nach der Annahme an SP existiert so eine Algebra, da Φ (falls rekursiv) eine wohlfundierte Rekursion ist. Dann ist $\mathcal{A}$ auch ein Modell von SP'.

Ähnlich, sei $\mathcal{A}^0 =_F \mathcal{B}^0$. In diesem entscheidenden Schritt verlangen wir, daß $[\![F]\!]^{\mathcal{A}^0} := [\![F]\!]^{\mathcal{M}}$. Somit, $\mathcal{A}^0 \simeq \mathcal{M}$, aber es bleibt zu zeigen, daß $\mathcal{A}^0$ auch eine Teilstruktur von $\mathcal{A}$ ist, d.h. wir müssen zeigen, daß $[\![F]\!]^{\mathcal{A}^0}$ mit $[\![F]\!]^{\mathcal{A}}$ auf $dom(\mathcal{A}^0)$ übereinstimmt:

$$\forall \overline{d} \in dom(\mathcal{A}^0).\ \overline{d} \in [\![F]\!]^{\mathcal{A}^0} \Leftrightarrow \overline{d} \in [\![F]\!]^{\mathcal{A}}$$

Falls Φ eine rekursive Definition ist, dann starten wir die Induktion über die Argumente der Operation $(d_1, \ldots, d_{n-1})$ für $\overline{d} = (d_1, \ldots, d_n) \in dom(\mathcal{A}^0)$. Die Induktion basiert auf der wohlfundierten Ordnung $\prec$, daß der wohlfundierten rekursiven Definition von F entspricht (für die KIV Definitionen ist es die strukturelle Induktion).

Sei $\overline{d} \in dom(\mathcal{A}^0)$. Wie vorher fixiert gilt: $\mathcal{A} \models \Phi$, $\mathcal{M} \models \Phi$ (damit auch $\mathcal{A}^0 \models \Phi$). Laut der Definition $\Phi \equiv \forall \underline{x}, y.\ F(\underline{x}, y) \leftrightarrow \Psi(\underline{x}, y)$ verwandelt sich die obere Beweisverpflichtung in die folgende zu zeigende Aussage:

$$[\![\Psi]\!]^{\mathcal{A}^0, \alpha[(\underline{x},y):=\overline{d}]} \Leftrightarrow [\![\Psi]\!]^{\mathcal{A}, \alpha[(\underline{x},y):=\overline{d}]}$$

wo $\Psi \equiv \mathcal{Q}_1 v_1, \ldots \mathcal{Q}_k v_k.\ \chi(\underline{v}, \underline{x}, y)$, $Free(\Psi) = \{\underline{x}, y\}$, $\alpha : Vars_i \to dom(\mathcal{A}^0)$ (Ψ hat die gleiche Bedeutung in $\mathcal{A}$ und $\mathcal{A}^0$). Sei gemerkt (später wichtig), dass für α gilt (trivialerweise): $\forall v \in Free(\Psi).\ \bigvee_{u \in \underline{x} \cup y} \alpha(v) \preceq \alpha(u)$, weil $Free(\Psi) = \emptyset$. Wir setzen voraus, dass α dise Eigenschaft hat.

Jetzt starten wir die strukturelle Induktion über den Aufbau der Formel Ψ:

Basisfall • $\Psi \equiv r(\underline{v})$ vorausgesetzt für α gilt $\bigvee_{u \in \underline{x} \cup y} \alpha(v_i) \preceq \alpha(u)$.
$[\![r(\underline{v})]\!]^{\mathcal{A},\alpha} = (\alpha(v_1), \ldots, \alpha(v_k)) \in [\![r]\!]^{\mathcal{A}} =^* (\alpha(v_1), \ldots, \alpha(v_k)) \in [\![r]\!]^{\mathcal{A}^0} = [\![r(\underline{v})]\!]^{\mathcal{A}^0,\alpha}$. Im Schritt $=^*$ haben wir das folgende Wissen eingesetzt: die Annahme über α, die $\preceq$-Abgeschlossenheit von $\mathcal{A}^0$ (somit gilt $\alpha(v_i) \in dom(\mathcal{A}^0)$), $[\![G]\!]^{\mathcal{A}^0} = [\![G]\!]^{\mathcal{A}}$ auf $dom(\mathcal{A}^0)$ für $G \neq F$. Im Fall von $r = F$ (rekursive Aufruf) wird die Induktionshypothese der Induktion über $\overline{d}$ angewandt, die die Annahme benutzt, daß die Rekursion wohlfundiert ist. Die Fälle der logischen Operatoren $\wedge$ und $\neg$ sind trivial. Wir müssen nicht den Operator der transitiven Hülle $^\wedge r$ und der Komposition $r_1.r_2$ betrachten, da sie in Ψ nicht vorkommen.

Quantoren • $\Psi \equiv \exists z.\ \Psi_0$, angenommen $\forall v \in Free(\Psi).\ \bigvee_{u \in \underline{x} \cup y} \alpha(v) \preceq \alpha(u)$.
$[\![\exists z.\ \Psi_0]\!]^{\mathcal{A},\alpha} = \exists d_z \in dom(\mathcal{A}).\ [\![\Psi_0]\!]^{\mathcal{A},\alpha[z:=d_z]} =^*$
$\exists d_z \in dom(\mathcal{A}^0).\ [\![\Psi_0]\!]^{\mathcal{A},\alpha[z:=d_z]} =^{Ind.} [\![\exists z.\ \Psi_0]\!]^{\mathcal{A}^0,\alpha}$
Im Schritt $=^*$ haben wir die Eigenschaft der $\preceq$-Kompatibilität der Definition Ψ benutzt, d.h. wenn die Aussage $\exists d_z \in dom(\mathcal{A}) \ldots$ wahr ist, dann ist d_z^0 mit einem Wert d_z in $dom(\mathcal{A}^0)$ belegt (wegen $\bigvee_{u \in \underline{x} \cup y} d \preceq \alpha(u)$, $\alpha(u) \in dom(\mathcal{A}^0)$ und $\preceq$-Abgeschlossenheit von $\mathcal{A}^0$). Anschließend, kann die Induktion angewandt werden, weil $\alpha' \equiv \alpha[z := d_z^0]$ die Eigenschaft der Begrenztheit erfüllt: $\forall v \in Free(\Psi).\ \bigvee_{u \in \underline{x} \cup y} \alpha'(v) \preceq \alpha'(u)$.

• $\Psi \equiv \forall z.\ \Psi_0$ vorausgesetzt $\forall v \in Free(\Psi).\ \bigvee_{u \in \underline{x} \cup y} \alpha(v) \preceq \alpha(u)$. Ähnlich wie für $\exists$ Fall und der Benutzung der $\preceq$-Kompatibilität von Ψ.

Damit ist die Struktur $\mathcal{A}^0$ eine echte Teilstruktur von $\mathcal{A}$ und es gilt $\mathcal{A}^0 \simeq \mathcal{M}$. □

Theorem 4 *[Korrektheit: Erweiterung II]*
Seien SP eine algebraische Spezifikation der freien Datentypen, deren Signatur nur Selektorfunktionen enthält. Sei

$$SP' = SP + (f_1, \Phi_1) + \ldots + (f_n, \Phi_n)$$

eine Erweiterung von SP um die Operationen $f_1, \ldots, f_n$ mit kompatiblen Definitionen $\Phi_1, \ldots, \Phi_n$ in der relationalen Form.

Dann gilt:

$$\mathcal{M} \models \tau_{\mathtt{fin}}(SP') \quad \Leftrightarrow \quad \begin{array}{l} \text{es gibt } \mathcal{A}, \mathcal{A}^0 \ :\ \mathcal{A} \models SP',\ \mathcal{A}^0 \subseteq \mathcal{A},\ \mathcal{M} \simeq \mathcal{A}^0, \\ \mathcal{M} \text{ ist abgeschlossen} \end{array}$$

Proof.
Induktiv über die Operationen $f_1, \ldots, f_n$ unter Anwendung von Theorem 2 (Basisfall) und Theorem 3 (Induktionsschritt). □

4.3.5 Effizienter Kompatibilitätstest

Um zu wissen, ob eine Spezifikation mit Alloy sich analysieren lässt, müssen die Definitionen der einzelnen Operationen auf die Kompatibilität mit der vorgegebenen Präordnung $\preceq$ überprüft werden. Dafür müsste für die Definition

$$\Phi \equiv \forall \underline{x}, y.\ F(\underline{x}, y) \leftrightarrow \mathcal{Q}v_1.\ \ldots \mathcal{Q}v_k.\ \chi(\underline{v}, \underline{x}, y)$$

folgende Beweisverpflichtung gezeigt werden:

$$\vdash \Upsilon(\mathcal{Q}v_1.\ \ldots \mathcal{Q}v_k.\ \chi(\underline{v}, \underline{x}, y), \underline{x} \cup \{y\})$$

Aus Gründen der Effizienz würde man diese Überprüfung einfacher machen. Die Axiomatisierungen der Operationen in KIV sind häufig uniform, weil der Mensch gerne nach einem bestimmten Schema vorgeht. Z.B., die Quantoren $Qv_1.\ \ldots Qv_n.$ in Definitionen sind alle universell bzw. alle existentiell. Für solche Fälle können effiziente Heuristiken zur Kompatibilitätsüberprüfung entworfen werden. Diese Heuristiken können dann automatisch mittels syntaktischer Analyse die Tests durchführen.

Wir betrachten zwei Klassen von Definitionen (in der relationalen Form):

$$\begin{aligned} \textit{existentiell} &: \forall \underline{x}, y.\ F(\underline{x}, y) \leftrightarrow \exists \underline{z}.\ \chi(\underline{x}, y, \underline{z}) \\ \textit{universell} &: \forall \underline{x}, y.\ F(\underline{x}, y) \leftrightarrow \forall \underline{z}.\ \chi(\underline{x}, y, \underline{z}) \end{aligned}$$

Die Formel χ ist quantorenfrei und $Free(\chi) = \underline{v} \cup \underline{x} \cup \{y\}$. Zunächst betrachten wir die Definitionen in der existentiellen Form. Als eine zusätzliche Annahme sei die Formel $\chi(\underline{x}, y, \underline{z})$ in der disjunktiven Normalform (DNF):

$$\bigvee_i \bigwedge_j \tilde{A}_{i,j}$$

wobei

$$\tilde{A}_{i,j} \in \{r(\overline{v}),\ \neg\, r(\overline{v}),\ x_1 = x_2,\ \neg x_1 = x_2\}$$

Zu beachten ist die folgende Eigenschaft, die sich aus der Konstruktion der relationalen Form der Definition ergibt:

$$\bigcap_i Free(\bigwedge_j \tilde{A}_{i,j}) \subseteq \{\underline{x}, y\}$$

D.h. die Mengen der freien Variablen der einzelnen Konjunktionen sind paarweise disjunkt modulo $\{\underline{x}, y\}$. Nun muss es für jedes i überprüft werden, ob die Eigenschaft der Beschränktheit durch $\{\underline{x}, y\}$ für die erfüllbare Belegung der freien Variablen in der jeweiligen Konjunktion gilt:

$$SP \models \forall v \in Free(\bigwedge_j \tilde{A}_{i,j}).\ \bigwedge_j \tilde{A}_{i,j} \rightarrow \bigvee_{u \in \{\underline{x}, y\}} v \preceq u$$

Die obere Eigenschaft wird separat für jede Konjunktion mit Hilfe einer Heuristik überprüft. Die Heuristik verwendet ausschließlich die syntaktische Analyse der Formel $\bigwedge_j \tilde{A}_{i,j}$. Es wird ein gerichteter Graph $G = (V, E)$ konstruiert. Die Menge der Knoten ist $V = \{\underline{x}, y\} \cup Vars(\bigwedge_j \tilde{A}_{i,j})$. Die Menge der Kanten E wird aus der Operationsdefinition nach der folgenden Regel generiert:

$$(v_1, v_2) \in E \ \Leftrightarrow \exists j.\ \tilde{A}_{ij} = C(\ldots, v_2, \ldots, v_1) \tag{4.2}$$

wo C eine Relation ist, die einer Konstruktorfunktion entspricht.

Definition 12 *[Heuristik]*
Gegeben eine existentielle Definition Φ und der nach oberen Regel konstruierte Graph $G = (V, E)$. Φ ist kompatibel bezüglich der Präordnung $\preceq$ genau dann wenn jedes $z \in V$ von einem Knoten aus der Menge $\underline{x} \cup \{y\}$ in G erreichbar ist:

$$\forall z \in \underline{z}.\ \exists v \in \underline{x} \cup \{y\}.\ Path(v, z)$$

Beispiel (Definition von *reverse*).

Betrachten wir als Beispiel die Definition der Funktion *reverse*. Dieses Beispiel wurde bereits in Abschnitt 4.3.2 behandelt. Dabei wurde festgestellt, daß die Abgeschlossenheit bezüglich der Subtermrelation für die korrekte Modellgenerierung nicht ausreicht. Daher nehmen wir hier die stärkere $\#_{list}$-Abgeschlossenheit (*size* Funktion $\#_{list} : list \rightarrow nat$), siehe Definition 13.

Definition 13 *[#-Abgeschlossenheit]*
Das endliche Teilmodell $\mathcal{M}$ des Modells $\mathcal{A}$ ist #-abgeschlossen genau dann wenn

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

Es ist zu zeigen, daß diese Definition bezüglich der #-abgeschlossenen endlichen Modellen kompatibel ist. Die entsprechende Präordnung ist größer als die Subtermrelation: für eine Liste x müssen alle #-kleineren Listen auch im Modell enthalten sein. Dies verletzt die Anforderung der Endlichkeit der Modelle nicht, da die Domäne der Elemente endlich ist. Die KIV Axiomen $rev([\,]) = [\,]$ und $rev(a + x) = rev(x) + (a + [\,])$ werden in die relationale Form gebracht:

$$\forall x, y.\ REV(x, y) \leftrightarrow \exists a, z_1, z_2, z_3, z_4. \qquad \begin{array}{l} NIL(x) \wedge NIL(y)\ \vee CONS(a, z_1, x) \wedge \\ REV(z_1, z_2) \wedge NIL(z_3) \wedge \\ CONS(a, z_3, z_4) \wedge APP(z_2, z_4, y) \end{array}$$

Dieser Definition entspricht ein gerichteter Graph, siehe die Abbildung 4.9. Die mit Dreieck kennzeichneten Knoten sind die quantifizierten Variablen auf der rechten Seite der Äquivalenz in der Definition.

Die durchgezogenen Kanten sind Ergebnisse der Anwendung der Heuristik auf die Fälle mit dem Konstruktor *CONS*. Die gestrichelten Kanten sind zusätzlich eingefügt und sind notwendig um die Erreichbarkeit der Dreieckknoten zu bekommen. Diese Information basiert auf der Eigenschaft der *append* Funktion im Zusammenhang mit der #-Abgeschlossenheit:

$$SP \models APP(x_1, x_2, y) \rightarrow x_1 \preceq_{\#} y \wedge x_2 \preceq_{\#} y$$

D.h., die APP-Ausgabe y ist immer #-größer als die APP-Eingaben x_1, x_2. Damit kann APP an der Stelle vom Konstruktor C in der Heuristik (4.2) verwendet werden.

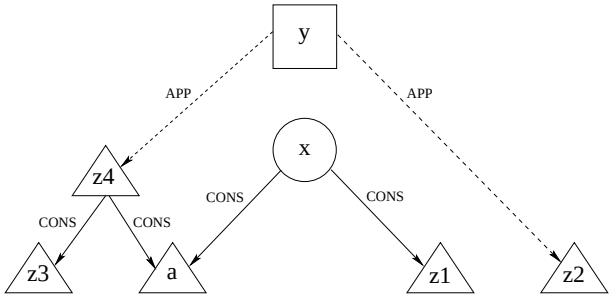


Abbildung 4.9: Graph für den heuristischen Kompatibilitätstest der Definition von *reverse*.

Im Allgemeinen, kann es für manche Operationen f notwendig sein den Zusammenhang zwischen ihren Eingaben und der Ausgabe bezüglich der Präordnung $\preceq$ zu wissen:

$$SP \models f(\underline{x}) = y \rightarrow x_i \preceq y \tag{4.3}$$

Strenggenommen müsste der Formale Beweis für (4.3) gemacht werden. In der Praxis wird dieser Schritt meistens informell durchgeführt.

Ein Punkt ist noch nicht untersucht worden und muss offen bleiben: kann Alloy automatisch die Eigenschaften (4.3) der bereits definierten Funktionen auf den endlichen Modellen $\mathcal{M}$ überprüfen ? Dies wäre vor allem aus praktischer Sicht interessant.

Der Fall einer universellen Definition kann trivial zu einer existentiellen Form umgewandelt werden. Die ganze Formel wird negiert. Z.B., für die Definition des Prädikats *disjoint* in der universellen Form:

$$disj(x, y) \leftrightarrow \forall a.\ \neg(a \in x \wedge a \in y)$$

bekommt man nach der Negation eine Definition in der existentiellen Form:

$$\neg disj(x, y) \leftrightarrow \exists a.\ a \in x \wedge a \in y$$

Diese kann einfach auf die Kompatibilität bezüglich, z.B., der Subtermrelation überprüft werden.

4.4 Nicht-Freie Datentypen

Die nicht-frei generierten Datentypen werden praktisch in jeder KIV Fallstudie verwendet. Die typischen Beispiele sind Sets, Stores, Integers, Arrays. In diesem Kapitel betrachten wir den bekanntesten nicht-freien Datentyp *Store*

als Beispiel. Stores repräsentieren das abstrakte Speichermodell mit Selektoren *address* und *data*. Meistens werden sie bei der Spezifikation von komplexeren Datenstrukturen wie eine Halde, Hashtabelle, Dateisystem oder Java VM Speicher verwendet.

4.4.1 Algebraische Spezifikation

Die Algebraische Spezifikation von Stores ist mit *elem* (Adressen) und *data* parametrisiert. Diese können später beliebig anwendungsspezifisch instanziert werden. Z.B., um ein Dateisystem zu spezifizieren würde man für *elem* die Pfade (als Listen repräsentiert) nehmen und für *data* - Dateien (freie Datentyp *Datei*).

Der Datentyp *Store* wird mit Konstruktoren

- $\oslash : store$ (Leere Store, Konstante)
- $.\,[\,.\,,\,.\,] : store \times elem \times data \rightarrow store$ (Daten unter einer Adresse im Store ablegen, wenn nötig Adresse im Store allozieren)

generiert.

Zusätzlich werden noch die Operationen

- $.\,[\,.\,] : store \times elem \rightarrow data$ (Zugriff auf die Daten unter einer Adresse)
- $.\,--\,. : store \times elem \rightarrow store$ (Delete)
- $.\,\subseteq\,. : store \times store$ (Teilmengenbeziehung) spezifiziert

Die Spezifikation der natürlichen Zahlen *nat* wird importiert, da sie später gebraucht wird.

Neben der Axiomatisierung der einzelnen Operationen wird die Äquivalenz zweier Stores mittels des Extensionalitätsaxioms spezifiziert:

$$st_1 = st_2 \leftrightarrow \forall a.\ (a \in st_1 \leftrightarrow a \in st_2) \wedge st_1[a] = st_2[a]$$

Damit gehören zwei Stores in eine Äquivalenzklasse genau dann wenn die gleichen Adressen alloziert sind und der Inhalt auch gleich ist.

Definition 14 *[Σ-Kongruenz]*
Sei $\Sigma = (S, \mathcal{F}, \mathcal{P})$ und $\mathcal{A} \in Alg(\Sigma)$. Σ-Kongruenz R ist eine durch S induzierte Familie der Äquivalenzrelationen $(R_s)_{s \in S}$, die mit den Operationen $\mathcal{F} \cup \mathcal{P}$ verträglich sind:

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

```
generic specification
      parameter elem, data
      target sorts store
      using nat
      constants
            ⊘ : store;      comment : empty
      functions
            . [ . , . ] : store × elem × data → store;      comment : put
            . [ . ] : store × elem → data;      comment : at
      predicates
            . ∈ . : elem × store;
      variables
            st, st0, st1, st2 : store;
      induction
            store generated by ⊘, . [ . , . ];
      axioms
            Ext : st1 = st2 ↔ ∀a. (a ∈ st1 ↔ a ∈ st2) ∧ st1[a] = st2[a];
            In-empty : ¬ a ∈ ⊘;
            In-put : a ∈ st[b, d] ↔ a = b ∨ a ∈ st;
            At-same : (st[a, d][a]) = d;
            At-other : a ≠ b → (st[b, d][a]) = st[a];
end generic specification

enriched specification
      functions
            . -- . : store × elem → store;      comment : delete
      predicates
            . ⊆ . : store × store;
      axioms
            Subset : st1 ⊆ st2 ↔ (∀a. a ∈ st1 → a ∈ st2 ∧ st1[a] = st2[a]);
            Del-in : a ∈ st--b ↔ a ≠ b ∧ a ∈ st;
            Del-at : a ≠ b → (st--b)[a] = st[a];
end enriched specification
```

Abbildung 4.10: Die Basisspezifikation des nicht-freien Datentyps *Store* und die Anreicherung mit zwei Operationen: *delete* und *subset*.

Für eine vorgegebene Σ-Kongruenz heißen die entsprechenden Modelle *Quotientenalgebren.*

Definition 15 *[Quotientenalgebra]*
Für eine Σ-Kongruenz R auf $\mathcal{A}$ ist $\mathcal{A}/_R = ((Q_s)_{s\in S}, (f^\circ)_{f\in\mathcal{F}}, (p^\circ)_{p\in\mathcal{P}})$ die entsprechende Quotientenalgebra:

1. *$Q_s = \{[a] : a \in \mathcal{A}_s\}$ wobei $[a] = \{b : R_s(a, b)\}$*
2. *Für die Operationen $f \in \mathcal{F}$, $p \in \mathcal{P}$ und $[a_i] \in Q_{s_i}$:*
   $f^\circ([a_1], \ldots, [a_n]) = [f_\mathcal{A}(a_1, \ldots, a_n)]$
   $p^\circ([a_1], \ldots, [a_n]) \Leftrightarrow [p_\mathcal{A}(a_1, \ldots, a_n)]$

Da R eine Kongruenz ist, sind alle Operationen wohldefiniert und $\mathcal{A}/_R \in Alg(\Sigma)$. Trivialerweise, wenn $\mathcal{A}$ eine termgenerierte Algebra ist, dann ist $\mathcal{A}/_R$ auch termgeneriert.

4.4.2 Alloy Spezifikation

Bei der Generierung der endlichen abgeschlossenen Modelle für nicht-freien Datentypen verwenden wir den Ansatz für freie Datentypen, der auf der SUA-Generierung beruht. Allerdings werden leichte Modifikationen vorgenommen, da die Trägermenge eine Quotientenalgebra ist. Sie besteht aus den Äquivalenzklassen, die durch die vorgegebene Σ-Kongruenz induziert sind.

Jede solche Klasse kann durch einen Term mit der minimalen Konstruktionskomplexität repräsentiert werden (solche Repräsentation ist *nicht eindeutig*!). Die Konstruktionskomplexität eines Terms wird durch die maximale Tiefe des Syntaxbaumes des entsprechenden Konstruktorterms ($\in T(C_s, X/X_s)$) bestimmt. Sei die entsprechende *size*-Funktion $\#_s : s \to nat$.

Lemma 1 *Eine Struktur, die aus den minimalen Repräsentanten der Äquivalenzklassen von $\mathcal{A}/_R$ besteht, ist subtermabgeschlossen.*

Es existiert eine noethersche Ordnung $\prec_m$ auf den minimalen Repräsentanten, die sich auf die Komplexität der Termkonstruktion bezieht. Wenn $[\ .\ ]_m : store \to store$ eine Abbildung ist, die zu einem Term einen minimalen Repräsentanten berechnet, dann ist $\prec_m$ wie folgt definiert:

$$[t_1] \prec_m [t_2] \Leftrightarrow \#_s([t_1]_m) \leq \#_s([t_2]_m)$$

Damit lassen sich die endlichen subtermabgeschlossenen Teilstrukturen mit *SUA*-basierten Ansatz generieren. Die grundlegende Idee, die das Lemma 1 verwendet, ist ein zusätzliches restriktives Axiom zu spezifizieren, das die SUA-Generierung nur auf die "richtigen" subtermabgeschlossenen Strukturen einschränkt. Unter "richtig" sind die Strukturen, die nur die minimalen Repräsentanten enthalten, gemeint. Ein Term wird nur dann dazugeneriert, wenn er nicht zu der gleichen

Äquivalenzklasse gehört und minimal in seiner Klasse ist. Aus der Minimalität folgt, daß die neu dazugenerierte Äquivalenzklasse größer bezüglich $\prec_m$ ist.

Diese Eigenschaft mit Hilfe des Prädikats *BIGGER* in die SUA-Generierung integriert. *BIGGER* legt fest, wenn ein Konstruktor angewandt werden darf, ohne daß die Minimalität verletzt wird.

Z.B. für Stores:

$$\texttt{bigger}: \quad \forall st : store,\ a : elem,\ d : data.\ BIGGER(st, a, d) \leftrightarrow \neg IN(a, st)$$

Für ein Store st ist die Anwendung von dem Konstruktor $put : store \times elem \times data \rightarrow store$ zulässig genau dann wenn die entsprechende Adresse a noch nicht in st alloziert wurde. Dann führt das Speichern von Daten d unter der Adresse a zu einem Store st', das zu einer anderen Äquivalenzklasse gehört und minimal ist. Die Minimalität folgt trivialerweise aus der Minimalität von st.

Das Axiom, das diesen Prädikat in die SUA-Generierung integriert, sieht für Stores wie folgt:

$$\texttt{put-bigger}: \quad \forall st, st_1 : store,\ a : elem,\ d : data.\ PUT(st, a, d, st_1) \rightarrow BIGGER(st, a, d)$$

Da wir *PUT* für die Generierung der Trägermenge verwendet haben, weicht jetzt seine Semantik von der ursprünglichen Semantik der *put* Funktion ab: es ist eine echte Teilmenge davon. Jetzt muss die ursprüngliche *put* Operation neu spezifiziert werden. Wir bezeichnen sie mit $\widehat{PUT}$[2]:

$$\begin{aligned}\texttt{new-put}: \quad & \forall st, st_1 : store,\ a : elem,\ d : data.\ \widehat{PUT}(st, a, d, st_1) \leftrightarrow \\ & IN(a, st_1) \wedge AT(st_1, a, d) \wedge \\ & (\forall b : elem.\ a \neq b \rightarrow (IN(b, st_1) \leftrightarrow IN(b, st)) \wedge \\ & \quad (\exists d_1, d_2 : data.\ AT(st, b, d_1) \wedge AT(st_1, b, d_2) \rightarrow d_1 = d_2))\end{aligned}$$

Annahme.

Die Axiome `bigger` und `new-put` verwenden die Hilfsoperationen *AT* und *IN*. Die beiden Operationen sind rekursiv definiert und ihre Definitionen verwenden die Konstruktorfunktion *PUT*. Wie vorher erwähnt gilt $PUT \subset \widehat{PUT}$. Als eine wichtige Voraussetzung, nehmen wir an, daß die Verwendung von *PUT* in den Definition von Hilfsoperationen *IN* und *AT* die richtige Semantik der jeweiligen Operation ergibt.

Im Folgenden wird die Alloy Spezifikation beschrieben. Das Alloy Modul ist mit Sorten *Data* und *Elem* parametrisiert. Es wird die Spezifikation der natürlichen Zahlen importiert, da sie für die Generierung benötigt werden.

Alloy Spezifikation des nicht-freien Datentyps *Store*:

[2] In Alloy Spezifikation die Relation `put`.

```
module store[Elem,Data]
open nat as Nat

sig Store {
   cplx: one Nat,
   isin: set Elem,
   at: Elem -> lone Data,
   put: Elem -> Data -> lone Store,

-- enriched operations
   size: lone Nat,
   delete: Elem -> lone Store,
   sub: set Store
}
one sig Empty extends Store {}

sig Put extends Store {
   address : Elem,
   data: Data,
   rest: Store
}

// SUA generation
fact Uniqueness {
   all st1,st2: Store |
      st1.address = st2.address and
      st1.data = st2.data and
      st1.rest = st2.rest
   => st1 = st2
}

fact Acyclic {
   no st: Store | st in st.^rest
}


fact EXTENSION {
   all st1,st2: Store |
      st1 = st2 <=>
      all a: Elem |
         ((st1->a) in isin <=> (st2->a) in isin) and
         at[st1][a] = at[st2][a]
}

// constraint for generation of minimal representatives of quotients
fact PUT_BIGGER {
   all st,st1: Store, a: Elem, d: Data | PUT[st,a,d,st1] => BIGGER[st,a,d]
}

fact FINITE_GENERATOR {
   all st: Store, a: Elem, d: Data | lt[st.cplx,Zero.~prede.~prede] and BIGGER[st,a,d]=>
      some st1: Store | (st->a->d->st1) in put
}


fact BOUND {
   all st: Store | lte[st.cplx,Zero.~prede.~prede]
}


fact CPLX {
   cplx = {st: Store, n: Nat |
```

```
      st= Empty and n = Zero
      or
      some st1: Store, a: Elem, d: Data, m: Nat | {
         PUT[st1,a,d,st]
         (st1->m) in cplx
         m = n.prede
      }
   }
}


fact ISIN {
   isin = {st: Store, a: Elem |
      not st = Empty and
      some st1: Store, b: Elem, d: Data |
         PUT[st1,b,d,st] and
         (b = a or
         (st1->a) in isin)
   }
}

fact AT {
   at = {st: Store, a: Elem, d: Data |
      {
         some st1: Store |
            PUT[st1,a,d,st]
      } or
      {
         some st1: Store, b: Elem, d1: Data |
            not a = b and
            PUT[st1,b,d1,st] and
            (st1->a->d) in at
      }
   }
}

fact PUT_constructor {
   put = {st: Store, a: Elem, d: Data, st1: Store |
      (st1->a) in isin and at[st1][a] = d and
      (all b: Elem | not a = b => ((st->b) in isin <=>
                                   (st1->b) in isin) and at[st][b] =at[st1][b])
   }
}


// predicates
pred PUT [st: Store, a: Elem, d: Data, st1: Store] {
   st1.address = a and st1.data = d and st1.rest = st
}
pred BIGGER [st: Store, a: Elem, d: Data] {
   not (st->a) in isin
}

pred testRun() {}

run testRun for exactly 2 Elem, exactly 2 Data, 9 Store, 3 Nat
```

Die Kernoperationen *isin, at, put, cplx* (Notwendig für die Generierung des Termmodells) sind in die Signatur von *Store* als Relationen eingetragen. Entsprechend der SUA-Generierung ist die Sorte *Store* in die Untersorten *Empty* und *Put* unterteilt, siehe die Abbildung 4.11. Für die Untersorte *Put*, die dem

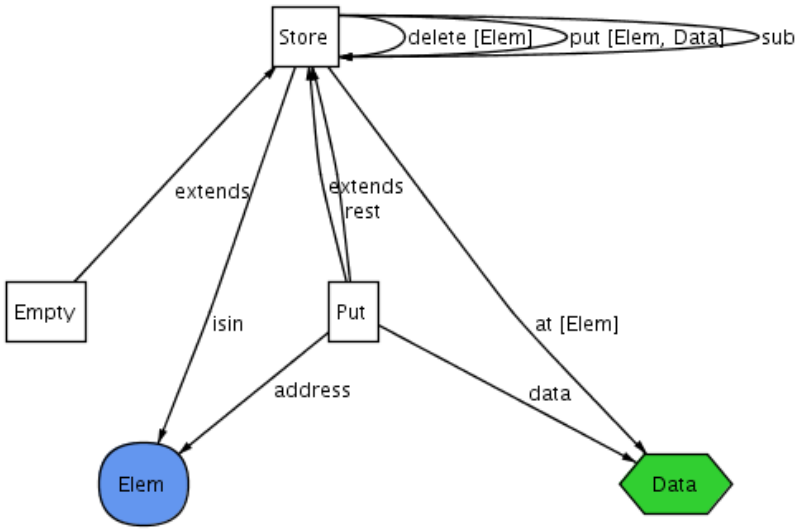


Abbildung 4.11: Metamodell für die Alloy Spezifikation von *Stores*.

Konstruktor *put* entspricht, sind die Selektoren *address, data, rest* in die Signatur eingetragen. Sie ermöglichen die SUA-Generierung, siehe Axiome *Uniqueness* und *Acyclic* in der Alloy Spezifikation. Das Axiom *PUT_BIGGER* erzwingt die Generierung von minimalen Repräsentanten. Mit den Axiomen *CPLX, FINITE_GENERATOR, BOUND* wird die *store*, 2-Abgeschlossenheit (#-Abschluss, alle *store*-Terme bis zu Größe 2) des Teilmodells spezifiziert. Der Ausdruck `Zero,~ prede.~ prede` wird in Alloy als die Zahl 2 interpretiert. Schließlich mit dem Befehl `run` wird Alloy aufgefordert unter der Berücksichtigung der vorgegebenen Schranken ein Teilmodell von Stores zu berechnen.

Die Abbildung 4.12 zeigt das Teilmodell $\mathcal{M}$, das zu der oberen Alloy Spezifikation berechnet wurde. Die Domänen enthalten:

$$\begin{aligned}\mathcal{M}_{store} = \{&\oslash, \oslash[a,d_1], \oslash[a,d_2], \oslash[b,d_1], \oslash[b,d_2], \oslash[a,d_1][b,d_1], \oslash[a,d_1][b,d_2],\\ &\oslash[a,d_2][b,d_1], \oslash[a,d_2][b,d_2]\}\\ \mathcal{M}_{elem} = \{a,b\},\ &\mathcal{M}_{data} = \{d_1,d_2\},\ \mathcal{M}_{nat} = \{0,1,2\}\end{aligned}$$

Für bessere Überschaubarkeit sind nur die Relationen sichtbar, die für die Identifikation der Werte der Atome notwendig sind. Die Operationen *at* (Daten auf der gegebenen Speicherposition) und *cplx* ($\#_{store}$) sind als Beschriftungen der Atome sichtbar.

4.4.3 Korrektheit

Um die korrekte Anwendung des Ansatzes im vorherigen Abschnitt zu garantieren, müssen die notwendigen Randbedingungen für die algebraische Spezifikation der nicht-freien Datentypen identifiziert werden. Bei der Argumentation werden die vier zusätzlichen Operationen verwendet, siehe Definition 16. Drei

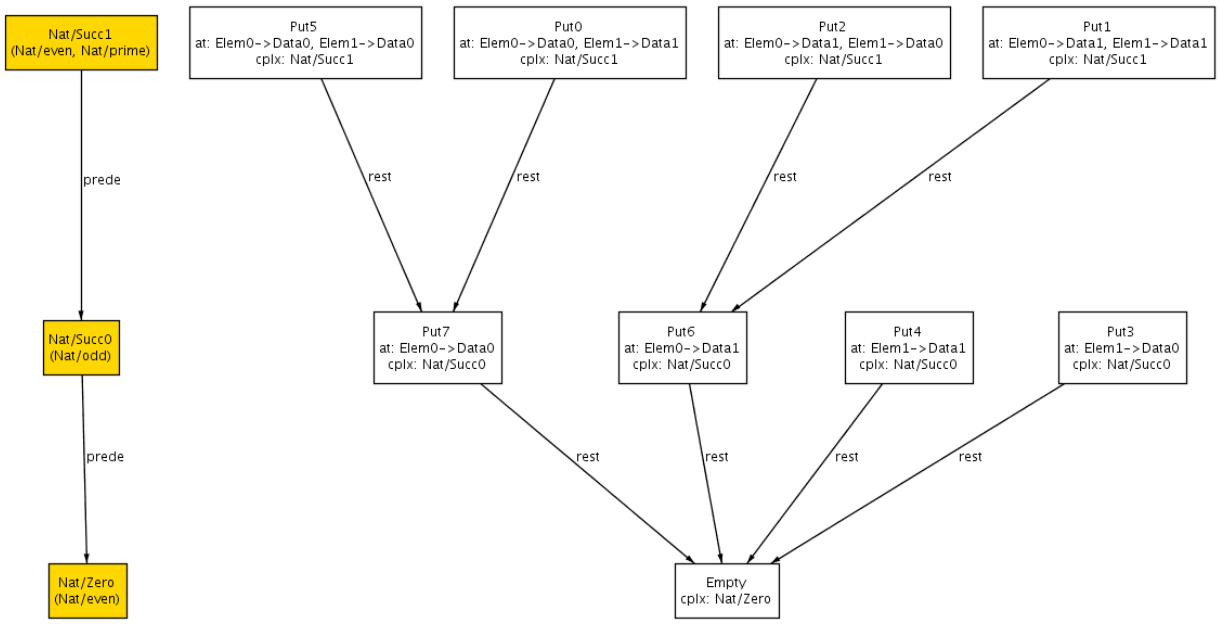


Abbildung 4.12: Eine mit Alloy generierte endliche subtermabgeschlossene Teilstruktur $\mathcal{M}$ für die Spezifikation von Stores.

von ihnen wurden bei den Überlegungen über die Korrektheit für freie Datentypen bereits verwendet (letzten drei). Der neue dazugekommene Konstrukt ist das Prädikat $\prec$: $s \times s$ auf der nicht-freien Sorte s (siehe Prädikat *BIGGER* für *Stores*).

Definition 16 *[Hilfsoperationen]*

1. $\prec$: $s \times s$ *ordnet die Terme bezüglich der minimalen Repräsentanten ihrer Klasse*
2. $\#_s : s \rightarrow nat$ *berechnet die Konstruktionskomplexität der s-wertigen Terme*
3. $\preceq$: $s_1 \times s_1 \cup s_1 \times s_2 \cup \ldots \cup s_k \times s_k$ *eine Präordnung auf Domänen*
4. $\Upsilon : For(SP) \times Vars$ *testet Definitionen auf die* $\preceq$*-Kompatibilität*

Für $\preceq$ wird in den meisten praktischen Anwendungen die Subtermrelation genommen. Manchmal wird die stärkere s, k - Abgeschlossenheit verwendet (alle Terme t der Sorte s der Größe $\#(t) \leq k$). Dafür wird auch die Operation $\#_s$ (*size*) benötigt. Die Definition der Operation $\prec$ kostet den meisten Aufwand bei der Anwendung des Ansatzes und ist zentral bei den Überlegungen über die Korrektheit.

Die Trägermenge $\mathcal{A}_s/_R$ für den nicht-freien Datentyp s enthält die Quotienten $[t]$. Diese können (nicht eindeutig) mit den minimalen Repräsentanten charak-

terisiert werden, d.h. Termen $t_m \in [t]$ mit der minimalen Konstruktionsgröße ($\#_s$). Sei $[\ .\ ]_m : s \to s$ die Funktion, die einen minimalen Repräsentanten berechnet. Das Prädikat $\prec: s \times s$ ist formal definiert:

$$t_1 \prec t_2 \leftrightarrow \#_s[t_1]_m < \#_s[t_2]_m$$

In der Alloy Spezifikation wird $\prec$ für die Generierung ausschließlich der Terme $[t]_m$ mit SUA Axiomen verwendet. Zusätzlich verlangen wir, daß $\prec$ mit $\preceq$ übereinstimmt. Für die weiteren Operationen, mit denen die Spezifikation später angereichert werden kann, gelten die gleichen $\preceq$-Kompatibilitätsanforderungen wie im Fall der freien Datentypen.

Zusammengefasst, können die folgenden drei Schritte zur Zeit nicht automatisch gemacht werden. Sie verlangen die **Kreativität** und die **Sorgfalt** seitens des Benutzers.

1. Spezifikation des Prädikats $\prec: s \times s$ für die SUA-basierte Generierung der minimalen Repräsentanten
2. Spezifikation der Konstruktorfunktionen $\hat{c}$ für jeden Konstruktor $c \in C_s$ für einen nicht-freien Datentyp s (vergleiche PUT und $\widehat{PUT}$ bei *Stores*)
3. Nachweis der $\preceq$-Kompatibilität für SP

Bei den ersten zwei Schritten wird eine besonders sorgfältige Betrachtung und Verständnis von $\mathcal{A}_s/_R$ verlangt. Obwohl sie aus Sicht des Benutzers äußerst kritisch und herausfordernd sind, sind sie in der Praxis auch am wenigsten arbeits- und zeitintensiv. Es liegt daran, daß in allen KIV Fallstudien die gewöhnlichen nicht-freien Datentypen aus der KIV Bibliothek (*Stores, Sets, Arrays, Integers, Graphs*) verwendet werden. Da diese Datentypen bereits nach Alloy portiert sind, sind die Kosten für ihre Verwendung gleich Null.

Der dritte Schritt ist der langwierigste. Hier muss jede neue anwendungsspezifische Operation auf die Kompatibilität untersucht werden. Für eine endgültige Sicherheit könnte sogar ein formaler Beweis in Frage kommen.

4.5 Analysierbare Klassen von Formeln

Definition 17 *[Bounded Quantoren, UBE Formeln]*

- *Ein bounded existentieller Quantor* $\exists_{v<t} v.\ \psi$ *ist die Abkürzung für*

$$\exists v.\ v < t \wedge \psi$$

- *ein bounded universeller Quantor* $\forall_{v<t} v.\ \psi$ *ist die Abkürzung für*

$$\forall v.\ v < t \to \psi$$

wobei t *ein beliebiger Term und* $<$ *die Subtermrelation auf Termen sind.*

Eine UBE Formel verwendet universelle und bounded existentielle Quantoren.

Kuncak und Jackson [26] definieren die EBU Formeln (existentiell-bounded universell), da sie sich für die Erfüllbarkeit interessieren. Für uns ist die Widerlegbarkeit (Gegenbeispiele) einer Formel interessant. Deshalb befassen wir uns mit den UBE Formeln (negierte EBU Formeln). Die Beschränktheit wird mit Hilfe der Ordnung $<$ auf den Termen formuliert: die Zusatzbedingung $v < t$ schränkt die quantifizierte Variable v ein, siehe Definition 17. Die beschränkte Quantifizierung in [26] erlaubt $v \in S$ für eine Menge S an der Stelle von $v < t$. Auf den ersten Blick sieht die Variante mit $v \in S$ viel liberaler aus, ist es aber nicht, da die Sprache in deren Ansatz nur auf Selektoren beschränkt ist. Wenn wir nur die beliebigen Funktionssymbole für den Ausdruck S erlauben, funktioniert der Ansatz nicht mehr. Deshalb setzen wir voraus, daß der Ausdruck t in bounded Quantoren nur aus Selektorfunktionen und bereits quantifizierten Variablen besteht. Das Theorem 5 schließt die Überlegungen über die Korrektheit von unserem Ansatz der begrenzten Analyse.

Theorem 5 *[Widerlegung mittels endlicher Gegenbeispiele]*
Sei SP eine algebraische Spezifikation der ersten Stufe und eine φ eine UBE Formel. Dann gilt:

$$\texttt{exists } \mathcal{M}: \ \mathcal{M} \models \tau_{\texttt{fin}}(SP),\ \mathcal{M} \not\models \tau(\varphi) \ \Rightarrow\ \texttt{es gibt } \mathcal{A}: \ \mathcal{A} \models SP,\ \mathcal{A} \not\models \varphi$$

Beweis.
Sei $\mathcal{M}$ ein endliches Modell der Alloy Spezifikation $\tau_{\texttt{fin}}(SP)$, das gleichzeitig ein Gegenbeispiel für die Formel $\tau(\varphi)$ ist: $\mathcal{M} \not\models \tau(\varphi)$. Wir müssen ein Modell $\mathcal{A}$ der KIV Spezifikation identifizieren mit $\mathcal{A} \not\models \varphi$.

Die relationale Form der Formel φ ist laut der Definition 6:

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

Da φ eine *UBE* Formel ist, ist die obere Formel auch eine *UBE* Formel. Die Formel $\tau(\varphi)$ hat keine freien Variablen (ist $\forall$-abgeschlossen). Folglich, gilt für ein Modell $\mathcal{M}$ die Äquivalenz:

$$\mathcal{M} \not\models \tau(\varphi) \Leftrightarrow \mathcal{M} \models \neg\tau(\varphi)$$

Betrachten wir die Formel $\neg\tau(\varphi)$:

$$\begin{aligned}\neg\tau(\varphi) \quad &\equiv \widehat{Q}_1 v_1 :: s_1 \ldots \widehat{Q}_n v_n :: s_n.\ \exists\, \vartheta(\mathfrak{T}_{all}).\\ &\bigwedge_{f(t_1,\ldots,t_k)\in\mathfrak{T}_{all}} (\vartheta(t_1),\ldots,\vartheta(t_k),\vartheta(f(t_1,\ldots,t_k))) \in F \wedge\ \neg\psi[\mathfrak{T}_{top} \diagdown \vartheta(\mathfrak{T}_{top})]\end{aligned}$$

Dies ist eine *EBU* Formel: mit $\widehat{Q}$ wird der inverse Quantor zum Quantor Q bezeichnet. Dabei wurde die Eigenschaft von bounded Quantoren verwendet:

$$\neg\exists_{v<t} v.\ \varphi \equiv \forall_{v<t} v.\ \neg\varphi$$

Da $\mathcal{M} \models \tau_{\mathtt{fin}}(SP)$ gilt, existiert eine Algebra $\mathcal{A}$ mit $\mathcal{A} \models SP$ und $\mathcal{M} \subset \mathcal{A}$ (Theorem 3, über die Korrektheit der Modellgenerierung).

Nun bleibt es zu zeigen, daß die Formel $\neg\tau(\varphi)$ auf der unendlichen Struktur $\mathcal{A}$ die gleiche Bedeutung wie auf dem endlichen Modell $\mathcal{M}$ hat: $\mathcal{A} \models \neg\tau(\varphi)$.

Grob gesagt gilt es, weil $\neg\tau(\varphi)$ eine *EBU* Formel ist und $\mathcal{M}$ ein Teilmodell von $\mathcal{A}$ ist: $\mathcal{M} \subset \mathcal{A}$. D.h., bei der Auswertung der existentiellen Quantoren in $\neg\tau(\varphi)$ können die gleichen Zeugen in $\mathcal{A}$ wie in $\mathcal{M}$ ausgewählt werden.

Wir zeigen mit der strukturellen Induktion über den Aufbau der Formel $\neg\tau(\varphi)$, daß für eine Belegung der Variablen $\alpha : Vars \to dom(\mathcal{M})$ gilt:

$$[\![\neg\tau(\varphi)]\!]^{\mathcal{M},\alpha} \Rightarrow\ [\![\neg\tau(\varphi)]\!]^{\mathcal{A},\alpha}$$

Die quantorenfreie Teilformel $\neg\psi[\mathfrak{T}_{top} \diagdown \vartheta(\mathfrak{T}_{top})]$ enthält nur Variablen aus $\vartheta(\mathfrak{T}_{top}) \cup \{v_1, \ldots, v_n\}$ und die Basisoperatoren $\wedge, \neg, \vee, =$. Folglich, gilt:

$$[\![\neg\psi[\mathfrak{T}_{top} \diagdown \vartheta(\mathfrak{T}_{top})]]\!]^{\mathcal{M},\alpha} \Rightarrow\ [\![\neg\psi[\mathfrak{T}_{top} \diagdown \vartheta(\mathfrak{T}_{top})]]\!]^{\mathcal{A},\alpha}$$

Die Disjunktion von $(\vartheta(t_1), \ldots, \vartheta(t_k), \vartheta(f(t_1, \ldots, t_k))) \in F$ bewahrt die Semantik beim Übergang von $\mathcal{M}$ nach $\mathcal{A}$ ebenfalls.

Es bleibt noch der interessanteste Fall von Quantoren zu betrachten.

Vorausgesetzt, gilt $[\![\varphi]\!]^{\mathcal{M},\alpha} \Rightarrow\ [\![\varphi]\!]^{\mathcal{A},\alpha}$ für $\varphi_1 \equiv Qv.\ \varphi$. Es ist zu zeigen, daß $[\![\varphi_1]\!]^{\mathcal{M},\alpha} \Rightarrow\ [\![\varphi_1]\!]^{\mathcal{A},\alpha}$ gilt. Betrachten wir die beiden Fälle von existentiellen und bounded-universellen Quantoren:

- Sei $\varphi_1 \equiv \exists v.\ \varphi$ und es gilt $[\![\varphi_1]\!]^{\mathcal{M},\alpha}$ für $\mathcal{M} \subset \mathcal{A}$. Dann existiert $d \in \mathcal{M}_i$ mit $[\![\varphi]\!]^{\mathcal{M},\alpha[\mathtt{v}:=\mathtt{d}]}$. Dann nach der Induktionshypothese gilt $[\![\varphi]\!]^{\mathcal{A},\alpha[\mathtt{v}:=\mathtt{d}]}$ und damit auch $[\![\varphi_1]\!]^{\mathcal{A},\alpha}$.
- Sei $\varphi_1 \equiv \forall_s v.\ \varphi$ und es gilt $[\![\varphi_1]\!]^{\mathcal{M},\alpha}$ für $\mathcal{M} \subset \mathcal{A}$. Wir benutzen die Annahme, daß der Ausdruck s nur Selektorfunktionen und bereits quantifizierten Variablen verwenden darf. Dann gilt $[\![s]\!]^{\mathcal{M},\alpha} = [\![s]\!]^{\mathcal{A},\alpha}$. Die Formel $\varphi_1 \equiv \forall v.\ v < s \wedge \varphi$. Vorausgesetzt $\mathcal{M}$ ist ein subtermabgeschlossenes Teilmodell von $\mathcal{A}$ gilt für alle $d \in \mathcal{M}_i$: $[\![v < s \wedge \varphi]\!]^{\mathcal{M},\alpha[\mathtt{v}:=\mathtt{d}]}$. Damit aber auch das gleiche aber für alle $d \in \mathcal{A}$, da in der Teilformel Constraint $v < s$ auftaucht und $\alpha(s) \in \mathcal{M}$. Unter der Benutzung der Induktionsannahme, daß die Teilformel φ in $\mathcal{M}$ und $\mathcal{A}$ mit α gleich ausgewertet werden, zeigen wir $[\![\varphi_1]\!]^{\mathcal{A},\alpha}$.

□

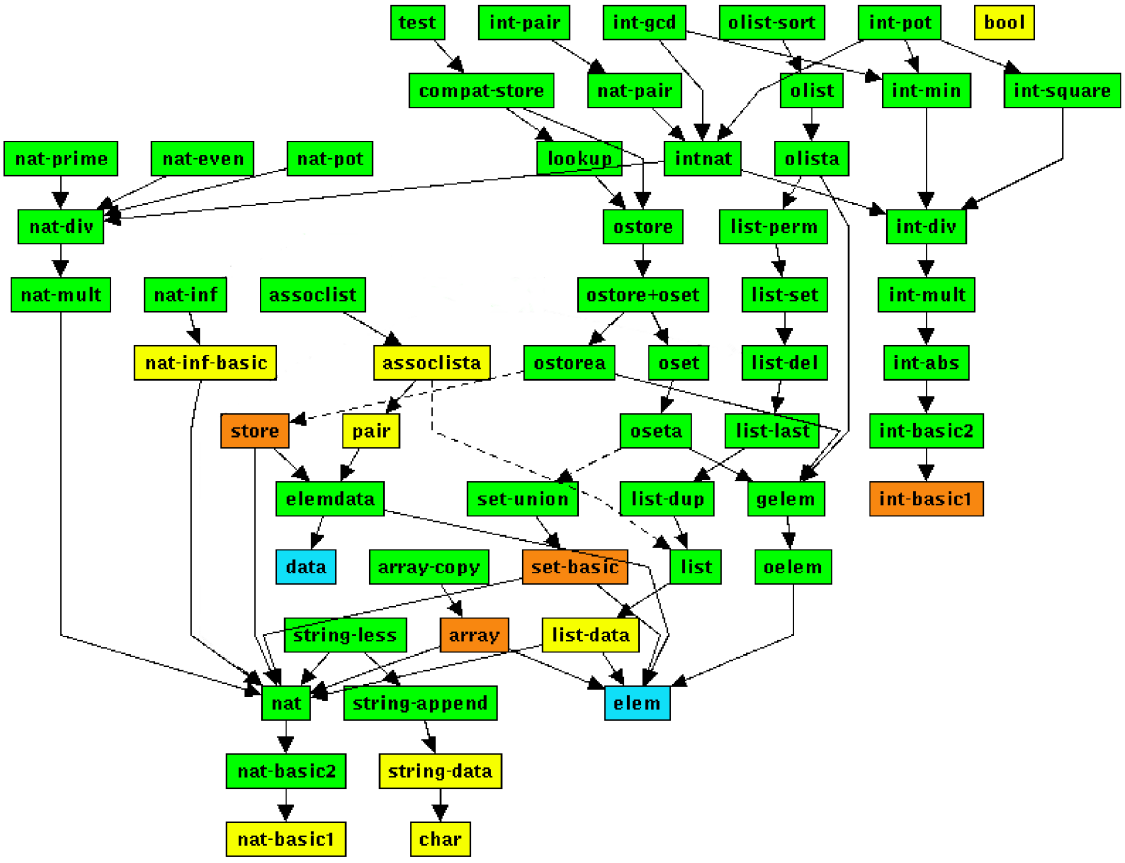


Abbildung 4.13: Spezifikationshierarchie für die KIV Bibliothek der grundlegenden Datentypen.

4.6 Erste Ergebnisse: Datentypen in der KIV Bibliothek

Die KIV Bibliothek ist eine Sammlung von Spezifikationen der essentiellen Datentypen, wie *Listen*, *Stores*. *Sets*, *Nats* oder *Integers*, die in jeder Fallstudie zur Spezifikation der anwendungsspezifischen Datenstrukturen verwendet werden. Neben den spezifizierten Datentypen sind auch zahlreichen nützlichen Operationen spezifiziert.

Die Fallstudien in KIV verwenden in erheblichem Maße die Datentypen und Operationen aus der Bibliothek. Da wir die KIV Bibliothek nach Alloy portiert haben, ist damit auch der große Teil des Aufwands für Portierung der zukünftigen Fallstudien erledigt.

Der **nicht automatisierte** Anteil an der Portierung beinhaltet:

I. Spezifikation von nicht frei generierten Datentypen

II. Kompatibilitätsüberprüfung der Operationsdefinitionen

Teil I. ist aufwendig aber in der Praxis glücklicherweise extrem selten und damit auch akzeptabel. Teil II. ist potentiell der aufwendigste Schritt im gesamten Ansatz, wird aber in der Praxis meistens informell (durch Draufgucken) abgehandelt.

Teil I. kam in Fallstudien glücklicherweise nie zum Zuge, da alle anwendungsspezifischen ($\notin$ KIV Bibliothek) Datentypen frei generiert waren. Theoretisch ist es aber möglich, daß neue nicht freie Datentypen in einer Fallstudie auftauchen, z.B. *Graphen* ($\notin$ KIV Bibliothek). Eine praxisrelevante Situation, wo laufend *verschiedene* nicht freie Datentypen spezifiziert werden mussten ist uns nicht bekannt.

Die Abbildung 4.13 zeigt die Spezifikationshierarchie für die KIV Bibliothek. Die grün gefärbten Spezifikationen sind Erweiterungen der Datentypspezifikationen um weitere Operationen. Alle anderen sind die Spezifikationen der Datentypen, die in folgende Arten eingeteilt sind:

blau: nicht generierten Datentypen (*data, elem*)

gelb: frei generierten Datentypen (*nat,char,string,list,pair,assoclist,nat-inf,bool*)

orange: nicht frei generierten Datentypen (*array, set, integer,store*)

Um die ersten quantitativen und qualitativen Ergebnisse zu illustrieren betrachten wir folgende fünf Spezifikationen:

- $SP_1 = (\{nat\}, (\mathcal{F}, \mathcal{P}), Gen_{nat}, Ax)$
  - $Gen_{nat} \equiv \{nat = 0 \mid . \ +1 \ (\ . \ -1 : nat)\}$
  - $\#\mathcal{F} \cup \mathcal{P} = 21$, $\#Ax = 43$, $\#Theorems = 753$
- $SP_2 = (\{list, elem, nat\}, (\mathcal{F}, \mathcal{P}), Gen_{list} \cup Gen_{nat}, Ax)$
  - $Gen_{list} \equiv \{list = [] \mid . \ + \ . (\ . \ .first : elem; \ . \ .rest : list)\}$
  - $\#\mathcal{F} \cup \mathcal{P} = 31$, $\#Ax = 64$, $\#Theorems = 868$
- $SP_3 = (\{store, elem, data, nat\}, (\mathcal{F}, \mathcal{P}), Gen_{store} \cup Gen_{nat}, Ax)$,
  - $Gen_{store} \equiv \{store$ `generated by` $\oslash, \ . [\ .\ ,\ .\ ]\}$
  - $\#\mathcal{F} \cup \mathcal{P} = 12$, $\#Ax = 16$, $\#Theorems = 84$
- $SP_4 = (\{set, elem, nat\}, (\mathcal{F}, \mathcal{P}), Gen_{set} \cup Gen_{nat}, Ax)$,
  - $Gen_{set} \equiv \{set$ `generated by` $\emptyset, ++\}$
  - $\#\mathcal{F} \cup \mathcal{P} = 16$, $\#Ax = 19$, $\#Theorems = 341$
- $SP_5 = (\{array, elem, nat\}, (\mathcal{F}, \mathcal{P}), Gen_{array} \cup Gen_{nat}, Ax)$.
  - $Gen_{array} \equiv \{array$ `generated by` $mkarray \ .\ ,\ .\ [\ .\ ,\ .\ ]\}$
  - $\#\mathcal{F} \cup \mathcal{P} = 12$, $\#Ax = 16$, $\#Theorems = 14$

Die oberen Spezifikationen wurden informell auf die Kompatibilität der definierten Operationen überprüft. Genauso, wurden die darin enthaltenen Theoreme auf die Zugehörigkeit zu der *UBE* Klasse überprüft. Mit Ausnahme von der Spezifikation SP_1 waren alle Definitionen bezüglich der #-Abgeschlossenheit kompatibel. In der Spezifikation SP_1 waren 4 Operationen nicht kompatibel: *div, mod, log, log2*. D.h., wir kennen keine Präordnung $\preceq$ bezüglich welcher die Definitionen der genannten Operationen kompatibel sein könnten.

Betrachte, z.B., die Operation $div : nat \times nat \rightarrow nat$ (Division auf natürlichen Zahlen). Die entsprechende Definition in KIV ist:

$$n \neq 0 \rightarrow mult(div(m,n),n) \leq m \wedge m < mult(succ(div(m,n)),n)$$

In der Äquivalenzform:

$$\begin{aligned} div(m,n) = y \leftrightarrow (\exists z_1, z_2, z_3. \quad & n \neq 0 \wedge z_1 \leq m \wedge m < z_2 \wedge z_1 = mult(y,n) \wedge \\ & z_2 = mult(succ(y),n) \vee \\ & n = 0 \wedge y = z_3) \end{aligned}$$

Das Problem ist in den Variablen z_1 und z_2 versteckt. Nichts kann garantieren, daß diese bezüglich irgendeiner Präordnung $\preceq$ durch m oder n gebunden werden, siehe Definition 11 der Kompatibilität. Eine kompatible Spezifikation von div, die eine wohlfundierte Rekursion benutzt, wäre:

$$\begin{aligned} & n \neq 0 \wedge m < n \rightarrow div(m,n) = 0 \\ & m \geq n \rightarrow div(m,n) = div(m-n,n) + 1 \end{aligned}$$

Einige der Theoreme waren mit unseren Methode nicht analysierbar ($\notin$ *UBE*). In der Spezifikation SP_1 waren 8 Theoreme von 753 nicht in der *UBE* Klasse. Allerdings, da in SP_1 4 Operationen in Alloy nicht spezifizierbar waren, führte es dazu, daß die Theoreme die diese Funktionssymbole benutzten auch nicht analysierbar waren. Damit waren zusätzlich noch 180 Theoreme von 753 nicht analysierbar.

Für die anderen Spezifikationen waren alle Operationen in Alloy spezifizierbar. In SP_2 nur 17 Theoreme von 868 waren nicht analysierbar, in SP_3 - 3 von 84 und in SP_4 - 10 von 341. In SP_5 waren alle Theoreme analysierbar. Unter den gefundenen 38 Theoremen, die nicht in *UBE* Klasse waren gab es folgende Verteilung: 21 vom Typ $\varphi \leftrightarrow \exists x.\ \psi$, 15 vom Typ $\varphi \leftrightarrow \forall x.\ \psi$, und 2 vom Typ $\exists x.\ \forall y.\ \psi$.

Es ist nicht möglich einen Spezifikationsfehler oder ein falsches Theorem in der KIV Bibliothek zu finden, weil alle Theoreme bereits bewiesen sind. Trotzdem wurden für die Evaluierungszwecke folgende Schritte durchgeführt. Zuerst wurden für jede Spezifikation 40 ihrer Theoreme ausgewählt, die insgesamt eine hohe Abdeckung des intendierten Verhaltens erreichen. Diese Theoreme sind zwar bereits bewiesen, wurden aber nochmal mit Alloy gecheckt. Es gab keine Überraschungen: es wurden keine falschen Gegenbeispiele generiert. Nach

Sorten	Bounds für Modellgröße	#-closed	#Ops	#Clauses	Zeit
nat	$\|N\| \leq 8$	-	17	3 x 10^5	9 s
	$\|N\| \leq 10$	-	17	5 x 10^5	30 s
	$\|N\| \leq 12$	-	17	1 x 10^6	2 min
list,	$\|L\| \leq 3$, $\|E\| = 2$, $\|N\| = 2$	(list,1)	43	3 x 10^4	0.5 s
elem,	$\|L\| \leq 4$, $\|E\| = 3$, $\|N\| = 2$	(list,1)	43	1 x 10^5	1 s
nat	$\|L\| \leq 7$, $\|E\| = 2$, $\|N\| = 3$	(list,2)	43	1 x 10^6	9 s
	$\|L\| \leq 13$, $\|E\| = 3$, $\|N\| = 3$	(list,2)	20	9 x 10^6	4 min
	$\|L\| \leq 15$, $\|E\| = 2$, $\|N\| = 4$	(list,3)	10	12 x 10^6	$>$ 5 min
store,	$\|S\| \leq 9$, $\|E\| = 2$, $\|D\| = 2$, $\|N\| = 3$	(store,2)	12	8 x 10^4	2 s
elem,	$\|S\| \leq 10$, $\|E\| = 3$, $\|D\| = 3$, $\|N\| = 2$	(store,1)	12	3 x 10^5	30 s
data,	$\|S\| \leq 16$, $\|E\| = 1$, $\|D\| = 15$, $\|N\| = 3$	(store,2)	12	5 x 10^5	33 s
nat	$\|S\| \leq 16$, $\|E\| = 2$, $\|D\| = 3$, $\|N\| = 3$	(store,2)	12	3 x 10^5	$>$ 5 min
set,	$\|S\| \leq 4$, $\|E\| = 2$, $\|N\| = 3$	(set,2)	16	2 x 10^4	0.3 s
elem,	$\|S\| \leq 7$, $\|E\| = 3$, $\|N\| = 3$	(set,2)	16	6 x 10^4	1 s
nat,	$\|S\| \leq 8$, $\|E\| = 3$, $\|N\| = 4$	(set,3)	16	1 x 10^5	2 s
	$\|S\| \leq 11$, $\|E\| = 4$, $\|N\| = 3$	(set,2)	16	2 x 10^5	30 s
	$\|S\| \leq 15$, $\|E\| = 4$, $\|N\| = 4$	(set,3)	16	6 x 10^5	1 min
	$\|S\| \leq 16$, $\|E\| = 4$, $\|N\| = 5$	(set,4)	16	8 x 10^5	14 min
array,	$\|A\| \leq 7$, $\|E\| = 2$, $\|N\| = 3$	(array,2)	5	5 x 10^4	2 s
elem,	$\|A\| \leq 14$, $\|E\| = 2$, $\|N\| = 4$	(array,2)	5	2 x 10^5	1 min
nat	$\|A\| \leq 17$, $\|E\| = 2$, $\|N\| = 4$	(array,3)	5	4 x 10^5	$>$ 5 min

Tabelle 4.1: Benchmark: Durchschnittlichen Zeiten für die Generierung des Modells der vorgegebenen Größe (zChaff [58] SAT Solver, 2.4 GHz Dual).

diesem Korrektheitstest wurden verschiedene Fehler und Anomalien, die in der Regel während des Designprozesses gemacht werden, in die Spezifikation eingebaut. Z.B. einfache Tippfehler, Vergessen von wichtigen Fallunterscheidungen in den Definitionen der Operationen, *OutOfBounds* Zugriffe auf die Daten usw. Alle diese Fehler wurden in Sekundenschnelle von Alloy entdeckt.

Alloy kann in zwei verschiedenen Betriebsarten benutzt werden:

- `run` φ - Suche ein Modell mit $SP_{Alloy} \wedge \varphi$
- `check` φ - finde ein Gegenmodell mit $SP_{Alloy} \wedge \neg\varphi$

Um die Dynamik des Ressourcenverbrauchs zu untersuchen wurden die betrachteten Spezifikationen mit unterschiedlichen Laufzeitparametern in Alloy analysiert. Insbesondere wurden die oberen Schranken für die Anzahl der Atome im Universum, Anzahl der definierten Operationen in der Spezifikation oder auch die Art der Abgeschlossenheit variiert. Tabelle 4.1 zeigt die Laufzeiten und die Größen der SAT-Instanzen für die jeweiligen Spezifikationen in Abhängigkeit zu diesen Parametern. Für alle betrachteten Spezifikationen war es ausreichend nur die moderaten Modegrößen zu betrachten. Z.B. für Stores hat es gereicht die Schranken für die Atome im Universum $\mathcal{M}$ auf $|S| \leq 9$, $|E| = 2$, $|D| = 2$, $|N| = 3$ zu setzen und die $\#^2_{store}$-Abgeschlossenheit (alle Stores bis zu Größe 2) zu verlangen um alle Fehler zu finden.

Die Anzahl der Atome im Universum ist ausschlaggebend für die Größe der SAT-Instanz. Ferner, ist für die Laufzeit auch die Stelligkeit der Relationen in der Signatur der Alloy Spezifikation von Bedeutung. Dies gilt sogar für relativ kleine Universen bestehend aus einigen wenigen Atomen. Z.B., für einen einsortigen Universum mit nur 4 Atomen und einer einzigen Relation, die aber 5-stellig ist, dauert die Generierung eines Modells bereits 3 Minuten. Anderseits, ist die SAT-Instanz winzig: 10^5 Klauseln und wird in 55 ms gelöst. Offensichtlich, liegt es an der Kodierungsalgorithmus, der die Relationen in die aussagenlogische Formeln und anschließen in die SAT-Instanzen übersetzt. Der Ansatz explodiert mit der Stelligkeit der betrachteten Relationen. Folglich, waren wir nicht in der Lage die Funktion $fill : array \times elem \times nat \times nat \rightarrow array$, der eine 5-stellige Relation entspricht, in Alloy zu berechnen.

4.7 Verwandte Arbeiten

Fast zu gleicher Zeit ist in an der TU München eine sehr ähnliche Arbeit von Tjark Weber [57] entstanden (2005 - 2009). Hier wurde ein Generator der endlichen Modelle für die Logik der höheren Stufe (HOL) entwickelt und in Isabelle [34] integriert. Die Theorie in HOL wird dabei in eine SAT-Instanz kodiert und an einen SAT-Solver übergeben. Falls erfüllbar, wird aus der berechneten Belegung der Variablen, die die aussagenlogische Formel erfüllt, ein endliches Modell berechnet, das die HOL Theorie erfüllt. Die rekursive Datentypen und rekursiven Funktionen werden hier nur am Rande theoretisch behandelt ohne praktischer Evaluierung. Nach einer Nachforschung haben wir festgestellt, daß die Gegenbeispielsuche in Isabelle für rekursive Datentypen und Funktionen manchmal falsche Gegenbeispiele generiert. Dieses Problem im Gegensatz zu uns hat Weber in seiner Arbeit nicht behandelt.

Neben der Arbeit von Weber existieren bereits verschiedene Modellgeneratoren: MACE [28], Kodkod [55] (wird in der neuen Version von Alloy benutzt), Paradox [8], SEM [59], FINDER [48]. Die Kombinierung von Theorembeweiser und Modellgeneratoren wurde in mehreren Fallstudien untersucht. Sie alle habe symmetrische Ziele gemeinsam: stärken der interaktiven Werkzeuge durch hinzufügen automatischer Techniken auf einer Seite, und Erweiterungen der automatischen Werkzeuge mit der formalen Beweisführung auf der anderen. Beide Beweismethoden werden als sich ergänzende Aktivitäten betrachtet. Ein Ansatz [2, 37] benutzt automatische Theorembeweiser basierend auf der Resolution um die first-order Theoreme in KIV zu analysieren. Der automatische Theorembeweiser Prover9 [30] (Nachfolger von Otter [29]) verwendet Modellgenerator MACE um die Gegenbeispiele für falsche Theoreme zu berechnen. Umgekehrt, wurde in Modellgenerator Paradox ein automatischer Theorembeweiser Equinox [7] integriert.

Die automatischen Theorembeweiser können nur eingeschränkt in KIV benutzt werden, weil sie keine Induktion unterstützen. Die Induktion wird für die Beweisführung über die Datentypen in algebraischen Spezifikationen und die rekursiven Axiomatisierungen der Operationen gebraucht. Meng [31] beschreibt eine automatische Prozedur, die higher-order Theorien nach FOL übersetzt und damit die Unterschiede zwischen Isabelle (HOL) und den automatischen Beweisern Vampire und SPASS überbrückt.

Einen ähnlichen Wer geht Pike [35] mit der Integration von SPIN Modell Checker [21] und dem Theorembeweiser PVS [33], die eine effiziente Identifikation von falschen Theoremen ermöglicht.

Steigerung der Qualität der Spezifikation spielt eine sehr wichtige Rolle in der Praxis. Die Arbeiten von Thums et al. [52, 39] und Ahrendt [1] behandeln die Fehlersuche in algebraischen Spezifikationen der abstrakten Datentypen. Ahrendt präsentiert die Modellgenerierungsprozedur die für eine Spezifikation der freien Datentypen SPmit Funktionen und ein Lemma φ ein Modell konstruiert, das $SP \wedge \neg\varphi$ erfüllt. Die Modellsuche erfolgt gesteuert durch Modellgenerierungsregeln, die aus der Signatur, Axiomen und der angezweifelten Aussage konstruiert werden.

4.8 Zusammenfassung

Die freien und die nicht-freien Datentypen bilden die Grundbausteine der algebraischen Spezifikationen in KIV. Dieser Kapitel beschreibt den Ansatz zur Spezifikation der endlichen Teilmodellen im Alloy. Ferner, wird die formale Definition der Klasse der kompatiblen Spezifikationen sowie der Klasse der analysierbaren Formeln beschrieben. Die Korrektheitsüberlegungen und die Evaluation der Methodik auf Beispielen aus der KIV Bibliothek runden es ab.

Bereits die ersten Ergebnisse aus der Anwendung der Technik haben die Gründlichkeit der Analyse deutlich gemacht: in Sekundenschnelle hat Alloy alle eingebauten Fehler entdeckt. Diese Exaktheit kommt daraus, daß schlicht alle möglichen Teilmodelle bis zu der vorgegebenen Größe k berücksichtigt werden. Damit war Alloy viel effizienter als der Mensch, wurde nicht müde und seine Zeit war auch viel günstiger.

Kapitel 5

Fallstudie: Listen von Intervallen

Im letzten Kapitel wurde der Grundansatz zur automatischen Analyse der algebraischen Spezifikationen mit Alloy vorgestellt. Die qualitativen und die quantitativen Eigenschaften der Technik wurden auf Beispielen aus der KIV Bibliothek verdeutlicht.

Als nächstes ist es wichtig die praktische Relevanz der Technik zu untersuchen, die potentiellen Schwächen aufzudecken und eventuell zu beheben. In diesem und im nächsten Kapiteln werden zwei realistischen aber trotzdem überschaubaren KIV Fallstudien präsentiert, die für diese Zwecke verwendet wurden. Anschließend, werden die erzielten Ergebnisse besprochen und ein Vergleich zu der bisher benutzten Technik zur Fehlersuche gemacht.

5.1 Fehlersuche mit Alloy

Eine Beschreibung dieser Fallstudie wurde bereits im Kapitel 3 gemacht. Nun wenden wir uns konkret den Einzelheiten der Anwendung des Ansatzes zur Fehlersuche zu. Die ursprüngliche KIV Spezifikation enthielt einen Fehler. Die rekursive Definition der *insert* Operation hatte eine unvollständige Fallunterscheidung über den betrachteten Intervall. Es wurde ein Fall vergessen: sollten durch das Vergrößern eines Intervalls zwei zusammenhängende Intervalle in der Liste entstehen, dann müssen die beiden zu einem neuen Intervall verschmelzen. Dieser Defekt führte zur Verletzung der Invariante durch die Operation *insert*: nach dem Einfügen einer Zahl kann eventuell eine nicht wohlgeformte Liste von Intervallen entstehen.

Dieser Fehler entpuppt sich bereits bei relativ kleinen Gegenbeispielen. Am Anfang ist die minimal notwendige Schranke für die Anzahl der Atome im Modell unbekannt. Daher fängt man zuerst mit kleineren Modellen an und vergrößert sukzessiv die Suchreichweite, siehe die Abbildung 5.1. Erst ab der Schrankengröße $k = 4$ kann das Gegenbeispiel Modell in 2 Sekunden generiert werden. Die Gesamtzeit für die Suche beträgt also 4 Sekunden.

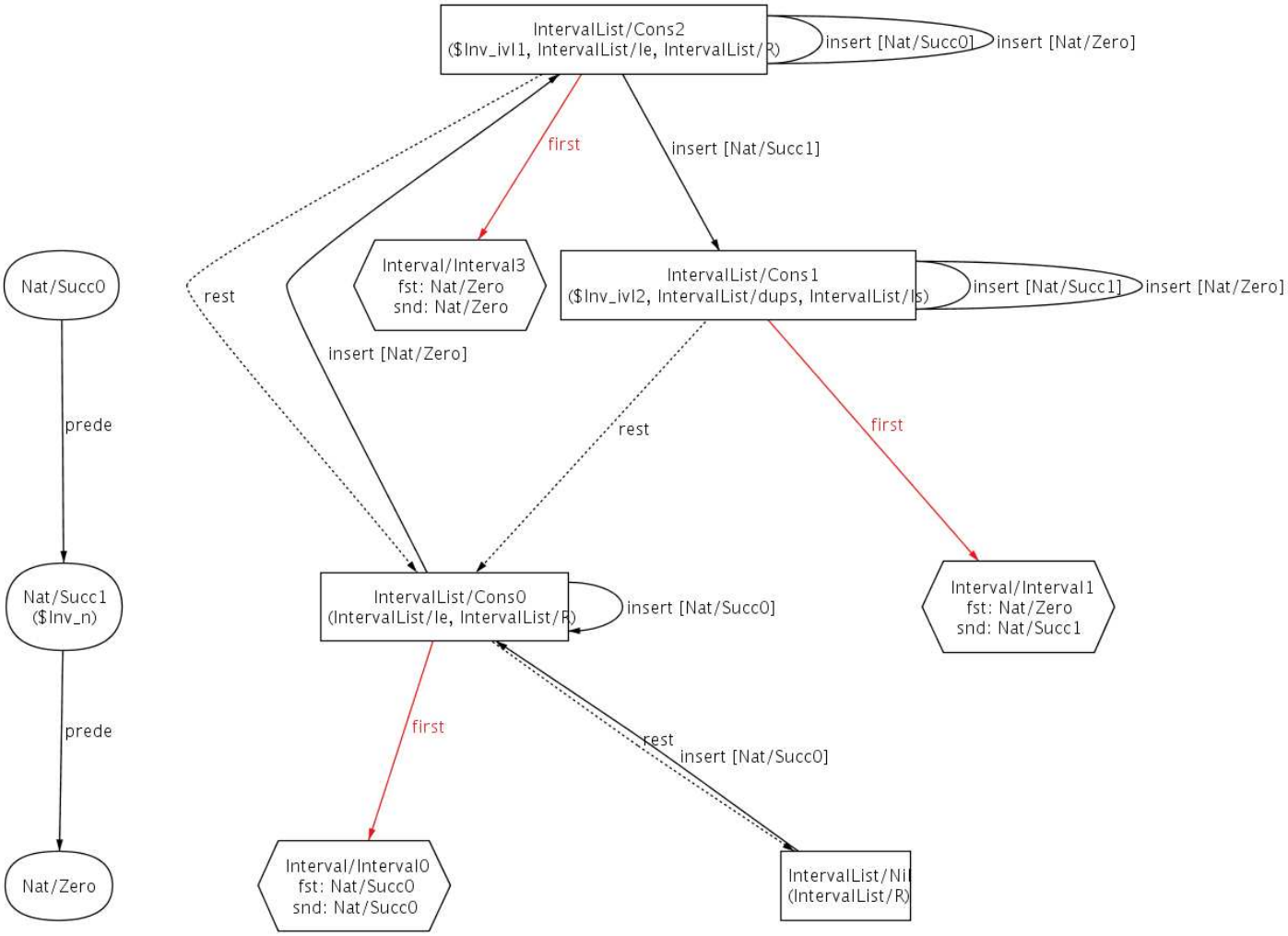


Abbildung 5.1: Mit Alloy gefundenes und generiertes Gegenbeispiel Modell.

scope (Modellgröße)	Gegenbeispiel	clauses	Zeit
1	nein	2000	0.1 s
2	nein	5000	0.3 s
3	nein	13252	0.7 s
4	ja	83302	2 s

Tabelle 5.1: Benchmark: Listen von Intervallen (Berkmin SAT Solver, 2.4 GHz Dual Core)

Die Invariante wird durch *insert* verletzt, weil die Verschmelzung zweier Intervalle nicht durchgeführt wird. Alloy generiert das kleinste Gegenbeispiel (ein endliches Termmodell), für das die Formel ψ_{INV} falsch ist. ψ_{INV} drückt die Erhaltung der Invariante R durch die Operation *insert* aus:

$$\psi_{INV} \equiv \forall\, ivl_1, ivl_2 \in intervallist,\ n \in nat.\\ R(ivl_1)\ \wedge\ ivl_2 = insert(ivl_1, n) \rightarrow R(ivl_2)$$

Die kleinste Liste, wo der Fehler auftritt, ist eine wohlgeformte Liste mit genau zwei Intervallen:

$$[(m_1, n_1), (m_2, n_2)],\quad wobei\ \ n_1 = m_2 - 2$$

Der Fehler kommt durch das Einfügen der Zahl $n_1 + 1$ zu Stande. Die Abbildung 5.1 zeigt das mit Alloy berechnetes Gegenbeispiel. Alloy berechnet auch eine passende Belegung der Variablen ivl_1, ivl_2, n, die die analysierte Formel falsifiziert. Mit einer Aufschrift $\$varname$ sind die entsprechenden Atome im Teilmodell markiert, siehe Aufschriften $\$ivl1$ (Cons2), $\$ivl2$ (Cons1) und $\$n$ (Succ1).

Die Identifikation der Werte der Atome $Cons1$, $Cons2$ und $Succ1$ erfolgt anhand der Selektorrelationen:

- $Cons2.rest = Cons0$, $Cons0.rest = Nil$
- $Cons2.first = Interval3$, $Cons0.first = Interval0$
- $Interval3.fst = Zero$, $Interval3.snd = Zero$
- $Interval0.fst = Succ0$, $Interval0.fst = Succ0$
- $Succ1.prede = Zero$, $Succ0.prede = Succ1$

Daraus folgern wir:

- $n : Succ1 = 1$
- $ivl1 : Cons2 = [(0,0),(2,2)]$
- $ivl2 : Cons1 = [(0,1),(2,2)]$

Die Liste ivl_2 ist das Ergebnis der Anwendung von *insert* auf ivl_1. Gleichzeitig ist ivl_2 nicht wohlgeformt: $\neg R(ivl_2)$, was zur Verletzung der Invariante führt. Die Entdeckung der Fehlerursache geschieht durch die sorgfältige Analyse der Operationsdefinition für den konkreten Fall, der von Alloy aufgedeckt wurde.

Der gleiche Fehler wird automatisch mit Kodkod in KIV gefunden. Dabei werden die Konstruktorterme im Gegenbeispiel zusammengebaut und gut lesbar angezeigt. Falls es sich um die Konstanten handelt, werden diese an Stelle von Konstruktortermen angezeigt.

```
counterexample for INV:

variables assignment:
n=1
ivl2=(0 x 1) + (2 x 2) + []
ivl1=(0 x 0) + (2 x 2) + []

MODEL:

function table for insert : ivlist x nat -> ivlist
--------------------------------------------------------------------------------
(0 x 0) + (2 x 2) + []  1  (0 x 1) + (2 x 2) + []
...
================================================================================

function table for R : ivlist -> bool
--------------------------------------------------------------------------------
(0 x 0) + (2 x 2) + []
(2 x 2)
[]
================================================================================

statistics
p cnf 75087 236872
primary variables: 764
translation time: 5435 ms
solving time: 885 ms
```

Die Fehlerursache ist ein vergessener Randfall, der zu einer Liste mit zusammenhängenden Intervallen führt. Die korrigierte Version der Axiomatisierung von *insert* enthält eine explizite Behandlung von diesem Fall:

(1) $insert([\,], k) = [(k, k)]$

(2) $k < m \wedge k \neq m - 1 \rightarrow insert((m, n) + x, k) = (k, k) + (m, n) + x$

(3) $k < m \wedge k = m - 1 \rightarrow insert((m, n) + x, k) = (k, n) + x$

(4) $k \geq m \wedge k \leq n \rightarrow insert((m, n) + x, k) = (m, n) + x$

(5 a) $k \geq m \wedge k = n + 1 \wedge x = [] \rightarrow insert((m, n) + x, k) = (m, k) + x$

(5 b) $k \geq m \wedge k = n + 1 \wedge x = (m_1, n_1) + y \wedge m_1 = k + 1 \rightarrow$
$insert((m, n) + x, k) = (m, n_1) + y$

(5 c) $k \geq m \wedge k = n + 1 \wedge x = (m_1, n_1) + y \wedge m_1 \neq k + 1 \rightarrow$
$insert((m, n) + x, k) = (m, k) + x$

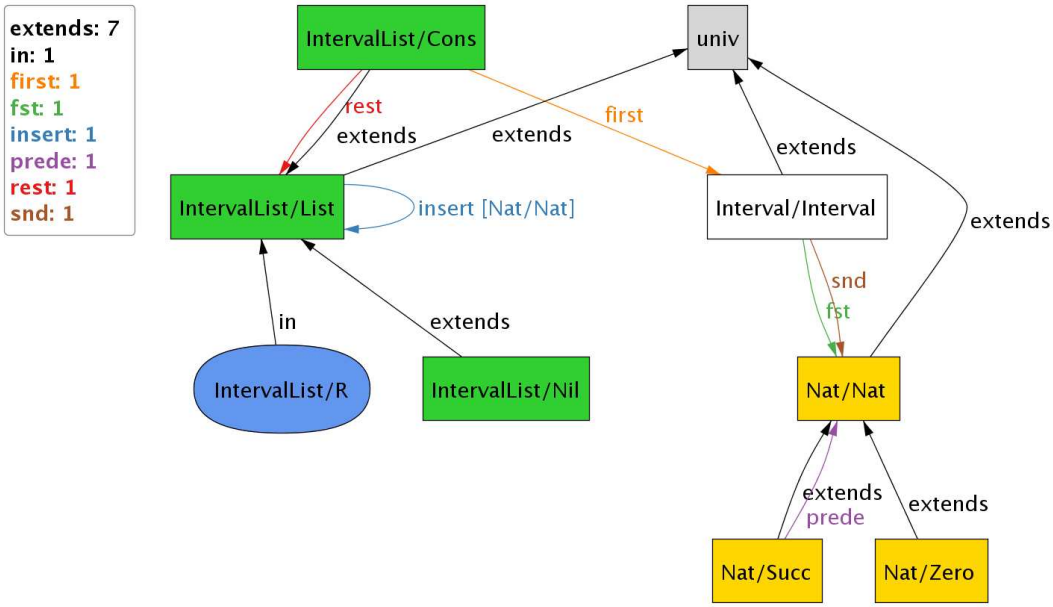


Abbildung 5.2: Das Metamodell zur Alloy Spezifikation der Listen von Intervallen.

(6) $k \geq m \wedge k > n + 1 \rightarrow insert((m, n) + x, k) = (m, n) + insert(x, k)$

Das Axiom (5b) behandelt genau diesen Fall: hier verschmelzen die beiden zusammenhängenden Intervalle zu einem.

5.2 Alloy Spezifikation

Die Erstellung der Alloy Spezifikation ist nicht aufwendig. Die verwendeten Datentypen *List*, *Pair* (Intervalle) und *Nat* sind bereits in der KIV Bibliothek spezifiziert und damit auch bereits nach Alloy portiert. Die Alloy Module *nat, interval* und *list* werden importiert, wobei *list* mit Intervallen aktualisiert wird:

```
module intervallist

open nat as Nat
open interval as Interval
open list[Interval] as IntervalList

fact insert_def {...}
fact R_def {...}
```

Die Abbildung 5.2 zeigt das Metamodell der Alloy Spezifikation. Die Operation *insert* ist als eine Relation $List \times Nat \times List$ darauf zu sehen. Das 1-stellige Prädikat R für die Wohlgeformtheit ist als eine Untermenge der Sorte *Liste* spezifiziert. Diese beiden Operationen müssen in die Signatur von Listen eingetragen werden (im Alloy Modul *list*):

```
module list[Element]
...
sig List {
...
   insert: Nat -> lone List,
...
}
...
sig R in List
```

Die KIV Definitionen der beiden Operationen werden als Alloy *facts* in die Spezifikation eingefügt. Die relationale Form der Definition von R ist:

```
fact R_def {
   R = { ivl: List |
      ivl = Nil or
      (not ivl = Nil and
       lte[ivl.first.fst,ivl.first.snd] and
       (ivl.rest = Nil or
        not ivl.rest = Nil and
        lt[ivl.first.snd,ivl.rest.first.fst.prede] and
        ivl.rest in R
       )
      )
   }
}
```

Das Theorem über die Invariantenerhaltung wird mit τ in die relationale Form gebracht und als *assert* in Alloy Modul eingetragen. Die Alloy *checks* überprüfen das Theorem für die vorgegebenen Schranken:

```
assert Inv {
   all ivl,ivl': List, n: Nat |
      ivl in R and (ivl->n->ivl') in insert => ivl' in R
}

check Inv for 1
check Inv for 2
check Inv for 3
check Inv for 4
```

Für eine sichere Anwendung dieses Ansatzes auf diese Fallstudie muss die Frage über die Kompatibilität der Spezifikation geklärt werden. Hierfür werden folgende Schritte gemacht:

1. Wahl der passenden Abgeschlossenheit der endlichen Teilmodelle ($\preceq$: $dom(\mathcal{M}) \times dom(\mathcal{M})$)
2. Überprüfung der Kompatibilität der Spezifikation bezüglich $\preceq$

Sollte der zweite Schritt fehlschlagen, muss der erste Schritt wiederholt werden. In unserem Fall reduziert sich die erste Frage zu der Wahl zwischen der Subtermabgeschlossenheit oder #-Abgeschlossenheit der Listen. Es wird die erste Variante genommen, die auch weniger restriktiv ist, d.h. mehr endlichen Teilmodelle sind zulässig.

Die Kompatibilitätsüberprüfung reduziert sich zu der Überprüfung der beiden Definitionen bezüglich der Subtermabgeschlossenheit. Ohne weitere Komplikationen fallen beide Tests positiv aus. Die Definition von R ist trivialerweise kompatibel, weil sie nur Selektoren verwendet. Bei der Definition von *insert* ist es aber nicht sofort ersichtlich.

Betrachten wir die Kompatibilitätsüberprüfung der Definition von *insert* als Beispiel. Die Definition ist zwar groß dafür aber strukturell einfach:

$$
\begin{aligned}
\forall x, y : List,\ k : Nat \qquad & insert(x, n, y) \leftrightarrow \\
& \exists \underline{z}.\ \varphi_1 \wedge x = [] \wedge y = [(k,k)] \vee x = (m,n) + x_0 \wedge (\\
& \qquad \varphi_2 \wedge y = (k,k) + (m,n) + x_0 \vee \\
& \qquad \varphi_3 \wedge y = (k,n) + x_0 \vee \\
& \qquad \varphi_4 \wedge y = (m,n) + x_0 \vee \\
& \qquad \varphi_{5a} \wedge y = (m,k) + x_0 \vee \\
& \qquad \varphi_{5b} \wedge y = (m,n_1) + x_0 \vee \\
& \qquad \varphi_{5c} \wedge y = (m,k) + x_0 \vee \\
& \qquad \varphi_6 \wedge y = (m,n) + insert(x_0,k))
\end{aligned}
$$

In $\underline{z}$ sind die Zwischenergebnisse gespeichert. Die Formeln φ_i repräsentieren die Vorbedingungen der einzelnen Fälle in der Axiomatisierung von *insert*. Z.B. spezifiziert φ_{5b} den Fall, wo durch das Einfügen der Zahl k zwei zusammenhängende Intervalle entstehen, die dann zu einem Intervall (m, n_1) verschmelzen sollen.

$$\varphi_{5b} \equiv k \geq m \wedge k = n + 1 \wedge x_0 = (m_1, n_1) + x_1 \wedge m_1 = n + 1$$

Es handelt sich um eine existentielle Definition der Form:

$$\forall \underline{x}, y.\ F(\underline{x}, y) \leftrightarrow \exists \underline{z}. \bigvee_{i\ \in\{1,\ldots,6\}} \bigwedge_{j} \tilde{A}_{i,j}$$

Wir verwenden die Heuristik aus dem Abschnitt 4.3.5 für einen effizienten Test. In jeder der Konjunktionen müssen die Zwischenergebnisse $\underline{z}$ durch $\underline{x}$ oder y "gebunden sein." Betrachten wir den Fall von $i = 5b$. Die Formel φ_{5b} wird jetzt zu der relationalen Form normalisiert. Zusammengefasst, sieht der Fall (5b) in der relationalen Form wie folgt:

$$
\begin{aligned}
& z_1 = n + 1 \wedge z_2 = (m_1, n_1) \wedge z_3 = z_2 + x_1 \wedge z_4 = (m,n) \wedge z_5 = z_4 + x_0 \wedge \\
& z_5 = (m, n_1) \wedge \\
& x = z_5 \wedge k \geq m \wedge k = z_1 \wedge x_0 = z_3 \wedge m_1 = z_1 \wedge \\
& y = z_5 + x_0
\end{aligned}
$$

Nun kann der Graph, der zu dieser Konjunktion gehört, mit der Heuristik konstruiert werden. Dann wird die Erreichbarkeit der Knoten $\{\underline{z}, x_0, x_1, m, n, m_1, n_1\}$

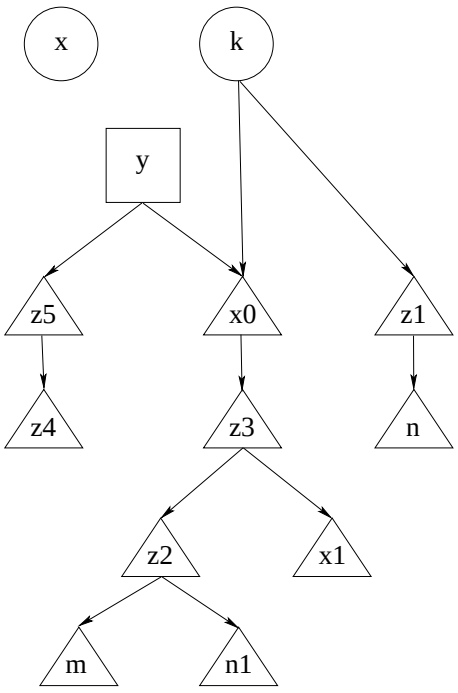


Abbildung 5.3: Abhängigkeitsgraph für die Konjunktion aus dem Fall (5b) der Axiomatisierung von *insert*.

aus der Menge der Knoten $\{x, y, k\}$ überprüft. Die Abbildung 5.3 zeigt den Graphen. Offensichtlich sind alle Knoten von der Knotenmenge $\{x, y, k\}$ erreichbar, und damit ist der Test positiv.

5.3 Vergleich zu dem bisherigen Gegenbeispiel-Ansatz

Das gleiche Beispiel wurde bereits früher von Thums et al. [40] mit einem anderen Ansatz zur Fehlersuche ausprobiert. Die Vorgehensweise ist ungefähr wie folgt: zuerst wird ein gewöhnlicher Beweis für das Theorem gestartet. Durch die Anwendung der vorher definierten Heuristiken wird in 4 Sekunden ein Beweisbaum aufgebaut. Da das Theorem falsch ist, wird man irgendwann auf ein offenes *Goal* (Knoten im Baum) stossen. Der Benutzer muss dabei die Sequenz betrachten und entscheiden ob sie beweisbar ist. Sollte er bemerken, daß die Sequenz nicht beweisbar ist ($true \vdash false$), kann er das System auffordern ein Gegenbeispiel zu konstruieren. An dieser Stelle kommt die Technik zum Einsatz.

Die Grundidee ist, die quantifizierten Variablen systematisch (mit Heuristiken) zu instanzieren in der Hoffnung einen Volltreffer zu erzielen. Die Heuristiken wenden automatisch *Rewrite* Beweisregeln an. Die Variablen im Theorem werden mit unterschiedlichen Konstruktortermen belegt, die systematisch durch

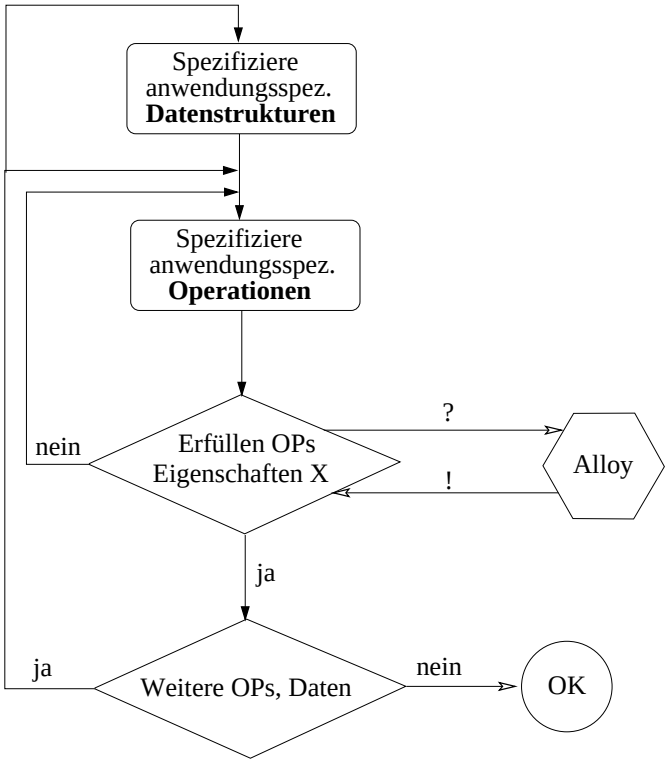


Abbildung 5.4: Ein einfaches Anwendungsszenario für Alloy bei der Entwicklung einer Spezifikation.

Fallunterscheidungen[1] und *Simplifier* Regeln konstruiert werden. Die Suchstrategie ist die heuristische Breitensuche. Ähnlich wie bei der A*-Suche werden vom Benutzer Heuristiken definiert, die die bestimmte Knoten im Beweisbaum bevorzugen.

Für den Beispiel von Intervall Listen hat KIV 2 Sekunden für die Suche gebraucht. Eine erfolgreiche Anwendung der Technik hängt kritischerweise von den vorher definierten Heuristiken für die Zustandsraumexploration ab. Damit sind die 2 Sekunden, die Alloy für die Berechnung gebraucht hat, eine klare Verbesserung im Vergleich zu 6 Sekunden + Zeit für die Untersuchung des Goals durch den Benutzer mit der alten Technik. Im Abschnitt 6.4 wird anhand eines Beispiels ein weiterer Vergleich gemacht.

5.4 Fazit

Die Abbildung 5.3 fasst das grundlegende Anwendungsszenario der Technik zur Fehlersuche bei der Entwicklung einer Spezifikation zusammen. Alloy wird hier ähnlich eingesetzt, wie die zahlreichen automatischen Programverifizierer, die mittlerweile in die Entwicklungstools für die meisten Programmiersprachen in-

[1] *constructor cuts*: z.B. $x = nil \vee x = cons(a, nil)$ für Listen.

tegriert sind. Ähnlich wie beim Programmieren werden während des Entwurfs und der Spezifikation eines komplexen Systems laufend Fehler gemacht. Die Fehler sind dann nur schwer, wenn überhaupt, durch das Testen zu finden.

Während der Entwicklung der algebraischen Spezifikationen werden Entwurfsfehler gemacht. Meistens liegt es an der Komplexität der verwendeten Datenstrukturen sowie der Ausdrucksstärke der Logik der ersten Stufe.

Bis jetzt wurden die Defekte in der Spezifikation während des Beweisens der Theoreme, die die Korrektheit und die Funktionalität des Systementwurfs ausdrücken, entdeckt. Auf diese Weise Fehler aufzudecken ist zwar qualitativ der beste Weg, gleichzeitig aber sehr aufwendig.

Am Anfang der Entwicklung, scheitern die Beweise andauernd, da einerseits die Spezifikation Fehler enthält und anderseits die Theoreme öfters falsch formuliert werden. Offensichtlich, wäre es ein Fortschritt die falschen Theoreme bzw. die von dem System nicht erfüllten Eigenschaften automatisch zu lokalisieren.

Im Appendix B.1 ist eine ausführliche Liste der Ergebnisse für die Fallstudie enthalten.

Kapitel 6

Fallstudie: Nicht-blockierende Implementierung der Mengen

Dieses Kapitel berichtet über die Erfahrungen aus der Anwendung der Fehlersuche mit Alloy auf eine weitere Fallstudie. Hier werden bereits komplexere Datenstrukturen und darauf definierten Operationen verwendet. Wir betrachten die verwendeten Datenstrukturen aus einer nicht-blockierenden (*lock-free*) parallelen Implementierung der Mengen von Objekten. Die Spezifikation stammt aus den Arbeiten von Harris [18] und Heller et al. [20].

6.1 Beschreibung

Diese Fallstudie wurde in KIV in Rahmen einer Forschungsarbeit über die nebenläufigen Algorithmen gemacht [24, 9]. Dabei wurde unter anderem die *Linearisierbarkeit* und somit auch die Korrektheit des Algorithmus nachgewissen. Die nicht-blockierende Algorithmen haben sich schon in der Praxis als Alternative zu den blockierenden Algorithmen etabliert. Sie haben die folgende Eigenschaft. Immer wenn die Methodenaufrufe durch mehrere Prozesse stattfinden und zumindest einer der Prozesse die Ausführungsrechte besitzt, dann beendet immer zumindest ein Prozess erfolgreich seinen Aufruf. Diese Eigenschaft garantiert, daß das System als Ganzes immer einen Fortschritt macht. Anderseits gibt es keine Fortschrittsgarantie für jeden einzelnen Prozess. *Wait*-Freiheit (*wait-free*) ist eine stärkere Eigenschaft, die zusichert, daß immer wenn ein Prozess fortschreitet wird er auch irgendwann fertig.

Die *abstrakte* Spezifikation beschreibt eine Menge von Objekten und die darauf-definierten abstrakten Operationen:

- *add(x)* - Objekt x wird in die Menge eingefügt,

- *remove(x)* - Entfernen eines Objektes,
- *contains(x)* - ein Test ob x in der Menge enthalten ist

Damit es für einen nebenläufigen nicht-blockierenden Algorithmus verwendet werden kann, wird diese Menge durch eine komplexere verlinkte Datenstruktur realisiert. Diese Struktur wird auf einer Halde abgespeichert.

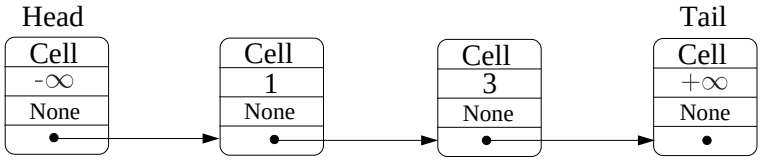


Abbildung 6.1: Menge von zwei Objekten als verlinkte Liste.

Die Objekte werden mit dem freien Datentyp *Cell* repräsentiert. Dieser enthält auch die zusätzlichen implementierungsrelevanten Datenfelder:

$$\begin{aligned}\texttt{Cell} = \texttt{mkcell}(&\\ &\texttt{i}_\infty : \texttt{IntegerInf},\\ &\texttt{pin} : \texttt{PIDorNone},\\ &\texttt{nxt} : \texttt{Ref}\\ )&\end{aligned}$$

Das Datenfeld i_∞ ordnet einer Zelle eine Ganzzahl zu. Damit werden die Zellen als eine geordnete Menge betrachtet. Es gibt ausgezeichneten Zellen *Head* und *Tail*, denen die Zahlen $-\infty$ und $+\infty$ zugeordnet sind. Die Ganzzahlen werden mit dem freien Datentyp *IntegerInf* spezifiziert. Dieser enthält auch die beiden Unendlichkeiten:

$$\texttt{IntegerInf} = \lceil\ .\ \rceil\ (\texttt{i} : \texttt{Integer}) \mid\ -\infty \mid\ +\infty$$

Der nicht-freie Datentyp *Integer* spezifiziert die eigentlichen Ganzzahlen:

$$\begin{aligned}&\texttt{Integer} = \texttt{generated by}\ 0,\ +1, -1\\ &\texttt{succpred} : i\ +1\ -1 = i\\ &\texttt{predsucc} : i\ -1\ +1 = i\end{aligned}$$

Der Datenfeld *pin* speichert Id des Prozesses, der diese Zelle gerade sperrt:

$$\texttt{PIDorNone} = \lceil\ .\ \rceil\ (\texttt{pid} : \texttt{PID}) \mid \texttt{None}$$

Mit *nxt* wird die Referenz auf die nächste Zelle gespeichert. Die Abbildung von Referenzen auf die Zellen wird auf einer Halde abgespeichert. Auf der Abbildung 6.1 ist die verlinkte Liste von Zellen gezeigt, die der Menge $\{1, 3\}$ entspricht.

Die Halde wird als eine Instantiierung von nicht-freiem Datentyp *Store* mit *Ref* (für *Elem*) und *Cell* (für *Data*) spezifiziert:

$$\texttt{Heap} = \texttt{Store}[\texttt{Ref}, \texttt{Cell}]$$

Datentyp *RefList* spezifiziert die Listen von Referenzen. Es ist als eine Instantiierung von *List* mit *Ref* spezifiziert:

$$\texttt{RefList} = \texttt{List}[\texttt{Ref}]$$

Die Menge von Ganzzahlen wird als eine Instantiierung vom Datentyp *Set* mit *IntegerInf* spezifiziert:

$$\texttt{IntSet} = \texttt{Set}[\texttt{IntegerInf}]$$

Insgesamt ergibt sich für die KIV Spezifikation eine Hierarchie von Datentypen, die auf der Abbildung 6.2 zu sehen ist.

Die Halde, sowie die darin gespeicherten Listen von Referenzen, müssen gewisse Eigenschaften (*Darstellungsinvariante*) erfüllen, die die Korrektheit des darauf arbeitenden nebenläufigen Algorithmus sichern. Die Invariante wird mittels des Prädikats $SetAbs : Ref \times Heap \times IntSet$ formalisiert. $SetAbs(Head, H, S)$ gilt genau dann wenn die Halde H eine *korrekte* verlinkte Liste von Zellen speichert. Der Inhalt der Halde repräsentiert die Menge S der Ganzzahlen.

$$\begin{aligned}
\texttt{SetAbs} - \texttt{def}: \quad & SetAbs(Head, H, S) \\
\leftrightarrow \quad & Head \in H \wedge Head \neq Null \wedge H[Head].nxt \neq Null \\
& \wedge\ H[Head].i_\infty = -\infty\ \wedge\ SetAbsRec(-\infty, H[Head].nxt, H, S)
\end{aligned}$$

Das Hilfsprädikat $SetAbsRec : IntegerInf \times Ref \times Heap \times IntegerSet$ spezifiziert den rekursiven Anteil der Invariante:

$$\begin{aligned}
\texttt{SetAbsRec} - \texttt{base}: \quad & SetAbsRec(i_\infty, r, H, \emptyset) \\
\leftrightarrow \quad & r \in H \wedge r \neq Null \wedge H[r].nxt = Null \wedge H[r].i_\infty = \infty \\
\texttt{SetAbsRec} - \texttt{rec}: \quad & S \neq \emptyset \rightarrow (SetAbsRec(i_\infty, r, H, S) \\
\leftrightarrow \quad & r \in H \wedge r \neq Null \wedge i_\infty < H[r].i_\infty \wedge H[r].i_\infty \neq \infty \\
& \wedge\ H[r].i_\infty.i \in S \wedge H[r].nxt \neq Null \\
& \wedge\ SetAbsRec(H[r].i_\infty, H[r].nxt, H, S - - H[r].i_\infty.i))
\end{aligned}$$

Mit dem Prädikat $path : RefList \times Heap$ wird überprüft, ob eine Liste von Referenzen in der Halde als verkettete Liste von Zellen gespeichert ist:

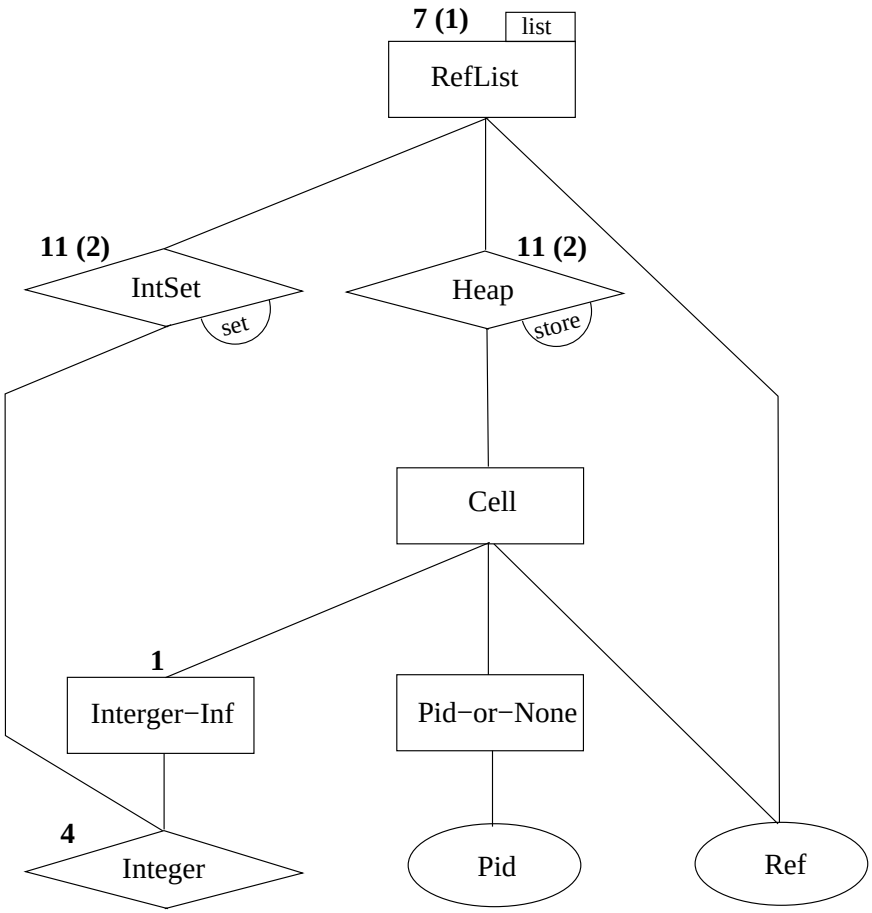


Abbildung 6.2: Hierarchie der verwendeten Datentypen.

$$
\begin{aligned}
\texttt{path}-\texttt{empty}: &\quad \neg\, pat([\,], H)\\
\texttt{path}-\texttt{one}: &\quad path(r + [], H) \leftrightarrow (r \in H \wedge r \neq Null)\\
\texttt{path}-\texttt{two}: &\quad path(r + r_0 + [], H)\\
\leftrightarrow &\quad r \in H \wedge rinH \wedge r \neq Null \wedge r_0 \neq Null \wedge H[r].nxt = r_0\\
\texttt{path}-\texttt{cons}: &\quad path(r + r_0 + x, H)\\
\leftrightarrow &\quad r \in H \wedge r \neq Null \wedge H[r].nxt = r_0 \wedge pat(r_0 + x, H)
\end{aligned}
$$

Darauf aufbauend wird ein weiteres Prädikat definiert: $reachable : Ref \times Ref \times Heap$. Es formalisiert die Erreichbarkeit zwischen Referenzen in einer Halde:

$$
\begin{aligned}
\texttt{reachable}-\texttt{def}: &\quad reachable(r, r_0, H)\\
\leftrightarrow &\quad (\exists x.\ path(r + x, H) \wedge (r + x).last = r_0)
\end{aligned}
$$

6.2 Alloy Spezifikation

Der Aufwand die KIV Spezifikation nach Alloy zu portieren reduziert sich im wesentlichen auf die Überprüfungen der Kompatibilität der vorhandenen Operationssymbolen. Dabei wurde fesgestellt, daß die Definition der Operation *reachable* bezüglich der verwendeten Abgeschlossenheiten nicht kompatibel war. Laut Heuristik zur Kompatibilitätsüberprüfung aus Abschnitt 4.3.5 sind in der Definition von *reachable* weder die Variable x noch $r{+}x$ durch eine der Variablen r, r_0, H $\preceq$-gebunden. In diesem Fall handelt es sich bei $\preceq$ um die Subtermrelation. Es ist einfacher die Operation *reachable* neu zu definieren, als nach einer passenden Ordnung $\preceq$ zu suchen, siehe unten.

Alle verwendeten komplexeren Datentypen waren bereits in der KIV Bibliothek definiert. Anwendungsspezifisch waren nur die einfachen *Container*-Datentypen wie *PID-or-None*, *Cell*. Diese Datentypen sind frei generiert und deren Übersetzung nach Alloy automatisch abläuft.

Die Abb. 6.2 zeigt neben der Hierarchie der Datentypen die Anzahl der definierten Operationen in der jeweiligen Spezifikation. In Klammern steht die Anzahl der anwendungsspezifischen Operationen, die noch nicht in KIV Bibliothek spezifiziert sind. Insgesamt sind es die folgenden Operationen:

$$
\begin{array}{rcl}
\texttt{IntSet}: & \textbf{SetAbs} & : Ref \times Heap \times IntSet, \\
 & \textbf{SetAbsRec} & : IntegerInf \times Ref \times Heap \times IntegerSet \\
\texttt{Heap}: & \textbf{Reachable} & : Ref \times Ref \times Heap \\
\texttt{Reflist}: & \textbf{Path} & : RefList \times Heap
\end{array}
$$

Der Hauptaufwand bei der Übersetzung nach Alloy war die Kompatibilität der Definitionen dieser Operationen zu überprüfen. Mit Ausnahme der Operation *reachable* sind alle anderen Definitionen bezüglich der Subtermabgeschlossenheit kompatibel.

Die Definition von *reachable* wird mittels des Prädikats *path:* $RefList \times Heap$ gemacht. Die Referenz r_0 ist von der Referenz r in der Halde H erreichbar genau dann wenn es in H einen Pfad gibt, der mit r anfängt und r_0 aufhört.

Damit diese Definition auf den endlichen Modellen korrekt funktioniert, muss Folgendes gelten. Für jeden in H existierenden Pfad $r_1 \rightarrow \ldots \rightarrow r_n$ muss im endlichen Teilmodell eine entsprechende Liste vorhanden sein. Ansonsten liefert das Prädikat *reachable false* zurück, weil es keine für die semantische Auswertung der Definition notwendige Liste im endlichen Modell findet. Die Abgeschlossenheit der endlichen Modelle gegen die Subtermrelation garantiert dies nicht. Damit ist die Definition von *reachable* nicht kompatibel.

Wir definieren eine Hilfsrelation *Step*, die einen Schritt in der iterativen Pfadbildung bei der Erreichbarkeit spezifiziert:

$$
\begin{array}{rcl}
\texttt{step}-\texttt{def}: & Step(H, r, r_0) & \\
 \leftrightarrow & r \neq Null \wedge r_0 \neq Null \wedge r \in H \wedge r_0 \in H \wedge r_0 = H[r].nxt &
\end{array}
$$

Da der transitive Abschluss in der Logik der ersten Stufe nicht ausdrückbar ist, wird ein in Alloy vordefinierter Operator dafür verwendet. Wenn R eine binäre Relation ist, dann ist $^\wedge R$ der transitive Abschluss und $^* R$ die reflexiv-transitive Hülle von R.

$$\begin{aligned} \texttt{reachable} - \texttt{def} - \texttt{new}: \quad & reachable(r, r_0, H) \\ \leftrightarrow \quad & r \neq Null \wedge r \in H \wedge r_0 \in r.*(H.Step) \end{aligned}$$

6.3 Fehlersuche

Bei der formalen Analyse der Eigenschaften der *Linearisierbarkeit* [24, 9] wurden zahlreichen Lemmas formuliert. Sie wurden in Beweisen der zentralen Theoremen als Heuristiken verwendet. Wir haben diese Lemmas der automatischen Überprüfung mit Kodkod unterzogen. Im folgenden beschreiben wir drei instruktive Beispiele für die falschen Lemmas, die mit Kodkod aufgedeckten wurden. Eine Vollständige Liste der Ergebnisse der Fehlersuche mit Kodkod in KIV angewandt auf diese Fallstudie ist im Anhang B.2 zu finden.

6.3.1 Lemma `reachable-next`

Das Lemma `reachable-next` beschreibt die folgende Eigenschaft von *reachable*. Wenn die Referenz r_1 von r aus in der Halde H errechbar ist, dann ist auch die Referenz $H[r_1].nxt$ in errechbar solange $H[r_1].nxt \in H$ gilt:

$$\begin{aligned} \texttt{reachable} - \texttt{next}: \quad & reachable(r, r_1, H) \rightarrow \\ & (reachable(r, H[r_1].nxt, H) \leftrightarrow H[r_1].nxt \in H) \end{aligned}$$

Das Lemma ist falsch, weil der Fall des Nullpointers vergessen wurde: $H[r_1].nxt = null$. Kodkod berechnet folgenden Gegenbeispiel[1]:

```
counterexample for reachable-next:

variables assignment:
r=Ref2
r1=Ref2
H={(null,mkcell(∞,⌈ PID2 ⌉,Ref2)),
   (Ref2,mkcell(-∞,⌈ PID1 ⌉,null))}

MODEL:

function table for reachable : Ref x Ref x heap -> bool
--------------------------------------------------------------------------------
Ref2   Ref2   {(null,mkcell(∞,⌈ PID2 ⌉,Ref2)),
```

[1] Für bessere Lesbarkeit wird Halde als Menge von Tupeln (address,data) gezeigt: $\{(a_0, d_0), (a_1, d_1), (a_2, d_2)\}$

```
               (Ref2,mkcell(-∞,⌈ PID1 ⌉,null))}
================================================================================
```

```
statistics
p cnf 15249 39606
primary variables: 882
translation time: 3358 ms
solving time: 40 ms
```

Die korrigierte Version des Lemmas enthält die Zusatzbedingung $H[r_1].nxt \neq null$ in der Äquivalenz:

$$\texttt{reachable} - \texttt{next}: \quad reachable(r, r_1, H) \rightarrow (reachable(r, H[r_1].nxt, H) \leftrightarrow \mathbf{H[r_1].nxt} \neq \mathbf{null} \wedge H[r_1].nxt \in H)$$

6.3.2 Lemma `reachable-addnew`

Das folgende Lemma behauptet, daß das Hinzufügen einer Zelle die Erreichbarkeit nicht beeinflusst. Referenz r_1 ist von r aus in der modifizierten Halde $H[r_2, ce]$ erreichbar, wobei r_2 eine in H nicht allokierte Adresse ist. Es wird erwartet, dass die Erreichbarkeit auch in der ursprünglichen Halde H gilt:

$$\texttt{reachable} - \texttt{addnew}: \quad reachable(r, r_1, H[r_2, ce]) \wedge r_1 \in H \wedge r_2 \notin H \rightarrow reachable(r, r_1, H)$$

Das Lemma ist falsch, weil der Fall nicht beachtet wird, wo die allokierte Zelle ce für die Erreichbarkeit eventuell von Bedeutung ist. Kodkod findet ein Gegenbeispiel:

```
counterexample for reachable-addnew:

variables assignment:
r=Ref2
r1=Ref1
ce=mkcell(-∞,⌈ PID1 ⌉,Ref1)
r2=Ref2
H={(Ref1,mkcell(-∞,⌈ PID1 ⌉,Ref2))}

MODEL:

function table for reachable : Ref x Ref x heap -> bool
--------------------------------------------------------------------------------
Ref1   Ref1   {(Ref1,mkcell(-∞,⌈ PID1 ⌉,Ref2)),
               (Ref2,mkcell(-∞,⌈ PID1 ⌉,Ref1))}
Ref1   Ref1   {(Ref1,mkcell(-∞,⌈ PID1 ⌉,Ref2))}
```

```
Ref1   Ref2   {(Ref1,mkcell(-∞,⌈ PID1 ⌉,Ref2)),
               (Ref2,mkcell(-∞,⌈ PID1 ⌉,Ref1))}
Ref2   Ref1   {(Ref1,mkcell(-∞,⌈ PID1 ⌉,Ref2)),
               (Ref2,mkcell(-∞,⌈ PID1 ⌉,Ref1))}
Ref2   Ref2   {(Ref1,mkcell(-∞,⌈ PID1 ⌉,Ref2)),
               (Ref2,mkcell(-∞,⌈ PID1 ⌉,Ref1))}
================================================================================

statistics
p cnf 15436 39804
primary variables: 891
translation time: 2127 ms
solving time: 23 ms
```

Die korrigierte Version des Lemmas enthält die Zusatzbedingung $ce.nxt = null$, die garantiert, dass die Zelle ce sich neutral zu der Erreichbarkeit verhält:

$$\texttt{reachable} - \texttt{addnew}: \quad \begin{aligned} & reachable(r, r_1, H[r_2, ce]) \wedge \\ & \mathbf{ce.nxt} = \mathbf{null} \wedge r_1 \in H \wedge r_2 \notin H \rightarrow \\ & reachable(r, r_1, H) \end{aligned}$$

Auf dieses Lemma wurde auch der Ansatz von Thums angewandt. Der Ansatz hat 6000 Kalkül Schritte gebraucht (etwa 5 Minuten) und hat das gleiche Gegenbeispiel ausgerechnet.

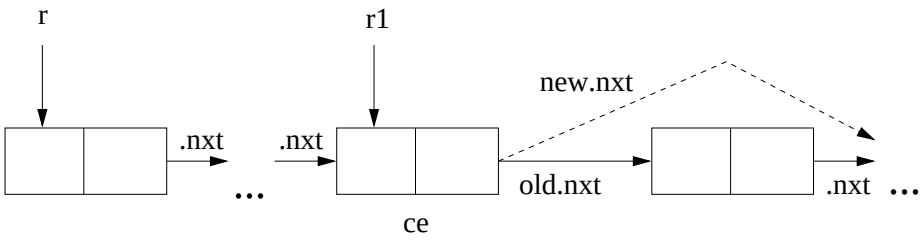


Abbildung 6.3: Lemma `rem-reachable-rev`.

6.3.3 Lemma `reachable-remove`

Das Lemma beschreibt das Entfernen einer Zelle aus der Kette. Unter gegebenen Bedingungen sollte nach dem Entfernen einer Zelle die Erreichbarkeit erhalten bleiben:

$$\texttt{reachable} - \texttt{remove}: \quad \begin{aligned} & reachable(r, r_0, H[r_1, ce]) \\ & H[r_1].nxt \in H \wedge r_1 \neq null \wedge r_1 \in H \\ & r_0 \neq H[r_1].nxt \wedge ce.nxt = H[H[r_1].nxt].nxt \rightarrow \\ & reachable(r, r_0, H) \end{aligned}$$

Die Abbildung 6.3 visualisiert dieses Lemma. Die Referenz r_1 zeigt auf eine Zelle, die von r aus erreichbar ist. Die Zelle $H[r_1]$ wird mit der Zelle *ce* überschrieben. Die Zelle *ce* zeigt auf $H[H[r_1].nxt].nxt$ (gestrichelte Linie $new.nxt$). Das Lemma behauptet, daß die Erreichbarkeit unverändert bleibt, vorausgesetzt die entfernte Zelle *ce* die gegebenen Vorbedingungen erfüllt.

Das Lemma ist falsch, weil ein Randfall vergessen wurde: $H[r_1].nxt = null$. Falls $H[r_1].nxt$ ein Nullpointer ist, ist die Zelle *ce* von Bedeutung für die Erreichbarkeit. Kodkod findet ein Gegenbeispiel:

```
counterexample for reachable-remove:

variables assignment:
r=Ref1
r0=Ref2
r1=Ref1
ce=mkcell(∞,⌈ PID0⌉),Ref2)
H={(null,mkcell(∞,⌈ PID0 ⌉,Ref2)),
   (Ref2,mkcell(⌜ int3 ⌝,⌈ PID0 ⌈,null)),
   (Ref1,mkcell(⌜ int3 ⌝,⌈ PID0 ⌉,null))}

MODEL:

function table for reachable : Ref x Ref x heap -> bool
--------------------------------------------------------------------------------
Ref1   Ref1   {(null,mkcell(∞,⌈ PID0 ⌉,Ref2)),
               (Ref2,mkcell(⌜ int3 ⌝,⌈ PID0 ⌉,null)),
               (Ref1,mkcell(∞,⌈ PID0 ⌉,Ref2))}
Ref1   Ref1   {(null,mkcell(∞,⌈ PID0 ⌉,Ref2)),
               (Ref2,mkcell(⌜ int3 ⌝,⌈ PID0 ⌉,null)),
               (Ref1,mkcell(⌜ int3 ⌝,⌈ PID0 ⌉,null))}
Ref1   Ref2   {(null,mkcell(∞,⌈ PID0 ⌉,Ref2)),
               (Ref2,mkcell(⌜ int3 ⌝,⌈ PID0 ⌉,null)),
               (Ref1,mkcell(∞,⌈ PID0 ⌉,Ref2))}
Ref2   Ref2   {(null,mkcell(∞,⌈ PID0 ⌉,Ref2)),
               (Ref2,mkcell(⌜ int3 ⌝,⌈ PID0 ⌉,null)),
               (Ref1,mkcell(∞,⌈ PID0 ⌉,Ref2))}
Ref2   Ref2   {(null,mkcell(∞,⌈ PID0 ⌉,Ref2)),
               (Ref2,mkcell(⌜ int3 ⌝,⌈ PID0 ⌉,null)),
               (Ref1,mkcell(⌜ int3 ⌝,⌈ PID0 ⌉,null))}
Ref2   Ref2   {(null,mkcell(∞,⌈ PID0 ⌉,Ref2)),
               (Ref2,mkcell(⌜ int3 ⌝,⌈ PID0 ⌉,null))}
================================================================================

statistics
p cnf 126832 333292
primary variables: 3506
translation time: 62496 ms
solving time: 25800 ms
```

Die korrigierte Version des Lemmas ist:

`reachable − remove` : $reachable(r, r_0, H[r_1, ce])$
$\mathbf{H[r_1].nxt \neq null} \wedge H[r_1].nxt \in H \wedge r_1 \neq null \wedge r_1 \in H$
$r_0 \neq H[r_1].nxt \wedge ce.nxt = H[H[r_1].nxt].nxt \rightarrow$
$reachable(r, r_0, H)$

Die Anwendung des Ansatzes zur Fehlersuche von Thums hat hier kein Fehler gefunden, weil die Information **ce.nxt = null** wird in den Kalkülschritten nie berücksichtigt. Es liegt daran, dass die Konstante $null : Ref$ unterspezifiziert ist. Die Fehlersuche von Thums ist insofern nicht vollständig, weil in deren Breitensuche auf mit Konstruktoren generierten Datentypen die unterspezifizierten Funktionen nicht untersucht wertden. Dieses Problem wird im nächsten Abschnitt nochmal diskutiert.

6.4 Alternative Generische Spezifikation

In einer alternativen generischen Spezifikation dieser Fallstudie wurde von den Zellen abstrahiert:

lincell =
enrich Ref **with**
sorts cell;
functions . .nxt : cell → Ref ;
variables ce: cell;

end enrich

Die unterspezifizierte Funktion $.nxt : cell \rightarrow Ref$ modelliert den Zeiger der jeweiligen Zelle. Der Inhalt der Zelle selbst ist für die Eigenschaft *reachable* nicht relevant.

Das besondere an dieser generischen Variante ist die unterspezifizierte Funktion *.nxt*. Der Ansatz zur Fehlersuche von Thums funktioniert hier nicht. Im Ansatz von Thums werden systematisch die Fallunterscheidungen über die Variablen im Theorem gemacht. Diese Fallunterscheidungen sind *constructor cuts*. Z.B. für die Variable x der Sorte List wird die folgende Fallunterscheidung gemacht: $x = nil \vee x = cons(a, nil)$. Für die unterspezifizierten Funktionen wie *nxt* oder *null* wird über die Werte von entsprechenden Termen keine Fallunterscheidung gemacht ($f(x) = y \vee f(x) \neq y$). Die *konkreten .nxt*-Werte sind aber entscheidend für die erfolgreiche Fehlersuche, weil sie die Graphstruktur vom verzeigerten Heap speichern.

Betrachten wir das fehlerhafte Lemma `reachable-next` aus dem Abschnitt 6.3.1. Kodkod findet auch einen Gegenbeispiel in der generischn Spezifikation:

```
counterexample for reachable-next:

variables assignment:
```

```
r=Ref2
r1=Ref2
H={(null,cell1),(Ref2,cell2)}


MODEL:

function table for reachable : Ref x Ref x heap -> bool
--------------------------------------------------------------------------------
Ref2                Ref2                {(null,cell1),(Ref2,cell2)}
================================================================================

function table for .nxt : cell -> Ref
--------------------------------------------------------------------------------
cell2               null
================================================================================

statistics
p cnf 13059 33240
primary variables: 696
translation time: 1437 ms
solving time: 18 ms
```

Das Gegenbeispiel ist das gleiche wie für die speziellere Spezifikation.

6.5 Zusammenfassung

In größeren Fallstudien werden bei interaktiven Beweisen Hunderte von Lemmas nebenbei definiert und als Rewrite-Regeln weiterverwendet. Die Wahrscheinlichkeit dabei falsche Aussagen zu formulieren ist groß. Öfters werden dadurch die komplizierten Beweise fälschlicherweise abgeschlossen, weil nicht geltende Lemmas benutzt wurden. Die automatischen "on-the-fly" Überprüfungen von solchen Aussagen sind äußerst hilfreich und sind eine attraktive Alternative zu langwierigen interaktiven Beweisen. Informell die Lemmas zu überprüfen ist kein vergleichbarer Ersatz, da es zeitaufwendiger und insgesamt weniger effizient als die Gegenbeispielsuche mit Alloy ist.

In dieser Fallstudie wurden erstmal die nicht-freien Datentypen verwendet (*Stores*, *Sets*, *Integers*). Eine vollständige Liste der Ergebnisse ist im Anhang B.2 zu finden. Die Fehlersuche war schnell (wenige Sekunden) und erfolgreich. Nur für größere Gegenbeispiele bei manchen Lemmas hat es mehrere Minuten gedauert. Im Vergleich zum Ansatz von Thums war unser Ansatz in dieser Fallstudie weit überlegen. Insbesondere weil hier die unterspezifizierten Funktionen bei der Entstehung von Fehlern sehr kritisch waren. Die ursprüngliche Spezifikation der Erreichbarkeit im Graphen war nicht kompatibel und musste mit dem Operator für die transitive Hülle neuspezifiziert werden. Da Alloy/Kodkod diesen Operator zur Verfügung stellen, war die Anpassung sehr leicht.

Kapitel 7

Fallstudie: Sicherheitsmodell für SmacOS

Diese Fallstudie ist ein Beispiel aus der Praxis. Es handelt sich um eine Spezifikation des Sicherheitsmodells des multiapplikativen Smartcard Betriebssystems (SmacOS) [46]. Wegen der Größe der Spezifikation werden die Grenzen für die mit dieser Technik noch handhabbaren Spezifikationen ganz deutlich. Die Fallstudie hat auch die Entwicklung der inkrementellen Technik mit Kodkod zwecks besseren Skalierbarkeit, siehe Kapitel 8, motiviert.

7.1 Sicherheitsmodell

Das Modell behandelt die zentralen Sicherheitsaspekte wie die Geheimhaltung, die Integrität, die sichere Kommunikation zwischen den Anwendungen auf der Karte sowie das sichere Hochladen von Anwendungen auf der Karte. Die Formalisierung basiert auf der Authentifikation und transitiver Noninterference [41]. Die formale Spezifikation erlaubt eine Verifikation des Systems auf der Ebene der abstrakten Systemaufrufe bezüglich der Sicherheitsanforderungen. Im Prinzip ist es auch für andere mobile Geräte wie Handys oder PDAs geeignet. Das Design war von einem informellen Sicherheitsmodell beeinflusst worden, das von IBM für ein Smartcard Betriebssystem für den Philips SmartXA Chip entwickelt wurde.

Die Abbildung 7.1 zeigt ein Beispiel-Szenario, wo auf der Karte zwei Anwendungen installiert sind: ein elektronischer Türöffner der Hotelkette H und eine Anwendung für die elektronische Flugscheine einer Fluglinie A. Da die beiden Gesellschaften H und A kooperieren, sollte eine Kommunikation zwischen den beiden Anwendungen möglich sein. Die Kunden, die in Hotels von H übernachten bekommen Loyalitätspunkte und entsprechende Ermäßigungen bei A und umgekehrt. Die Kommunikation wird durch das Schreiben in die Dateien, die den jeweiligen Anwendungen gehören, modelliert.

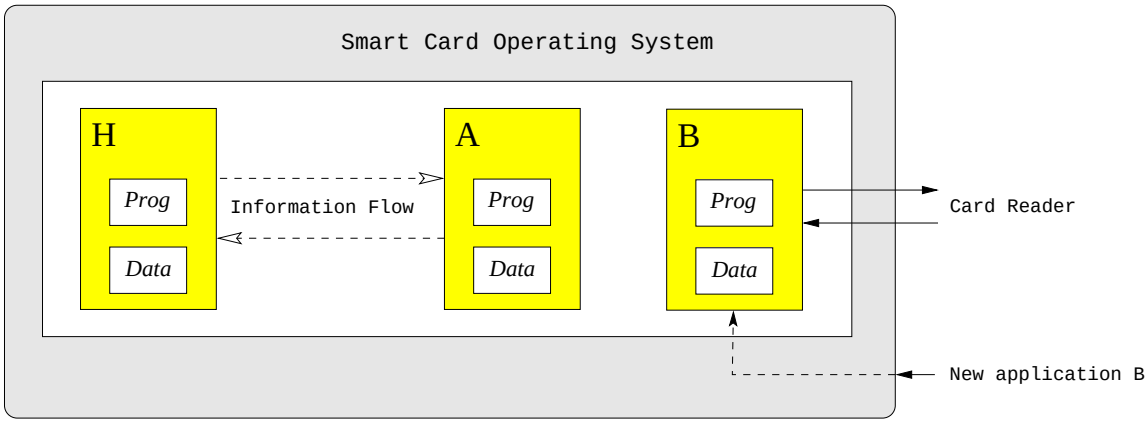


Abbildung 7.1: Beispiel-Szenario: Hochladen einer Anwendung auf der Karte.

Jetzt, wird eine neue Anwendung hochgeladen: eine elektronische Geldbörse der Bank B. Dabei muss sichergestellt werden, daß es zu keiner Interferenz (unerwünschter Informationsfluss) zwischen B und H bzw. A kommt. D.h. durch das Hochladen neuer Anwendungen sollte die bisherige Konfiguration nicht beeinträchtigt werden, sowie die geladene Anwendung nicht von den auf der Karte bestehenden Anwendungen korrumpiert werden. Dieses Problem könnte insbesondere an Schärfe gewinnen, wenn z.B. zwei geschäftlich konkurrierende Anwendungen auf der Karte installiert sind.

Aus Sicht des sicheren Informationsflusses unterscheidet man zwischen den *Subjekten* (*Agenten*), die die sicherheitsrelevanten Aktionen durchführen, und den Objekten, die durch die Subjekte manipuliert werden. Zu den Subjekten gehören das Betriebssystem (*OS*) und die Anwendungen (*Apps*). Um die Sicherheitsrichtlinien definieren zu können werden die Zugriffsklassen (*access classes*) definiert, die relativ zueinander geordnet sind: $ac_1 \prec \ldots \prec ac_n$.

Basierend auf den Zugriffsklassen bekommt jedes Subjekt eine *Sicherheitsdomäne*, der seine **Zugriffsrechte** festlegt, siehe Definition 18.

Definition 18 *[Sicherheitsdomäne (security clearance)]*
Ein Subjekt A mit der Sicherheitsdomäne $(ircl_A, iwcl_A, srcl_A, swcl_A)$ kann die Information dem Subjekt B mit der Sicherhheitsdomäne $(ircl_B, iwcl_B, srcl_B, swcl_B)$ übertragen genau dann wenn:

$$ircl_B \preceq iwcl_A \wedge swcl_A \preceq srcl_B$$

Jeder Aktion (Schreiben in eine bzw. Lesen aus einer Datei) wird ein Paar von Zugriffsklassen zugeordnet: $(iwcl, swcl)$ (für Schreiben) bzw. $(ircl, srcl)$ (für Lesen). Die beiden in so einem Paar enthaltenen Zugriffsklassen legen die Integritätsstufe (*integrity level*) und die Geheimhaltungsstufe (*secrecy level*) der jeweiligen Aktion fest. Die Übertragung von Informationen von A zu B erfolgt

durch Schreiben in eine Datei durch A, die dann von B gelesen wird. Damit diese Übertragung erfolgreich ist, muss laut der oberen Definition für die Paare der Zuggriffsklassen $(iwcl_A, swcl_B)$ für die Schreibaktion von A und $(ircl_A, srcl_B)$ für die Leseaktion von B gelten: $ircl_B \preceq iwcl_A$ (A hat höhere Integritätsklasse als B, d.h. kann mindestens das gleiche verändern wie B) und $swcl_A \preceq srcl_B$ (B hat höhere Geheimhaltungsklasse als A, d.h. kann mindestens das gleiche lesen wie A).

Im Gegensatz zu Subjekten haben die Objekte (Dateien mit Daten) nur zwei Zugriffsklassen zugeordnet: für die Integrität (icl) und für die Geheimhaltung (scl). Ein Subjekt mit dem Sicherheitsfreiraum dm, der lesend / schreibend auf die Datei fp zugreifen will, muss entsprechend höhere / niedrigere Geheimhaltungsklasse aber niedrigere / höhere Integrität für die Lese- bzw. Schreibaktion besitzen. Die Prädikate

$$rdable : domain \times filepath \times filesystem$$
$$wrable : domain \times filepath \times filesystem$$

testen ob eine Datei unter dem Pfad fp im Dateisystem fs für ein Subjekt mit der *clearance* dm lesbar bzw. beschreibbar ist:

$$rdable(dm, fp, fs) \leftrightarrow fp \in fs \wedge dm.ircl \leq fs[fp].icl \wedge fs[fp].scl \leq dm.srcl$$
$$wrable(dm, fp, fs) \leftrightarrow fp \in fs \wedge fs[fp].icl \leq dm.iwcl \wedge dm.swcl \leq fs[fp].scl$$

Aufbauend auf den Zugriffsrechten für die Aktionen kann die Sicherheitsrichtlinie definiert werden, die den Informationsfluss zwischen den Agenten regelt.

Definition 19 *[Sicherheitsrichtlinie (security policy)]*
Eine Sicherheitsrichtlinie ist als eine Relation $\rightsquigarrow$: $domain \times domain$ auf den Domänen (clearances) definiert. Dabei bedeutet $A \rightsquigarrow B$, daß es dem Subjekt mit der Domäne A erlaubt ist die Information zu dem Subjekt mit der Domäne B zu übertragen. Formal ist die Relation wie folgt spezifiziert:

$$dm_1 \rightsquigarrow dm_2 \leftrightarrow dm_1.swcl \leq dm_2.srcl \wedge dm_2.ircl \leq dm_1.iwcl$$

Das Gesamtmodell des Systems kann grob als ein Zustandsübergangsdiagramm gesehen werden. Es fängt im Zustand *init* und führt sequentiell die Systemaufrufe aus, die den Systemzustand verändern und eine Ausgabe produzieren. Die zentrale Operation

$$exec : systemstate \times command \rightarrow systemstate \times output$$

berechnet den nachfolgenden Systemzustand und die entsprechende Ausgabe. Der Systemzustand speichert neben den wichtigen Informationen über das Dateisystem, Authentifizierung, Zugriffsklassen der Subjekte (Anwendungen) und der

Objekte (Dateien) auch die Information über den Subjekt, der gerade bei der Ausführung dran ist (sequentielle Ausführung der Systemaufrufe).

Die Sicherheitsrichtlinie erlaubt es die Sicherheit des Gesamtsystems basierend auf dem Prinzip der *Noninterference* zu definieren. Das Prinzip der Noninterference wurde von Rushby [41] beschrieben. Das System ist also sicher gdw. das Ergebnis der Ausführung der Systemaufrufe durch eine Anwendung A unabhängig von vorher ausgeführten Systemaufrufe durch eine beliebige Anwendung B, die mit A nicht interferiert. Ob das System diese zentrale Sicherheitseigenschaft erfüllt hängt vor allem von der Spezifikation der Systemaufrufe. Sollte einer der Systemaufrufe fehlerhaft spezifiziert sein, z.B., beim Lesen einer Datei werden die Zugriffsrechte nicht beachtet, würde dies die Eigenschaft der Noninterference verletzen und damit zu einer Sicherheitslücke im System führen.

Um die Noninterference auszudrücken wird eine Relation auf Systemzuständen eingeführt, die ihre Gleichheit bezüglich einer bestimmten Clearence spezifiziert: $\sim$: $systemstae \times domain \times systemstate$. D.h., $sys_1 \sim^A sys_2$ gilt gdw. für den Subjekt mit Clearence A sind die Zustände sys_1, sys_2 ununterscheidbar (gleiche verfügbare Daten etc.). Eine Formale Definition ist:

$$
\begin{aligned}
eqd(sys_1, d, sys_2) \leftrightarrow \quad & sys_1.as = sys_2.as \wedge sys_1.ck = sys_2.ck \wedge \\
& sys_1.cap = sys_2.cap \wedge eqrd(sys_1.fs, d, sys_2.fs) \wedge \\
& eqap(sys_1.fs, sys_2.fs) \wedge eqdircont(sys_1.fs, d, sys_2.fs) \wedge \\
& eqcont(sys_1.fs, d, sys_2.fs)
\end{aligned}
$$

7.2 Algebraische Spezifikation in KIV

Die KIV Spezifikation definiert im wesentlichen die grundlegenden Datenstrukturen, die den Systemzustand beschreiben und die darauf arbeitende Operationen wie *exec* (Ausführung eines Systemaufrufs) oder Prädikate, die die Sicherheitseigenschaften ausdrücken.

Alle "komplizierten" Datentypen sind bereits in der KIV Bibliothek definiert und können direkt übernommen werden. Es sind die frei generierten *Listen* (siehe Abschnitt 4.3) und die nicht-freien Datentypen: *Stores* (siehe Abschnitt 4.4) und *Sets*.

$$\texttt{list} = [] \mid . \ + \ . \ (\ . \ .\texttt{first} : \texttt{elem};\ . \ .\texttt{rest} : \texttt{list})$$

$$. \ + + \ . \ : \texttt{set} \times \texttt{elem} \rightarrow \texttt{set} \quad (:\ \textit{insert}\ :)$$
$$\texttt{set generated by } \emptyset,\ ++$$

$$. \ [\ .\ ,\ .\ ] \ : \texttt{store} \times \texttt{elem} \times \texttt{data} \rightarrow \texttt{store} \quad (:\ \textit{put}\ :)$$
$$\texttt{store generated by } \oslash,\ .\ [\ .\ ,\ .\ ]$$

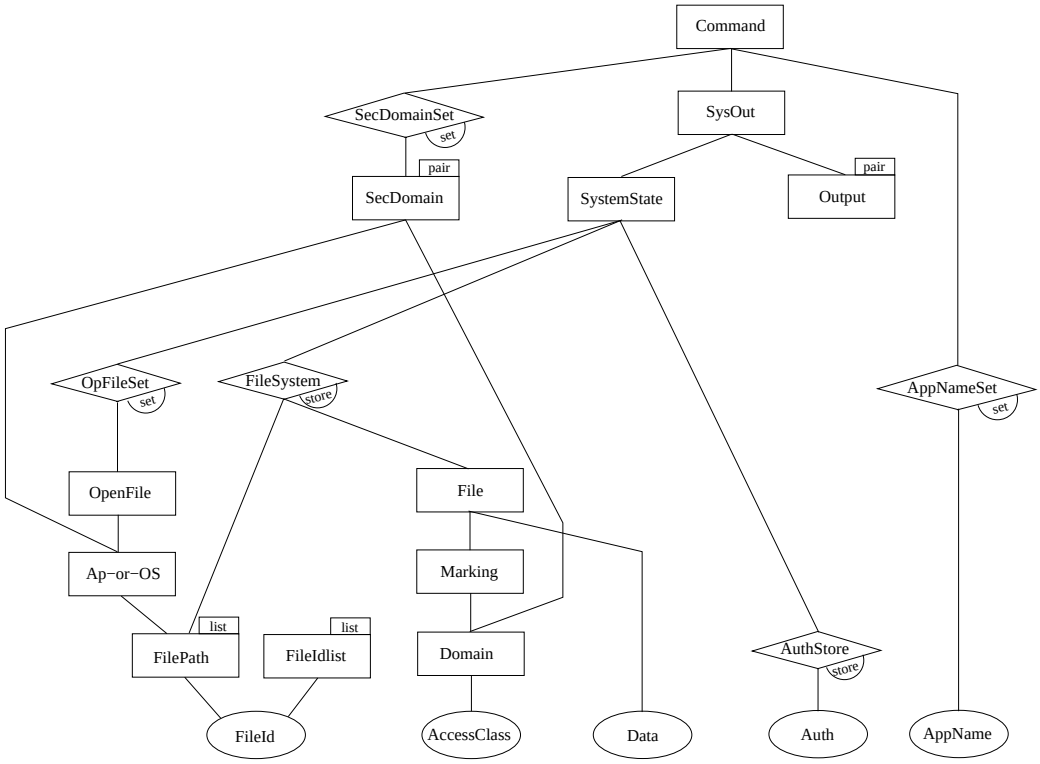


Abbildung 7.2: Hierarchie der Datentypen in der KIV Spezifikation.

Alle anderen Datentypen sind "einfach" in dem Sinne, daß sie frei generiert werden und nicht rekursiv sind, d.h. sie kapseln andere Datenstrukturen zu einem Bündeldatenobjekt. Dabei reflektieren sie die anwendungsspezifischen Aspekte der Spezifikation. Der zentrale Datentyp *SystemState* beschreibt den Gesamtzustand des Systems zu einem bestimmten Zeitpunkt:

$$\texttt{SystemState} = \texttt{mksys}(\texttt{fs} : \texttt{FileSystem},\ \texttt{as} : \texttt{AuthStore},\ \texttt{ck} : \texttt{Auth},\ \texttt{cap} : \texttt{Ap} - \texttt{or} - \texttt{OS},\ \texttt{ofs} : \texttt{OpenFileSet})$$

Zu der Beschreibung des Systemzustands gehören das Dateisystem (Datentyp *FileSystem*), Authentifizierungstabelle (Datentyp *AuthStore*, eine Zuordnung zwischen Anwendungen und dazugehörigen Authentifizierungsinformationen), Authentifizierungsinformation der Karte (Datentyp *Auth*), der aktuelle Besitzer der Ausführungsrechte (Datentyp *Ap-or-OS* spezifiziert eine Anwendung oder das Betriebssystem), die zurzeit geöffneten Dateien (*OpenFileSet*, eine Menge).

Die Spezifikation von *Stores* ist mit *Elem* (Adressen) und *Data* (gespeicherten Daten) parametrisiert. Um damit das Dateisystem zu spezifizieren wird sie mit den Datentypen *FilePath* (für *Elem*, der Pfad wo die Datei gespeichert ist) und *File* (Dateien, eine Datei mit Daten oder eine Anwendung) instanziert:

$$\texttt{FileSystem} = \texttt{Store}[\texttt{FilePath}, \texttt{File}]$$

Ein Pfad (*FilePath*) wird als eine Liste spezifiziert. Dafür wird die Spezifikation von *Listen* mit dem nicht-generierten Datentyp *FileId* instanziert:

$$\texttt{FilePath} = \texttt{List}[\texttt{FileId}]$$

FileId ist ein eindeutiger Dateideskriptor, das jeder Datei bzw. jedem Dateiverzeichnis zugeordnet ist. Der Datentyp *File* spezifiziert Dateien und Dateiverzeichnisse;

$$\begin{aligned}\texttt{File} = \quad & \texttt{mkdir}(\texttt{dircont} : \texttt{FileIdList}, \texttt{icl}, \texttt{scl} : \texttt{AccessClass}) \\ & \texttt{mkfile}(\texttt{cont} : \texttt{Data}, \texttt{icl}, \texttt{scl} : \texttt{AccessClass}, \texttt{mrk} : \texttt{Marking})\end{aligned}$$

Ein Verzeichnis hat als Attribute die Liste der enthaltenen Dateien (*dircont*, Liste der Dateideskriptoren) und die Zugriffsklassen (*icl,scl*). Für eine Datei neben den anderen Attributen enthält der Konstruktor den Feld *mrk* vom Typ *Marking*:

$$\texttt{Marking} = \texttt{None} \mid \texttt{mkmrk}(\texttt{dm} : \texttt{Domain})$$

Die Dateien mit Daten haben keine Markierung, d.h. das Feld *mrk* ist gleich *None*. Die Anwendungen besitzen Markierungen die ihren Zugriffsfreiraum (*security clearance*) festlegen.

$$\texttt{Domain} = \texttt{mkdom}(\texttt{ircl}, \texttt{iwcl}, \texttt{srcl}, \texttt{swcl} : \texttt{AccessClass})$$

Neben dem Datentyp *File*, der für das Dateisystem benutzt wird, gib es auch den Datentyp *OpenFile*, der eine geöffnete Datei spezifiziert. Der Systemzustand enthält die Menge (Datentyp *Set*) der geöffneten Dateien.

$$\begin{aligned}\texttt{OpenFile} = \quad & \texttt{rd}(\texttt{owner} : \texttt{Ap}-\texttt{or}-\texttt{OS},\ \texttt{file} : \texttt{FilePath}) \mid \\ & \texttt{wr}(\texttt{owner} : \texttt{Ap}-\texttt{or}-\texttt{OS},\ \texttt{file} : \texttt{FilePath})\end{aligned}$$

Die Information über den Subjekt (eine Anwendung oder das Betriebssystem), der die Datei geöffnet hat, wird im Datentyp *Ap-or-OS* gespeichert:

$$\texttt{Ap}-\texttt{or}-\texttt{OS} = \texttt{mkap}(\texttt{ap} : \texttt{FilePath}) \mid \texttt{OS}$$

Es gibt 23 Systemaufrufe, wie z.B. Öffnen, Lesen, Schreiben einer Datei oder Hochladen einer Anwendung. Diese werden mit dem Datentyp *Command* spezifiziert. Die Systemaufrufe sind in zwei Klassen eingeteilt. *register, loadapp,*

delapp werden durch das Betriebssystem der Karte als Reaktion auf die externen Anfragen aufgerufen. *create, read, write, remove, setintsec* werden von der aktuell laufenden Anwendung zwecks Datenmanipulation aufgerufen.

Die Operation $exec : SystemState \times Command \rightarrow SysOut$ berechnet den Nachfolgezustand und die Ausgabe nach der Ausführung des jeweiligen Systemaufrufs. Der Datentyp *Output* speichert die Systemausgabe nach der Ausführung:

$$\begin{aligned}\texttt{SysOut} = \quad & \texttt{YES} \mid \texttt{NO} \mid \texttt{mkoutdata}(\texttt{data} : \texttt{Data}) \mid \texttt{mkoutap}(\texttt{outap} : \texttt{AppName}) \mid \\ & \texttt{mkoutbool}(\texttt{b} : \texttt{Bool}) \mid \texttt{mkoutlist}(\texttt{li} : \texttt{FileIdList}) \mid \\ & \texttt{mkoutcls}(\texttt{icl}, \texttt{scl} : \texttt{AccessClass})\end{aligned}$$

Die Ausgabe *NO* wird für die ungültigen Systemaufrufe gemacht, z.B., der Zugriff auf eine Datei trotz der nicht ausreichenden Rechte. Je nach der aufgerufenen Systemoperation können auch komplexere Daten zurückgegeben werden, z.B., das Auflisten eines Verzeichnisses liefert die Liste der Dateideskriptoren.

Die Axiomatisierung der *exec* Operation wird mittels der strukturellen Induktion über den zweiten Argument (der auszuführende Systemaufruf) gemacht. Es wird dabei im wesentlichen mit Hilfe einer Fallunterscheidung der Nachfolgezustand konstruiert. Die Axiome haben die Form:

$$exec(sysstate, syscall(x, y)) = \textbf{if } \psi \textbf{ then } expr_1 \textbf{ else } expr_2 \tag{7.1}$$

Die Schreibweise *if* φ *then* ψ_1 *else* ψ_2 ist Abkürzung für die Formel $\varphi \wedge \psi_1 \vee \neg\varphi \wedge \psi_2$. Z.B., der Systemaufruf *create(parent: FilePath, fid: FileId)* wird wie folgt spezifiziert:

$$\begin{aligned}
&exec(sys, create(fp, fid)) = \\
&\quad \textbf{if} \\
&\qquad sys.cap \neq OS \wedge \\
&\qquad rdable(sys.fs[sys.cap.ap].mrk.dm, fp, sys.fs) \wedge \\
&\qquad wrable(sys.fs[sys.cap.ap].mrk.dm, fp, sys.fs) \wedge \\
&\qquad dirp(sys.fs[sys.cap.ap]) \wedge \\
&\qquad \neg fid \in sys.fs[fp].dircont \wedge \\
&\quad \textbf{then} \\
&\qquad mksys(\\
&\qquad\quad sys.fs[fp + fid, \\
&\qquad\qquad\quad mkfi(nodata, sys.fs[sys.cap.ap].mrk.dm.ircl, \\
&\qquad\qquad\qquad sys.fs[sys.cap.ap].mrk.dm.srcl, none)] \\
&\qquad\qquad\quad [fp, \\
&\qquad\qquad\quad mkdir(sys.fs[fp].dircont + fid, sys.fs[fp].icl, fs[fp].scl)] \\
&\qquad\quad sys.as, sys.ck, sys.cap, sys.ofs
\end{aligned}$$

$$) \times YES$$
else
$$sys \times NO$$

Die Bedingung in der Fallunterscheidung spezifiziert den positiven Fall, wo eine Datei mit dem Deskriptor fid im Verzeichnis mit Pfad fp von dem Subjekt *sys.cap*, der aktuell die Ausführungsrechte hat, angelegt werden kann. Es wird die Datei $sys.fs[sys.cap.ap]$, die der Anwendung *sys.cap* entspricht, abgerufen und überprüft ob die Domäne (Sicherheitsfreiraum) es erlaubt lesend bzw. schreibend auf den Ordner fp zuzugreifen (Prädikate $rdable$, $wrable$). Zum Schluß muss überprüft werden ob der Pfad fp wirklich zu einem Ordner führt ($dirp$) und der Dateideskriptor fid nicht in diesem Ordner bereits enthalten ist ($\neg fid \in sys.fs[fp].dircont$). Im negativen Fall liefert *exec* einen Tupel $sys \times no$, d.h. der Systemzustand bleibt unverändert und die Ausgabe ist *NO*. Im positiven Fall, wird der Nachfolgezustand mittels Konstruktoren entsprechend aufgebaut und die Systemausgabe ist *YES*. Nach dem erfolgreichen Anlegen einer Datei wird im Nachfolgezustand nur der Feld mit dem Dateisystem ($sys.fs$) verändert. Unter der Adresse $fp + fid$ wird die mit dem Konstruktor

$$mkfi : Data \times AccessClass \times AccessClass \times Marking \rightarrow File$$

erstellte Datei angelegt. Diese Datei ist eine Datendatei (keine Markierung, *none*) und hat die gleichen Zugriffsrechte wie der Subjekt, der diese Datei erstellt hat. Zusätzlich wird der unter der Adresse fp gespeicherte Verzeichnis abgeändert: die Liste der darin enthaltenen Dateien wird um fid erweitert.

7.3 Alloy Spezifikation

Die Erstellung der Alloy Spezifikation für die Fallstudie "SmacOS" bedeutete im wesentlichen die Transformation zu der relationalen Form[1] und die Überprüfung auf die Kompatibilität aller Definitionen der Operationen. Der Hauptwand lag bei den Definitionen der 23 Systemaufrufe und der Definition von des $\sim$: $systemstate \times domain \times systemstate$. Allerdings, sind diese Definitionen uniform aufgebaut, siehe (7.1) in Abschnitt 7.2, was zu ähnlichen Kompatibilitätschecks führt. Die Zwischenergebnisse in ψ, $expr_1$ und $expr_2$ müssen durch die Ein-/Ausgaben der Operation *exec* $\preceq$-begrenzt sein, siehe die Abschnitte 4.3.3 und 4.3.5 über die Kompatibilität der algebraischen Spezifikationen mit der endlichen begrenzten Analyse. Nach einer informellen Analyse wurden alle verwendeten Definitionen als kompatibell eingestuft.

Die beiden verwendeten nicht-freien Datentypen *Set* und *Store* sind in der KIV Bibliothek und damit auch bereits nach Alloy portiert. Bei den anwendungsspezifischen freien Datentypen wird das automatische *SUA*-basierte Vorgehen bei der Erstellung der Alloy Spezifikationen angewandt. Nach einer informellen Untersuchung wurde festgestellt, daß die Subtermabgeschlossenheit für die korrekte Modellgenerierung ausreicht. Nur für die beiden Datentypen *FilePath*

[1] Bei Kodkod Versuchen wird dieser Schritt vollautomatisch durchgeführt.

und *FileIdList*, die Instanzen vom Datentyp *List* sind, wurde die stärkere $\#_{list}$-Abgeschlossenheit verlangt. Insgesamt hat die Spezifikation folgende Größe, siehe Abb. 7.1.

Modul	Zeilen	Sorten	Typ	Operationen
accessclass	45	AccessClass	-	2
aporos	30	APorOS	frei	0
appname	26	AppName	-	1
appnameset	152	AppNameSet	nicht-frei	3
auth	8	Auth	-	1
authstore	201	AuthStore	nicht-frei	5
bool	8	Bool	frei	0
command	271	Command	frei	0
data	5	Data	-	1
domain	41	Domain	frei	2
file	90	File	frei	0
fileid	19	FileId	-	1
fileidlist	196	FileIdList	frei	6
filepath	184	FilePath	frei	8
filesystem	493	FileSystem	nicht-frei	12
mark	23	Mark	frei	0
nat	114	Nat	frei	0
openfile	45	OpenFile	frei	0
openfileset	244	OpenFileSet	nicht-frei	9
output	95	Output	frei	0
secdomain	38	SecDomain	frei	1
systemstate	1169	SystemState	frei	3

Tabelle 7.1: Alloy Spezifikation.

7.4 Qualitätssicherung mit Alloy

Die Algebraische Spezifikation in KIV für die Fallstudie "SmacoOS" hat mehr als 800 Zeilen. Die zahlreichen Querverbindungen zwischen verschiedenen Teilen der Spezifikation machen es schwierig alleine durch das sorgfältige Anschauen die Fehler zu lokalisieren. Wie bereits früher an verschiedenen Beispielen gezeigt wurde, ist die begrenzte Analyse mit Alloy viel sorgfältiger und auch effizienter. Der Ansatz, den wir für eine größere Fallstudie wie "SmacOS" anwenden, besteht aus den folgenden sequentiell nacheinander durchgeführten Schritten:

Schritt 1 ("Lebendigkeit") : Sind die gewünschten Szenarien (Systemabläufe) möglich ?
⇒ `run` $Szenario_1$, ..., `run` $Szenario_n$

Schritt 2 ("Begrenzte Sicherheit") : Gibt es endliche abgeschlossene Modelle, die die gewünschten Eigenschaften verletzen?
⇒ `check` φ_1, ..., `check` φ_m

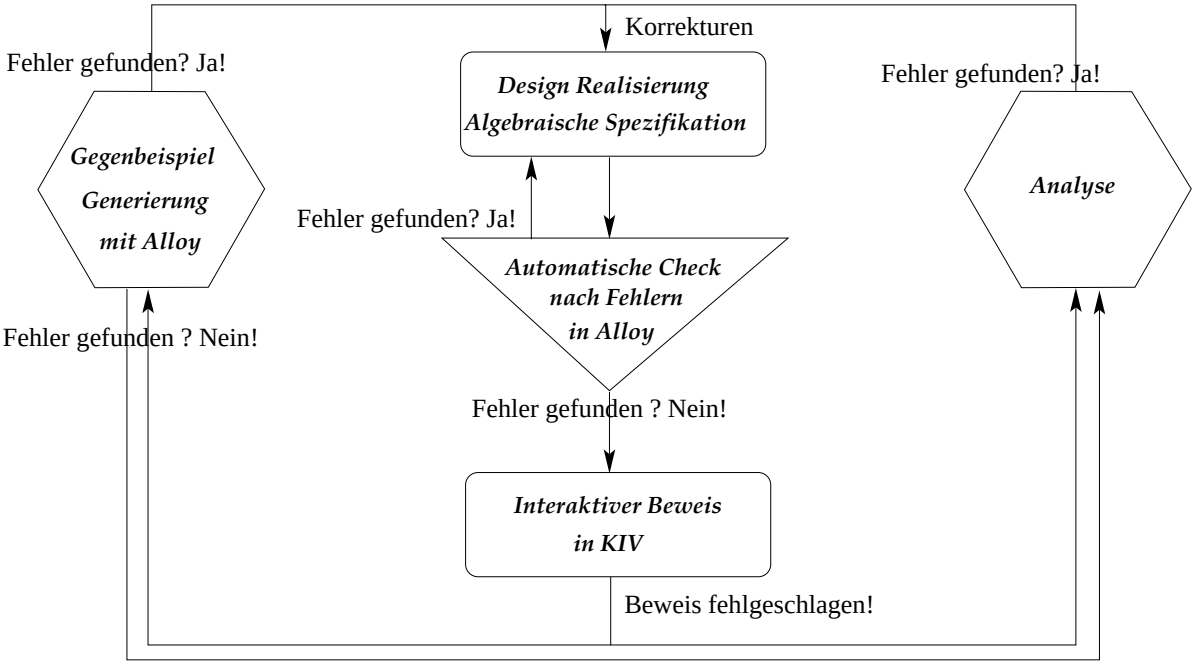


Abbildung 7.3: Qualitätssicherung mit Alloy.

7.4.1 Lebendigkeit

Die Komplexität der verwendeten Datenstrukturen, zahlreiche und komplizierte Operationen führen bereits in frühen Phasen der Entwicklung zu Fehlern in Spezifikationen. Ähnlich wie beim Programmieren wird dieser Art von Fehlern am effizientesten mit Tests aufgedeckt. Wenn man, z.B., in der Verifikation nur universelle Eigenschaften nachweist, was häufig der Fall ist, dann können viele Fehlerarten erst gar nicht erkannt werden. Z.B., die Sicherheitseigenschaft würde trivialerweise für das System gelten, das wegen eines Bugs immer im Initialzustand verharrt.

```
initial state: [[],[],ck,OS,[]]
step 1: register(akey,ckey) --> [[],[(ap,akey)],ck,OS,[]]
step 2: loadapp(A,Afi)   --> [[A],[(ap,akey)],ck,OS,[]]
step 3: startapp(A)  --> [[A],[(ap,akey)],ck,A,[]]
step 3: create(fp,fi)   --> [[A,fi],[(ap,akey)],ck,A,[]]
step 4: exitapp(A)   --> [[A,fi],[(ap,akey)],ck,OS,[]]
```

Abbildung 7.4: Systemzustandsübergänge für das Testszenario.

Betrachten wir ein Testszenario, wo eine Anwendung auf der Karte geladen und gestartet wird. Anschließend wird eine neue Datei durch diese Anwendung im Dateisystem angelegt. Am Anfang befindet sich das System im Initialenzustand: das Dateisystem enthält keine Dateien außer dem *Root*-Verzeichnis, der Speicher mit Authentifizierungsinformationen ist leer. In unserem Testszenario werden sequentiell folgende Systemschritte durchgeführt: der Authentifizierungsschlüssel

für die Anwendung wird auf der Karte gespeichert, die Anwendung wird als eine Datei im Dateisystem der Karte gespeichert, die Anwendung wird gestartet, eine neue Datei wird angelegt und die Anwendung wird beendet. Die Abbildung 7.4 zeigt die Übergänge zwischen den Systemzuständen, die als Tupeln mit Systeminformationen dargestellt werden.

Ein Systemzustand sys wird mit einem 5-Tupel dargestellt:

$$\begin{aligned} sys = (fs &: FileSystem, \\ as &: AuthStore, \\ ck &: Auth, \\ cap &: Ap - or - OS, \\ ofs &: OpenFileSet) \end{aligned}$$

Am Anfang gehören die Ausführungsrechte dem Betriebssystem: $sys.cap = OS$. Im ersten Schritt wird die Authentifizierungsinformation für die neue Anwendung auf der Karte gespeichert. Im zweiten Schritt wird die Anwendungsdatei im Kartendateisystem gespeichert. Dann folgen Starten, Anlegen einer Datei und Beenden.

Die Abbildung 7.5 zeigt ein endliches Modell, das automatisch für das betrachtete Testszenario mit Alloy generiert wurde. Jeder Pfad bestehend aus Systemzuständen und Übergängen (Systemaufrufe) beschreibt eine mögliche Evolution des Systems. Insbesondere ist das obere Szenario darin enthalten. Durchführung solcher Tests beweist, daß die gewünschten Abläufe im entworfenem System möglich sind.

Die Szenarien werden mit Alloy als Prädikate spezifiziert. Z.B., ein Szenario der mindestens 4 ausgewählte Systemaufrufe enthält wird wie folgt formuliert:

```
pred Szenario[] {

     some cmd1: CreateApp, cmd2: LoadApp, cmd3: StartApp, cmd4: ExitApp,
          sys: SystemState, so1,so2,so3,so4: SysOut |
               so1 = exec[sys][cmd1] and
               so1.snd = YES and
               so2 = exec[so1.fst][cmd2] and
               so2.snd = YES and
               so3 = exec[so2.fst][cmd3] and
               so3.snd = YES and
               so4 = exec[so3.fst][cmd4] and
               so4.snd = YES
}
```

Die Generierung erfolgt mit dem Befehl

```
run Szenario for 5
```

Im nächsten Schritt werden die UBE-artigen Sicherheitseigenschaften überprüft werden.

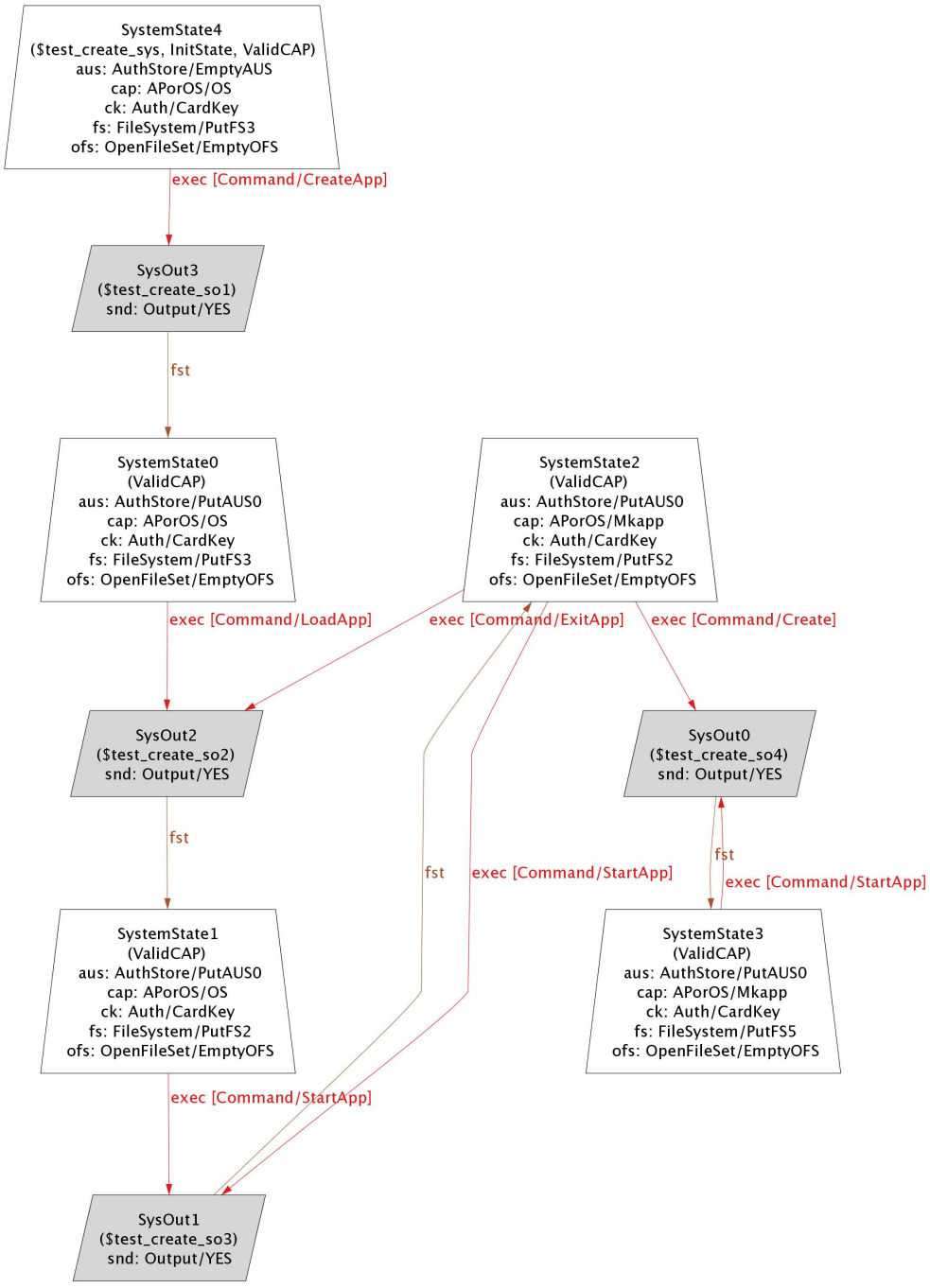


Abbildung 7.5: Ein möglicher Ablauf, der mit Alloy generiert wurde und das Hochladen einer Anwendung auf der Karte demonstriert.

7.4.2 Begrenzte Sicherheit

Testen einzelner interessanter Systemabläufe ist nur der erste Schritt in der Suche nach Fehlern. Eine viel größere Abdeckung bietet die Gegenbeispielsuche mit Alloy. Dabei werden *alle* endlichen Modelle bis einer vorgegebenen Größe auf die Erhaltung einer Eigenschaft untersucht: `check` φ `for scope`.

Zwei Klassen von Theoremen wurden betrachtet: gewöhnliche Lemmas, die die gewünschten Eigenschaften der spezifizierten Operationen beschreiben und die Gruppe von Theoremen, die die zentrale Sicherheitseigenschaft der "Noninterference" beschreiben. Schellhorn et al. [46] reduzieren die *globale* Eigenschaft der Noninterference, die über alle möglichen Systemabläufe aussagt, zu 8 *lokalen* Eigenschaften, die für jeden einzelnen Systemaufruf gelten sollen. Betrachten wir die dritte Eigenschaft. Sie verlangt, daß das System die Interferenzrelation $\rightsquigarrow$: $Domain \times Domain$ beachtet:

$$inv(sys) \wedge dom(sys, co) \not\rightsquigarrow d_0 \rightarrow sys \sim^{d_0} exec(sys, co).1 \tag{7.2}$$

Die Invariante $inv : SystemState$ gilt im initialen und in allen nachfolgenden Systemzuständen. Die Funktion $dom : systemstate \times command \rightarrow domain$ liefert die Domäne des Agenten, der im Systemzustand sys aktiv ist. Falls die Sicherheitsdomäne des im Zustand sys aktiven Agenten ($dom(sys, co)$) die Sicherheitsdomäne d_0 nicht beeinflüssen kann (Noninterference), dann ist nach der Ausführung von co der Nachfolgezustand $exec(sys, co).1$ für d_0 äquivalent[2] zu sys, d.h. aus Sicht von d_0 sind die beiden Zustände nicht zu unterscheiden und insbesondere gleiche Informationen zugreifbar sind.

Bereits während der Verifikation in KIV [46] wurde ein Fehler in der Spezifikation der *SetIntSec* Systemoperation gefunden. Wir haben diesen Fehler auch mit Alloy gefunden. *SetIntSec(fp,iac,sac)* setzt die Zugriffsrechte der unter fp gespeicherten Datei auf (iac, sac) neu. Die Axiomatisierung von *SetIntSec* enthält einen Fehler:

$$\begin{array}{l} exec(sys, setintsec(fp, iac, sac) = \\ \quad \textbf{if} \\ \quad\quad sys.cap \neq OS \wedge fp \in sys.fs \wedge filep(sys.fs[fp]) \\ \quad \textbf{then} \\ \quad\quad mksys(\\ \quad\quad\quad sys.fs[fp, mkfi(sys.fs[fp].cont, iac, sac, sys.fs[fp].mrk)], \\ \quad\quad\quad sys.as, sys.ck, sys.cap, sys.ofs) \times YES \\ \quad \textbf{else} \\ \quad\quad sys \times NO \end{array}$$

Die Vorbedingung für eine erfolgreiche Ausführung von *setintsec* ist zu schwach und lässt auch unerwünschte Abläufe zu. Nun wird die Sicherheitseigenschaft (7.2) mit Kodkod gecheckt.

[2] Die Relation $\sim$: $SystemState \times Domain \times SystemState$ identifiziert die Systemzustände,

scope (Modellgröße)	Gegenbeispiel	clauses	Zeit
2	nein	91330	4 s
3	nein	2412657	36 s
3 (4 File, 4 FileIdList, 5 FileSystem)	ja	7194817	10 min

Tabelle 7.2: Benchmark: Gegenbeispielsuche für die SmacOS Sicherheitseigenschaft (7.2) (MiniSAT Solver, 2.4 GHz Dual Core)

Ein Gegenbeispiel wird bei der Schranke 3 gefunden, wobei für die Sorten, die das Dateisystem modellieren, werden mindestens 4 Atome benötigt. Die generelle Strategie bei der Gegenbeispielsuche ist die Schranken schrittweise zu erhöhen bis das Gegenbeispiel gefunden wird oder die Suche lange braucht und abgebrochen werden muss. Ein gutes Versändniss vom Modell kann gut helfen die passenden Schranken zu wählen. Ansonsten bleibt eine einfache Strategie die Schranken $2, 3, 4, \ldots$ für alle Sorten gleich durchzuprobieren. Jedoch bei größeren Spezifikationen, wie z.B. SmacOS, kann eine passende Schrankenauswahl viel Zeit sparen, siehe Tabelle 7.2. Die erfolgreiche Fehlersuche mit Kodkod liefert ein Gegenbeispiel[3]:

```
counterexample for security:

variables assignment:
d0=mkdm(system-high,system-high,system-low,system-low)
co=setintsec([fileid2],system-low,system-low)
sys=({([],        mkdir([fileid2],mkacc(system-high,system-low))),
      ([fileid2], mkfi(data2,mkacc(access-class0,system-high),
                       mkmrk(mkdm(system-high,access-class0,system-low,system-high)))),
     {}, authentication0, mkap([fileid2]), {})

MODEL:

function table for exec : systemstate x command -> systemstateandoutput
--------------------------------------------------------------------------------
({([],        mkdir([fileid2],mkacc(system-high,system-low))),
  ([fileid2], mkfi(data2,mkacc(access-class0,system-high),
                   mkmrk(mkdm(system-high,access-class0,system-low,system-high)))),
 {}, authentication0, mkap([fileid2]), {})
setintsec([fileid2],system-low,system-low)
({([],        mkdir([fileid2],mkacc(system-high,system-low))),
 ([fileid2], mkfi(data2,mkacc(system-low,system-low),
             mkmrk(mkdm(system-high,access-class0,system-low,system-high)))),
 {}, authentication0, mkap([fileid2]), {})
...
================================================================================

statistics
p cnf 3911142 10883853
primary variables: 5534
translation time: 186508 ms
solving time: 439039 ms
```

Die Abbildung 7.6 visualisiert nochmal das Gegenbeispiel. Das Agent mit der

die aus Sicht der Sicherheitsdomäne d_0 äquivalent sind, siehe Abschnitt 7.1.

[3]Ein Systemzustand sys wird mit dem 5-Tupel dargestellt: $sys = (fs : FileSystem, as : AuthStore, ck : Auth, cap : Ap - or - OS, ofs : OpenFileSet)$.

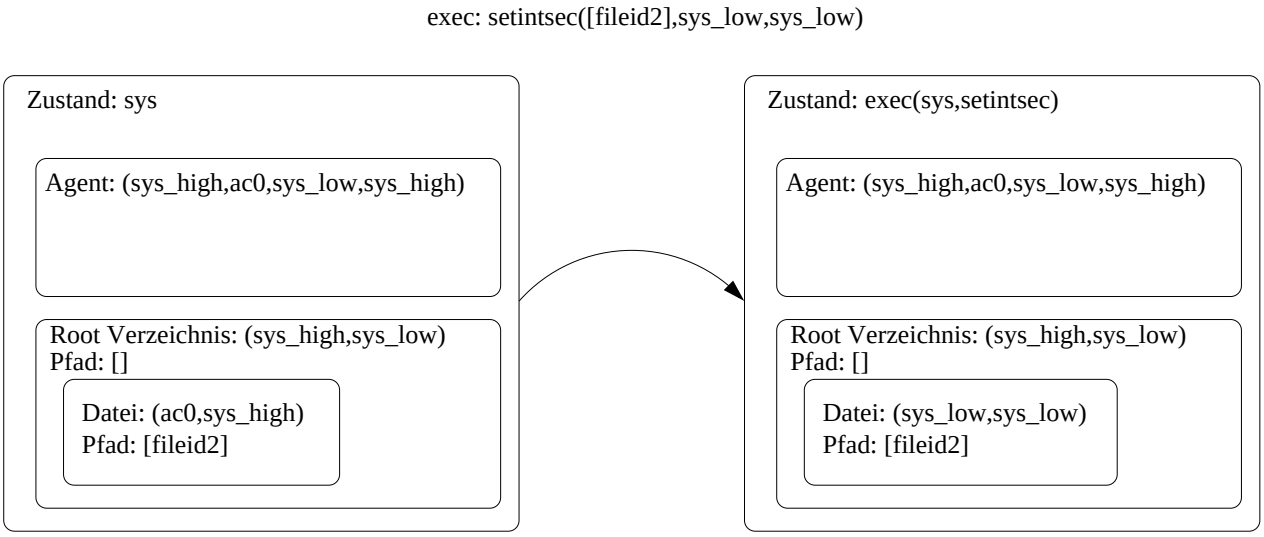


Abbildung 7.6: Gegenbeispiel für die Sicherheitseigenschaft (7.2).

Sicherheitsdomäne

$$cdom(sys, co) = (sys_{high}, ac_0, sys_{low}, sys_{high})$$

kann die Domäne

$$d_0 = (sys_{high}, ac_0, sys_{low}, sys_{high})$$

nicht beeinflüssen: $cdom(sys, co) \not\rightsquigarrow d_0$. Nun werden von dem Agenten die Zugriffsrechte der Datei $[fileid2]$ von (ac_0, sys_{high}) auf (sys_{low}, sys_{low}) verändert. Die Datei war für d_0 vorher lesbar und jetzt ist nicht mehr lesbar, d.h. der aktive Agent hat d_0 beeinflusst, was die Sicherheitseigenschaft verletzt.

Dieser Fehler wird durch die Verstärkung der Vorbedingung der Operation *SetIntSec* beseitigt. Das Verzeichnis in dem die abgeänderte Datei liegt muss lesbar und beschreibbar für den Agenten sein:

$$rdable(sys.fs[sys.cap.ap].mrk.dm, butlast(fp), sys.fs) \wedge$$
$$wrable(sys.fs[sys.cap.ap].mrk.dm, butlast(fp), sys.fs)$$

Eine Liste von weiteren Ergebnissen zu der Fallstudie ist im Anhang B.3 zu finden.

7.4.3 Change Management

Selbst nachdem die Entwurfsphase zum großen Teil abgeschlossen ist und der Entwickler sich der formalen Analyse zuwendet wird die Spezifikation trotzdem immer wieder verändert und angepasst. Sei es ein beim Beweisen entdeckter

Fehler oder ein vergessener Systemaufruf, die eventuellen Änderungen des Systementwurfs sind mit dem Risiko verbunden einen versteckten Fehler einzubauen.

Bis jetzt wurde dieses Problem so gelöst, daß nach jeder Änderung der Spezifikation alle betroffenen Beweise für ungültig erklärt werden. Nun bietet das KIV System dem Benutzer die Möglichkeit diese Beweise hoch automatisiert zu wiederholen ("Replay" Funktion). Allerdings, in vielen Fällen wird die Benutzerinteraktion verlangt, was bei hunderten Beweisen viel Zeit beansprucht. Mit Hilfe von Alloy, können jetzt die zahlreichen Theoreme vollautomatisch durchgecheckt werden. Falls, welche falsifiziert wurden, wird der Benutzer auf diese hingewiesen und kann sie gezielt analysieren.

7.5 Zusammenfassung

Die Spezifikation des Sicherheitsmodells ist eine Fallstudie realistischer Größe und ist für die in der Praxis vorkommenden Beispiele repräsentativ. Es waren vorallem die folgenden Fragen zu beantworten:

- Wo liegen die Grenzen für die berechenbaren Teilmodelle ?

Für den Ansatz mit Alloy war diese Fallstudie bereits an der Grenze der analysierbaren Spezifikationen. Generierung der Teilmodelle und Analyse der Theoreme dauerte im Schnitt von einigen Minuten bis zu 10 Minuten. Diese Erfahrung motiviert den inkrementellen Ansatz mit Kodkod. Im Abschnitt 8.3 wird über die ersten Experimente in diese Richtung berichtet.

- Was sind die Hauptprobleme für die "Zustandsexplosion" in größeren Spezifikationen ?

Bereits in kleinen Spezifikationen mit nur wenigen Datentypen explodieren die SAT-Instanzen, wenn die Schranken für die Anzahl der Atome in Teilmodellen erhöht werden.

- Wie schnell steigt der Aufwand bei der Portierung der Spezifikationen nach Alloy mit steigender Spezifikationsgröße ? Wo liegen dabei die Hauptschwierigkeiten ?

Insgesamt steigt der Aufwand moderat mit der Spezifikationsgröße und es müssen nur die Definitionen auf die Kompatibilität überprüft werden. Allerdings ist es bei größeren Spezifikationen auch viel aufwendiger, diese Überprüfung informell durchzuführen. Oft gibt es in Definitionen gut versteckte Abhängigkeiten zwischen den Datentypen, die für die Kompatibilität von Bedeutung sind.

Eine Ausführliche Liste der Ergebnisse aus der Evaluation ist im Anhang B.3 zu finden. Alle Versuche wurden mit Kodkod Engine durchgeführt, die bereits in KIV eingebaut wurde.

Kapitel 8

Kodkod Integration in KIV

Dieses Kapitel beschreibt die Realisierung des Kernansatzes zur endlichen Gegenbeispielsuche. Die Technik wurde mit Hilfe von Constraint Solver Kodkod [54, 53] umgesetzt und in KIV eingebaut.

8.1 Constraint Solver Kodkod

Kodkod ist eine Weiterentwicklung von Alloy, die drei zentrale Verbesserungen mit sich bringt.

- **Klare API-Schnittstelle für die einfachere Einbindung in andere Werkzeuge.** Bisher gab es in Alloy nur eine Zeichenkette-basierte Schnittstelle nach außen. Die Alloy Spezifikationen mussten vorher als Textdateien erstellt werden, was langsam, ineffizient und fehleranfällig war. Kodkod bietet eine Java API für die Erstellung der Spezifikationen und Modellberechnungen. Damit kann Kodkod viel besser als ein Backend-Tool verwendet werden.

- **Unterstützung für die Teillösungen.** Das Teilwissen über das zu berechnende Modell müsste in Alloy als Constraints formuliert werden, die dann hoffentlich die SAT-Prozedur schneller machen sollten. Mit Kodkod können die partiellen Lösungen als Teilrelationen direkt in die Spezifikation integriert werden. Damit ist Kodkod auf den Problemen, wo die Teillösungen (*partial instances*) bekannt sind, dramatisch schneller als Alloy.

- **Effizientere Kodierung in boolesche SAT-Probleme.** Kodkod überholt Alloy sogar auf den Problemen, für die Alloy entwickelt wurde. Dies gilt Dank der aufwendigeren Kodierung in die boolesche Formeln: neue Schema zur Auflösung von Symmetrien in den Modellen, effizientere Speicherung der Relation mit Sparse-Matrizen, bessere Erkennung und Ausnutzung von gemeinsamen Teilausdrücken im abstrakten Syntaxbaum, der die Constraints darstellt.

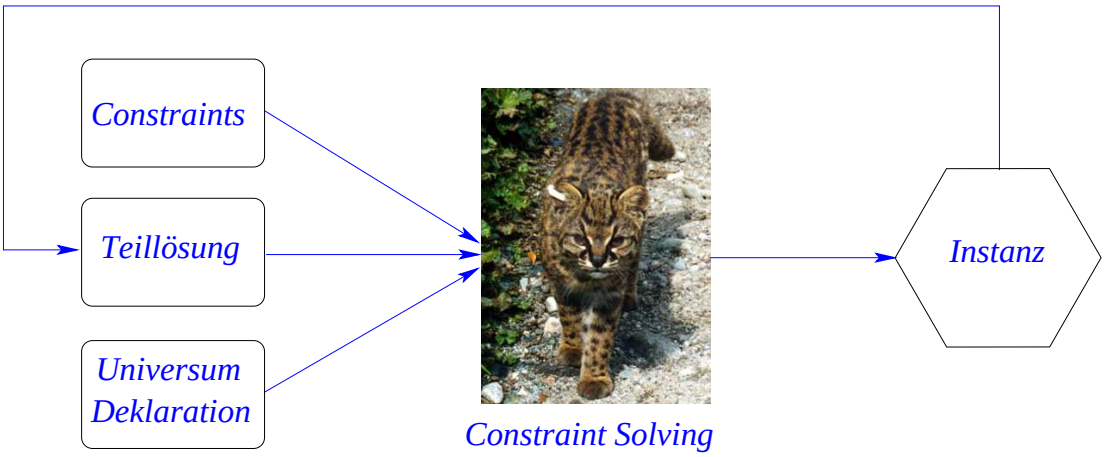


Abbildung 8.1: Constraint Solving mit Kodkod.

Für den Benutzer ist der Umstieg von Alloy nach Kodkod leicht. Die Sprache in der die Constraints an die Relationen im Modell formuliert werden ist die gleiche wie für Alloy: relationale Logik der ersten Stufe mit dem transitiven Abschluß.

Die Abbildung 8.1 zeigt die Benutzung von Kodkod. Die Definition eines Kodkod Problems erfolgt indem man (ähnlich wie in Alloy) zuerst das Universum deklariert. Dazu gehören die Deklarationen der Sorten und der Relationen. Es werden auch die Schranken (*Bounds*) für die Modellgröße und für die einzelnen Relationen. Eine Schranke für eine Relation ist ein Tupel $(lower, bound)$, das aus der unteren und der oberen Schranke besteht. Z.B. für eine binäre Relation $r : s_1 \times s_2$ auf Sorten s_1 und s_2 eine mögliche Schranke ist $(\{\}, domain(s_1) \times domain(s_2))$.

Im zweiten Schritt werden in Alloy Sprache die Constraints auf die Relationen im Modell spezifiziert. Kodkod berechnet ein Modell (*Instanz*), das die vorgegebenen Deklarationen, Schranken und Constraints erfüllt.

Die Instanzberechnung für ein Kodkod Problem P erfolgt intern in folgenden fünf Schritten [54]:

1. Aufspüren von Symmetrien.
2. Übersetzung von P in Compact Boolean Circuit, $CBC(P)$.
3. Berechnung von $SBP(P)$, Symmetry Breaking Prädikat für P.
4. Transformation von $CBC(P) \wedge SBP(P)$ in $CNF(P)$ (konjunktive Normalform).
5. Anwendung von einem SAT-Solver für $CNF(P)$. Falls erfüllbar, dann Interpretation des Modells als eine Instanz von P.

Die letzten drei Schritte sind die Standardprozeduren aus der Literatur [13, 47, 58]. Die ersten zwei wurden speziell für Kodkod an MIT (CSAIL) entwickelt [54, 53].

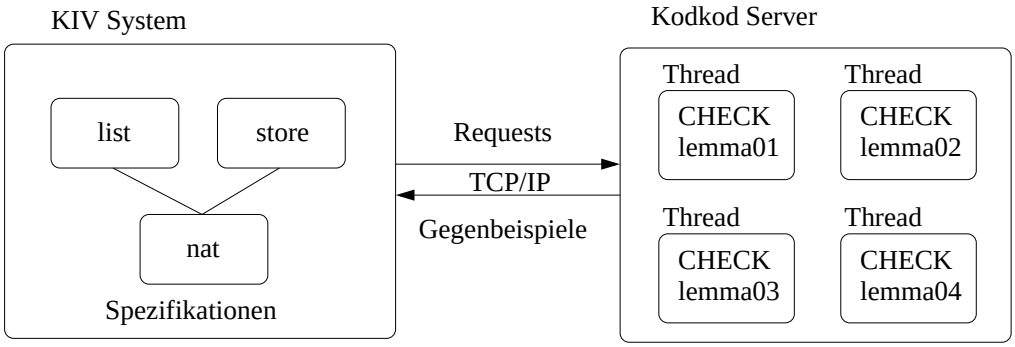


Abbildung 8.2: Kodkod Service in KIV.

8.2 Kodkod Service in KIV

Kodkod ist als ein Service in KIV System integriert. Der Service wird aktiviert sobald im Projektverzeichnis in KIV in der Datei *config* die Kodkod Option ausgewählt ist: `use kodkod`. Der gestartete Java Process kommuniziert mit KIV System über TCP/IP. Sobald auf einer KIV Spezifikation *Work On* gemacht wird, speichert KIV alle notwendigen Informationen zu der Spezifikation als ein PPL-Objekt in eine Datei und schickt ein *INIT* Request an den Kodkod Server. Auf der Server Seite wird die Spezifikation eingelesen und in die Kodkod Sprache übersetzt. Der Benutzer kann auf die ausgewählten Lemmas die Strategie *Kodkod Check* anwenden. KIV speichert das Lemma als ein PPL-Objekt und schickt ein *CHECK* Request an den initialisierten Server. Server startet einen Thread, der für das ausgewählte Lemma die Kodkod API Prozedur `Solution solve(Formula formula, Bounds bounds)` ausführt. Nach einer erfolgreichen Gegenbeispielberechnung wird das Gegenbeispiel in einer benutzerfreundlichen Form abgespeichert und an das KIV System geschickt. KIV zeigt das Gegenbeispiel in einem Fenster.

Das Gegenbeispiel zeigt die Belegung der Variablen und die Funktionstabellen. Z.B. das gefundene Kodkod Gegenbeispiel für den Fehler aus der Fallstudie über die Listen von Intervallen (Kapitel 5) sieht wie folgt:

```
counterexample for INV:

variables assignment:
n=1
ivl2=(0 × 1) + (2 × 2) + []
ivl1=(0 × 0) + (2 × 2) + []

MODEL:

function table for insert : ivlist x nat -> ivlist
------------------------------------------------------------------------
(0 × 0) + (2 × 2) + []  1  (0 × 1) + (2 × 2) + []
...
========================================================================
```

```
function table for R : ivlist -> bool
--------------------------------------------------------------------------------
(0 × 0) + (2 × 2) + []
(2 × 2) + []
[]
================================================================================

function table for + : interval x ivlist -> ivlist
--------------------------------------------------------------------------------
...

statistics
p cnf 75087 236872
primary variables: 764
translation time: 5435 ms
solving time: 885 ms
```

Die Konstruktorterme verraten die Werte für die Variablen $n, ivl1, ivl2$, die das Lemma *INV* falsifizieren. Früher mussten die Konstruktorterme aus dem Graphen ausgelesen werden, indem man die Selektorfunktionen verfolgt hat. Nun geschiet dies automatisch und die zusammengebauten Konstruktorterme werden lesbar angezeigt. An Stelle eines Atoms, das eine Konstante repräsentiert, wird die Konstante angezeigt. Unten werden die Funktionstabellen für alle spezifizierten Operationen angezeigt. Kodkod sortiert die angezeigten Funktionstabellen nach dem folgenden Prinzip:

1. Operationen, die im Lemma vorkommen
2. unterspezifizierten (anonymen) Operationen
3. alle anderen

Ohne spezielle Einstellung werden bei der Fehlersuche die Schranken inkrementell erhöht $2, 3, 4, \ldots$ bis entweder das Gegenbeispiel gefunden wird oder ein Timeout eintritt (nach 5 Minuten) und der Thread wird beendet. Es besteht für den Benutzer die Möglichkeit die Parameter für die Suche zu konfigurieren. Im Verzeichnis der aktuellen Spezifikation wird das Verzeichnis *kodkod* angelegt, wo die Kodkod relevanten Dateien liegen. Wenn man eine längere Suche braucht, z.B. 10 Minuten, dann wird in Kodkod Verzeichnis die Datei *time* mit der gespeicherten Zahl 600 (Sekunden) angelegt. Für größere Spezifikation ist es sinnvoll für effizientere Suche die Schranken für die einzelnen Sorten speziell vorzugeben. Dafür wird die Datei *scope* mit den Schranken angelegt. Z.B. für viele gefundene Fehler in der SmacOS Fallstudie wurden die Schranken vordefiniert:

```
all,3
nat,4
file,4
filesystem,5
```

Für die aufgelisteten Sorten werden spezielle Schranken verwendet und für alle anderen die Schranke 3.

Features.

In KIV besteht die Möglichkeit Axiome oder Lemmas mit Flags (*features*) zu markieren, z.B. *ls* (*local simplifier*). Damit der Benutzer in der Lage ist die Kodkod Behandlung von KIV Axiome zu steuern wurden folgende neue Features eingebaut:

1. *nokodkodax*: Das Axiom wird ignoriert.
2. *kodkodax*: Das Lemma wird als ein Axiom von Kodkod aufgenommen.
3. *kodkodaxstrong*: Das Axiom wird besonders nach Kodkod übersetzt (*starke* Variante).
4. *kodkodaxweak*: Das Axiom wird besonders nach Kodkod übersetzt (*schwache* Variante).

Das Flag *nokodkodax* ist sehr praktisch wenn KIV Spezifikation eine nicht kompatible Definition enthält. Somit wird das Axiom für Kodkod "deaktiviert" ohne dass die KIV Spezifikation geändert werden muss.

Um eine alternative kompatible Definition hizuzufügen wird ein Lemma definiert und mit dem Flag *kodkodax* markiert. Dabei bleibt die KIV Spezifikation wieder unverändert. Das markierte Lemma wird von Kodkod als ein Axiom behandelt.

Die Übersetzung von KIV Axiomen nach Kodkod setzt voraus, dass die Axiome bestimmte Form haben. Für Funktionsdefinitionen (siehe Definition 8 im Abschnitt 4.3.2):

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

und für die Prädikatendefinitionen

$$\forall \underline{v}.\ P(\underline{v}) \leftrightarrow \varphi$$

Die Transformation τ, siehe Definition 8, wird dabei benutzt die oberen KIV Axiome in die folgende Form zu übersetzen:

$$\Phi \equiv \forall \underline{x}, y.\ F(\underline{x}, y) \leftrightarrow \mathcal{Q}v_1.\ \ldots \mathcal{Q}v_k.\ \chi(\underline{v}, \underline{x}, y)$$

Manchmal haben KIV Axiome aber eine abweichende Form. Z.B. die Funktionen $last : list \rightarrow elem$, $butlast : list \rightarrow list$, $new : store \rightarrow elem$ sind auf eine andere Weise spezifiziert:

$$x \neq [] \rightarrow x.butlast + x.last = x$$
$$\neg new(st) \in st$$

Um die Axiome in beliebiger Form nach Kodkod zu übersetzt werden sie mit Flags *kodkodaxstrong* bzw. *kodkodaxweak* markiert. Wir definieren eine spezielle Transformation die *stark* oder *schwach* durchgeführt wird. Bei dieser Transformation gelten die Korrektheitsüberlegungen nicht mehr und es kann eventuell zu falschen Gegenbeispielen kommen (*spurious counter examples*).

Definition 20 *[Spezielle Transformation]*
Für eine Formel φ werden die starke Transformation τ_{strong} und die schwache Transformation τ_{weak} mittels der strukturellen Induktion über Aufbau von φ gemacht:

- *Operator* $\wedge$
  - $\tau_{strong}(\varphi_1 \wedge \varphi_2) = \tau_{strong}(\varphi_1) \wedge \tau_{strong}(\varphi_2)$
  - $\tau_{weak}(\varphi_1 \wedge \varphi_2) = \tau_{weak}(\varphi_1) \wedge \tau_{weak}(\varphi_2)$
- *Operator* $\vee$
  - $\tau_{strong}(\varphi_1 \vee \varphi_2) = \tau_{strong}(\varphi_1) \vee \tau_{strong}(\varphi_2)$
  - $\tau_{weak}(\varphi_1 \vee \varphi_2) = \tau_{weak}(\varphi_1) \vee \tau_{weak}(\varphi_2)$
- *Operator* $\neg$
  - $\tau_{strong}(\neg\varphi_1) = \neg\tau_{weak}(\varphi_1)$
  - $\tau_{weak}(\neg\varphi_1) = \neg\tau_{strong}(\varphi_1)$
- *Quantor* $\forall$
  - $\tau_{strong}(\forall \underline{v}.\ \varphi_1) = \forall \underline{v}.\ \tau_{weak}(\varphi_1)$
  - $\tau_{weak}(\forall \underline{v}.\ \varphi_1) = \forall \underline{v}.\ \tau_{strong}(\varphi_1)$
- *Quantor* $\exists$
  - $\tau_{strong}(\exists \underline{v}.\ \varphi_1) = \exists \underline{v}.\ \tau_{strong}(\varphi_1)$
  - $\tau_{weak}(\exists \underline{v}.\ \varphi_1) = \exists \underline{v}.\ \tau_{weak}(\varphi_1)$
- *Prädikat* $P(t_1, \ldots, t_n)$
  - $\tau_{strong}(P(t_1, \ldots, t_n)) = \exists z_1, \ldots, z_n.\ z_1 = t_1 \wedge \ldots z_n = t_n \wedge P(z_1, \ldots, z_n)$
  - $\tau_{weak}(P(t_1, \ldots, t_n)) = \forall z_1, \ldots, z_n.\ z_1 = t_1 \wedge \ldots z_n = t_n \rightarrow P(z_1, \ldots, z_n)$
- *Gleichheit* $t_1 = t_2$
  - $\tau_{strong}(t_1 = t_2) = \exists z_1, z_2.\ z_1 = t_1 \wedge z_2 = t_2 \wedge z_1 = z_2$
  - $\tau_{weak}(t_1 = t_2) = \forall z_1, z_2.\ z_1 = t_1 \wedge z_2 = t_2 \rightarrow z_1 = z_2$

Wenn man τ_{strong} und τ_{weak} mit der ursprünglichen Transformation τ vergleicht, dann stellt man Folgendes fest. Die Axiome werden mit τ_{strong} und die Theoreme mit τ_{weak} übersetzt. Bei Theoremen gilt $\tau_{weak}(\varphi) = \tau(Prenex(\varphi))$.

Die Standardübersetzung der Axiome ist etwas komplizierter. Prädikatendefinitionen bzw. Funktionsdefinitionenwerden werden wie folgt übersetzt:

$$\forall \underline{v}.\ P(\underline{v}) \leftrightarrow Prenex(\tau_{strong}(\varphi))$$
$$\forall x, y.\ F(x, y) \rightarrow Prenex(\tau_{strong}(\varphi_1 \wedge x = t_1 \wedge y = t_2 \vee \ldots))$$

Bei dem Axiom für *last,butlast* wird τ_{strong} verwendet:

$$\tau_{strong}(x \neq [] \rightarrow x.butlast + x.last = x) =$$
$$x \neq [] \rightarrow (\exists z_1.\ z_1 = x.butlast + x.last \wedge z_1 = x)$$

Das Axiom verlangt impliziert, dass das endliche Teilmodell gegen das Präfix (*butlast*) abgeschlossen ist. Für $x = []$ sind *last,butlast* unterspezifiziert und für $x \neq []$ muss insbesondere immer das Präfix im Modell enthalten sein.

Das Axiom für *new* wird mit τ_{weak} übersetzt, da man nicht verlangen kann das für jedes *Store* auch immer eine nicht allokierte Adresse im endlichen Teilmodell existiert:

$$\tau_{weak}(\neg new(st) \in st) =$$
$$\neg(\exists a.\ a = new(st) \wedge a \in st =$$
$$\forall a.\ a = new(st) \rightarrow a \notin st$$

Damit liefert *new* im endlichen Teilmodell nicht für jeden *Store* ein Ergebnis zurück. Meistens wird die starke Übersetzung benutzt. Nur wenn das Axiom Funktionen benutzt, gegen die die endlichen Modelle eventuell nicht abgeschlossen sind, ist die schwache Variante zu verwenden.

8.3 Inkrementelle Modellgenerierung mit Kodkod

Der große Vorteil von Kodkod ist die Möglichkeit die bereits berechneten Teillösungen zusätzlich zu den Constraints in die neuen Problemstellungen zu integrieren. Uzuncaova et al. [56] zeigt einen inkrementellen Ansatz bei der Behandlung von Constraints in deklarativen Spezifikationen. Es wird eine *optimale* Priorisierung für die Constraints bestimmt und die Instanzen schrittweise berechnet. Damit wird die Berechnung in Teilschritte zerlegt und viel effizienter durchgeführt. Der Ansatz ist als ein Werkzeug Kato realisiert.

Motiviert durch die Arbeit von Uzuncaova et al. [56] untersuchen wir die Möglichkeiten für einen ähnlichen Ansatz im Kontext von algebraischen Spezifikationen in KIV. Das Ziel ist ein automatisiertes Vorgehen bei der inkrementellen Generierung der endlichen Modelle für die KIV Spezifikationen mittels Kodkod.

Betrachten wir eine kleine Spezifikation der Listen aus dem Abschnitt 4.3.2, siehe Abbildung 4.7. Sie enthält die Generiertheitsklauseln für den freien Datentyp

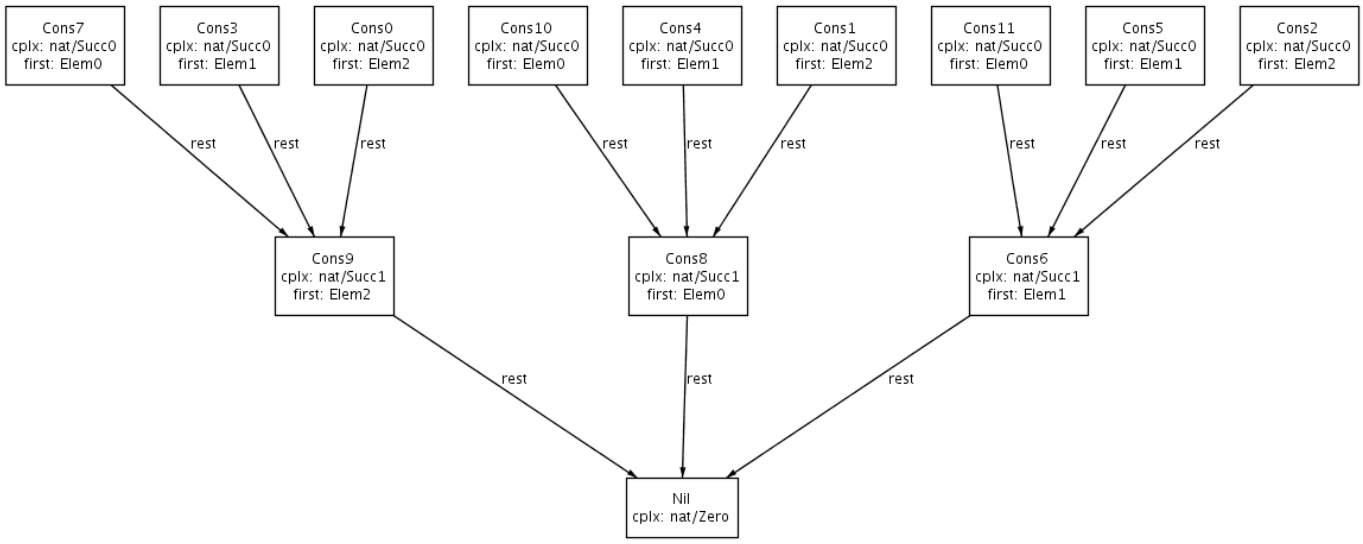


Abbildung 8.3: Endliches Grundmodell mit 7 Listenatomen und 2 Elementen.

List und die Definitionen von zwei Operationen: *append* und *reverse*. Anstatt wie vorher mit Alloy ein endliches Modell in einem Schritt zu generieren wird die Berechnung in drei Teilschritte zerlegt. Zuerst wird das Grundmodell der Listen berechnet, das nur Selektorfunktionen enthält, siehe Abbildung 8.3. Hier wird das endliche Universum deklariert, das aus Atomen der Sorten *Elem* und *List* besteht:

$$ELEM = \{Elem0, Elem1, Elem2\}$$
$$LIST = \{Nil, Cons0, Cons1, \ldots, Cons11\}$$

Das Ergebnis der Berechnung wird in den Mengen von Tupeln, die die Relationen *FIRST* und *REST* wiedergeben, gespeichert und für den nächsten Schritt als ein Constraint in die Kodkod-Problembeschreibung aufgenommen:

$$FIRST = \{(Cons0, Elem2), \ldots, (Cons11, Elem0)\}$$
$$REST = \{(Cons0, Cons9), \ldots, (Cons11, Cons6)\}$$

Jetzt kann das generierte endliche Modell schrittweise um die anderen Operationen erweitert werden. Dabei müssen die Abhängigkeiten zwischen den einzelnen Schritten berücksichtigt werden: *Grundmodell* $\Leftarrow$ *append* $\Leftarrow$ *reverse*. Dieser Abhängigkeitsgraph wird mit einer syntaktischen Analyse aus den Axiomen der jeweiligen Operationen berechnet. Diese Prozedur wurde für KIV bereits in [37] implementiert. Es baut vor allem darauf auf, dass die KIV Spezifikationen Hierarchie persistent sind. D.h., jedes Modell von unteren Spezifikationen kann zu einem Modell der oberen erweitert werden. In [37, 2] werden automatischen Beweiser (SPASS, Protein und andere) in das KIV System integriert und die

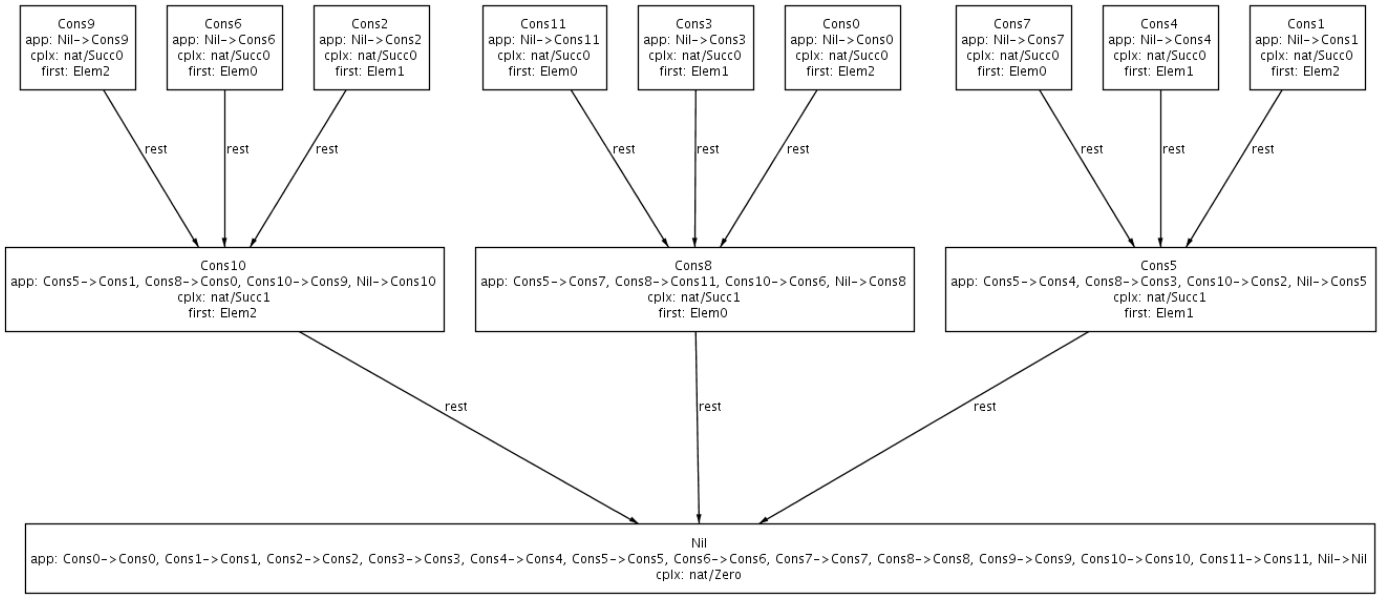


Abbildung 8.4: Erweiterung um die Operation *append*.

Eigenschaft der Hierarchiepersistenz ausreichend für eine korrekte Partinierung ist. Für Kodkod muss zusätzlich noch beachtet werden, dass bei der inkrementellen Modellberechnung $\mathcal{M}_1 \Rightarrow \ldots \Rightarrow \mathcal{M}_n$ die notwendige Abgeschlossenheit der endlichen Teilmodelle berücksichtigt wird. Wenn in $\mathcal{M}_i$ eine Operation f dazukommt, die #-Abgeschlossenheit bei Listen für die Kompatibilität benötigt, dann müssen Listen in $\mathcal{M}_1, \ldots, \mathcal{M}_{i-1}$ auch #-abgeschlossen berechnet werden.

Die Tabelle 8.1 zeigt die Laufzeiten bei der Berechnung eines endlichen abgeschlossenen Modells der vorgegebenen Größe ($|L| \leq 15$, $|E| = 2$) für die Spezifikation der Listen auf drei verschiedene Weisen: mit Alloy (siehe Abbildungen 8.3,8.4,8.5), mit Kodkod (gleiches Vorgehen wie bei Alloy: alles in einem Schritt berechnen), mit Kodkod (inkrementell neue Operationen dazuberechnen, vorherige Berechnungen wiederverwenden). Wie erwartet bringt das inkrementelle

Sorten	Schranken	Ops	#Clauses	#Vars	Zeit
list, elem	$\|L\| \leq 15$, $\|E\| = 3$	Grundmodell	1 x 10^5	5 x 10^4	4 s
	$\|L\| \leq 15$, $\|E\| = 3$	{app}	2 x 10^7	5 x 10^6	20 min
	$\|L\| \leq 15$, $\|E\| = 3$	{app,rev}	2 x 10^7	6 x 10^6	35 min
list, elem	$\|L\| \leq 15$, $\|E\| = 3$	Grundmodell	4 x 10^3	2 x 10^3	1 s
	$\|L\| \leq 15$, $\|E\| = 3$	{app}	1 x 10^6	4 x 10^5	3 min
	$\|L\| \leq 15$, $\|E\| = 3$	{app,rev}	2 x 10^6	5 x 10^5	20 min
list, elem	$\|L\| \leq 15$, $\|E\| = 3$	Grundmodell	4 x 10^3	2 x 10^3	1 s
	$\|L\| \leq 15$, $\|E\| = 3$	{app}	1 x 10^5	4 x 10^4	8min
	$\|L\| \leq 15$, $\|E\| = 3$	{app,rev}	5 x 10^3	5 x 10^3	20 s

Tabelle 8.1: Vergleich der Versuche: Modellgenerierung in einem Schritt mit Alloy, in einem Schritt mit Kodkod, inkrementell mit Kodkod

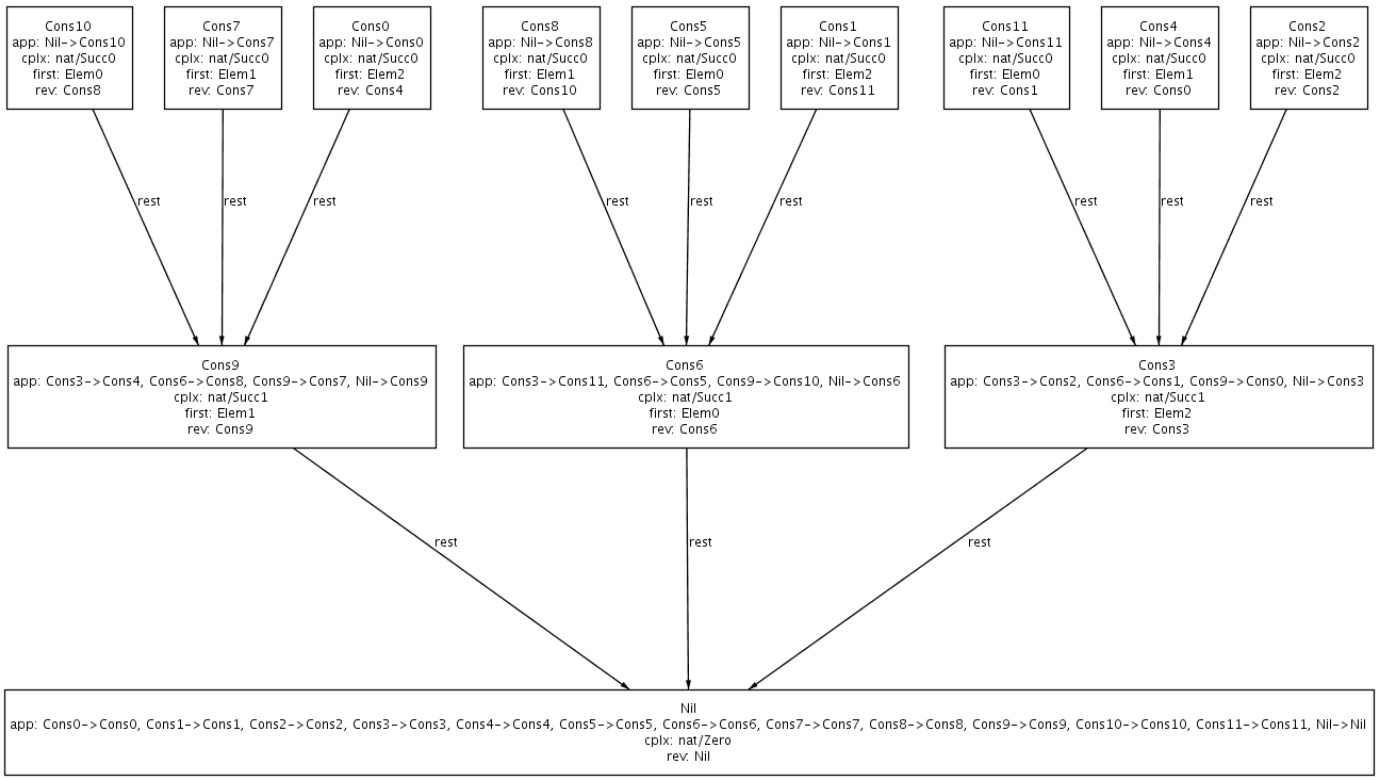


Abbildung 8.5: Erweiterung um die Operation *reverse*.

Vorgehen dramatische Verbesserungen, siehe die letzten drei Zeilen in der Tabelle 8.1. Ein weiterer Unterschied zwischen den Berechnungen ist die effizientere Kodierung in die CNF bei Kodkod und dadurch schnelleres SAT-Lösen. Z.B., für die Berechnung des Grundmodells mit Alloy wird ein SAT-Problem mit 10^5 Klauseln in 4 Minuten generiert und gelöst. Im Gegensatz bei der Anwendung von Kodkod wird ein SAT-Problem mit 30 000 Klauseln in einer Sekunde generiert und gelöst.

8.3.1 Partitionierung der Spezifikation

Die Feststellung der Reihenfolge der Partitionierung der Spezifikation wird komplizierter sobald man größere Spezifikationen betrachtet. Die Spezifikationen können viele Datentypen und Operationen enthalten, mit beliebigen Abhängigkeiten untereinander. Betrachten wir die Fallstudie aus dem Kapitel 6 als Beispiel. Folgende Sorten kommen hier vor:

PID, Ref (*nicht generiert*)
RefList, PID-or-None, Cell, Integer-Inf (*freie*)
Integer, Heap, IntSet (*nicht freie*)

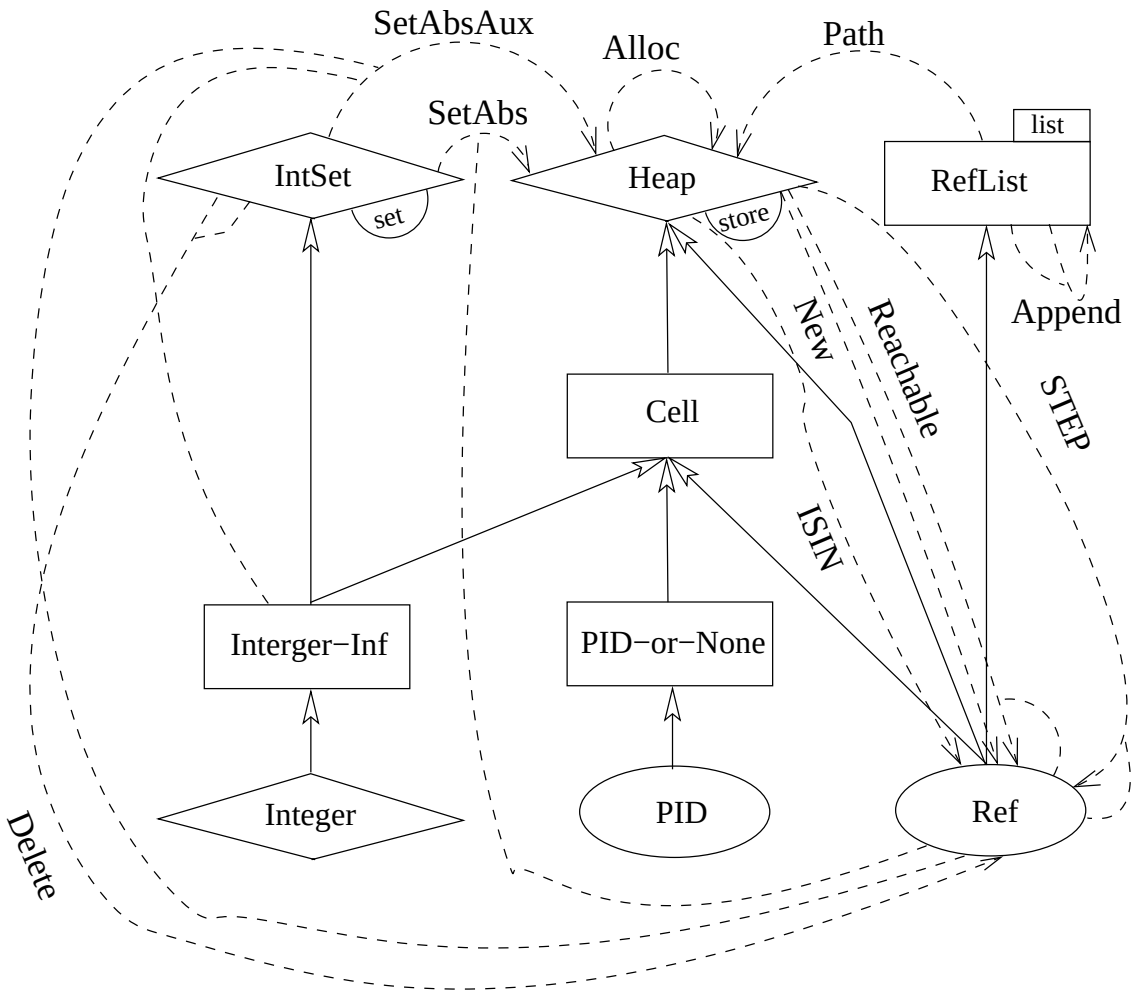


Abbildung 8.6: Hierarchie der Datentypen und die Abhängigkeiten der Operationen.

Die Spezifikationshierarchie gibt bereits den ersten Hinweis für die Partitionierungsreihenfolge. Im ersten Schritt werden die endlichen Grundmodelle der vorgegebenen Größen für die rekursiven Datentypen unter Berücksichtigung der Abhängigkeiten berechnet:

1 *Integer*
2 *IntSet, Integer-Inf (nicht-interpretiert)*
3 *Heap, Cell (nicht-interpretiert), Ref*
4 *RefList, Ref*

Bei der Berechnung des Grundmodells für *IntSet* werden die nicht-interpretierten Atome der Sorte *Integer-Inf* im Modell verwendet, d.h. sie besitzen noch keine Bedeutung, da die entsprechenden Relationen und Atome der Sorte *Integer*

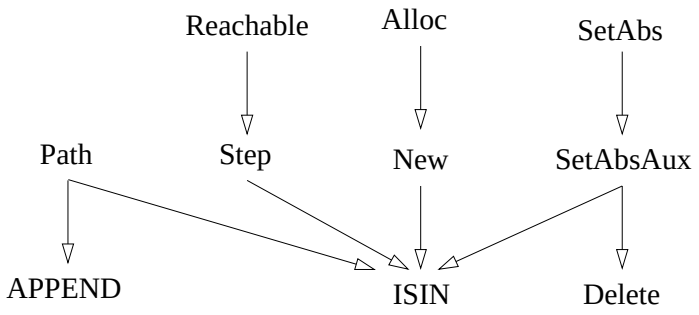


Abbildung 8.7: Partielle Ordnung auf der Menge der Operationssymbole.

fehlen. Nun werden Schrittweise die Operationen dazuberechnet. Die Priorisierung resultiert wieder aus der Spezifikationshierarchie: zuerst werden die "lokalen" Operationen betrachtet, die nur die benachbarten Datentypen involvieren. Anschließend betrachtet man die Operationen, in denen die Datentypen aus verschiedenen Grundmodellen vorkommen. Dabei werden die Abhängigkeiten berücksichtigt, siehe Abbildung 8.6.

Das Ziel ist die Berechnung einer partielle Ordnung[1] auf der Menge von Operationssymbolen *Ops* aufbauen. Die Blätter (keine ausgehenden Kanten) sind die Operationen, die nur Konstruktoren verwenden oder unspezifizierten Operationen. Für unseren Beispiel erhält man folgendes Diagramm, siehe Abbildung 8.7. Aus der Ordnung wird die Reihenfolge, in der die Operationen schrittweise mit Kodkod berechnet werden, bestimmt. Z.B., für die Berechnung der Operation *SetAbsAux* werden zuerst die notwendigen Datentypen bestimmt: $\{Heap, IntSet, Integer, Cell\}$. Dann werden im Diagramm 8.7 alle Nachfahren von *SetAbsAux* ermittelt: $ISIN : Heap \times Ref,\ Delete : Set \times Integer \rightarrow Set$. Für die Berechnung von *SetAbsAux* müssen vorher die folgenden Teile berechnet werden:

- Grundmodelle vorgegebener Größe für $\{Heap, IntSet, Integer, Cell\}$ und
- die Operationen *ISIN, Delete*

Falls es in dem oberen Graphen Zyklen gibt (gegenseitige Abhängigkeiten), dann werden die starken Zusammenhangskomponenten bestimmt und die betreffenden Operationen immer gleichzeitig behandelt. Diese Berechnungen sind bereits in der entsprechenden Prozedur in Rahmen der früheren Arbeit [37] zur Integration der automatischen Beweisern implementiert.

8.3.2 Aktualisierungen

Bei der Kombinierung der Grundmodelle von *Heap* und *Cell* stoßen wir auf ein prinzipielles Problem, das die das inkrementelle Vorgehen bei Operationen, die die beide Datentypen verwenden, verhindert. Z.B. die Operation *SetAbsAux*.

[1] Vorausgesetzt gibt es keine gegenseitig abhängigen Definitionen.

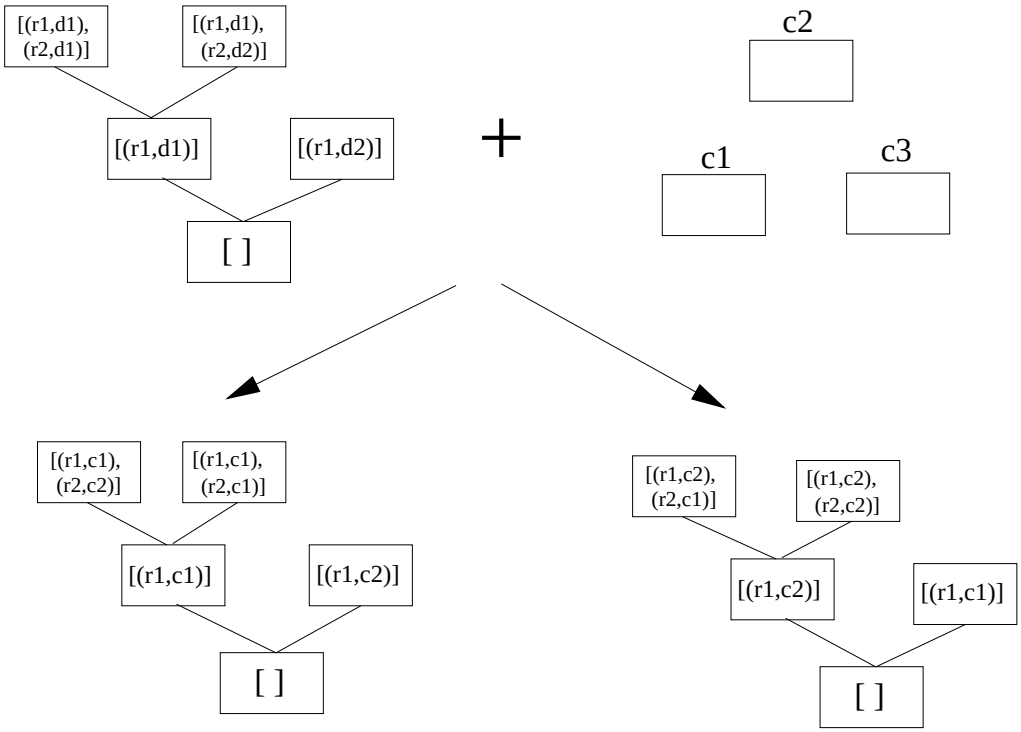


Abbildung 8.8: Möglichkeiten bei der Kombination zweier endlichen Grundmodelle.

Betrachten wir ein endliches Termmodell von *Heap*, wo *Cell* erstmal nicht interpretiert ist, und ein endliches Termmodell für *Cell*. Es gibt viele Möglichkeiten wie die beiden Grundmodelle zu einem Modell kombiniert werden können, siehe Abbildung 8.8. Die kombinatorisch vielen Möglichkeiten die Modelle zu kombinieren stellt ein Problem für die Fehlersuche dar. Die Entscheidung eine bestimmte Zuordnung zu wählen schränkt den Suchraum erheblich ein. Es besteht die Gefahr, das dabei viele Fehler verloren gehen und unentdeckt bleiben. Um eine möglichst maximalen Abdeckung der endlichen Teilmodelle zu erreichen verzichten wir an der Stelle auf das inkrementelle Berechnung. D.h. solche Operationen werden nicht inkrementell berechnet.

8.4 Zusammenfassung

Die Kodkod Integration in KIV hat sehr viele Möglichkeiten für die praktische Evaluation des Ansatzes ermöglicht. Vorher müssten die Alloy Spezifikationen manuell eingetippt werden. Jetzt wird automatisch aus der KIV Spezifikation ein Kodkod Problem erstellt und gelöst. Dem Benutzer bleibt nur der analytische Aufwand für die Kompatibilitätsüberprüfungen nicht erspart. Die praktischen Erfahrungen mit mehreren Fallstudien haben gezeigt, daß dies informell gelöst wird und nicht aufwändig ist. Die Kodkod Berechnungen werden auf einem externen Prozess als Threads ausgeführt und beeinflussen damit das KIV sy-

stem minimal. Die Schnittstelle zum Kodkod Service erlaubt es die Suchweite explizit einzustellen und die oberen Zeitschranken für die Fehlersuche zu definieren. Durch die *feature* Flags *kodkodax, nokodkodax, kodkodaxstrong, kodkodaxweak* für die Axiome und Lemmas in KIV Spezifikationen kann der Benutzer die spezielle Behandlung für bestimmte Axiome konfigurieren. Da ein Kodkod Gegenbeispiel als Menge von Tupeln sehr schlecht lesbar ist, wird es in eine benutzerfreundliche Form transformiert und angezeigt. Dabei wird die Belegung der Variablen im falschen Lemma mit Konstruktortermen gezeigt, sowie die Funktionstabellen für die Operationen.

Die ersten Untersuchungen zum inkrementellen Ansatz haben positive Ergebnisse gebracht. Die Technik steigert die Effizienz der Fehlersuche und weist bessere Skalierbarkeit der Methodik auf. Dabei wird das Prinzip ausgenutzt, daß die bereits gemachten Berechnungen gespeichert und für die späteren Berechnungen wiederverwendet werden können. Dies betrifft nicht nur die Fälle der Erweiterungen um Operationen (*enrichment*) sondern auch das Hinzufügen von weiteren Datentypen in die analysierte KIV Spezifikation. Die Umsetzung und die Integration in das KIV System ist eine vielversprechende Aufgabe für die Zukunft.

Kapitel 9

Zusammenfassung

9.1 Resultate und Erfahrungen

Abstrakte Datentypen eignen sich sehr gut für die High-Level Spezifikation von Software. Die algebraischen Spezifikationen von abstrakten Datentypen bringen durch die Strukturierung gute Überschaubarkeit des Entwurfs und erlauben die Ausdrucksstärke der Logik der ersten Stufe. Wir haben einen Ansatz zur Fehlersuche mittels endlichen Modellsuche entworfen und auf mehreren Fallstudien evaluiert. Es wurde eine prototypische Implementierung in Java gemacht, die den Modellgenerator und Constraint Solver Kodkod benutzt.

Wir haben eine Methodik [12] entwickelt die für die freien Datentypen automatisch und für die nicht-freien Datentypen automatisiert die endliche Modellsuche ermöglicht. Der ursprüngliche Ansatz von Kuncak und Jackson [26] funktionierte nur für die freien Datentypen und eine Sprache, die auf Selektorfunktionen beschränkt war. Wir haben ihn auf beliebige rekursive oder nicht-rekursive Operationen [11], sowie nicht-freien Datentypen [10] erweitert. Es wurden die formalen Kriterien aufgestellt, die die Korrektheit der Erweiterung garantieren. In dieser Arbeit wird die Klasse der analysierbaren Formeln formal definiert (UBE Formeln). Die meisten Theoreme in betrachteten KIV Fallstudien gehörten zu dieser Klasse und waren damit für die Widerlegung mittels der endlichen Gegenbeispielsuche geeignet. Der Ansatz zur Spezifikation der endlichen Strukturen, die isomorph zu endlichen Teilmodellen der unendlichen Termalgebren sind, liegt im Kern der Arbeit. Dabei wird vorausgesetzt, daß die endlichen Teilstrukturen abgeschlossen bezüglich einer festen Präordnung auf der Domäne der unendlichen Struktur sind. Diese Abgeschlossenheit ist für die Erhaltung der Semantik der Operationsdefinitionen beim Übergang auf die endlichen Modelle entscheidend.

Die Effizienz, mit der Kodkod die falschen Lemmas aufgedeckt hat, ist beeindruckend. Obwohl die Vergrößerung der Suchweite zu einer Zustandsexplosion führt, waren in den meisten Fällen relativ kleinen oberen Schranken ausreichend. Um eine bessere Skalierbarkeit zu erreichen haben wir eine inkrementelle Technik zur Modellgenerierung entwickelt. Die Technik führt die Modellberechnungen iterativ aus und ermöglicht somit effizientere und schnellere Modellsuche.

Um die praktische Relevanz der Methodik nachzuweisen haben wir sie auf die Fallstudien realistischer Größe angewandt. Die Stärken der automatischen Fehlersuche waren sofort zu erkennen. In Sekundenschnelle konnten Dutzende von Lemmas überprüft werden. Für die Falschen Theoreme wurden endliche Gegenbeispiele generiert, die sie widerlegten. Es bleibt aber dem Benutzer nicht erspart die Fehlerursache zu finden und zu beheben. Allerdings, hilft das Werkzeug dabei indem es die Belegung der Variablen im falschen Lemma anzeigt, die das Lemma falsifiziert.

9.2 Ausblick

Wir betrachten die möglichen weiteren Entwicklungen dieser Arbeit aus drei Sichten: Verbesserung der Realisierung der Technik zur Fehlersuche, Erweiterung der Kernmethodik auf weitere formale Sprachen, Ausweitung der Anwendungsdomäne für die höhere Effizienz und Qualität in der Softwareentwicklung.

Verbesserung der Realisierung

Der Ansatz zur Inkrementellen Modellgenerierung in Kapitel 8 wurde nur prototypisch in Kodkod realisiert. Ein Werkzeug wie Kato [56], der automatisch die Reihenfolge für schrittweise Modellkonstruktion berechnet, bleibt noch umzusetzen. Es wurde experimentell gezeigt, daß die Größe der SAT-Instanz exponentiell mit der Modellgröße wächst. Vor allem für die Testfallgenerierung ist es interessant größere Instanzen zu berechnen. Um das zu erzielen, muss man den inkrementellen Ansatz in Kapitel 8 um die inkrementelle Berechnung der Domäne der Termalgebra erweitern. Die Kompatibilitätsüberprüfung der Operationsdefinitionen nimmt bei größeren Fallstudien viel Zeit in Anspruch, siehe Kapitel 7. Eine Realisierung der automatischen Prozedur in Abschnitt 4.3.5 muss noch implementiert werden.

Erweiterung der Kernmethodik

Das KIV System hat eine vielzahl an Formalismen zur formalen Spezifikation von Software. Diese Arbeit behandelt den grundlegenden Formalismus: algebraischen Spezifikationen der abstrakten Datentypen mit Sprache eingeschränkt auf die Logik der ersten Stufe. Als nächster Schritt ist es natürlich eine Erweiterung auf die Logik der höheren Stuffe (HOL) vorzunehmen. In vielen Fallstudien mit zustandsbasierten Spezifikationen werden die bekannten Formalismen Temporale Logik (TL) und Abstrakte Zustandsmaschinen verwendet. Diese wiederum verwenden Datentypen für die Beschreibung des Zustands. Als eine interessante Weiterentwicklung der Methodik in dieser Arbeit wäre eine Kombination von Alloy mit einem modernem Modell Checker, z.B. SMV [49]. Dabei konnte Alloy zur Berechnung der Abstraktion und SMV zur Zustandsexploration verwendet werden.

Ausweitung der Anwendungsdomäne

Die klassische Anwendung der Technik in dieser Arbeit ist die effiziente Aufdeckung von falschen Lemmas in algebraischen Spezifikationen. Alternativ können damit die Fehler in der Spezifikation entdeckt werden. Es ist für den Benutzer vor allem bei größeren Spezifikationen aufwendig die Fehlerursache zu lokalisieren. Nun können mit der neuen Technik, die in Kodkod realisiert wurde [53], die

Axiome gefunden werden, die zur Falsifizierung eines Theorems beitragen. Diese zusätzliche Information kann dem Benutzer die Fehlerlokalisierung erleichtern.

Anhang A

Spezifikationen

A.1 KIV Spezifikationen

A.1.1 KIV Bibliothek: Listen

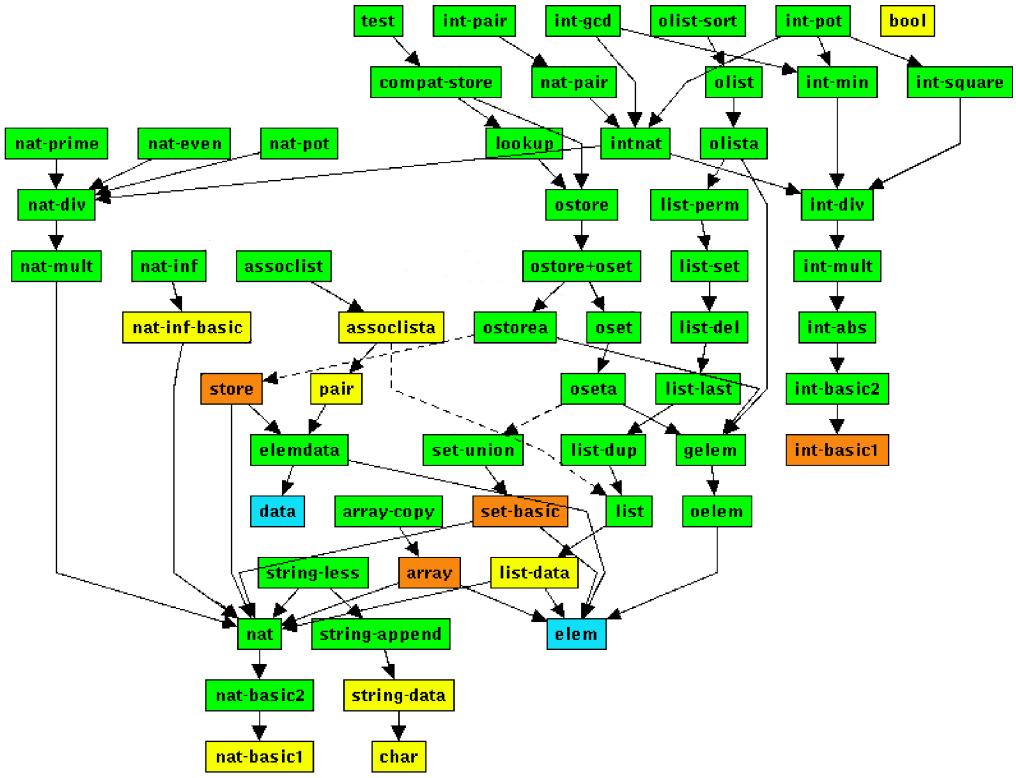


Abbildung A.1: Entwicklungsgraph der KIV Bibliothek.

Das Bibliothek enthält die essentiellen Datentypen, die in KIV Fallstudien verwendet werden,siehe Abbildung A.1. Wir unterscheiden zwischen drei Arten von Ddatentypen: frei generierten (gelb), nicht-frei generierten (orange) und nicht generierten Datentypen (blau). Alle anderen spezifikationen (grün) sind Erweiterungen der Datentypspezifikationen um beliebige Operationen (Funktionen oder Prädikate).

Der bekannte Beispiel für einen freien Datentyp sind die Listen. Im folgenden sind die KIV Spezifikation *list-data*) und Erweiterungen *list* (einfache Operationen auf Listen), *list-dup* (duplikat-freie Listen), *list-last*, *list-del* (Operation delete), *list-set* (Mengenoperationen auf Listen), *list-perm* (Permutationen), *olist* (geordnete Listen), *olist-sort* (Sortieren) aufgelistet:

list-data =
generic data specification
parameter elem **using** nat
list = []
| . + . **prio** 9 (. .first : elem ; . .rest : list ;) **prio** 9
;
variables x, y, z, x0, y0, z0, x1, y1, z1, x2, y2, z2: list;
size functions # . : list → nat ;
order predicates . < . : list × list;
end generic data specification

---

list =
enrich list-data **with**
functions

. ’	:	elem	→	list	;
. + .	:	list × list	→	list	**prio** 9;
. + .	:	list × elem	→	list	**prio** 9;
. + .	:	elem × elem	→	list	**prio** 9;

predicates . ∈ . : elem × list;

axioms

Nil : [] + x = x;

Cons : (a + x) + y = a + x + y;

One : a ’ = a + [];

Last : x + a = x + a ’;

Two : a + b = a ’ + b ’;

In : a ∈ x ↔ (∃ y, z. x = y + a + z);

end enrich

---

list-dup =

enrich list **with**
functions
. ++ . : list × elem → list **prio** 9 **left**;
rmdup : list → list ;
predicates
dups : list;
disj : list × list;

axioms

rmdup-e : rmdup([]) = [];
rmdup-y : a ∈ x → rmdup(a ' + x) = rmdup(x);
rmdup-n : ¬ a ∈ x → rmdup(a ' + x) = a + rmdup(x);
adjoin-in : a ∈ x → x ++ a = x;
adjoin-notin : ¬ a ∈ x → x ++ a = a + x;
dups : dups(x) ↔ (∃ a, x0, y, z. x = x0 + a + y + a + z);
disjoint : disj(x, y) ↔ (∀ a. ¬ (a ∈ x ∧ a ∈ y));

end enrich

---

list-last =
enrich list-dup **with**
functions
. .last : list → elem ;
. .butlast : list → list ;
butlastn : nat × list → list ;
rev : list → list ;
mklist : elem × nat → list ;
fillfirst : nat × elem × list → list ;
predicates
. ⊑ . : list × list;
. ⊒ . : list × list;

axioms

last : x ≠ [] → x.butlast + x.last = x;
rev-e : rev([]) = [];
rev-r : rev(a ' + x) = rev(x) + a;
mk-len : # mklist(a, n) = n;
mk-elem : a ∈ mklist(b, n) → a = b;
butlastN-base : butlastn(0, x) = x;
butlastN-rec : butlastn(n + 1, x) = butlastn(n, x.butlast);
fillfirst-longer : n ≤ # x → fillfirst(n, a, x) = x;
fillfirst-fill : # x < n → fillfirst(n, a, x) = mklist(a, n - # x) + x;

prefix : x $\sqsubseteq$ y $\leftrightarrow$ ($\exists$ z. x + z = y);

postfix : x $\sqsupseteq$ y $\leftrightarrow$ ($\exists$ z. z + x = y);

end enrich

---

list-del =
enrich list-last **with**
functions

. -l .	:	list × elem	→	list	**prio** 9;
. -1l .	:	list × elem	→	list	**prio** 9;
. -1l .	:	list × nat	→	list	**prio** 9;
. [.]	:	list × nat	→	elem	**prio** 2;
pos	:	elem × list	→	nat	;
. [.]	:	list × nat × elem	→	list	;
sublist	:	nat × nat × list	→	list	;
firstn	:	nat × list	→	list	;
restn	:	nat × list	→	list	;
lastn	:	nat × list	→	list	;
frome	:	list × elem	→	list	;

axioms

del-e : [] -l a = [];

del-y : (a ' + x) -l a = x -l a;

del-n : a $\neq$ b $\rightarrow$ (b ' + x) -l a = b ' + x -l a;

del1-e : [] -1l a = [];

del1-y : (a ' + x) -1l a = x;

del1-n : a $\neq$ b $\rightarrow$ (b ' + x) -1l a = b ' + x -1l a;

delpos-empty : [] -1l n = [];

delpos-base : (a ' + x) -1l 0 = x;

delpos-rec : (a ' + x) -1l n + 1 = a + x -1l n;

get-zero : a ' + x[0] = a;

get-succ : a ' + x[n + 1] = x[n];

pos-e : pos(a, []) = 0;

pos-y : pos(a, a ' + x) = 0;

pos-n : a $\neq$ b $\rightarrow$ pos(a, b ' + x) = pos(a, x) + 1;

put-zero : b ' + x[0, a] = a + x;

put-succ : b ' + x[n + 1, a] = b + x[n, a];

sublist-def : sublist(m, n, x) = firstn(n, restn(m, x));

firstN-zero : firstn(0, x) = [];

firstN-rec : firstn(n + 1, x) = x.first + firstn(n, x.rest);

restN-zero : restn(0, x) = x;

restN-rec : restn(n + 1, x) = restn(n, x.rest);
lastN-zero : lastn(0, x) = [];
lastN-rec : lastn(n + 1, x) = lastn(n, x.butlast) + x.last;
fromE-empty : frome([], a) = [];
fromE-yes : frome(a ' + x, a) = a ' + x;
fromE-no : a ≠ b → frome(a ' + x, b) = frome(x, b);

end enrich

---

list-set =
enrich list-del **with**
functions

. ∪ .	:	list × list	→	list **prio** 9;
. \ .	:	list × list	→	list **prio** 9;
filter	:	list × list	→	list ;

predicates . ⊆ . : list × list;

axioms

subset : x ⊆ y ↔ (∀ a. a ∈ x → a ∈ y);
union : x ∪ y = rmdup(x + y);
diff-e : [] \ y = [];
diff-y : a ∈ y → (a ' + x) \ y = x \ y;
diff-n : ¬ a ∈ y → (a ' + x) \ y = a + x \ y;
filt-e : filter([], y) = [];
filt-y : a ∈ y → filter(a ' + x, y) = a ' + filter(x, y);
filt-n : ¬ a ∈ y → filter(a ' + x, y) = filter(x, y);

end enrich

---

list-perm =
enrich list-set **with**
functions #oc : elem × list → nat ;
predicates

perm	:	list × list;
. ⊆m .	:	list × list;

axioms

oc-e : #oc(a, []) = 0;
oc-y : #oc(a, a ' + x) = #oc(a, x) + 1;
oc-n : a ≠ b → #oc(a, b ' + x) = #oc(a, x);
msubset : x ⊆m y ↔ (∀ a. #oc(a, x) ≤ #oc(a, y));
perm : perm(x, y) ↔ x ⊆m y ∧ y ⊆m x;

end enrich

---

olista =
actualize list-perm **with** gelem **by morphism**

end actualize

---

olist =
enrich olista **with**
functions
ins≤ : elem × list → list ;
merge : list × list → list ;
predicates
ordered≤ : list;
ordered< : list;

axioms

le-e : ordered≤([]);
le-o : ordered≤(a ');
le-r : ordered≤(a ' + b ' + x) ↔ ¬ b < a ∧ ordered≤(b + x);
ls-e : ordered<([]);
ls-o : ordered<(a ');
ls-r : ordered<(a ' + b ' + x) ↔ a < b ∧ ordered<(b + x);
ins-e : ins≤(a, []) = a ';
ins-y : ¬ b < a → ins≤(a, b ' + x) = a + b + x;
ins-n : b < a → ins≤(a, b ' + x) = b + ins≤(a, x);

end enrich

---

olist-sort =
enrich olist **with**
functions sort : list → list ;

axioms

Ordered : ordered≤(sort(x));
Perm : perm(x, sort(x));

end enrich

A.1.2 Nicht-blockierende Mengen

abs =
enrich intset, path **with**
predicates
abs : Ref × heap × intset;
absh : intinf × Ref × heap × intset;
variables
Head0, Head1, Head: Ref;
H0, H1, H2: heap;
pid: PId;
bv, bv': bool;

axioms

abs-def :
abs(Head, H, s)
↔ Head ∈ H
∧ Head ≠ null
∧ H[Head].nxt ≠ null
∧ H[Head].i∞ = -∞
∧ absh(-∞, H[Head].nxt, H, s);

absh-base : absh(i∞, r, H, ∅) ↔ r ∈ H ∧ r ≠ null ∧ H[r].nxt = null
∧ H[r].i∞ = ∞;

absh-rec :
s ≠ ∅
→ (absh(i∞, r, H, s)
↔ r ∈ H
∧ r ≠ null
∧ i∞ < H[r].i∞
∧ H[r].i∞ ≠ ∞
∧ H[r].i∞.i ∈ s
∧ H[r].nxt ≠ null
∧ absh(H[r].i∞, H[r].nxt, H, s – H[r].i∞.i));

end enrich

path =
enrich reflist, newalloc **with**
predicates
path : reflist × heap;
reachable : Ref × Ref × heap;
variables H: heap;

axioms

path-empty : ¬ path([], H);

path-one : path(r ', H) ↔ ¬ ¬ (r ∈ H ∧ r ≠ null);

path-two :
path(r ' + r0 ', H) ↔ ¬ ¬ (r ∈ H ∧ r0 ∈ H ∧ r ≠ null ∧ r ≠ null ∧ H[r].nxt = r0);

path-cons :
path(r ' + r0 ' + x, H) ↔ ¬ ¬ (r ∈ H ∧ r ≠ null ∧ H[r].nxt = r0 ∧ path(r0 ' + x, H));

reachable-def : reachable(r, r0, H) ↔ (∃ x. path(r ' + x, H) ∧ (r ' + x).last = r0);

end enrich

cell =
data specification
using Ref, PidorNone, int-inf
cell = mk (. .i∞ : intinf ; . .pin : PIdorNone ; . .nxt : Ref ;);
variables ce: cell;
end data specification

PidorNone =
data specification
using PId
PIdorNone = ⌜ . ⌝ (. .pid : PId ;)
| none
;
variables pin: PIdorNone;
end data specification

int-inf =
enrich int-inf-basic **with**
predicates . < . : intinf × intinf;

axioms

neginf-less : -∞ < i∞ ↔ i∞ ≠ -∞;

less-inf : i∞ < ∞ ↔ i∞ ≠ ∞;

less-int : ⌜i⌝ < ⌜i0⌝ ↔ i < i0;

not-less-neginf : ¬ i∞ < -∞;

not-inf-less : ¬ ∞ < i∞;

end enrich

int-inf-basic =
data specification
using int-basic1
intinf = ⌜ . ⌝ (. .i : int ;)
| -∞
| ∞

;
variables i∞: intinf;
end data specification

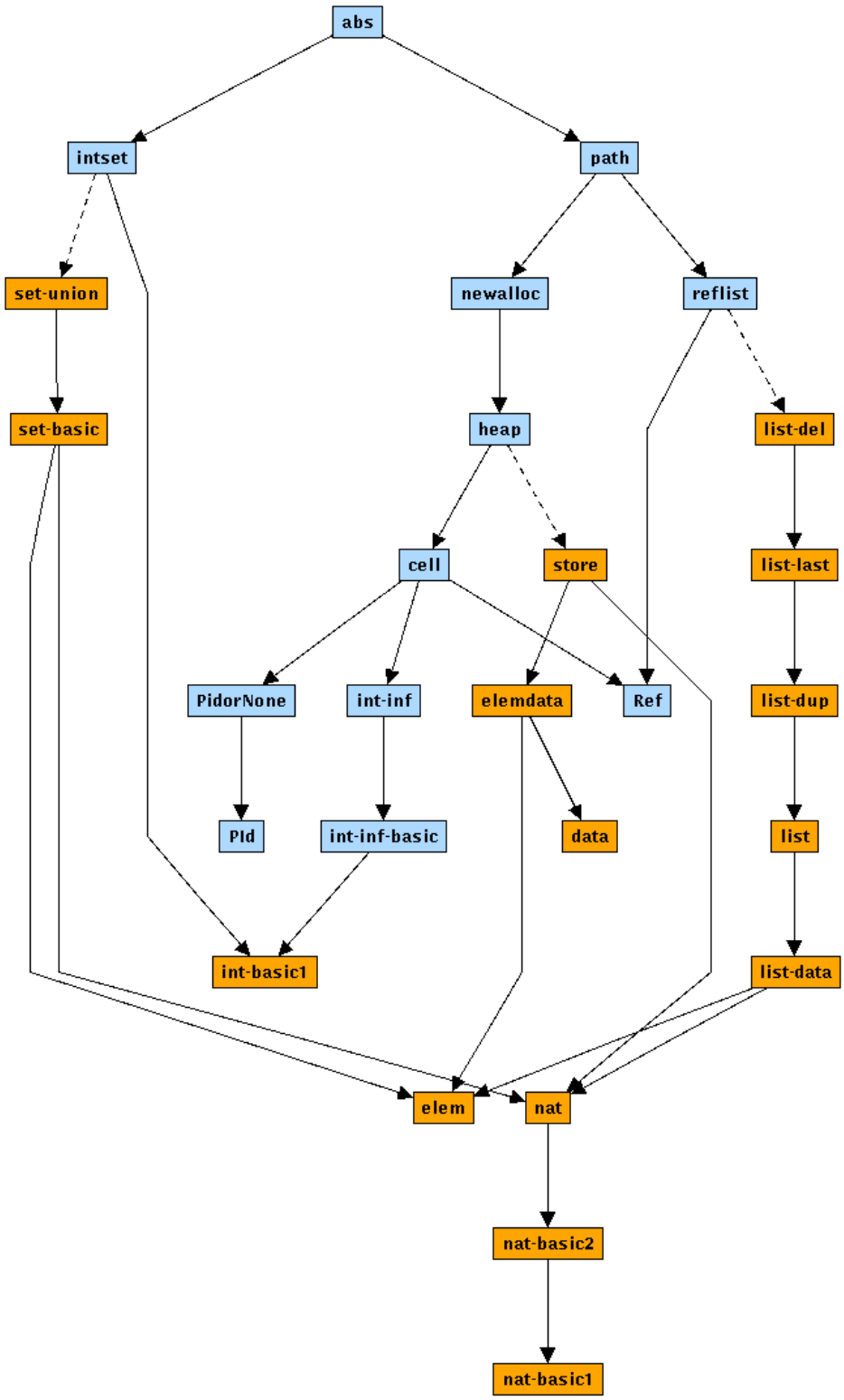


Abbildung A.2: Entwicklungsgraph der Fallstudie über die nicht-blockierenden Mengen.

A.2 Alloy Spezifikationen

A.2.1 KIV Bibliothek: Listen

```
module list_benchmark
open nat

--------------------------
----  signature: list  ----
--------------------------

abstract sig List {

-- list basis
   cplx: one Nat,

-- list functions and predicates

-- #1 specification: list
   app:  List -> lone List,
   lastcons:  Elem -> lone List,
   r_isin: set Elem, -- reversed arguments

-- #2 specification: list-dup
   adjoin:  Elem -> lone List,
   rmdup: lone List,
   disjoin: set List,

-- #3 specification: list-last
   last: lone Elem,
   butlast: lone List,
   r_butlastn: Nat -> lone List, -- reversed arguments
   rev: lone List,
   r_fillfirst: Nat -> Elem -> lone List, -- reversed arguments
   prefix: set List,
   postfix: set List,

-- #4 specification: list-del
   del:  Elem -> lone List,
   del1:  Elem -> lone List,
   delpos:  Nat -> lone List,
   get:  Nat -> lone Elem,
   r_pos: Elem -> lone Nat, -- reversed arguments
   put: Nat -> Elem -> lone List,
   r_sublist: Nat -> Nat-> lone List, -- reversed arguments
   r_firstn: Nat -> lone List, -- reversed arguments
   r_restn: Nat -> lone List, -- reversed arguments
   r_lastn: Nat -> lone List, -- reversed arguments
   frome: Elem -> lone List,

-- #5 specification: list-set
   union:  List -> lone List,
   diff:  List -> lone List,
   filter:  List -> lone List,
   subset: set List,

-- #6 specification: list-perm
   r_oc: Elem -> lone Nat, -- reversed arguments
   perm: set List,
   msubset: set List,
```

```
-- #7 specification: olist
   r_ins: Elem -> lone List, -- reversed arguments
   merge: List -> lone List,

-- #8 specification: olist-sort
   sort: lone List

}

one sig Nil extends List {}

sig Cons extends List {
   first:  Elem,
   rest:  List
}

----------------------------
--   monadic predicates   --
----------------------------

-- #2 specification: list-dup
sig dups in List {}

-- #7 specification: olist
sig le in List {}
sig ls in List {}


----------------------
----  SUA axioms  ----
----------------------

fact Uniqueness {
   all l,l': Cons |
      l.first = l'.first and l.rest = l'.rest => l = l'
}

fact Acyclic {
   no l: Cons | l in l.^rest
}


-- CPLX (term size)
-- KIV view
-- (1) cplx([]) = 0
-- (2) cplx(a + x) = 1 + cplx(x)

fact CPLX {
   cplx = {x: List, n: Nat |
      (x = Nil and n = Zero) or
      some z1: List, a: Elem, m1: Nat | {
         cons[a,z1,x]
         (z1->m1) in cplx
         succ[n,m1]
      }
   }
}

------------------------
----   predicates   ----
------------------------
```

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


-----------------------------
----   signature: elem   ----
-----------------------------

sig Elem {
-- basis
   els: set Elem,

-- #1 specification: list
   mklist01: lone List,
   mklist02: Elem -> lone List,

-- #3 specification: list-last
   mklist: Nat -> lone List
}


fact ORDER_IRREFLEX {
   no a: Elem | (a->a) in els
}

fact ORDER_TRANSITIVE {
   all a,b,c: Elem | (a->b) in els and (b->c) in els => (a->c) in els
}

fact ORDER_TOTAL {
   all a,b: Elem | (a->b) in els or (b->a) in els or a = b
}

--------------------------------
----   s,k - completeness   ----
--------------------------------

-- Finite Generator

fact FiniteGenerate {
   all x: List, a: Elem | lt[x.cplx,Zero.~prede.~prede] => some y: List | cons[a,x,y]
}

fact FiniteClose {
   all x: List | lte[x.cplx,Zero.~prede.~prede]
}

-- nat constants
-- K = 1 (Zero.~prede)
-- K = 2 (Zero.~prede.~prede)


-----------------------------------------
----  axioms for enriched operations   ----
-----------------------------------------

--------------------------------
---   #1 specification: list   ---
--------------------------------
```

```
-- MKLIST01
-- KIV axioms
-- (1)  a ' = a + @
fact MKLIST01 {
   mklist01 = {a: Elem, x: List | cons[a,Nil,x]
   }
}


-- APPEND
-- KIV axioms
-- (1)  Nil ++ x = x
-- (2)  (a + x) ++ y = a + (x ++ y)
fact APPEND {
   app = {x,y,z: List |
      (x = Nil and z=y)
         or
      some z1,z2: List, a: Elem | {
         cons[a,z1,x]
         (z1->y->z2) in app
         cons[a,z2,z]
         }
   }
}


-- LASTCONS
-- KIV axioms
-- (1)  x + a = x + a '
fact LASTCONS {
   lastcons = {x: List, a: Elem, y: List | app[x][mklist01[a]] = y
   }
}


-- MKLIST02
-- KIV axioms
-- (1)  a + b = a ' + b '
fact MKLIST02 {
   mklist02 = {a,b: Elem, x: List | app[mklist01[a]][mklist01[b]] = x
   }
}


-- ISIN
-- KIV axioms
-- a in x <-> exists z1,z2. x = z1 + (a + z2)
fact ISIN {
   r_isin = {x : List, a: Elem | some z1,z2,z3 : List |
      cons[a,z2,z3] and (z1->z3->x) in app
   }
}


--------------------------------------
---   #2 specification: list-dup   ---
--------------------------------------

-- ADJOIN
-- KIV axioms
-- (1) a in x -> x ++ a = x
-- (2) not a in x -> x ++ a = a + x
```

```
fact ADJOIN {
   adjoin = {x: List, a: Elem, y: List |
      (x->a) in r_isin and y = x
      or
      not (x->a) in r_isin and cons[a,x,y]
   }
}


-- RMDUP
-- KIV axioms
-- (1)  rmdup([]) = []
-- (2)  a in x -> rmdup(a ' + x) = rmdup(x)
-- (3) not a in x -> rmdup(a ' + x) = a + rmdup(x)
fact RMDUP {
   rmdup = {x: List, y: List |
      x = Nil and y = Nil
      or
      (some z: List, a: Elem | cons[a,z,x] and (z->a) in r_isin and (z->y) in rmdup)
      or
      (some z1,z2: List, a: Elem | cons[a,z1,x] and not (z1->a) in r_isin and
      (z1->z2) in rmdup and cons[a,z2,y])
   }
}


-- DISJOIN
-- KIV axiom
-- disjoin(x,y) <-> forall a. not (a in x and a in y)
fact DISJOIN {
   disjoin = {x,y: List | all a : Elem |
      not ((x->a) in r_isin and (y->a) in r_isin)
   }
}

/*
-- OutOfMemory: checkable only for K = 1
-- DUPS
-- KIV axiom
-- dups(x) <-> (exist a, x0, y, z. x = x0 + (a + (y + (a + z))))
fact DUPS {
   dups = {x: List | some a: Elem, x0,y,z,z1,z2,z3: List |
      cons[a,z,z1] and z2 = app[y][z1] and cons[a,z2,z3] and x = app[x0][z3]
   }
}
*/
-- DUPS-LIGHT
fact DUPS {
   dups = {x: List | some a: Elem, x0,y0: List |
      x = app[x0][y0] and (x0->a) in r_isin and (y0->a) in r_isin
   }
}


----------------------------------------
---   #3 specification: list-last   ---
----------------------------------------


-- LAST_BUTLAST
-- KIV axiom
-- x != @ -> x .butlast + x .last = x
```

```
fact LAST_BUTLAST {
   all x: List | not x = Nil => x = lastcons[butlast[x]][last[x]]
}


-- BUTLASTN
-- KIV axiom
-- (1) butlastn(0, x) = x
-- (2) butlastn(n + 1, x) = butlastn(n, x .butlast)
fact BUTLASTN {
   r_butlastn = {x: List, n: Nat, y: List |
      n = Zero and y = x
      or
      (some m: Nat | n = m.~prede and (butlast[x]->m->y) in r_butlastn)
   }
}

-- REV
-- KIV axiom
-- (1) rev([]) = []
-- (2) rev(a ' + x) = rev(x) + a
fact REV {
   rev = {x,y: List |
      x = Nil and y = Nil
      or
      (some a: Elem, z1,z2: List | app[mklist01[a]][z1] = x and (z1->z2) in rev and
      lastcons[z2][a] = y)
   }
}


-- MKLIST
-- KIV axiom
-- (1) # mklist(a, n) = n
-- (2) a in mklist(b, n) -> a = b
fact MKLIST {
   mklist = {a: Elem, n: Nat, x: List |
      (x->n) in cplx and
      (all b: Elem | (x->b) in r_isin => b = a)
   }
}



-- FILLFIRST
-- KIV axiom
-- (1) n <= # x -> fillfirst(n, a, x) = x
-- (2) # x < n -> fillfirst(n, a, x) = mklist(a, n - # x) + x
fact FILLFIRST {
   r_fillfirst = {x: List, n: Nat, a: Elem, y: List |
      lte[n,cplx[x]] and y = x
      or
      lt[cplx[x],n] and y = app[mklist[a][sub[n][cplx[x]]]][x]
   }
}



--------------------------------------
---   #4 specification: list-del   ---
--------------------------------------
```

```
-- DEL
-- KIV axiom
-- (1) [] -l a = [];
-- (2) (a ' + x) -l a = x -l a;
-- (3) a != b -> (b ' + x) -l a = b ' + (x -l a)
fact DEL {
   del = {x: List, a: Elem, y: List |
      x = Nil and y = Nil
      or
      (some z: List | cons[a,z,x] and y = del[z][a])
      or
      (some b: Elem, z: List | not a = b and cons[b,z,x] and cons[b,del[z][a],y])
   }
}


-- DEL1
-- KIV axiom
-- (1) @ -1l a = @
-- (2) (a ' + x) -1l a = x
-- (3) a != b -> (b ' + x) -1l a = b ' + (x -1l a)
fact DEL1 {
   del1 = {x: List, a: Elem, y: List |
      x = Nil and y = Nil
      or
      (some z: List | cons[a,z,x] and y = z)
      or
      (some b: Elem, z: List | not a = b and cons[b,z,x] and cons[b,del1[z][a],y])
   }
}


-- DELPOS
-- KIV axiom
-- (1) [] -1l n = []
-- (2) (a ' + x) -1l 0 = x
-- (3) (a ' + x) -1l (n + 1) = a + (x -1l n)
fact DELPOS {
   delpos = {x: List, n: Nat, y: List |
      x = Nil and y = Nil
      or
      (some a: Elem, z: List | cons[a,z,x] and n = Zero and y = z)
      or
      (some a: Elem, z: List, m: Nat | cons[a,z,x] and n = m.~prede and
      cons[a,delpos[z][m],y])
   }
}

-- GET
-- KIV axiom
-- (1) (a ' + x)[0] = a
-- (2) (a ' + x)[n + 1] = x[n]
fact GET {
   get = {x: List, n: Nat, a: Elem |
      not x = Nil and
      (n = Zero and a = x.first
      or
      not n = Zero and a = get[x.rest][n.prede])
   }
}
```

```
-- POS
-- KIV axiom
-- (1) pos(a, []) = 0
-- (2) pos(a, a ' + x) = 0
-- (3) a != b -> pos(a, b ' + x) = pos(a, x) + 1
fact POS {
   r_pos = {x: List, a: Elem, n: Nat |
      x = Nil and n = Zero
      or
      (some z: List | cons[a,z,x] and n = Zero)
      or
      (some b: Elem, z: List | not a = b and cons[b,z,x] and n = r_pos[z][a].~prede)
   }
}


-- PUT
-- KIV axiom
-- (1) (b ' + x)[0, a] = a + x
-- (2) (b ' + x)[n + 1, a] = b + x[n, a]
fact PUT {
   put = {x: List, n: Nat, a: Elem, y: List |
      not x = Nil and
      (n = Zero and cons[a,x.rest,y]
      or
      not n = Zero and cons[x.first,put[x.rest][n.prede][a],y])
   }
}


-- SUBLIST
-- KIV axiom
-- (1) sublist(m, n, x) = firstn(n, restn(m, x))
fact SUBLIST {
   r_sublist = {x: List, m,n: Nat, y: List |
      y = r_firstn[r_restn[x][m]][n]
   }
}


-- FIRSTN
-- KIV axiom
-- (1) firstn(0, x) = []
-- (2) firstn(n + 1, x) = x .first + firstn(n, x .rest)
fact FIRSTN {
   r_firstn = {x: List, n: Nat, y: List |
      n = Zero and y = Nil
      or
      not n = Zero and cons[x.first,r_firstn[x.rest][n.prede],y]
   }
}


-- RESTN
-- KIV axiom
-- (1) restn(0, x) = x
-- (2) restn(n + 1, x) = restn(n, x .rest)
fact RESTN {
   r_restn = {x: List, n: Nat, y: List |
      n = Zero and y = x
      or
      not n = Zero and y = r_restn[x.rest][n.prede]
```

```
   }
}


-- LASTN
-- KIV axiom
-- (1) lastn(0, x) = []
-- (2) lastn(n + 1, x) = lastn(n, x .butlast) + x .last
fact LASTN {
   r_lastn = {x: List, n: Nat, y: List |
      n = Zero and y = Nil
      or
      not n = Zero and y = lastcons[r_lastn[butlast[x]][n.prede]][last[x]]
   }
}


-- FROME
-- KIV axiom
-- (1) frome([], a) = []
-- (2) frome(a ' + x, a) = a ' + x
-- (3) a != b -> frome(a ' + x, b) = frome(x, b)
fact FROME {
   frome = {x: List, a: Elem, y: List |
      x = Nil and y = Nil
      or
      (some z: List | cons[a,z,x] and cons[a,z,y])
      or
      (some b: Elem, z: List | not a = b and cons[b,z,x] and y = frome[z][a])
   }
}


--------------------------------------
---   #5 specification: list-set   ---
--------------------------------------

-- SUBSET
-- KIV axiom
-- x sub y  <-> all a in x -> a in y
fact SUBSET {
   subset = {x,y : List |
      all a: Elem | (x->a) in r_isin => (y->a) in r_isin
   }
}


-- UNION
-- KIV axiom
-- (1) x U y = rmdup(x + y)
fact UNION {
   union = {x,y,z: List | z = rmdup[app[x][y]]
   }
}


-- DIFF
-- KIV axiom
-- (1) []  y = []
-- (2) a in y -> (a ' + x)  y = x  y
-- (3) not a in y -> (a ' + x)  y = a + (x  y)
fact DIFF {
```

```
   diff = {x,y,z: List |
      x = Nil and z = Nil
      or
      (some a: Elem, z1: List | cons[a,z1,x] and (y->a) in r_isin and z = diff[z1][y])
      or
      (some a: Elem, z1: List | cons[a,z1,x] and not (y->a) in r_isin and
      cons[a,diff[z1][y],z])
   }
}


-- FILTER
-- KIV axiom
-- (1) filter([], y) = []
-- (2) a in y -> filter(a ' + x, y) = a ' + filter(x, y)
-- (3) not a in y -> filter(a ' + x, y) = filter(x, y)
fact FILTER {
   filter = {x,y,z: List |
      x = Nil and z = Nil
      or
      (some a: Elem, z1: List | cons[a,z1,x] and (y->a) in r_isin and
      cons[a,filter[z1][y],z])
      or
      (some a: Elem, z1: List | cons[a,z1,x] and not (y->a) in r_isin and
      z = filter[z1][y])
   }
}


---------------------------------------
---   #6 specification: list-perm   ---
---------------------------------------


-- OC
-- KIV axiom
-- (1) #oc(a, []) = 0
-- (2) #oc(a, a ' + x) = #oc(a, x) + 1
-- (3) a != b -> #oc(a, b ' + x) = #oc(a, x)
fact OC {
   r_oc = {x: List, a: Elem, n: Nat |
      x = Nil and n = Zero
      or
      (some z: List | cons[a,z,x] and n = r_oc[z][a].~prede)
      or
      (some b: Elem, z: List | cons[b,z,x] and not a = b and n = r_oc[z][a])
   }
}


-- MSUBSET
-- KIV axiom
-- (1) msubset(x,y) <-> (all a. #oc(a, x) <= #oc(a, y))
fact MSUBSET {
   msubset = {x,y: List |
      all a: Elem | lte[r_oc[x][a],r_oc[y][a]]
   }
}


-- PERM
-- KIV axiom
```

```
-- (1) perm(x, y) <-> msubset(x,y) and msubset(y,x)
fact PERM {
   perm = {x,y: List | (x->y) in msubset and (y->x) in msubset
   }
}


-----------------------------------
---   #7 specification: olist   ---
-----------------------------------

-- INS
-- KIV axiom
-- (1) ins(a, []) = a '
-- (2) not b < a -> ins(a, b ' + x) = a + b + x
-- (3) b < a -> ins(a, b ' + x) = b + ins(a, x)
fact INS {
   r_ins = {x: List, a: Elem, y: List |
      x = Nil and y = mklist01[a]
      or
      (some b: Elem, z,z2: List | cons[b,z,x] and not (b->a) in els and
      cons[b,z,z2] and cons[a,z2,y])
      or
      (some b: Elem, z: List | cons[b,z,x] and (b->a) in els and
      cons[b,r_ins[z][a],y])
   }
}


-- LE
-- KIV axiom
-- (1) ordered([])
-- (2) ordered(a ')
-- (3) ordered(a ' + b ' + x) <-> not b < a and ordered(b + x)
fact LE {
   le = {x: List |
      x = Nil
      or
      (some a: Elem | cons[a,Nil,x])
      or
      (some a,b: Elem, z1,z2: List | cons[b,z1,z2] and cons[a,z2,x] and
      not (b->a) in els and z2 in le)
   }
}


-- LS
-- KIV axiom
-- (1) ordered<([])
-- (2) ordered<(a ')
-- (3) ordered<(a ' + b ' + x) <-> a < b and ordered<(b + x)
fact LS {
   ls = {x: List |
      x = Nil
      or
      (some a: Elem | cons[a,Nil,x])
      or
      (some a,b: Elem, z1,z2: List | cons[b,z1,z2] and cons[a,z2,x] and
      (a->b) in els and z2 in ls)
   }
}
```

```
--------------------------------------
---   #8 specification: olist-sort   ---
--------------------------------------

-- SORT
-- KIV axiom
-- (1)
-- (2)
fact SORT {
   sort = {x,y: List |
      y in le and (x->y) in perm
   }
}


---------------------------------
----   #8 specification: olist-sort   ----
---------------------------------


-- KIV theorem:
-- app: sort(x + ins(a, y)) = ins(a, sort(x + y))

assert T08_01 {
   all x,y: List, a: Elem | all z1,z2,z3,z4,z5,z6: List |
      z1 = r_ins[y][a] and z2 = app[x][z1] and z3 = sort[z2] and
      z4 = app[x][y] and z5 = sort[z4] and z6 = r_ins[z5][a]
      =>
      (z3 = z6)
}


-- KIV theorem:
-- o: ordered(x) -> sort(x) = x

assert T08_02 {
   all x: List | all z1: List | z1 = sort[x]
      =>
      (x in le => z1 = x)
}


-- KIV theorem:
-- ins: ins(a, sort(x)) = y -> sort(x) = y -1l a

assert T08_03 {
   all x,y: List, a: Elem | all z1,z2,z3: List |
      z1 = sort[x] and z2 = r_ins[z1][a] and z3 = del[y][a]
      =>
      (z2 = y => z1 = z3)
}


-- KIV theorem:
-- s: sort(sort(x) + y) = sort(x + y)

assert T08_04 {
   all x,y: List | all z1,z2,z3,z4,z5: List |
      z1 = sort[x] and z2 = app[z1][y] and z3 = sort[z2] and z4 = app[x][y] and z5 = sort[z4]
      =>
      (z3 = z5)
```

```
}


-- KIV theorem:
-- s: # sort(x) = # x

assert T08_05 {
   all x: List | all z1: List, k1,k2: Nat |
      z1 = sort[x] and k1 = cplx[z1] and k2 = cplx[x]
      =>
      (k1 = k2)
}


-- KIV theorem:
-- s: sort(x) = sort(y) <-> perm(x, y)

assert T08_06 {
   all x,y: List | all z1,z2: List | z1 = sort[x] and z2 = sort[y]
      =>
      (z1 = z2 <=> (x->y) in perm)
}


-- KIV theorem:
-- s: sort(x -1l a) = sort(x) -1l a

assert T08_07 {
   all x: List, a: Elem | all z1,z2,z3,z4: List |
      z1 = del[x][a] and z2 = sort[z1] and z3 = sort[x] and z4 = del[z3][a]
      =>
      (z2 = z4)
}


-- KIV theorem:
-- s: perm(ins(a, sort(x)), y) <-> perm(a ' + x, y)

assert T08_08 {
   all x,y: List, a: Elem | all z1,z2,z3: List |
      z1 = sort[x] and z2 = r_ins[z1][a] and z3 = app[mklist01[a]][x]
      =>
      ((z2->y) in perm <=> (z3->y) in perm)
}


-- KIV theorem:
-- s: perm(sort(x), y) <-> perm(x, y)

assert T08_09 {
   all x,y: List | all z1: List | z1 = sort[x]
      =>
      ((z1->y) in perm <=> (x->y) in perm)
}


-- KIV theorem:
-- in: a in sort(x) <-> a in x

assert T08_10 {
   all x: List, a: Elem | all z1: List | z1 = sort[x]
      =>
```

```
      ((z1->a) in r_isin <=> (x->a) in r_isin)
}


--------------------
----   checks   ----
--------------------

-----------------------------------------
----   #8 specification: olist-sort   ----
-----------------------------------------

-- 10 theorems

check T08_01 for exactly 2 Elem, 8 List, 3 Nat
check T08_02 for exactly 2 Elem, 8 List, 3 Nat
check T08_03 for exactly 2 Elem, 8 List, 3 Nat
check T08_04 for exactly 2 Elem, 8 List, 3 Nat
check T08_05 for exactly 2 Elem, 8 List, 3 Nat
check T08_06 for exactly 2 Elem, 8 List, 3 Nat
check T08_07 for exactly 2 Elem, 8 List, 3 Nat
check T08_08 for exactly 2 Elem, 8 List, 3 Nat
check T08_09 for exactly 2 Elem, 8 List, 3 Nat
check T08_10 for exactly 2 Elem, 8 List, 3 Nat


------------------
----   runs   ----
------------------

run cons for exactly 2 Elem, 8 List, 3 Nat

-- K = 1
run cons for exactly 2 Elem, exactly 3 List, 2 Nat
run cons for exactly 3 Elem, exactly 3 List, 2 Nat

-- K = 2
run cons for exactly 2 Elem, exactly 7 List, 3 Nat
run cons for exactly 3 Elem, exactly 13 List, 4 Nat
```

Die Alloy Anweisung *run cons for 2 Elem, exactly 7 List, 3 Nat* generiert ein endliches Modell auf Abb. A.3. Die Anweisung *run cons for 3 Elem, exactly 13 List, 4 Nat* generiert die Instanz auf Abb. A.4.

A.2.2 Nicht-blockierende Mengen

```
module abs

----------------
---   imports   ---
----------------

open nat as Nat
open integer as Integer
open integerinf as IntegerInf
open pid as PID
open ref as Ref
open pidornone as PIDorNone
open cell as Cell
open heap as Heap
```

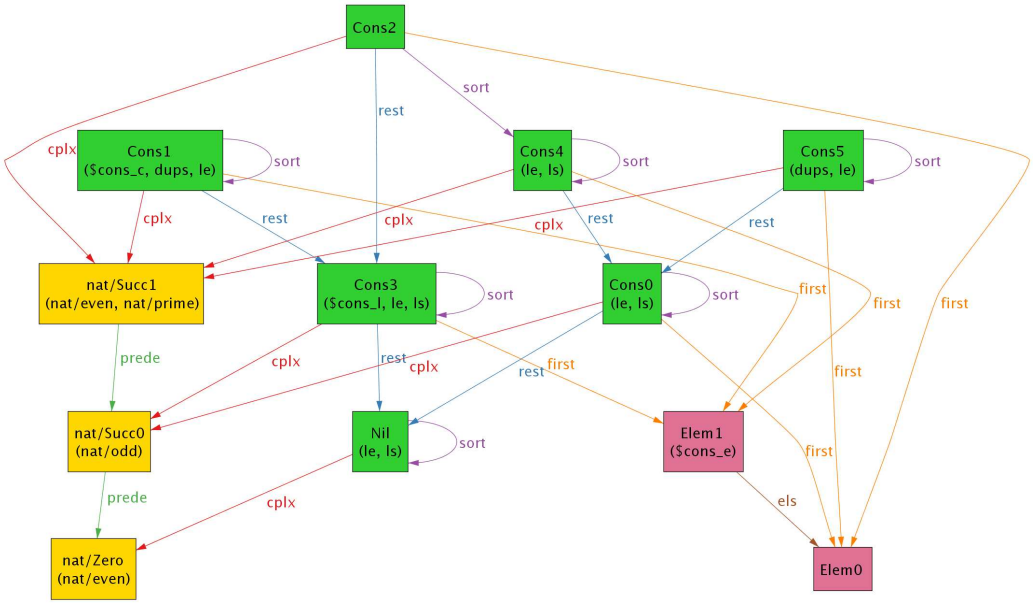


Abbildung A.3: Nur essentiellen Relationen und die *sort* Relation sind sichtbar. *Elems* sind mit *els* geordnet.

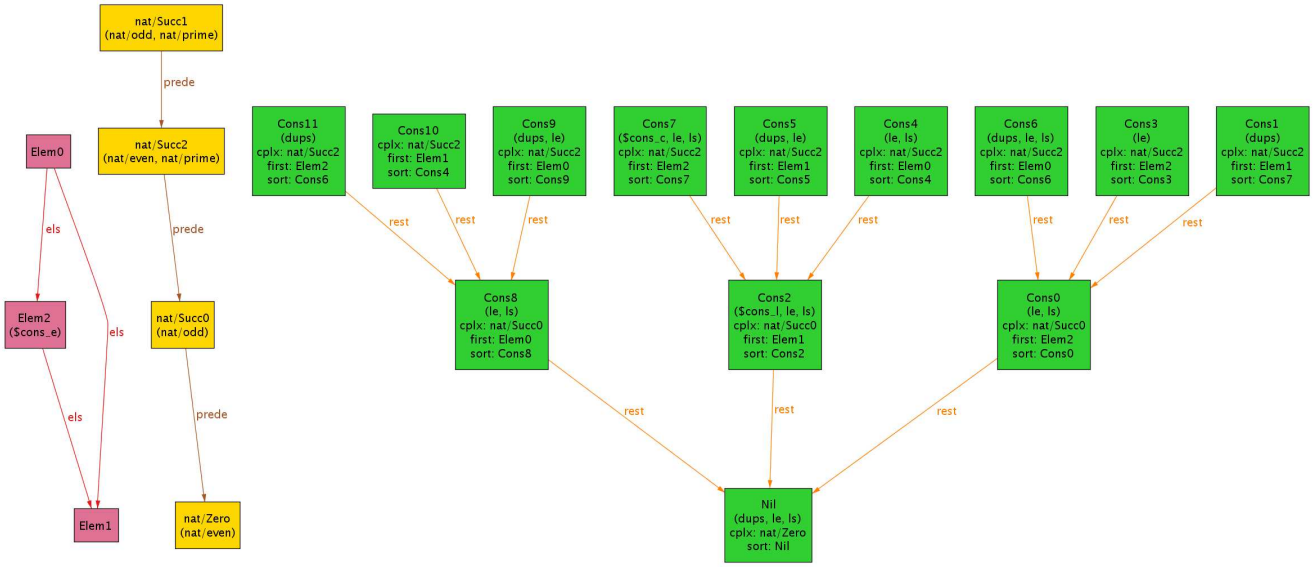


Abbildung A.4: Alternative Visualisierung: *first*, *cplx* and *sort* sind als Attribute der Atome sichtbar. *Elems* sind mit der Ordnung *els* geordnet.

```
open reflist as RefList
open status as Status
open intset as IntSet
open bool as Bool



----------------
---   axioms   ---
----------------

fact ABSH {
        absh = {is: IntSet, ii: IntegerInf, r: Ref, h: Heap |
                -- base
                is = Empty and (h->r) in isin and not r = Null and at[h][r].nxt = Null and
                at[h][r].intinf = PlusInf
                or
                -- rec
                not is = Empty and (h->r) in isin and not r = Null and
                (ii->at[h][r].intinf) in iilt and
                not at[h][r].intinf = PlusInf and (is->at[h][r].intinf.elem) in isin and
                not at[h][r].nxt = Null and
                (delete[is][at[h][r].intinf.elem]->at[h][r].intinf->at[h][r].nxt->h) in absh
        }
}

fact ABS {
        abs = {s: IntSet, r: Ref, h: Heap |
                (h->r) in isin and not r = Null and not at[h][r].nxt = Null and
                at[h][r].intinf = MinusInf and
                (s->MinusInf->at[h][r].nxt->h) in absh
        }
}


fact PATH {
        path = {rl: RefList, h: Heap |
                not rl = Nil and
                (
                        (some r: Ref |
                                rl = mklist01[r] and (h->r) in isin and not r = Null
                        )
                        or
                        (some r1,r2: Ref |
                                rl = app[mklist01[r1]][mklist01[r2]] and (h->r1) in isin and
                                (h->r2) in isin and
                                not r1 = Null and not r2 = Null and
                                at[h][r1].nxt = r2
                        )
                        or
                        (some r1,r2: Ref, rl0: RefList |
                                rl = app[app[mklist01[r1]][mklist01[r2]]][rl0] and
                                (h->r1) in isin and not r1 = Null and
                                at[h][r1].nxt = r2 and
                                (app[mklist01[r2]][rl0]->h) in path
                        )
                )
        }
}

/*
```

```
-- requires too strong completeness !! (not practical)
fact REACHABLE {
        reachable = {h: Heap, r1,r2: Ref |
                (some rl: RefList |
                        (app[mklist01[r1]][rl]->h) in path and
                        last[app[mklist01[r1]][rl]] = r2
                )
        }
}
*/


fact STEPH {
        steph = {h: Heap, r1,r2: Ref |
                not r1 = Null and not r2 = Null and (h->r1) in isin and (h->r2) in isin and
                at[h][r1].nxt = r2
        }
}

fact REACHABLE {
        reachable = {h: Heap, r1,r2: Ref |
                not r1 = Null and (h->r1) in isin and
                -- r2 is in reflexive transitive closure of steph
                r2 in r1.*(h.steph)
        }
}



module integer

open nat as Nat

--------------------------
----  signature  ----
--------------------------

sig Integer {
-- basis
                cplx: one Nat,

                plus: lone Integer,
                minus: lone Integer,

-- operations

-- specification #1 int
                ilt : set Integer,

}

one sig IntZero extends Integer {}

sig Plus extends Integer {
        prep : Integer
}

sig Minus extends Integer {
        prem : Integer
}

-------------------------------------
```

```
----                  constructors                 ----
-------------------------------------

pred PLUS [i,i1: Integer] {
        i1 in Plus and i1.prep = i
}

pred MINUS [i,i1: Integer] {
        i1 in Minus and i1.prem = i
}

pred BIGGER [i,i1: Integer] {
        i in IntZero and (i1 in Plus or i1 in Minus)
                or
                i in Plus and i1 in Plus
                or
                i in Minus and i1 in Minus
}


fact CONSTR_BIGGER {
        all i: Integer |
                         i in Plus and BIGGER[i.prep,i]
                         or
                         i in Minus and BIGGER[i.prem,i]
                         or
                         i in IntZero
}


--------------------
----  SUA axioms  ----
--------------------

fact Uniqueness_Plus {
   all i1,i2: Plus |
        i1.prep = i2.prep
        => i1 = i2
}

fact Uniqueness_Minus {
   all i1,i2: Minus |
        i1.prem = i2.prem
        => i1 = i2
}


fact Acyclic_Plus {
   no i: Plus | i in i.^prep
}

fact Acyclic_Minus {
   no i: Minus | i in i.^prem
}


-----------------------------------
----  Generation axioms  ----
-----------------------------------
```

```
fact FINITE_GENERATOR {
        all s: Set, a: Elem | lt[s.cplx,Zero.~prede.~prede] and BIGGER[s,a]=>
                some s1: Set | (s->a->s1) in put
}


fact BOUND {
        all s: Set | lte[s.cplx,Zero.~prede.~prede]
}



--------------------
----  axioms  ----
--------------------


-- CPLX

-- KIV view
-- (1) cplx(empty) = 0
-- (2) cplx(put(s,a)) = 1 + cplx(s)

fact CPLX {
        cplx = {i: Integer, n: Nat |
                i = IntZero and n = Zero
                or
                some i1: Integer, m: Nat |
                                        (PLUS[i1,i] or MINUS[i1,i]) and
                                        (i1->m) in cplx and m = n.prede
        }
}


-- PLUS (constructor)

fact PLUS_function {
        plus = {i1,i2: Integer |
                        i2 in IntZero and i1 in Minus and i2 = i1.prem
                        or
                        i2 in Minus and i1 in Minus and i1.prem = i2
                        or
                        i2 in Plus and i1 = i2.prep
        }
}

-- MINUS (constructor)

fact MINUS_function {
        minus = {i1,i2: Integer |
                        i2 in IntZero and i1 in Plus and i2 = i1.prep
                        or
                        i2 in Minus and i1 = i2.prem
                        or
                        i2 in Plus and i1 in Plus and i2 = i1.prep
        }
}


-- Less operator: <

fact LESS {
```

```
        ilt = {i,j: Integer |
                i in IntZero and j in Plus
                or
                i in Minus and (j in IntZero or j in Plus or j in i.^prem)
                or
                i in Plus and j in Plus and i in j.^prep
        }

}

-------------------------------------
----              checks & runs                ----
-------------------------------------

pred test [] {
        some i1,i2: Plus, j1,j2: Minus |
                not i1 = i2 and not j1 = j2
}

run test for exactly 5 Integer, 7 Nat
```

Anhang B

Ergebnisse der Evaluation

Die folgenden Ergebnisse der Fehlersuche wurden in KIV mit Kodkod erzielt.

B.1 Listen von Intervallen

Das untersuchte Lemma beschreibt die Eigenschaft der *insert* operation, dass die Wohlgeformtheit der Listen erhalten bleibt:

$$\psi_{INV} \equiv \forall\, ivl_1, ivl_2 \in intervallist,\ n \in nat.\\ R(ivl_1)\ \wedge\ ivl_2 = insert(ivl_1, n) \rightarrow R(ivl_2)$$

Die korrekte Spezifikation der Operation *insert* ist:

(1) $insert([\,], k) = [(k, k)]$

(2) $k < m \wedge k \neq m - 1 \rightarrow insert((m, n) + x, k) = (k, k) + (m, n) + x$

(3) $k < m \wedge k = m - 1 \rightarrow insert((m, n) + x, k) = (k, n) + x$

(4) $k \geq m \wedge k \leq n \rightarrow insert((m, n) + x, k) = (m, n) + x$

(5 a) $k \geq m \wedge k = n + 1 \wedge x = [] \rightarrow insert((m, n) + x, k) = (m, k) + x$

(5 b) $k \geq m \wedge k = n + 1 \wedge x = (m_1, n_1) + y \wedge m_1 = k + 1 \rightarrow$
$insert((m, n) + x, k) = (m, n_1) + y$

(5 c) $k \geq m \wedge k = n + 1 \wedge x = (m_1, n_1) + y \wedge m_1 \neq k + 1 \rightarrow$
$insert((m, n) + x, k) = (m, k) + x$

(6) $k \geq m \wedge k > n + 1 \rightarrow insert((m, n) + x, k) = (m, n) + insert(x, k)$

Auf dem Weg zur korrekten Axiomatisierung von *insert* wurden mehrere Fehler gemacht die mit Kodkod entdeckt wurden. Im folgenden werden diese Fehler beschrieben.

Fehler 1.

Die Axiome (5a-5c) wurden durch ein einfaches Axiom (5) ersetzt:

$$k \geq m \wedge k = n + 1 \rightarrow insert((m, n) + x, k) = (m, k) + x$$

Diese Axiom behandelt aber den Fall der zusammenhänden Interv alle nicht Korrekt. Damit entstehen zusammenhängende Intervalle in der Liste und die Wohlgeformtheit nicht gilt. Kodkod Produziert folgen Ausgabe[1]:

```
counterexample for INV:

variables assignment:
n=1
ivl2=[(0,1), (2,2)]
ivl1=[(0,0), (2,2)]

MODEL:

function table for insert : ivlist x nat -> ivlist
------------------------------------------------------------------------------
[(0,0), (2,2)]  1  [(0,1), (2,2)]
...
==============================================================================

function table for R : ivlist -> bool
------------------------------------------------------------------------------
[(0,0), (2,2)]
[(2,2)]
[]
==============================================================================

statistics
p cnf 75087 236872
primary variables: 764
translation time: 5435 ms
solving time: 885 ms
```

In die Liste $ivl1 = [(0, 0), (2, 2)]$ wird die Zahl 1 eingefügt. Die resultierende Liste $ivl2 = [(0, 1), (2, 2)]$ ist aber nicht wohlgeformt, weil sie zusammenhängende Intervalle enthält. Die korrekte Spezifikation von *insert* berücksichtigt den Fall von zusammenhänden Intervallen und fasst sie zu einem Intervall.

Fehler 2.

Die korrekte Spezifikation der Wohlgeformtheit R auf Listen ist:

$$R([\,]);$$

[1] In der Kodkod Ausgabe werden Werte mit Konstruktortermen repräsentiert. Z.B Werte 0, 1 für natürlichen Zahlen werden mit Termen 0 und $+1(0)$ repräsentiert. Für die Liste $[(0, 2)]$ bekommt man den Term $+(\times(0, +1(+1(0))), [])$. In folgenden Tabellen zu Übersichtlichkeit werden die Werte direkt geschrieben.

$$R((m,n)+x) \leftrightarrow m \leq n \wedge n+1 < x.first.fst \wedge R(x);$$

Die ursprüngliche Spezifikation war aber fehlerhaft:

$$R([\,]);$$
$$R((m,n)+x) \leftrightarrow m \leq n \wedge n < x.first.fst - 1 \wedge R(x);$$

Der Fehler wurde bei der Kodkod Analyse des folgenden Theorems entdeckt:

$$R - same: \qquad \neg R((n \times n) + (n \times n + m))$$

Das Theorem formuliert die gewünschte Eigenschaft von R, dass es keine zusammenhängende Intervalle in wohlgeformten Listen gibt. Kodkod hat das Theorem als falsh kennzeichnet:

```
counterexample for R-same:

variables assignment:
n=0
m=1

MODEL:

function table for R : ivlist -> bool
--------------------------------------------------------------------------------
[(0,0),(0,1)]
[(0,1)]
[]
================================================================================

statistics
p cnf 4618 12402
primary variables: 345
translation time: 220 ms
solving time: 7 ms
```

Die Liste $[(0,0),(0,1)]$ wird als wohlgeformt kennzeichnet. Der Fehler steckt in der Formel $n < x.first.fst - 1$ in der Spezifikation von *insert*. Im von Kodkod aufgedeckten Fall ist $(m,n) + x = [(0,0),(0,1)]$, $x = [(0,1)]$, $x.first.fst = 0$. Der Ausdruck $x.first.fst - 1$ hat den Wert $0 - 1$ und dait undefiniert. Die korrigierte Version der Formel ist $n + 1 < x.first.fst$.

Fehler 3.

Die Axiome (2) und (3) wurden weggelassen und durch das folgende Axiom ersetzt:

$$k < m \rightarrow insert((m,n)+x, k) = (k,k) + (m,n) + x$$

Kodkod generiert einen Gegenbeispiel:

```
counterexample for INV:

variables assignment:
n=0
ivl2=[(0,0),(1,1)]
ivl1=[1,1]

MODEL:

function table for R : ivlist -> bool
--------------------------------------------------------------------------------
[(1,1)]
[]
================================================================================

function table for insert : ivlist x nat -> ivlist
--------------------------------------------------------------------------------
[(1,1)]  0  [(0,0),(1,1)]
...
================================================================================

statistics
p cnf 103607 329807
primary variables: 366
translation time: 11202 ms
solving time: 606 ms
```

Das Axiom berücksichtigt nicht den Fall, wo $k = m - 1$ gilt. In diesem Fall sollten die Intervalle (k, k) und (m, n) zu (k, n) verschmelzen, siehe Axiom (3) in der richtigen Axiomtisierung.

Fehler 4.

Die Axiome (5a-5c) wurden weggelassen und das Axiom (6) durch das folgende Axiom ersetzt:

$$k \geq m \wedge k > n \rightarrow insert((m, n) + x, k) = (m, n) + insert(x, k)$$

Kodkod generiert folgenden Gegenbeispiel:

```
counterexample for INV:

variables assignment:
n=3
ivl2=[(2,2),(3,3)]
ivl1=[(2,2)]

MODEL:

function table for R : ivlist -> bool
--------------------------------------------------------------------------------
```

```
[(2,2)]
[(3,3)]
[]
================================================================================

function table for insert : ivlist x nat -> ivlist
--------------------------------------------------------------------------------
[(2,2)]  3  [(2,2),(3,3)]
...
================================================================================

statistics
p cnf 65599 208376
primary variables: 764
translation time: 4454 ms
solving time: 459 ms
```

Der Fall $k = n+1$ blieb unberücksichtigt und es entstanden zusammenhängende Intervalle.

Fehler 5.

Das Axiom (2) in der korrekten Axiomatisierung wurde durch das folgende Axiom ersetzt:

$$k < m \wedge k \neq m - 1 \rightarrow insert((m, n) + x, k) = (m, n) + (k, k) + x$$

Die Reihenfolge in der die Intervalle (m, n) und (k, k) eingefügt wurden wurde getauscht. Als Folge findet Kodkod ein Gegenbeispiel:

```
counterexample for INV:

variables assignment:
n=0
ivl2=[(2,2),(0,0)]
ivl1=[(2,2)]

MODEL:

function table for R : ivlist -> bool
--------------------------------------------------------------------------------
[(2,2)]
[(0,0)]
[]
================================================================================

function table for insert : ivlist x nat -> ivlist
--------------------------------------------------------------------------------
[(2,2)]  0    [(2,2),(0,0)]
...
================================================================================
```

```
statistics
p cnf 1244447 4003000
primary variables: 764
translation time: 245766 ms
solving time: 10872 ms
```

Die Ergebnisliste $[(2,2),(0,0)]$ ist nicht wohlgeformt.

Fehler 6.

Die Axiome (5b) und (5c) wurden falsch spezifiziert:

(5*b*) $k \geq m \wedge k = n+1 \wedge x = (m_1, n_1) + y \wedge m_1 = n+1 \rightarrow insert((m,n)+x, k) = (m, n_1) + y$

(5*c*) $k \geq m \wedge k = n+1 \wedge x = (m_1, n_1) + y \wedge m_1 \neq n+1 \rightarrow insert((m,n)+x, k) = (m, k) + x$

In den Vorbedingungen $m_1 = k+1$ und $m_1 \neq k+1$ wurde k durch n ersetzt.
Kodkod findet das folgende Gegenbeispiel:

```
counterexample for INV:

variables assignment:
n=2
ivl2=[(1,2),(3,3)]
ivl1=[(1,1),(3,3)]

MODEL:

function table for R : ivlist -> bool
--------------------------------------------------------------------------------
[(1,1),(3,3)]
[(3,3)]
[]
================================================================================

function table for insert : ivlist x nat -> ivlist
--------------------------------------------------------------------------------
[(1,1),(3,3)]  2  [(1,2),(3,3)]
...
================================================================================

statistics
p cnf 2139935 6689464
primary variables: 764
translation time: 257603 ms
solving time: 31644 ms
```

Es handelt sich um den Fall der Axiom (5c). $(m,n)+x = [(1,1)(3,3)]$, $n=2$, $(m_1,n_1) = (3,3)$. Es gilt $m_1 \neq n+1$ und damit wird die Liste $(m,k)+x = [(1,2),(3,3)]$ konstruiert, die nicht wohlgeformt ist.

Fehler 7.

Das Axiom (3) ist dur das folgende Axiom ersetzt:

$$k < m \wedge k = m - 1 \rightarrow insert((m, n) + x, k) = (n, k) + x$$

In der Ergebnisliste ist das Intervall (k, n) durch (n, k) ersetzt worden. Kodkod berechnet ein Gegenbeispiel:

```
counterexample for INV:

variables assignment:
n=1
ivl2=[(2,1)]
ivl1=[(2,2)]

MODEL:

function table for R : ivlist -> bool
------------------------------------------------------------------------------
[(2,2)]
[]
==============================================================================

function table for insert : ivlist x nat -> ivlist
------------------------------------------------------------------------------
[(2,2)]  1  [(2,1)]
...
==============================================================================

statistics
p cnf 175913 545947
primary variables: 300
translation time: 12462 ms
solving time: 511 ms
```

Fehler 8.

Wir spezifizieren das Prädikat $memeber : nat \times ivlist$, das die Midgliedschaft einer Zahl in einer Liste testet:

$$member(n, x) \leftrightarrow x \neq [] \wedge (x.first.1 \leq n \wedge x.first.2 \geq n \vee member(n, x.rest))$$

Nun wollen wir überprüefen ob die Operation *insert* sich bezüglich *memeber* richtig verhält:

$$member : \quad R(ivl) \rightarrow (member(m, ivl) \vee m = n \leftrightarrow member(m, insert(ivl, n)))$$

Das Lemma *member* wurde mit Kodkod gecheckt, wobei das Axiom (4) durch ein fehlerhaftes Axiom ersetzt war:

$$k \geq m \wedge k \leq n \rightarrow insert((m, n) + x, k) = (m, k) + x$$

Die richtige Ergebnisliste wäre $(m, n) + x$. In der Liste $(m, k) + x$ ist der Teil (k, n) abgeschnitten und es fehlen möglicherweise Zahlen die es in $(m, n) + x$ ursprünglich gab. Kodkod berechnet folgendes Gegenbeispiel:

```
counterexample for member:

variables assignment:
n=1
m=2
ivl=[(1,2)]

MODEL:

function table for R : ivlist -> bool
------------------------------------------------------------------------------
[(1,1)]
[(1,2)]
[]
==============================================================================

function table for member : nat x ivlist -> bool
------------------------------------------------------------------------------
2   [(1,2)]
1   [(1,1)]
1   [(1,2)]
==============================================================================

function table for insert : ivlist x nat -> ivlist
------------------------------------------------------------------------------
[(1,2)]  1  [(1,1)]
...
==============================================================================

statistics
p cnf 176041 547051
primary variables: 306
translation time: 13501 ms
solving time: 838 ms
```

In die Liste $ivl = [(1, 2)]$ wird die Zahl $n = 1$ eingefügt. Das Ergebnis ist die Liste $[(1, 1)]$, die offensichtlich die untersuchte Eigenschaft verletzt.

Fehler 9.

Das Axiom (5a) wird durch das folgende Axiom ersetzt:

$$k \geq m \wedge k = n + 1 \wedge x = [] \rightarrow insert((m, n) + x, k) = (m, n) + x$$

Die richtige Ergebnisliste wäre $(m, k) + x$. In der Liste $(m, n) + x$ fehlt die Zahl $k = n + 1$. Damit ist das Lemma *memeber* falsch und Kodkod berechnet ein Gegenbeispiel:

```
counterexample for member:

variables assignment:
n=1
m=1
ivl=[(0,0)]

MODEL:

function table for R : ivlist -> bool
------------------------------------------------------------------------------
[(0,0)]
[]
==============================================================================

function table for member : nat x ivlist -> bool
------------------------------------------------------------------------------
0   [(0,0)]
==============================================================================

function table for insert : ivlist x nat -> ivlist
------------------------------------------------------------------------------
[(0,0)]   1   [(0,0)]
...
==============================================================================

statistics
p cnf 5963 17630
primary variables: 118
translation time: 196 ms
solving time: 5 ms
```

Fehler 10.

Das Axiom (3) wird durch das folgende Axiom ersetzt:

$$k < m \wedge k = m - 1 \rightarrow insert((m, n) + x, k) = (m, n) + x$$

In der Ergebnisliste wird die Zahl k abgeschnitten. Die richtige Liste wäre $(k, n) + x$. Als folge gilt ist das Lemma *memeber* falsch und Kodkod berechnet ein Gegenbeispiel:

```
counterexample for member:

variables assignment:
n=0
m=0
ivl=[(1,1)]

MODEL:
```

```
function table for R : ivlist -> bool
--------------------------------------------------------------------------------
[(1,1)]
[]
================================================================================

function table for member : nat x ivlist -> bool
--------------------------------------------------------------------------------
1  [(1,1)]
================================================================================

function table for insert : ivlist x nat -> ivlist
--------------------------------------------------------------------------------
[(1,1)]   0   [(1,1)]
...
================================================================================

statistics
p cnf 6031 17834
primary variables: 118
translation time: 216 ms
solving time: 8 ms
```

B.2 Nicht-blockierende Mengen

Lemma 1.

Die Referenz r_0 ist von $H[r].nxt$ aus in H erreichbar. Dann ist es auch von r aus erreichabar, vorausgesetzt r in H allokiert ist:

$$\texttt{lemma01}: \quad \begin{aligned} & reachable(H[r].nxt, r_0, H) \\ & r \in H \rightarrow \\ & reachable(r, r_0, H) \end{aligned}$$

Das Lemma ist falsch, weil der Fall $r = null$ nicht beachtet wurde. In diesem Fall kann von r keine Referenz erreicht werden. Kodkod findet das entsprechende Gegenbeispiel:

```
counterexample for lemma01:

variables assignment:
r=null
r0=Ref2
H={(Ref2,mkcell(-∞,⌈ PID1 ⌉,Ref2)),
   (null,mkcell(-∞,⌈ PID1 ⌉,Ref2))}

MODEL:
```

```
function table for reachable : Ref x Ref x heap -> bool
--------------------------------------------------------------------------------
Ref2   Ref2   {(Ref2,mkcell(-∞,⌈ PID1 ⌉,Ref2)),
               (null,mkcell(-∞,⌈ PID1 ⌉,Ref2))}
Ref2   Ref2   {(Ref2,mkcell(-∞,⌈ PID1 ⌉,Ref2))}
================================================================================

statistics
p cnf 15301 39585
primary variables: 888
translation time: 2260 ms
solving time: 33 ms
```

Die richtige Formulierung berücksichtigt den Fall der Nullreferenz:

$$\texttt{lemma01}: \quad reachable(H[r].nxt, r_0, H) \\ \mathbf{r} \neq \mathbf{null} \wedge r \in H \rightarrow \\ reachable(r, r_0, H)$$

Lemma 2.

Der Pointer r_0 ist von r aus in H erreichbar. Es wird die folgende Vermutung getestet:

$$\texttt{lemma02}: \quad reachable(r, r_0, H) \rightarrow reachable(H[r].nxt, r_0, H)$$

Kodkod berechnet ein Gegenbeispiel:

```
counterexample for lemma02:

variables assignment:
r=Ref0
r0=Ref0
H={(Ref0,mkcell(-∞,⌈ PID1 ⌉,null))}

MODEL:

function table for reachable : Ref x Ref x heap -> bool
--------------------------------------------------------------------------------
Ref0   Ref0   {(Ref0,mkcell(-∞,⌈ PID1 ⌉,null))}
================================================================================

statistics
p cnf 3020 6557
primary variables: 305
translation time: 61 ms
solving time: 2 ms
```

Falls $r = r_0$ gilt ist das Lemma falsch. Das Lemma wird nun korrigiert:

$$\texttt{lemma02}: \quad reachable(r, r_0, H) \wedge \mathbf{r} \neq \mathbf{r_0} \rightarrow reachable(H[r].nxt, r_0, H)$$

Lemma 3.

Das Einfügen einer neuen Zelle soll die Erreichbarkeit nicht beeinflüssen. Dies wird durch die Zusatzbedingung $r_2 \neq r \wedge r_2 \neq r_1$ erzielt:

$$\texttt{lemma03}: \quad reachable(r, r_1, H[r_2, ce]) \wedge r_2 \neq r_1 \wedge r_2 \notin H \rightarrow reachable(r, r_1, H)$$

Nun findet Kodkod ein Gegenbeispiel, dass die Schwäche der Zusatzbedingung zeigt:

```
counterexample for lemma03:

variables assignment:
r=Ref2
r1=Ref1
ce=mkcell(-∞,⌈ PID1 ⌉,Ref1)
r2=Ref2
H={(Ref1,mkcell(-∞,⌈ PID1 ⌉,Ref1))}

MODEL:

function table for reachable : Ref x Ref x heap -> bool
--------------------------------------------------------------------------------
Ref1   Ref1   {(Ref1,mkcell(-∞,⌈ PID1 ⌉,Ref1)),
               (Ref2,mkcell(-∞,⌈ PID1 ⌉,Ref1))}
Ref1   Ref1   {(Ref1,mkcell(-∞,⌈ PID1 ⌉,Ref1))}
Ref2   Ref1   {(Ref1,mkcell(-∞,⌈ PID1 ⌉,Ref1)),
               (Ref2,mkcell(-∞,⌈ PID1 ⌉,Ref1))}
Ref2   Ref2   {(Ref1,mkcell(-∞,⌈ PID1 ⌉,Ref1)),
               (Ref2,mkcell(-∞,⌈ PID1 ⌉,Ref1))}
================================================================================

statistics
p cnf 15426 39792
primary variables: 891
translation time: 2699 ms
solving time: 27 ms
```

Damit die neueingefügte Zelle ce keinen Einfluss auf die Erreichbarkeit hat, muss noch zusätzlich $ce.nxt = null$ gefordert werden:

$$\texttt{lemma03}: \quad reachable(r, r_1, H[r_2, ce]) \wedge r_2 \neq r_1 \wedge \mathbf{ce.nxt} = \mathbf{null} \rightarrow reachable(r, r_1, H)$$

Lemma 4.

Im Lemma 3 wird die Vorbedingung $r_2 \neq r_1$ weggelassen:

$$\texttt{lemma04}: \quad reachable(r, r_1, H[r_2, ce]) \wedge r_2 \notin H \wedge ce.nxt = null \rightarrow reachable(r, r_1, H)$$

Dies führt zu folgenem Gegenbeispiel:

```
counterexample for lemma04:

variables assignment:
r=Ref0
r1=Ref0
ce=mkcell(-∞,⌈ PID1 ⌉,null)
r2=Ref0
H={}

MODEL:

function table for reachable : Ref x Ref x heap -> bool
--------------------------------------------------------------------------------
Ref0   Ref0   {(Ref0,mkcell(-∞,⌈ PID1 ⌉,null))}
================================================================================

statistics
p cnf 3070 6639
primary variables: 309
translation time: 135 ms
solving time: 5 ms
```

Falls $r_1 = r_2$ gilt, kann H leer sein und das Lemma falsch.

Lemma 5.

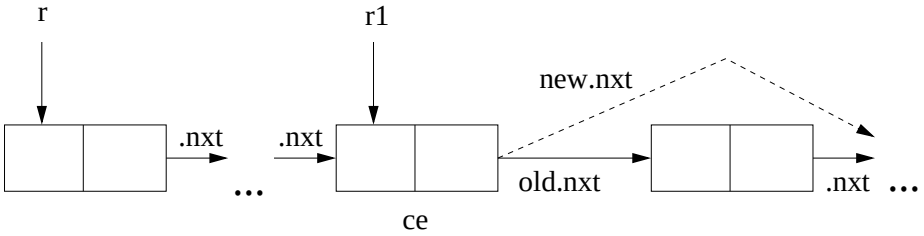


Abbildung B.1: Lemma 5.

Das folgende Lemma beschreibt das Überschreiben einer Zelle in der Halde. Dabei soll die Erreichbarkeit erhalten bleiben:

$$\texttt{lemma05}: \quad reachable(r, r_0, H) \; H[r_1].nxt \in H \wedge r_1 \neq null \wedge r_1 \in H \; ce.nxt = H[H[r_1].nxt].nxt \wedge r_0 \neq H[r_1].nxt \rightarrow reachable(r, r_0, H[r_1, ce])$$

Die Abbildung B.1 visualisiert dieses Lemma. Die Referenz r_1 zeigt auf eine Zelle, die von r aus erreichbar ist. Die Zelle $H[r_1]$ wird mit der Zelle *ce* überschrieben. Die Zelle *ce* zeigt auf $H[H[r_1].nxt].nxt$ (gestrichelte Linie $new.nxt$).

Das Lemma ist falsch, weil ein Randfall vergessen wurde: $H[r_1].nxt = null$. Falls $H[r_1].nxt$ ein Nullpointer ist, ist die Zelle *ce* von Bedeutung für die Erreichbarkeit. Kodkod findet ein Gegenbeispiel:

```
counterexample for reachable-remove:

variables assignment:
r=Ref1
r0=Ref2
r1=Ref1
ce=mkcell(∞,⌈ PID0 ⌉,Ref2)
H={(null,mkcell(∞,⌈ PID0 ⌉,Ref2)),
   (Ref2,mkcell(⌜ int3 ⌝,⌈ PID0 ⌉,null)),
   (Ref1,mkcell(⌜ int3 ⌝,⌈ PID0 ⌉,null))}

MODEL:

function table for reachable : Ref x Ref x heap -> bool
--------------------------------------------------------------------------------
Ref1   Ref1   {(null,mkcell(∞,⌈ PID0 ⌉,Ref2)),
               (Ref2,mkcell(⌜ int3 ⌝,⌈ PID0 ⌉,null)),
               (Ref1,mkcell(∞,⌈ PID0 ⌉,Ref2))}
Ref1   Ref1   {(null,mkcell(∞,⌈ PID0 ⌉,Ref2)),
               (Ref2,mkcell(⌜ int3 ⌝,⌈ PID0 ⌉,null)),
               (Ref1,mkcell(⌜ int3 ⌝,⌈ PID0 ⌉,null))}
Ref1   Ref2   {(null,mkcell(∞,⌈ PID0 ⌉,Ref2)),
               (Ref2,mkcell(⌜ int3 ⌝,⌈ PID0 ⌉,null)),
               (Ref1,mkcell(∞,⌈ PID0 ⌉,Ref2))}
Ref2   Ref2   {(null,mkcell(∞,⌈ PID0 ⌉,Ref2)),
               (Ref2,mkcell(⌜ int3 ⌝,⌈ PID0 ⌉,null)),
               (Ref1,mkcell(∞,⌈ PID0 ⌉,Ref2))}
Ref2   Ref2   {(null,mkcell(∞,⌈ PID0 ⌉,Ref2)),
               (Ref2,mkcell(⌜ int3 ⌝,⌈ PID0 ⌉,null)),
               (Ref1,mkcell(⌜ int3 ⌝,⌈ PID0 ⌉,null))}
Ref2   Ref2   {(null,mkcell(∞,⌈ PID0 ⌉,Ref2)),
               (Ref2,mkcell(⌜ int3 ⌝,⌈ PID0 ⌉,null))}
================================================================================

statistics
p cnf 134530 397657
primary variables: 3501
translation time: 267072 ms
solving time: 35132 ms
```

Die korrigierte Version des Lemmas ist:

`lemma05 :` $reachable(r, r_0, H)$
$\mathbf{H[r_1].nxt} \neq \mathbf{null} \wedge H[r_1].nxt \in H \wedge r_1 \neq null \wedge r_1 \in H$
$ce.nxt = H[H[r_1].nxt].nxt \wedge r_0 \neq H[r_1].nxt \rightarrow$
$reachable(r, r_0, H[r_1, ce])$

Lemma 6.

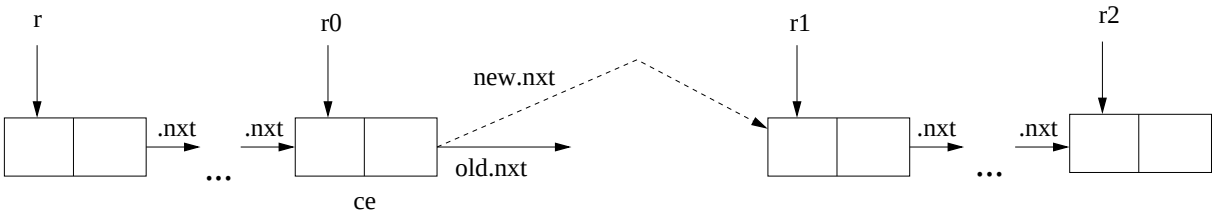


Abbildung B.2: Lemma 6.

Das folgende Lemma beschreibt das Verknüpfen von zwei Pointerketten in der Halde:

`lemma06 :` $reachable(r, r_0, H) \wedge reachable(r_1, r_2, H) \wedge ce = H[r_0] \rightarrow$
$reachable(r, r_2, H[r_0, mkcell(ce.i, ce.pin, r1)])$

Der *nxt*-Pointer der Zelle $ce = H[r_0]$ wird auf die Zelle $H[r_1]$ umgebogen. Damit wird die Erreichbarkeit der Referenz r_2 von r aus hergestellt. Kodkod findet ein Gegenbeispiel, das diese Behauptung falsifiziert:

```
counterexample for lemma06:

variables assignment:
r=Ref3
r0=Ref3
r1=Ref3
ce=mkcell(∞,⌈ PID1 ⌉,Ref2)
r2=Ref2
H={(Ref2,mkcell(∞,⌈ PID1 ⌉,Ref3)),
   (Ref3,mkcell(∞,⌈ PID1 ⌉,Ref2))}

MODEL:

function table for reachable : Ref x Ref x heap -> bool
--------------------------------------------------------------------------------
Ref2   Ref2   {(Ref2,mkcell(∞,⌈ PID1 ⌉,Ref3)),
               (Ref3,mkcell(∞,⌈ PID1 ⌉,Ref2))}
Ref2   Ref3   {(Ref2,mkcell(∞,⌈ PID1 ⌉,Ref3)),
               (Ref3,mkcell(∞,⌈ PID1 ⌉,Ref2))}
```

```
...
================================================================================

statistics
p cnf 58591 150326
primary variables: 2090
translation time: 13895 ms
solving time: 1278 ms
```

Im Gegenbeispiel wird der Fall $r_0 = r_1$ gezeigt. In diesem Fall das Überschreiben des nxt-Pointers in der Zelle $H[r_0]$ verhindert die Erreichbarkeit von r_2.

Lemma 7.

Der erste Versuch den Fehler im Lemma 6 zu korrigieren fügt die Zusatzbedingung $r_0 \neq r_1$ hinzu:

$$\texttt{lemma07}: \quad reachable(r, r_0, H) \wedge reachable(r_1, r_2, H) \wedge ce = H[r_0] \wedge r_0 \neq r_1 \rightarrow reachable(r, r_2, H[r_0, mkcell(ce.i, ce.pin, r1)])$$

Leider reicht es nicht aus und Kodkod findet ein Gegenbeispiel:

```
counterexample for lemma06:

variables assignment:
r=Ref3
r0=Ref3
r1=Ref2
ce=mkcell(-∞,None,Ref1)
r2=Ref1
H={(Ref1,mkcell(-∞,None,Ref2)),
   (Ref2,mkcell(∞,⌈ PID0 ⌉,Ref3)),
   (Ref3,mkcell(-∞,None,Ref1))}

MODEL:

function table for reachable : Ref x Ref x heap -> bool
--------------------------------------------------------------------------------
Ref1   Ref2    {(Ref1,mkcell(-∞,None,Ref2)),
                (Ref2,mkcell(∞,⌈ PID0 ⌉,Ref3)),
                (Ref3,mkcell(-∞,None,Ref2))}
Ref1   Ref3    {(Ref1,mkcell(-∞,None,Ref2)),
                (Ref2,mkcell(∞,⌈ PID0 ⌉,Ref3)),
                (Ref3,mkcell(-∞,None,Ref2))}
...
================================================================================

statistics
```

```
p cnf 127331 333966
primary variables: 3490
translation time: 62286 ms
solving time: 32460 ms
```

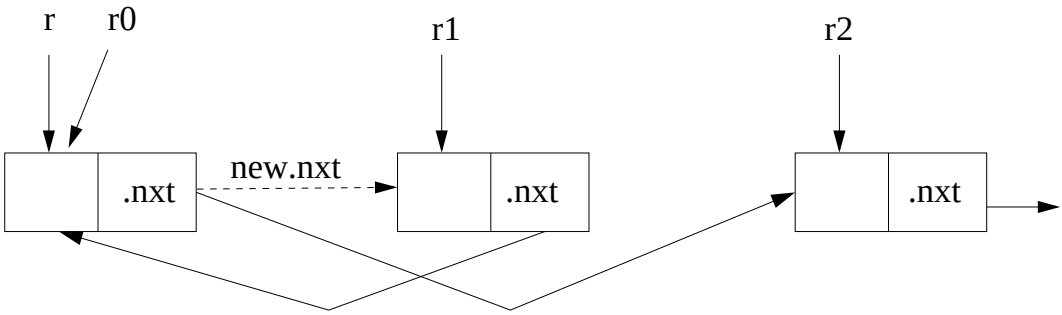


Abbildung B.3: Lemma 6.

Die Abbildung B.3 zeigt den Fall aus dem Gegenbeispiel. Das Umbiegen des Pointers $H[r_0].nxt$ auf r_1 verhindert die Erreichbarkeit von r_2.

Lemma 8.

Im zweiten Versuch das Lemma 6 zu korrigieren bekommt man:

$$\begin{aligned} \texttt{lemma08}: \quad & reachable(r, r_0, H[r_0, mkcell(ce.i, ce.pin, null)]) \wedge \\ & reachable(r_1, r_2, H) \wedge ce = H[r_0] \rightarrow \\ & reachable(r, r_2, H[r_0, mkcell(ce.i, ce.pin, r_1)]) \end{aligned}$$

Mit $reachable(r, r_0, H[r_0, mkcell(ce.i, ce.pin, null)])$ wird behauptet, dass der Wert von $H[r_0].nxt$ für die Erreichbarkeit von $r \rightarrow r_0$ unwichtig ist. Die Vorbedingung ist trotzdem nicht ausreichend und Kodkod findet ein Gegenbeispiel:

```
counterexample for lemma08:

variables assignment:
r=Ref3
r0=Ref3
r1=Ref3
r2=Ref2
ce=mkcell(⌜ int0 ⌝,⌡ PID0 ⌠,Ref2)
H={(Ref2,mkcell(⌜ int0 ⌝,⌡ PID0 ⌠,Ref2)),
   (Ref3,mkcell(⌜ int0 ⌝,⌡ PID0 ⌠,Ref2))}

MODEL:

function table for reachable : Ref x Ref x heap -> bool
--------------------------------------------------------------------------------
Ref3   Ref2   {(Ref2,mkcell(⌜ int0 ⌝,⌡ PID0 ⌠,Ref2)),
```

```
              (Ref3,mkcell(⌜ int0 ⌝,⌈ PID0 ⌉,Ref2))}
...
================================================================================

statistics
p cnf 128160 334696
primary variables: 3510
translation time: 63244 ms
solving time: 26113 ms
```

In der korrigierten Version des Lemmas wird zusätzlich noch verlangt, dass der Wert $H[r_0].nxt$ für die Errechnbarkeit $r_1 \rightarrow r_2$ unwichtig ist:

$$\begin{aligned} \texttt{lemma08}: \quad & reachable(r, r_0, H[r_0, mkcell(ce.i, ce.pin, null)]) \wedge \\ & \mathbf{reachable(r_1, r_2, H[r_0, mkcell(ce.i, ce.pin, null)])} \wedge \\ & ce = H[r_0] \rightarrow \\ & reachable(r, r_2, H[r_0, mkcell(ce.i, ce.pin, r_1)]) \end{aligned}$$

Lemma 9.

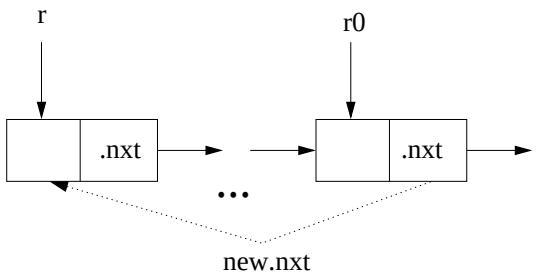


Abbildung B.4: Lemma 9.

Die Abbildung B.4 visualisiert das folgende Lemma:

$$\begin{aligned} \texttt{lemma09}: \quad & reachable(r, r_0, H[r_0, mkcell(ce.i, ce.pin, null)]) \wedge \\ & ce = H[r_0] \rightarrow \\ & reachable(H[r].nxt, r, H[r_0, mkcell(ce.i, ce.pin, r)]) \end{aligned}$$

Pointer $H[r_0].nxt$ ist für die Erreichbarkeit $r \rightarrow r_0$ unwichtig und wird auf r umgebogen. Jetzt muss $H[r].nxt$ von r aus erreichbar sein. Das Kodkod Gegenbeispiel zeigt, wann es nicht gilt:

```
counterexample for lemma09:

variables assignment:
r=Ref1
r0=Ref1
```

```
ce=mkcell(⌜ int0 ⌝,⌈ PID1 ⌉,null)
H={(Ref1,mkcell(⌜ int0 ⌝,⌈ PID1 ⌉,null))}

MODEL:

function table for reachable : Ref x Ref x heap -> bool
--------------------------------------------------------------------------------
Ref1   Ref1   {(Ref1,mkcell(⌜ int0 ⌝,⌈ PID1 ⌉,Ref1))}
Ref1   Ref1   {(Ref1,mkcell(⌜ int0 ⌝,⌈ PID1 ⌉,Ref1))}
================================================================================

statistics
p cnf 16065 38184
primary variables: 921
translation time: 1870 ms
solving time: 82 ms
```

In diesem Fall gilt $H[r].nxt = null$ und $H[r].nxt \notin H$, was die Erreichbarkeit verhindert.

Lemma 10.

Wir korrigieren das Lemma 9 indem wir die Zusatzbedingung $H[r].nxt = null \wedge H[r].nxt \notin H$ hinzufügen:

$$\begin{aligned} \texttt{lemma10}: \quad & reachable(r, r_0, H[r_0, mkcell(ce.i, ce.pin, null)]) \wedge \\ & H[r].nxt = null \wedge H[r].nxt \notin H \\ & ce = H[r_0] \rightarrow \\ & reachable(H[r].nxt, r, H[r_0, mkcell(ce.i, ce.pin, r)]) \end{aligned}$$

Leider reicht es nicht aus und Kodkod findet ein Gegenbeispiel:

```
counterexample for lemma09:

variables assignment:
r=Ref2
r0=Ref2
ce=mkcell(⌜ int4 ⌝,⌈ PID0 ⌉,Ref3)
H={(Ref3,mkcell(⌜ int4 ⌝,⌈ PID0 ⌉,null)),
   (Ref2,mkcell(⌜ int4 ⌝,⌈ PID0 ⌉,Ref3))}

MODEL:

function table for reachable : Ref x Ref x heap -> bool
--------------------------------------------------------------------------------
Ref2   Ref3   {(Ref3,mkcell(⌜ int4 ⌝,⌈ PID0 ⌉,null)),
               (Ref2,mkcell(⌜ int4 ⌝,⌈ PID0 ⌉,Ref3))}
...
================================================================================
```

```
statistics
p cnf 128825 335677
primary variables: 3529
translation time: 63896 ms
solving time: 21241 ms
```

Da $r = r_0$ gilt, ist $H[r_0].nxt$ für die Erreichbarkeit unwichtig solange $r \in H$ gilt. Damit ist r von $H[r].nxt$ aus unerreichbar.

Lemma 11.

Das Prädikat $SetAbs(Head, H, S)$ testet ob die Menge S in der Halde H abgespeichert ist, siehe die Definition in Abschnitt 6.1. Die Abbildung B.5 zeigt am Beispiel der Menge $\{1, 2\}$ die korrekte Speicherung auf der Halde.

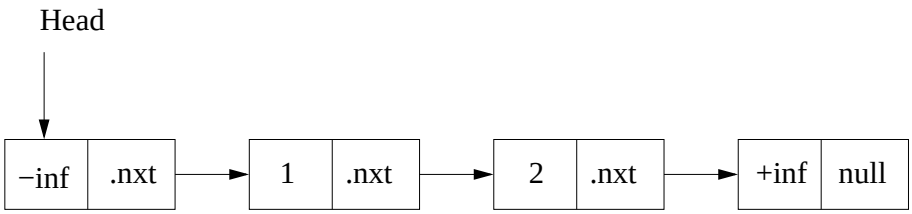


Abbildung B.5: SetAbs(Head,H,{1,2}).

Die Menge wird als eine verzeigerte Kette von aufsteigenden Zahlen abgespeichert, die mit $-\infty$ anfängt und $+\infty$ endet: $-\infty, 1, 2, +\infty$. Die Zeiger der letzten Zelle ist Null.

Folgende Lemma beschreibt eine Eigenschaft der Abstraktionsfunktion $SetAbs$:

$$\texttt{lemma11}: \quad reachable(r, r_1, H) \wedge H[r_1].nxt = null \wedge SetAbs(r, H, s) \rightarrow H[r_1].i\infty = -\infty$$

Das falsch weil die korrekten Ketten sollen mit einer Zelle enden, die Wert $+\infty$ enthält:

```
counterexample for lemma11:

variables assignment:
s={}
r=Ref1
r1=Ref0
H={(Ref1,mkcell(-∞,⌈ PID1 ⌉,Ref0)),
   (Ref0,mkcell(∞,⌈ PID1 ⌉,null))}

MODEL:

function table for SetAbs : Ref x heap x set -> bool
--------------------------------------------------------------------------------
```

```
Ref1   {(Ref1,mkcell(-∞,⌈ PID1 ⌉,Ref0)),   {}
        (Ref0,mkcell(∞,⌈ PID1 ⌉,null))}
================================================================================
```

```
statistics
p cnf 40640 117513
primary variables: 1233
translation time: 3563 ms
solving time: 249 ms
```

Korrigiertes Lemma:

$$\texttt{lemma11}: \quad reachable(r, r_1, H) \wedge H[r_1].nxt = null \wedge SetAbs(r, H, s) \rightarrow \mathbf{H[r_1].i\infty} = \infty$$

Lemma 12.
Das folgende Lemma ist falsch:

$$\texttt{lemma12}: \quad reachable(r, r_1, H) \wedge SetAbs(r, H, s) \rightarrow H[r_1].i\infty = -\infty \wedge H[r_1].i\infty = \infty$$

Kodkod berechnet ein Gegenbeispiel:

```
counterexample for lemma12:

variables assignment:
s={int0}
r=Ref3
r1=Ref2
H={(Ref2,mkcell(⌜ int0 ⌝,None,Ref0)),
   (Ref3,mkcell(-∞,None,Ref2)),
   (Ref0,mkcell(∞,⌈ PID1 ⌉,null))}

MODEL:

function table for SetAbs : Ref x heap x set -> bool
--------------------------------------------------------------------------------
Ref3   {(Ref2,mkcell(⌜ int0 ⌝,None,Ref0)),   {int0}
   (Ref3,mkcell(-∞,None,Ref2)),
   (Ref0,mkcell(∞,⌈ PID1 ⌉,null))}
================================================================================
```

```
statistics
p cnf 235271 744307
primary variables: 2892
translation time: 35355 ms
solving time: 23313 ms
```

Der Fall von $H[r_1].i\infty = \lceil i \rceil$ ist vergessen worden.

Lemma 13.

Im folgenden Lemma wird das ausklnken des mittleren Teils der Kette beschrieben:

$$\texttt{lemma13}: \quad reachable(r, r_1, H) \wedge SetAbs(r, H, s) \wedge H[r_1].nxt = null \rightarrow SetAbs(r, H[r, mkcell(i\infty, id, r_1)], \emptyset)$$

Das Lemma ist falsch, weil, dabei der Wert $H[r].i\infty = -\infty$ überschrieben wird und die Kette nicht mehr korrekt aufgebaut wird:

```
counterexample for lemma13:

variables assignment:
id=⌈ PID1 ⌉
s={}
r=Ref2
i_∞=∞
r1=Ref1
H={(Ref1,mkcell(∞,⌈ PID1 ⌉,null)),
   (Ref2,mkcell(-∞,⌈ PID1 ⌉,Ref1))}

MODEL:

function table for SetAbs : Ref x heap x set -> bool
--------------------------------------------------------------------------------
Ref2   {(Ref1,mkcell(∞,⌈ PID1 ⌉,null)),   {}
        (Ref2,mkcell(-∞,⌈ PID1 ⌉,Ref1))}
================================================================================


statistics
p cnf 236401 745985
primary variables: 2900
translation time: 33581 ms
solving time: 17172 ms
```

Lemma 14.

Laut folgendem Lemma verändert das Einfügen einer Zelle die Erreichbarkeit $Head \rightarrow r$ nicht:

$$\texttt{lemma14}: \quad reachable(Head, r, H[r_0, ce]) \rightarrow reachable(Head, r, H)$$

Das ist falsch:

```
counterexample for lemma14:

variables assignment:
Head=Ref1
r=Ref1
r0=Ref1
ce=mkcell(-∞,⌜ PID1 ⌝,Ref1)
H={}

MODEL:

function table for reachable : Ref x Ref x heap -> bool
--------------------------------------------------------------------------------
Ref1   Ref1   {(Ref1,mkcell(-∞,⌈ PID1 ⌉,Ref1))}
================================================================================

statistics
p cnf 5624 13497
primary variables: 410
translation time: 254 ms
solving time: 13 ms
```

Die Korrektur des Lemmas benutzt die Abstraktion *SetAbs*:

$$\texttt{lemma14}: \quad reachable(Head, r, H[r_0, ce]) \wedge \mathbf{SetAbs}(\mathbf{Head}, \mathbf{H}, \mathbf{S}) \rightarrow reachable(Head, r, H)$$

Lemma 15.

Das folgende Lemma beschreibt das Ausklinken einer Zelle in der Halde im Bezug auf *SetAbs*:

$$\texttt{lemma15}: \quad H[H[r_1].nxt].i\infty = \lceil i \rceil \wedge H[r_1].nxt \neq null \wedge reachable(r, r_1, H) \wedge SetAbs(r, H, s) \rightarrow SetAbs(r, H[r_1, mkcell(H[r_1].i\infty, H[r_1].pin, H[H[r_1].nxt].nxt)], s)$$

Dabei wird ein Fehler gemacht, dass von Kodkod gleich entdeckt wird:

```
counterexample for lemma15:

variables assignment:
s={int2}
r=Ref1
r1=Ref1
H={(Ref3,mkcell(⌜ int2 ⌝,None,Ref2)),
   (Ref2,mkcell(∞,None,null)),
   (Ref1,mkcell(-∞,⌈ PID0 ⌉,Ref3))}
i=int2
```

```
MODEL:

function table for SetAbs : Ref x heap x set -> bool
-------------------------------------------------------------------------------
Ref1   {(Ref3,mkcell(⌜ int2 ⌝,None,Ref2)),   {}
        (Ref2,mkcell(∞,None,null)),
        (Ref1,mkcell(-∞,⌈ PID0 ⌉,Ref2))}
Ref1   {(Ref3,mkcell(⌜ int2 ⌝,None,Ref2)),   {int2}
        (Ref2,mkcell(∞,None,null)),
        (Ref1,mkcell(-∞,⌈ PID0 ⌉,Ref3))}
===============================================================================

statistics
p cnf 754939 2453637
primary variables: 5053
translation time: 239126 ms
solving time: 616264 ms
```

Es wurde vergessen die Ergebnissmenge um i zuverkleinern: $s - -i$. Die Korrektur ist:

`lemma15` : $H[H[r_1].nxt].i\infty = \lceil i \rceil \wedge H[r_1].nxt \neq null \wedge reachable(r, r_1, H) \wedge SetAbs(r, H, s) \rightarrow SetAbs(r, H[r_1, mkcell(H[r_1].i\infty, H[r_1].pin, H[H[r_1].nxt].nxt)], \mathbf{s - -i})$

B.3 Sicherheitsmodell für SmacOS

Zwei Klassen von Theoremen wurden betrachtet: gewöhnliche Lemmas, die die gewünschten Eigenschaften der spezifizierten Operationen beschreiben und die Gruppe von Theoremen, die die zentrale Sicherheitseigenschaft der "Noninterference" beschreiben. Schellhorn et al. [46] reduzieren die *globale* Eigenschaft der Noninterference, die über alle möglichen Systemabläufe aussagt, zu 8 *lokalen* Eigenschaften, die für jeden einzelnen Systemaufruf gelten sollen:

(1) $\sim^d$ *ist reflexiv, symmetrisch, transitiv*

(2) $inv(sys) \wedge inv(sys') \wedge sys \sim^{dom(sys,co)} sys' \rightarrow exec(sys, co).2 = exec(sys', co).2$
(Systemausgabe ist konsistent)

(3) $inv(sys) \wedge dom(sys, co) \not\rightsquigarrow d_0 \rightarrow sys \sim^{d_0} exec(sys, co).1$
(System beachtet lokal $\rightsquigarrow$*)*

(4) $inv(sys) \wedge inv(sys') \wedge sys \sim^d sys' \wedge sys \sim^{dom(sys,co)} sys' \rightarrow exec(sys, co) \sim^d exec(sys', co)$
(System ist schwach schrittkonsistent)

(5) $sys \sim^d sys' \rightarrow (dom(sys, co) \rightsquigarrow d \leftrightarrow dom(sys', co) \rightsquigarrow d)$

$(Syscalls\ beachten\ \rightsquigarrow)$

(6) $sys \sim^{dom(sys,co)} sys' \rightarrow dom(sys, co) = dom(sys', co)$

$(Syscalls\ beachten\ \sim)$

(7) $inv(initstate)$

(8) $inv(sys) \rightarrow inv(exec(sys, co).1)$

$(Invariante)$

Das Checken von diesen Eigenschaften mit Kodkod hat mehrere Fehler in der SmacOS Spezifikation aufgedeckt, die wir im Folgenden einzeln betrachten.

Fehler 1.

Der Systemaufruf *openrd(fp)* öffnet eine Datei zum Lesen und fügt es in die Menge den geöffneten Dateien. Die Spezifikation von *openrd* ist:

$$
\begin{array}{l}
exec(sys, openrd(fp) = \\
\quad \textbf{if} \\
\quad\quad sys.cap \neq OS \wedge fp \in sys.fs \wedge filep(sys.fs[fp]) \\
\quad \textbf{then} \\
\quad\quad mksys(\\
\quad\quad\quad sys.fs, sys.as, sys.ck, sys.cap, sys.ofs + rd(sys.cap, fp)) \\
\quad\quad) \times YES \\
\quad \textbf{else} \\
\quad\quad sys \times NO
\end{array}
$$

Diese Spezifikation verletzt die Sicherheitseigenschaft (8), da die Überprüfung auf die Lesbarkeit der geöffneten Datei nicht stattfindet. Damit kann in die Menge der geöffnet Dateien $sys.ofs + rd(sys.cap, fp)$ eine Datei aufgenommen werden, die von der Anwendung $sys.cap$ nicht lesbar ist und das Prädikat $validrdofs : systemstate$ in dem Zustand falsch ist:

$$
\begin{array}{ll}
validrdofs(sys) \leftrightarrow & (\forall fp.\ \neg rd(os, fp) \in sys.ofs) \wedge \\
& (\forall fp, fp_0.\ rd(mkap(fp), fp_0) \in sys.ofs \rightarrow \\
& \qquad fp \in sys.fs \wedge filep(getfs(fp, sys.fs)) \wedge \\
& \qquad getfs(fp, sys.fs).mrk \neq none \wedge filep(getfs(fp_0, sys.fs)) \wedge \\
& \qquad rdable(getfs(fp, sys.fs).mrk.dm, fp_0, sys.fs) \wedge \\
& \qquad rdable(getfs(fp, sys.fs).mrk.dm, fp0.fpbutlast, sys.fs))
\end{array}
$$

Dieses Prädikat ist für die Invariante wichtig:

$$
\begin{array}{ll}
inv(sys) \leftrightarrow & legalp(sys.fs) \wedge validcap(sys) \wedge rootap(sys.fs) \wedge \\
& compatfsp(sys.fs) \wedge validrdofs(sys) \wedge validwrofs(sys)
\end{array}
$$

Kodkod findet ein Gegenbeispiel für die Eigenschaft (8) wo eine unlesbare Datei zum Lesen geöffnet wird und damit die Invariante im nachfolgenden Zustand[2] nicht mehr gilt:

```
counterexample for lemma08:

variables assignment:
co=openrd([fileid1])
sys=({([],mkdir([fileid1],mkacc(system-high,system-low))),
       ([fileid1],mkfi(data2,mkacc(system-high,access-class0),
                       mkmrk(mkdm(system-high,system-high, system-low,system-low))))},
     {},cardkey,mkap([fileid1]),{})

MODEL:

function table for inv : systemstate -> bool
------------------------------------------------------------------------------
({([],mkdir([fileid1],system-high,system-low)),
       ([fileid1],mkfi(data2,system-high,access-class0,
                       mkmrk(mkdm(system-high,system-high, system-low,system-low))))},
     {},cardkey,mkap([fileid1]),{})
initialstate
==============================================================================

function table for exec : systemstate x command -> systemstateandoutput
------------------------------------------------------------------------------
({([],mkdir([fileid1],system-high,system-low)),
       ([fileid1],mkfi(data2,mkacc(system-high,access-class0),
                       mkmrk(mkdm(system-high,system-high, system-low,system-low))))},
     {},cardkey,mkap([fileid1]),{})
openrd([fileid1])
@times(({([],mkdir([fileid1],system-high,system-low)),
               ([fileid1],mkfi(data2,system-high,access-class0,
                               mkmrk(mkdm(system-high,system-high, system-low,system-low))))},
     {},cardkey,mkap([fileid1]),{rd(mkap([fileid1]),[fileid1])}), yes)

...
initialstate        openrd([fileid1])  @times(initialstate,no)
==============================================================================

statistics
p cnf 2775271 7907272
primary variables: 4906
translation time: 109903 ms
solving time: 153740 ms
```

Die geöffnete Datei ist unter dem Adresse $[fileid1]$ gespeichert, hat die Zugriffsrechte (sys_{high}, ac_0) und ist damit für die Anwendung $mkap([fileid1])$ mit der Sicherheitsdomäne $mkdm(sys_{high}, sys_{high}, sys_{low}, sys_{low})$ nicht lesbar. Die Spezifikation von *openrd* wird verbessert indem eine Zusatzbedingung gefordert wird: $rdable(getfs(sys.cap.ap, sys.fs).mrk.dm, fp, sys.fs)$.

$$exec(sys, openrd(fp) = \\ \textbf{if}$$

[2]Ein Systemzustand sys wird mit dem 5-Tupel dargestellt: $sys = (fs : FileSystem, as : AuthStore, ck : Auth, cap : Ap - or - OS, ofs : OpenFileSet)$.

$$
\begin{array}{l}
\quad sys.cap \neq OS \wedge fp \in sys.fs \wedge filep(sys.fs[fp]) \wedge \\
\quad \mathbf{rdable}(\mathbf{getfs}(\mathbf{sys.cap.ap}, \mathbf{sys.fs}).\mathbf{mrk.dm}, \mathbf{fp}, \mathbf{sys.fs}) \\
\textbf{then} \\
\quad mksys(\\
\qquad sys.fs, sys.as, sys.ck, sys.cap, sys.ofs + rd(sys.cap, fp)) \\
\quad) \times YES \\
\textbf{else} \\
\quad sys \times NO
\end{array}
$$

Bei der Fehlersuche können dem Kodkod die Schranken explizit vorgegeben werden indem eine Datei "scope" im Unterverzeichniss "kodkod" angelegt wird. Für diese Suche wurden folgenden Schranken verwendet:

```
all,3
nat,4
fileid-list,4
file,4
filesystem,4
```

Fehler 2.

Beim Checken der Eigenschaft (8) wurde ein weiterer Fehler in der Spzifikation von *setintsec* endeckt. Den Endeckung des ersten Fehler in der Spzifikation von *setintsec* ist im Abschnitt 7.4.2 detailiert beschrieben. Die Spezifikation wurde durch Verstärkung der Vorbedingung korrigiert:

$$
\begin{array}{l}
exec(sys, setintsec(fp, iac, sac) = \\
\quad \textbf{if} \\
\quad\quad sys.cap \neq OS \wedge fp \in sys.fs \wedge filep(sys.fs[fp]) \\
\quad\quad rdable(sys.fs[sys.cap.ap].mrk.dm, butlast(fp), sys.fs) \wedge \\
\quad\quad wrable(sys.fs[sys.cap.ap].mrk.dm, butlast(fp), sys.fs) \\
\quad \textbf{then} \\
\quad\quad mksys(\\
\quad\quad\quad sys.fs[fp, mkfi(sys.fs[fp].cont, iac, sac, sys.fs[fp].mrk)], \\
\quad\quad\quad sys.as, sys.ck, sys.cap, sys.ofs) \times YES \\
\quad \textbf{else} \\
\quad\quad sys \times NO
\end{array}
$$

Nun ist hier ein weiterer Fehler versteckt, der erst beim Checken der Eigenschaft (8) aufgedeckt wird. Die Zugriffsrechte einer Datei werden so verändert, dass es im nachfolgenden Zustand von dem Agenten nicht mehr lesbar ist. Dies wäre kein Problem, wäre diese Datei nicht in der Menge der zum Lesen geöffneten Dateien enthalten. D.h. im nachfolgenden Zustan gilt die Invariante *inv* nicht mehr, weil eine unlesbare Datei geöffnet ist. Kodkod berechnet das entsprechende Gegenbeispiel:

```
counterexample for lemma08:

variables assignment:
co=setintsec([fileid1],access-class0,system-high)
sys=({([],mkdir([fileid1],system-high,system-low)),
      ([fileid1],mkfi(data2,system-low,access-class0,
                      mkmrk(mkdm(system-low,system-high,access-class0,system-low))))},
     {(applname2,authentication1)},cardkey,mkap([fileid1]),{rd(mkap([fileid1]),[fileid1])})

MODEL:

function table for inv : systemstate -> bool
------------------------------------------------------------------------------
({([],mkdir([fileid1],system-high,system-low)),
      ([fileid1],mkfi(data2,system-low,access-class0,
                      mkmrk(mkdm(system-low,system-high,access-class0,system-low))))},
     {(applname2,authentication1)},cardkey,mkap([fileid1]),{rd(mkap([fileid1]),[fileid1])})
initialstate
==============================================================================

function table for exec : systemstate x command -> systemstateandoutput
------------------------------------------------------------------------------
({([],mkdir([fileid1],system-high,system-low)),
  ([fileid1],mkfi(data2,system-low,access-class0,
                  mkmrk(mkdm(system-low,system-high,access-class0,system-low))))},
 {(applname2,authentication1)},cardkey,mkap([fileid1]),{rd(mkap([fileid1]),[fileid1])})
setintsec(@oplusfp(fileid1,@fp),access-class0,system-high)
@times(({([],mkdir([fileid1],system-high,system-low)),
         ([fileid1],mkfi(data2,access-class0,system-high,
                         mkmrk(mkdm(system-low,system-high,access-class0,system-low))))},
        {(applname2,authentication1)},cardkey,mkap([fileid1]),{rd(mkap([fileid1]),[fileid1])}),yes)
...
==============================================================================

statistics
p cnf 4047088 11379353
primary variables: 5528
translation time: 171166 ms
solving time: 350820 ms
```

Die Datei unter der Adresse $[fileid1]$ hat Zugriffsrechte (sys_{low}, ac_0), ist für den Agenten mit der Sicherheitsdomäne $(sys_{low}, sys_{high}, ac_0, sys_{low})$ lesbar und ist in der Menge der geöffneten Dateien enthalten. Nun wird der Systemaufruf $setintsec([fileid1], ac_0, sys_{high})$ gemacht und die Datei ist nicht mehr lesbar. Damit ist das Prädikat *validrdofs* im nachfolgenden Zustand falsch und die Invariante ist verletzt.

Die Spezifikation von *setintsec* wird verbessert indem nach einer erfolgreichen Veränderung der Zugriffsrechte die gegebenenfalls geöffnete Datei aus der Menge $sys.ofs$ entfernt wird:

$$
\begin{aligned}
&exec(sys, setintsec(fp, iac, sac) = \\
&\quad \textbf{if} \\
&\qquad sys.cap \neq OS \wedge fp \in sys.fs \wedge filep(sys.fs[fp]) \\
&\qquad rdable(sys.fs[sys.cap.ap].mrk.dm, butlast(fp), sys.fs) \wedge \\
&\qquad wrable(sys.fs[sys.cap.ap].mrk.dm, butlast(fp), sys.fs) \wedge
\end{aligned}
$$

then

$mksys($

$sys.fs[fp, mkfi(sys.fs[fp].cont, iac, sac, sys.fs[fp].mrk)],$

$sys.as, sys.ck, sys.cap, \mathbf{ofsrmfi(fp, sys.ofs)}) \times YES$

else

$sys \times NO$

Fehler 3.

Der Systemaufruf *openwr(fp)* öffnet eine Datei zum beschreiben. Die folgende Spezifikation enthält einen Fehler, der zur Verletzung der Invariante nach der Ausführung führt:

$exec(sys, openwr(fp)) =$

if

$sys.cap \neq OS \wedge fp \in sys.fs \wedge filep(sys.fs[fp]) \wedge$

$getfs(fp, sys.fs).mrk = none \wedge$

$rdable(getfs(sys.cap.ap, sys.fs).mrk.dm, fp.fpbutlast, sys.fs)$

then

$mksys($

$sys.fs, sys.as, sys.ck, sys.cap, sys.ofs + wr(sys.cap, fp))$

$) \times YES$

else

$sys \times NO$

Für eine erfolgreiche Ausführung wird insbesondere verlangt, dass das Verzeichniss in dem die Datei liegt lesbar ist. Die beschreibbarkeit der Datei selbst wird nicht überpüft. Kodkod berechnet ein Gegenbeispiel:

```
counterexample for lemma08:

variables assignment:
co=openwr([fileid2])
sys=({([],mkdir([fileid2,fileid1]),system-high,system-low),
      ([fileid2],mkfi(data2,system-high,system-low,none)),
      ([fileid1],mkfi(data2,system-high,system-low,
                      mkmrk(mkdm(system-high,system-high,system-low,system-high))))},
     {},cardkey,mkap([fileid1]),{})

MODEL:

function table for inv : systemstate -> bool
------------------------------------------------------------------------------
({([],mkdir([fileid2,fileid1]),system-high,system-low),
      ([fileid2],mkfi(data2,system-high,system-low,none)),
      ([fileid1],mkfi(data2,system-high,system-low,
                      mkmrk(mkdm(system-high,system-high,system-low,system-high))))},
     {},cardkey,mkap([fileid1]),{})
initialstate
==============================================================================
```

```
function table for exec : systemstate x command -> systemstateandoutput
------------------------------------------------------------------------------
({([],mkdir([fileid2,fileid1]),system-high,system-low),
      ([fileid2],mkfi(data2,system-high,system-low,none)),
      ([fileid1],mkfi(data2,system-high,system-low,
                      mkmrk(mkdm(system-high,system-high,system-low,system-high))))},
     {},cardkey,mkap([fileid1]),{})
openwr(@oplusfp(fileid2,@fp))
@times(
({([],mkdir([fileid2,fileid1]),system-high,system-low),
      ([fileid2],mkfi(data2,system-high,system-low,none)),
      ([fileid1],mkfi(data2,system-high,system-low,
                      mkmrk(mkdm(system-high,system-high,system-low,system-high))))},
     {},cardkey,mkap([fileid1]),{wr(mkap([fileid1]),[fileid2])}),yes)
...
==============================================================================

statistics
p cnf 2381585 6761608
primary variables: 5356
translation time: 105440 ms
solving time: 107016 ms
```

Die Spezifikation von *openwr* wird verbessert indem eine Zusatzbedingung gefordert wird:

$$wrable(getfs(sys.cap.ap, sys.fs).mrk.dm, fp, sys.fs)$$

Bei der Fehlersuche wurden folgenden Schranken verwendet:

```
all,3
nat,4
fileid-list,4
file,4
filesystem,5
```

Fehler 4.

Der Systemaufruf *write(fp,da)* schreibt Daten *da* in die Datei unter der Adresse *fp*. Für eine erfolgreiche Ausführung wird vorausgesetzt, dass die zu beschreibende Datei bereits zum Schreiben geöffneten ist:

$$\begin{array}{l} exec(sys, write(fp, da) = \\ \quad \textbf{if} \\ \quad\quad sys.cap \neq OS \wedge wr(sys.cap, fp) \in sys.ofs \\ \quad \textbf{then} \\ \quad\quad mksys(\\ \quad\quad\quad insfs(fp, mkfi(da, getfs(fp, sys.fs).icl, getfs(fp, sys.fs).scl, \\ \quad\quad\quad\quad\quad\quad\quad\quad getfs(fp, sys.fs).mrk), sys.fs), \\ \quad\quad\quad sys.as, sys.ck, sys.cap, sys.ofs) \end{array}$$

$$) \times YES$$

else

$$sys \times NO$$

Die Sicherheitseigenschaft (3) behauptet, dass die Systemafrufe die Interferenz Relation $\rightsquigarrow$ beachten:

$$(3) \qquad inv(sys) \wedge dom(sys, co) \not\rightsquigarrow d_0 \rightarrow sys \sim^{d_0} exec(sys, co).1$$

Der Zustand sys soll dabei die Invariante inv erfüllen. Die Invariante implizit insbesondere, dass die zum Schreiben geöffneten Dateien "legal" die Zuggriffsrechte respektieren:

$$\begin{aligned} validwrofs(sys) \leftrightarrow \quad & (\forall fp.\ \neg wr(os, fp) \in sys.ofs) \wedge \\ & (\forall fp, fp_0.\ wr(mkap(fp), fp_0) \in sys.ofs \rightarrow \\ & \qquad fp \in sys.fs \wedge filep(getfs(fp, sys.fs)) \wedge \\ & \qquad getfs(fp, sys.fs).mrk \neq none \wedge filep(getfs(fp_0, sys.fs)) \wedge \\ & \qquad wrable(getfs(fp, sys.fs).mrk.dm, fp_0, sys.fs) \wedge \\ & \qquad rdable(getfs(fp, sys.fs).mrk.dm, fp0.fpbutlast, sys.fs)) \end{aligned}$$

Diese Spezifikation läßt auch den Fall zu, wo die geöffnete Datei eine Anwendung ist: $getfs(fp_0, sys.fs).mrk \neq none$. In diesem Fall wird der Inhalt der Datei überschrieben. Dies führt zur Verletzung der Sicherheitseigenschaft (3). Kodkod findet ein Gegenbeispiel:

```
counterexample for lemma03:

variables assignment:
d0=mkdm(system-high,access-class0,system-low,access-class0)
co=write([fileid1],data1)
sys=({([],     mkdir([fileid1],system-high,system-low)),
      ([fileid1],mkfi(data2,access-class0,access-class0,
                      mkmrk(mkdm(system-high,access-class0,system-low,access-class0))))},
     {(applname0,cardkey)},cardkey,mkap(@oplusfp(fileid1,@fp)),
     {wr(mkap([fileid1]),[fileid1])})

MODEL:

function table for inv : systemstate -> bool
------------------------------------------------------------------------------
({([],     mkdir([fileid1],system-high,system-low)),
      ([fileid1],mkfi(data2,access-class0,access-class0,
                      mkmrk(mkdm(system-high,access-class0,system-low,access-class0))))},
     {(applname0,cardkey)},cardkey,mkap(@oplusfp(fileid1,@fp)),
     {wr(mkap([fileid1]),[fileid1])})

({([],     mkdir([fileid1],system-high,system-low)),
      ([fileid1],mkfi(data1,access-class0,access-class0,
                      mkmrk(mkdm(system-high,access-class0,system-low,access-class0))))},
     {(applname0,cardkey)},cardkey,mkap(@oplusfp(fileid1,@fp)),
     {wr(mkap([fileid1]),[fileid1])})
```

```
initialstate
================================================================================

function table for exec : systemstate x command -> systemstateandoutput
--------------------------------------------------------------------------------
({([],     mkdir([fileid1],system-high,system-low)),
      ([fileid1],mkfi(data2,access-class0,access-class0,
                      mkmrk(mkdm(system-high,access-class0,system-low,access-class0))))},
     {(applname0,cardkey)},cardkey,mkap(@oplusfp(fileid1,@fp)),
     {wr(mkap([fileid1]),[fileid1])})
write([fileid1],data1)
@times(({([],     mkdir([fileid1],system-high,system-low)),
      ([fileid1],mkfi(data1,access-class0,access-class0,
                      mkmrk(mkdm(system-high,access-class0,system-low,access-class0))))},
     {(applname0,cardkey)},cardkey,mkap(@oplusfp(fileid1,@fp)),
     {wr(mkap([fileid1]),[fileid1])}),yes)
...
================================================================================

statistics
p cnf 2303680 6489002
primary variables: 5380
translation time: 104156 ms
solving time: 152804 ms
```

Wir korrigieren die Spezifikation von *validwrofs*:

$$
\begin{aligned}
validwrofs(sys) \leftrightarrow \quad & (\forall fp.\ \neg wr(os, fp) \in sys.ofs) \wedge \\
& (\forall fp, fp_0.\ wr(mkap(fp), fp_0) \in sys.ofs \rightarrow \\
& \qquad fp \in sys.fs \wedge filep(getfs(fp, sys.fs)) \wedge \\
& \qquad getfs(fp, sys.fs).mrk \neq none \wedge filep(getfs(fp_0, sys.fs)) \wedge \\
& \qquad wrable(getfs(fp, sys.fs).mrk.dm, fp_0, sys.fs) \wedge \\
& \qquad rdable(getfs(fp, sys.fs).mrk.dm, fp_0.fpbutlast, sys.fs) \wedge \\
& \qquad \mathbf{getfs(fp_0, sys.fs).mrk = none})
\end{aligned}
$$

Am meisten wurden mit Gegenbeispielen nicht Fehler in der Spezifikation aufgedeckt sondern falsche Lemmas. Im folgenden werden diese Lemmas genau beschrieben.

Lemma 1.

Das Prädikat $eqcont(fs, dm, fs_0)$ testet ob es für die Domäne dm auf den Dateisystemen fs und fs_0 die gleichen Informationen lesbar sind:

$$eqcont(fs, dm, fs_0) \leftrightarrow \forall fp. \quad rdable(dm, fp, fs) \wedge filep(fp, fs) \rightarrow getfs(fp, fs) = getfs(fp, fs_0)$$

Das folgende Lemma vermutet die Transitivität von *eqcont*:

$$eqcont(fs_1, dm, fs_2) \wedge eqcont(fs_2, dm, fs_3) \rightarrow eqcont(fs_1, dm, fs_3)$$

Das Lemma ist falsch und Kodkod findet ein Gegenbeispiel:

```
counterexample for lemma01:

variables assignment:
fs1={([],mkfi(nodata,system-high,system-low,
              mkmrk(mkdm(system-high,system-high,system-low,system-low))))}
fs2={}
fs3={([],mkdir([],mkacc(system-high,system-low)))}
dm=mkdm(system-high,system-high,system-low,system-low)

MODEL:

function table for eqcont : filesystem x domain x filesystem -> bool
------------------------------------------------------------------------------
{([],mkfi(nodata,system-high,system-low,
              mkmrk(mkdm(system-high,system-high,system-low,system-low))))}
mkdm(system-high,system-high,system-low,system-low)
{}

{}
mkdm(system-high,system-high,system-low,system-low)
{([],mkdir([],mkacc(system-high,system-low)))}
...
==============================================================================

function table for rdable : domain x filepath x filesystem -> bool
------------------------------------------------------------------------------
mkdm(system-high,system-high,system-low,system-low)  []
{([],mkdir([],mkacc(system-high,system-low)))}

mkdm(system-high,system-high,system-low,system-low)  []
{([],mkfi(nodata,system-high,system-low,
          mkmrk(mkdm(system-high,system-high,system-low,system-low))))}
..
==============================================================================

statistics
p cnf 814858 2496996
primary variables: 4136
translation time: 32716 ms
solving time: 8194 ms
```

Dises Gegenbeispiel zeigt ein Randfall, wo $fs_2 = \{\}$ (leer) gilt. Es gilt $eqcont(fs_1, dm, fs_2)$ weil $getfs(fp, \{\})$ unterspezifiziert ist und damit auch $getfs([], \{\}) = getfs([], fs_1)$ gelten kann, wie, z.B., im Gegenbeispiel. $eqcont(fs_2, dm, fs_3)$ gilt trivialerweise, weil fs_2 leer ist. $eqcont(fs_1, dm, fs_3)$ gilt aber nicht, weil $getfs([], fs_1) \neq getfs([], fs_3)$ gilt.

Das Lemma wird korrigiert, indem die Vorbedingung verstärkt wird:

$$\mathbf{eqrd(fs_1, dm, fs_2)} \wedge eqcont(fs_1, dm, fs_2) \wedge eqcont(fs_2, dm, fs_3) \rightarrow eqcont(fs_1, dm, fs_3)$$

Mit dem Prädikat *eqrd* wird garantiert, dass die Mengen der lesbaren Dateien, die auch tatsächlich in fs_1 und fs_2 gespeichert sind, gleich sind:

$$eqrd(fs, dm, fs_0) \leftrightarrow (\forall fp.\ rdable(dm, fp, fs) \leftrightarrow rdable(dm, fp, fs_0))$$

Lemma 2.

Es wird untersucht ob die Systemaufrufe bestimmte Eigenschaften von Systemzuständen bzw. Dateisystemen erhalten, z.B. $legalp : filesystem$, $validwrofs : systemstate$ etc. Als erste Eigenschaft betrachten wir die Invarianz von *rootok*:

$$rootok(sys.fs) \rightarrow rootok(exec(sys, co).1.fs);$$

Das Prädikat *rootok* ist wie folgt spezifiziert:

$$\begin{aligned} rootok(fs) \leftrightarrow \quad & [] \in fs \wedge dirp(getfs([], fs)) \wedge \\ & getfs([], fs).icl = sys_{high} \wedge \\ & getfs([], fs).scl = sys_{low} \end{aligned}$$

Diese Eigenschaft gilt für den Systemaufruf *write(fp,data)* nicht, wie ein von Kodkod berechnetes Gegenbeispiel zeigt:

```
counterexample for lemma02:

variables assignment:
co=write([],data1)
sys=({([],mkdir([fileid1],system-high,system-low))},
     {(applname0,cardkey)},authentication1,mkap([fileid0]),
     {wr(mkap([fileid0]),[]),wr(os,[fileid0,fileid0])})

MODEL:

function table for exec : systemstate x command -> systemstateandoutput
------------------------------------------------------------------------------
({([],mkdir([fileid1],system-high,system-low))},
     {(applname0,cardkey)},authentication1,mkap([fileid0]),
     {wr(mkap([fileid0]),[]),wr(os,[fileid0,fileid0])})
write([],data1)
@times(({([],mkfi(data1,mkacc(system-high,system-low),none))},
```

```
     {(applname0,cardkey)},authentication1,mkap([fileid0]),
     {wr(mkap([fileid0]),[]),wr(os,[fileid0,fileid0])}),yes)
...
================================================================================

function table for rootok : filesystem -> bool
--------------------------------------------------------------------------------
{([],mkdir([fileid1],system-high,system-low))}
emptyfs
================================================================================

statistics
p cnf 2304211 6491185
primary variables: 5384
translation time: 102191 ms
solving time: 96640 ms
```

Wie Das Gegenbeispiel zeigt, gilt für das Dateisystem im nachfolgenden Zustand die *rootok* Eigenschaft nicht, weil das *root*-Verzeichnis durch eine Datei ersetzt wurde. Um das zuverhindern wird im Lemma zusätzlich verlangt, dass die zum Schreiben geöffneten Dateien valide sind, d.h., dass insbesondere keine Verzeichnisse übeschrieben werden können:

$$rootok(sys.fs) \wedge \mathbf{validwrofs}(\mathbf{sys}) \rightarrow rootok(exec(sys, co).1.fs);$$

Lemma 3.

Betrachten wir die Eigenschaft *upclosedp* von einem Dateisystem:

$$upclosedp(fs) \leftrightarrow (\forall fp, fid. \quad fp + fid \in fs \rightarrow fp \in fs \wedge dirp(getfs(fp, fs)) \wedge fid \in getfs(fp, fs).dircont)$$

Für jeden Pfad $fp \neq []$ soll im Dateisystem das Prefix $fp.butlast$ auch existieren und darunter ein Verzeichnis gespeichert sein. Es wird das Lemma untersucht, das die Erhaltung von *upclosedp* durch Systemmaufrufe formuliert:

$$upclosedp(sys.fs) \rightarrow upclosedp(exec(sys, co).1.fs);$$

Das Lemma ist falsch und Kodkod berechnet ein Gegenbeispiel:

```
counterexample for lemma03:

variables assignment:
co=write([],data1)
sys=({([],mkdir([fileid1],access-class0,system-low)),
      ([fileid1],mkdir([],system-high,system-low))},
     {(applname0,authentication1)},cardkey,mkap([fileid1,fileid1]),
     {wr(mkap([fileid1,fileid1]),[])})

MODEL:
```

```
function table for exec : systemstate x command -> systemstateandoutput
------------------------------------------------------------------------------
({([],mkdir([fileid1],access-class0,system-low)),
      ([fileid1],mkdir([],system-high,system-low))},
     {(applname0,authentication1)},cardkey,mkap([fileid1,fileid1]),
     {wr(mkap([fileid1,fileid1]),[])})
write([],data1)
@times(({([],mkfi(data1,access-class0,system-low,none)),
      ([fileid1],mkdir([],system-high,system-low))},
     {(applname0,authentication1)},cardkey,mkap([fileid1,fileid1]),
     {wr(mkap([fileid1,fileid1]),[])}),yes)
...
==============================================================================

function table for upclosedp : filesystem -> bool
------------------------------------------------------------------------------
emptyfs
{([],mkdir([fileid1],access-class0,system-low)),
      ([fileid1],mkdir([],system-high,system-low))}
{}
==============================================================================

statistics
p cnf 2304211 6491185
primary variables: 5384
translation time: 96976 ms
solving time: 144967 ms
```

Wie Das Gegenbeispiel zeigt, gilt für das Dateisystem im nachfolgenden Zustand die *upclosedp* Eigenschaft nicht, weil das *root*-Verzeichnis durch eine Datei ersetzt wurde. Um das zuverhindern wird im Lemma zusätzlich verlangt, dass die zum Schreiben geöffneten Dateien valide sind, d.h., dass insbesondere keine Verzeichnisse übeschrieben werden können:

$$upclosedp(sys.fs) \wedge \mathbf{validwrofs(sys)} \rightarrow upclosedp(exec(sys, co).1.fs);$$

Lemma 4.

Die Eigenschaft *rootap* von einem Dateisystem besagt, dass alle Anwendungen $file.mrk \neq none$ nur im *root*-Verzeichnis gespeichert sind:

$$\begin{aligned} rootap(fs) \leftrightarrow (\forall fp. \quad & fp \in fs \wedge filep(getfs(fp, fs)) \wedge \\ & getfs(fp, fs).mrk \neq none \rightarrow \\ & fp.fpbutlast = []) \end{aligned}$$

Das Lemma beschreibt die Invarianz von *rootap* bezüglich der Systemaufrufe:

$$rootap(sys.fs) \rightarrow rootap(exec(sys, co).1.fs);$$

Kodkod findet ein Gegenbeispiel, dass das Lemma falsifiziert:

```
counterexample for lemma04:

variables assignment:
co=write([],data2)
sys=({([],mkdir([],system-high,system-low))},{(applname2,cardkey)},
       authentication1,mkap([]),{wr(mkap([]),[])})

MODEL:

function table for exec : systemstate x command -> systemstateandoutput
--------------------------------------------------------------------------------
({([],mkdir([],system-high,system-low)))},{(applname2,cardkey)},
       authentication1,mkap([]),{wr(mkap([]),[])})
write([],data2)
@times(({([],mkfi(data2,system-high,system-low,
                  mkmrk(mkdm(access-class0,system-high,system-high,system-low)))))},
       {(applname2,cardkey)},authentication1,mkap([]),{wr(mkap([]),[])}),yes)

initialstate        write([],data2)    @times(initialstate,no)
================================================================================

function table for rootap : filesystem -> bool
--------------------------------------------------------------------------------
{([fileid2],mkdir([fileid2],access-class0,access-class0))}
{([fileid2,fileid2],mkdir([],system-high,system-low))}
emptyfs
{}
================================================================================

statistics
p cnf 2304211 6491185
primary variables: 5384
translation time: 89807 ms
solving time: 114171 ms
```

Systemaufruf verletzt das Lemma, weil in der Menge zum Schreiben geöffneten Dateien unberechtigt die Datei unter dem Pfad [] enthalten ist. Das Problem wird behoben indem die Bedingung $validwrofs(sys)$ zusätzlich verlangt wird:

$$rootap(sys.fs) \wedge \mathbf{validwrofs(sys)} \rightarrow rootap(exec(sys, co).1.fs);$$

Lemma 5.

Wir wollen untersuchen ob die Äquivalenz zweier Zustände bezüglich einer Domäne d dazu führt, dass die Invariante inv sich immer von sys auf sys_0 überträgt:

$$inv(sys) \wedge eqd(sys, d, sys_0) \rightarrow inv(sys_0)$$

Dies gilt nicht immer und Kodkod findet ein Gegenbeispiel:

```
counterexample for lemma05:

variables assignment:
d=mkdm(system-high,system-low,system-low,system-high)
```

```
sys=initialstate
sys0=({([],mkdir([],system-high,system-low)),
       ([fileid1],mkfi(data2,system-high,access-class0,none))},
      {},os,cardkey,{})

MODEL:

function table for inv : systemstate -> bool
--------------------------------------------------------------------------------
initialstate
================================================================================


function table for eqd : systemstate x secdomain x systemstate -> bool
--------------------------------------------------------------------------------
initialstate
mkdm(system-high,system-low,system-low,system-high)
({([],mkdir([],system-high,system-low)),
       ([fileid1],mkfi(data2,system-high,access-class0,none))},
      {},os,cardkey,{})

initialstate
mkdm(system-high,system-low,system-low,system-high)
initialstate
...
================================================================================

statistics
p cnf 3873186 10754748
primary variables: 5652
translation time: 168512 ms
solving time: 289919 ms
```

In diesem Beispiel sind die beiden Zustände sys, sys_0 äquivalent bzgl. d. Trotzdem gilt $upclosed(sys_0.fs)$ nicht:

$$upclosedp(fs) \leftrightarrow (\forall fp, fid. \quad fp + fid \in fs \rightarrow fp \in fs \wedge dirp(getfs(fp, fs)) \wedge fid \in getfs(fp, fs).dircont)$$

Damit gilt die Invariante inv in sys_0 nicht.

Lemma 6.

Das Lemma formuliert die Vermutung, dass wenn eine Datei für d nicht lesbar ist, dann kann der Lesesystemaufruf für diese Datei nicht erfolgreich sein:

$$\neg rdable(d.dom, fp, sys.fs) \rightarrow exec(sys, read(fp)).2 = no$$

Das Lemma ist falsch, da nicht überprüft wird ob die Menge der geöffneten Dateien korrekt ist, d.h. die Zugriffsrechte berückksichtigt. Kodkod findet ein Gegenbeispiel:

```
counterexample for lemma06:
```

```
variables assignment:
d=mkdm(system-high,system-high,system-low,system-low)
sys=({([],mkdir([fileid1,fileid1],system-low,system-high))},
     {},authentication1,mkap([fileid2]),{rd(mkap([fileid2]),[])})
fp=[]

MODEL:

function table for rdable : domain x filepath x filesystem -> bool
------------------------------------------------------------------------------
mkdm(system-high,system-low,system-low,system-high)     []  emptyfs
mkdm(system-high,system-high,system-low,access-class0)  []  emptyfs
mkdm(system-high,system-high,system-low,system-low)     []  emptyfs
==============================================================================

function table for exec : systemstate x command -> systemstateandoutput
------------------------------------------------------------------------------
({([],mkdir([fileid1,fileid1],system-low,system-high))},
     {},authentication1,mkap([fileid2]),{rd(mkap([fileid2]),[])})
read([])
@times(({([],mkdir([fileid1,fileid1],system-low,system-high))},
        {},authentication1,mkap([fileid2]),{rd(mkap([fileid2]),[])}),
       mkoutdata(data2))

initialstate        read([])           @times(initialstate,no)
==============================================================================

statistics
p cnf 3873464 10755218
primary variables: 5669
translation time: 176853 ms
solving time: 346323 ms
```

Eine einfache Korrektur beseitigt den Fehler:

$$\neg rdable(d.dom, fp, sys.fs) \wedge \mathbf{inv(sys)} \rightarrow exec(sys, read(fp)).2 = no$$

Lemma 7.

Zwei Zustände erfüllen die Inavriante und sind äquivalent bezüglich einer Domäne d: $eqd(sys, d, sys_0)$. Es wird vermutet, dass die Ausgaben eines Systemaufrufs auf diesen Zuständen gleich sind:

$$inv(sys) \wedge inv(sys_0) \wedge eqd(sys, d, sys_0) \rightarrow exec(sys, co).2 = exec(sys_0, co).2;$$

Kodkod findet ein Gegenbeispiel, wo die *read* Systemoperation diese Vermutung nicht erfüllt:

```
counterexample for lemma07:

variables assignment:
co=read([fileid2])
d=mkdm(system-high,system-low,access-class0,system-low)
sys=({([],mkdir([fileid2],system-high,system-low)),
      ([fileid2],mkfi(data2,system-low,system-low,
                      mkmrk(mkdm(system-low,system-high,access-class0,system-low))))},
```

```
     {},cardkey,mkap([fileid2]),{})
sys0=({(([],mkdir([fileid2],system-high,system-low))),
       ([fileid2],mkfi(data2,system-low,system-low,
                       mkmrk(mkdm(system-low,system-high,access-class0,system-low))))},
      {},cardkey,mkap([fileid2]),{rd(mkap([fileid2]),[fileid2])})

MODEL:

function table for inv : systemstate -> bool
--------------------------------------------------------------------------------
({([],mkdir([fileid2],system-high,system-low)),
      ([fileid2],mkfi(data2,system-low,system-low,
                      mkmrk(mkdm(system-low,system-high,access-class0,system-low))))},
     {},cardkey,mkap([fileid2]),{})

({(([],mkdir([fileid2],system-high,system-low))),
       ([fileid2],mkfi(data2,system-low,system-low,
                       mkmrk(mkdm(system-low,system-high,access-class0,system-low))))},
      {},cardkey,mkap([fileid2]),{rd(mkap([fileid2]),[fileid2])})

initialstate
================================================================================

function table for eqd : systemstate x secdomain x systemstate -> bool
--------------------------------------------------------------------------------
({([],mkdir([fileid2],system-high,system-low)),
      ([fileid2],mkfi(data2,system-low,system-low,
                      mkmrk(mkdm(system-low,system-high,access-class0,system-low))))},
     {},cardkey,mkap([fileid2]),{})
mkdm(system-high,system-low,access-class0,system-low)
({(([],mkdir([fileid2],system-high,system-low))),
       ([fileid2],mkfi(data2,system-low,system-low,
                       mkmrk(mkdm(system-low,system-high,access-class0,system-low))))},
      {},cardkey,mkap([fileid2]),{rd(mkap([fileid2]),[fileid2])})

...
================================================================================

function table for exec : systemstate x command -> systemstateandoutput
--------------------------------------------------------------------------------
({(([],mkdir([fileid2],system-high,system-low))),
       ([fileid2],mkfi(data2,system-low,system-low,
                       mkmrk(mkdm(system-low,system-high,access-class0,system-low))))},
      {},cardkey,mkap([fileid2]),{rd(mkap([fileid2]),[fileid2])})
read([fileid2])
@times(({(([],mkdir([fileid2],system-high,system-low))),
       ([fileid2],mkfi(data2,system-low,system-low,
                       mkmrk(mkdm(system-low,system-high,access-class0,system-low))))},
      {},cardkey,mkap([fileid2]),{rd(mkap([fileid2]),[fileid2])}),mkoutdata(data2))

({([],mkdir([fileid2],system-high,system-low)),
      ([fileid2],mkfi(data2,system-low,system-low,
                      mkmrk(mkdm(system-low,system-high,access-class0,system-low))))},
     {},cardkey,mkap([fileid2]),{})
read([fileid2])
@times(({([],mkdir([fileid2],system-high,system-low)),
      ([fileid2],mkfi(data2,system-low,system-low,
                      mkmrk(mkdm(system-low,system-high,access-class0,system-low))))},
     {},cardkey,mkap([fileid2]),{}),no)
================================================================================

statistics
```

```
p cnf 3873419 10755061
primary variables: 5667
translation time: 156219 ms
solving time: 285011 ms
```

Die Zustände sind aus Sicht von d gleich, weil die Datei unter der Adresse $[fileid2]$ für d nicht lesbar ist. Für die Domäne $cdom(sys, co)$ (Domäne des aktiven Agenten) ist die Datei lesbar. Da in sys_0 die Datei $[fileid2]$ zum Lesen geöffnet ist, sind die Zusände für die Domäne $cdom(sys, co)$ nicht gleich. Als Folge liefert der Systemaufruf *read([fileid2])* für sys und sys_0 unterschiedlieche Ergebnisse. Das Lemma wird korrigiert indem die Äquivalenz bezüglich $cdom(sys, co)$ verlangt wird:

$$inv(sys) \wedge inv(sys_0) \wedge \mathbf{eqd}(\mathbf{sys}, \mathbf{cdom}(\mathbf{sys}, \mathbf{co}), \mathbf{sys_0}) \rightarrow$$
$$exec(sys, co).2 = exec(sys_0, co).2;$$

Lemma 8.

In einem Zustand, der die Invariante erfüllt und der aktive Agent auf die Datei fp lesend zugreifen kann, sollte die Leseoperation *read(fp)* erfolgreich sein:

$$inv(sys) \wedge rdable(cdom(sys, read(fp)), fp, sys.fs) \rightarrow exec(sys, read(fp)).2 \neq no$$

Das Lemma beachtet nicht, dass die Datei möglicherweise noch nicht zum Lesen geöffnet wurde. Kodkod findet ein Gegenbeispiel:

```
counterexample for lemma08:

variables assignment:
sys=initialstate
fp=[]

MODEL:

function table for inv : systemstate -> bool
------------------------------------------------------------------------------
initialstate
==============================================================================

function table for exec : systemstate x command -> systemstateandoutput
------------------------------------------------------------------------------
initialstate        read([])           @times(initialstate,no)
...
==============================================================================

statistics
p cnf 3873563 10755302
primary variables: 5672
translation time: 153957 ms
solving time: 281762 ms
```

Das Lemma wird korrigiert indem verlangt wird, dass die Datei zum Lesen bereits geöffnet wurde:

$$inv(sys) \wedge rdable(cdom(sys, read(fp)), fp, sys.fs) \wedge$$
$$\mathbf{rd}(\mathbf{sys.cap}), \mathbf{fp} \in sys.ofs \rightarrow$$
$$exec(sys, read(fp)).2 \neq no$$

Lemma 9.

Das Lemma beschreibt die Erhaltung der *validrdofs* Eigenschaft durch Systemaufrufe;

$$validrdofs(sys) \rightarrow validrdofs(exec(sys, co).1)$$

Kodkod findet ein Gegenbeispiel wo durch den Systemaufruf *openrd(fp)* diese Eigenschaft verletzt wird:

```
counterexample for lemma09:

variables assignment:
co=openrd([])
sys=({([],mkfi(data2,system-high,system-low,
               mkmrk(mkdm(system-high,system-high,system-low,system-low))))},
     {},cardkey,mkap([]),{})

MODEL:

function table for valid-rdofs : systemstate -> bool
------------------------------------------------------------------------------
({([],mkfi(data2,system-high,system-low,
               mkmrk(mkdm(system-high,system-high,system-low,system-low))))},
     {},cardkey,mkap([]),{})

initialstate
==============================================================================

function table for exec : systemstate x command -> systemstateandoutput
------------------------------------------------------------------------------
({([],mkfi(data2,system-high,system-low,
               mkmrk(mkdm(system-high,system-high,system-low,system-low))))},
     {},cardkey,mkap([]),{})
openrd([])
@times(({([],mkfi(data2,system-high,system-low,
               mkmrk(mkdm(system-high,system-high,system-low,system-low))))},
     {},cardkey,mkap([]),{rd(mkap([]),[])}),yes)
...
==============================================================================

statistics
p cnf 2315669 6537367
primary variables: 5368
translation time: 100813 ms
solving time: 106556 ms
```

Die Datei [] wird von der Anwendung *mkap*([]) zum Lesen geöffnet. Der nachfolgende Zustand erfüllt *validrdofs* nicht, weil Die Datei in keinem Verzeichnis gespeichert ist. Dieser Fehler wird behoben indem man verlangt, dass der Zustand *sys* die Invariante *inv* erfüllt.

Lemma 10.

Sollten zwei Zustände bezüglich der Domäne des aktuell aktivem Agenten äquivalent sein ($sys \sim^{comd(sys,co)} sys_0$), dann liefern die Ausführungen eines Systemaufrufs die gleichen Ausgaben:

$$inv(sys) \wedge eqd(sys, cdom(sys, co), sys_0) \rightarrow exec(sys, co).2 = exec(sys_0, co).2;$$

Kodkod findet ein Gegenbeispiel, wo der Aufruf *startappl(fileid0)* diese Eigenschaft verletzt:

```
counterexample for lemma10:

variables assignment:
co=startappl(fileid0)
sys=({([],mkdir([fileid0],system-high,system-low)),
       ([fileid0],mkfi(data2,system-high,system-high,
                       mkmrk(mkdm(system-high,system-high,system-low,system-high))))},
     {},cardkey,mkap([fileid0]),{})

sys0=({([],mkdir([fileid0],system-high,system-low)),
        ([fileid0],mkfi(data2,system-high,system-high,none))},
      {},cardkey,mkap([fileid0]),{})

MODEL:

function table for exec : systemstate x command -> systemstateandoutput
--------------------------------------------------------------------------------
({([],mkdir([fileid0],system-high,system-low)),
        ([fileid0],mkfi(data2,system-high,system-high,none))},
      {},cardkey,mkap([fileid0]),{})
startappl(fileid0)
@times({({[],mkdir([fileid0],system-high,system-low)),
        ([fileid0],mkfi(data2,system-high,system-high,none))},
       {},cardkey,mkap([fileid0]),{}),no)

({([],mkdir([fileid0],system-high,system-low)),
       ([fileid0],mkfi(data2,system-high,system-high,
                       mkmrk(mkdm(system-high,system-high,system-low,system-high))))},
     {},cardkey,mkap([fileid0]),{})
startappl(fileid0)
@times({({[],mkdir([fileid0],system-high,system-low)),
       ([fileid0],mkfi(data2,system-high,system-high,
                       mkmrk(mkdm(system-high,system-high,system-low,system-high))))},
     {},cardkey,mkap([fileid0]),{}),yes)
================================================================================

statistics
p cnf 2315924 6537715
primary variables: 5380
translation time: 93403 ms
solving time: 232247 ms
```

Obwohl die beiden Zustände äquivalent bezüglich $\sim^{cdom(sys,co)}$ sind, unterscheiden sich die Ausgaben trotzdem. Die Markierung der Datei, die die Anwendung speichert in sys_0, ist gleich *none*. Damit ist die Vorbedingung für *startappl* in sys_0 nicht erfüllt:

$$
\begin{array}{l}
exec(sys, startappl(fid) = \\
\quad \textbf{if} \\
\quad\quad [fid] \in sys.fs \wedge filep(getfs([fid], sys.fs)) \wedge \\
\quad\quad getfs([fid], sys.fs).mrk \neq none \\
\quad \textbf{then} \\
\quad\quad mksys(sys.fs, sys.as, sys.ck, mkap([fid]), sys.ofs) \times YES \\
\quad \textbf{else} \\
\quad\quad sys \times NO
\end{array}
$$

Lemma 11.

Die Eigenschaft $eqofswr$ bleibt erhalten:

$$
\begin{array}{l}
inv(sys) \wedge eqofswr(sys.ofs, ap, sys0.ofs) \rightarrow \\
eqofswr(exec(sys, co).1.ofs, ap, exec(sys0, co).1.ofs)
\end{array}
$$

Das Lemma ist falsch wie Kodkod Gegenbeispiel zeigt:

```
counterexample for lemma11:

variables assignment:
co=openwr([])
ap=mkap([])
sys=initialstate
sys0=({(@fp,mkfi(data2,mkacc(access-class0,system-high),none))},{},
      cardkey,mkap([]),{})

MODEL:

function table for exec : systemstate x command -> systemstateandoutput
------------------------------------------------------------------------------
({(@fp,mkfi(data2,mkacc(access-class0,system-high),none))},{},
 cardkey,mkap([]),{})
openwr([])
@times(({(@fp,mkfi(data2,mkacc(access-class0,system-high),none))},{},
        cardkey,mkap([]),{wr(mkap([]),[])}),yes)

initialstate        openwr([])         @times(initialstate,no)
...
==============================================================================

function table for inv : systemstate -> bool
------------------------------------------------------------------------------
initialstate
==============================================================================

statistics
p cnf 2403646 6844293
primary variables: 5404
translation time: 103712 ms
solving time: 108836 ms
```

Lemma 12.

Systemaufrufe erhalten die *validwrofs* Eigenschaft:

$$validwrofs(sys) \rightarrow validwrofs(exec(sys, co).1)$$

Kodkod falsifiziert das Lemma:

```
counterexample for lemma12:

variables assignment:
co=openwr([])
sys=({([],mkfi(data0,access-class0,system-high,none)),
      ([fileid0,fileid0],mkfi(data2,system-high,system-low,
                              mkmrk(mkdm(system-high,system-low,system-low,system-low))))},
     {},cardkey,mkap([]),{})

MODEL:

function table for valid-wrofs : systemstate -> bool
------------------------------------------------------------------------------
({([],mkfi(data0,access-class0,system-high,none)),
      ([fileid0,fileid0],mkfi(data2,system-high,system-low,
                              mkmrk(mkdm(system-high,system-low,system-low,system-low))))},
     {},cardkey,mkap([]),{})
initialstate
==============================================================================

function table for inv : systemstate -> bool
------------------------------------------------------------------------------
initialstate
==============================================================================

function table for exec : systemstate x command -> systemstateandoutput
------------------------------------------------------------------------------
({([],mkfi(data0,access-class0,system-high,none)),
      ([fileid0,fileid0],mkfi(data2,system-high,system-low,
                              mkmrk(mkdm(system-high,system-low,system-low,system-low))))},
     {},cardkey,mkap([]),{})
openwr([])
@times({([],mkfi(data0,access-class0,system-high,none)),
      ([fileid0,fileid0],mkfi(data2,system-high,system-low,
                              mkmrk(mkdm(system-high,system-low,system-low,system-low))))},
     {},cardkey,mkap([]),{wr(mkap([]),[])}),yes)
...
==============================================================================

statistics
p cnf 2403176 6843610
primary variables: 5368
translation time: 103710 ms
solving time: 112048 ms
```

Lemma 13.

Falls der aktive Agent die Domäne d nicht beeinflussen kann, bleibt die Eigenschaft *eqrd* durch die Systemafrufe erhalten:

$$inv(sys_0) \wedge cdom(sys, co) \not\rightarrow d \wedge eqrd(sys.fs, d, sys_0.fs) \rightarrow$$
$$eqrd(exec(sys, co).1.fs, d, exec(sys_0, co).1.fs)$$

Kodkod findet ein Gegenbeispiel:

```
counterexample for lemma13:

variables assignment:
d=mkdm(system-low,system-high,system-low,access-class0)
co=setintsec([],system-high,access-class0)
sys=({([],mkfi(data2,system-low,system-low,
               mkmrk(mkdm(system-high,system-high,access-class0,access-class0)))),
      ([fileid2,fileid2],mkfi(data2,system-high,access-class0,
                              mkmrk(mkdm(system-high,system-high,access-class0,access-class0))))},
     {},cardkey,mkap(@fp),{})
sys0=initialstate

MODEL:

function table for exec : systemstate x command -> systemstateandoutput
------------------------------------------------------------------------------
({([],mkfi(data2,system-low,system-low,
               mkmrk(mkdm(system-high,system-high,access-class0,access-class0)))),
      ([fileid2,fileid2],mkfi(data2,system-high,access-class0,
                              mkmrk(mkdm(system-high,system-high,access-class0,access-class0))))},
     {},cardkey,mkap(@fp),{})
setintsec([],system-high,access-class0)
@times(({([],mkfi(data2,system-high,access-class0,
               mkmrk(mkdm(system-high,system-high,access-class0,access-class0)))),
      ([fileid2,fileid2],mkfi(data2,system-high,access-class0,
                              mkmrk(mkdm(system-high,system-high,access-class0,access-class0))))},
     {},cardkey,mkap(@fp),{}),yes)

initialstate        setintsec([],system-high,access-class0)  @times(initialstate,no)
...
==============================================================================

function table for inv : systemstate -> bool
------------------------------------------------------------------------------
initialstate
==============================================================================

statistics
p cnf 2479586 7117966
primary variables: 5481
translation time: 113531 ms
solving time: 307162 ms
```

Lemma 14.

Falls der aktive Agent die Domäne d nicht beeinflussen kann, bleibt die Eigenschaft *eqcont* durch die Systemafrufe erhalten:

$$inv(sys_0) \wedge cdom(sys, co) \not\mapsto d \wedge eqcont(sys.fs, d, sys_0.fs) \rightarrow$$
$$eqcont(exec(sys, co).1.fs, d, exec(sys_0, co).1.fs)$$

Kodkod findet ein Gegebeispiel:

```
counterexample for lemma14:

variables assignment:
```

```
d=mkdm(system-high,system-high,system-low,system-low)
co=setintsec([],system-high,system-low)
sys=({([],mkfi(data1,system-low,access-class0,
               mkmrk(mkdm(system-low,system-low,access-class0,access-class0))))},
     {},cardkey,mkap([]),{})
sys0=initialstate

MODEL:

function table for exec : systemstate x command -> systemstateandoutput
--------------------------------------------------------------------------------
({([],mkfi(data1,system-low,access-class0,
           mkmrk(mkdm(system-low,system-low,access-class0,access-class0))))},
     {},cardkey,mkap([]),{})
setintsec([],system-high,system-low)
@times(({([],mkfi(data1,system-high,system-low,
                  mkmrk(mkdm(system-low,system-low,access-class0,access-class0))))},
     {},cardkey,mkap([]),{}),yes)

initialstate        setintsec([],system-high,system-low)  @times(initialstate,no)
...
================================================================================

function table for inv : systemstate -> bool
--------------------------------------------------------------------------------
initialstate
================================================================================

statistics
p cnf 2479586 7117966
primary variables: 5481
translation time: 119538 ms
solving time: 189181 ms
```

Lemma 15.

Falls der aktive Agent die Domäne d nicht beeinflussen kann, bleibt die Eigenschaft *eqd* durch die Systemafrufe erhalten:

$$cdom(sys, co) \not\rightsquigarrow d \wedge eqd(sys, d, sys_0) \rightarrow$$
$$eqd(exec(sys, co).1, d, exec(sys_0, co).1)$$

Kodkod findet ein Gegebeispiel:

```
counterexample for lemma15:

variables assignment:
d=mkdm(mkacc(system-high,system-low),mkacc(system-low,system-high))
co=create([fileid1],fileid1)
sys=({([fileid1],mkdir([fileid2],access-class0,access-class0))},{(applname0,authentication1)},
     authentication1,mkap([fileid1]),{})

MODEL:

function table for exec : systemstate x command -> systemstateandoutput
--------------------------------------------------------------------------------
({([fileid1],mkdir([fileid2],access-class0,access-class0))},{(applname0,authentication1)},
     authentication1,mkap([fileid1]),{})
```

```
create([fileid1],fileid1)
@times(({([fileid1],mkdir([fileid2],access-class0,access-class0))},{(applname0,authentication1)
         ([fileid1,fileid1]),mkfi(nodata,system-low,system-high,none))},
     authentication1,mkap([fileid1]),{}),yes)
...
================================================================================

statistics
p cnf 2356282 6684091
primary variables: 5395
translation time: 101887 ms
solving time: 224342 ms
```

Literaturverzeichnis

[1] Wolfgang Ahrendt. Deductive search for errors in free data type specifications using model generation. In Andrei Voronkov, editor, *18th International Conference on Automated Deduction*, volume 2392 of *LNCS*. Springer, 2002.

[2] Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, Wolfram Menzel, Wolfgang Reif, Gerhard Schellhorn, and Peter Schmitt. Integrating Automated and Interactive Theorem Proving. In W. Bibel and P. Schmitt, editors, *Automated Deduction – A Basis for Applications*. Kluwer Academic Publishers, Dordrecht, 1998.

[3] Online Tutorial für Alloy4. URL: `http://alloy.mit.edu/alloy4/tutorial4/`.

[4] Michael Balser, Christoph Duelli, Wolfgang Reif, and Gerhard Schellhorn. Verifying concurrent systems with symbolic execution. *Journal of Logic and Computation*, 12(4):549–560, 2002.

[5] Michael Balser, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Andreas Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*. Springer LNCS 1783, 2000.

[6] Simon Bäumler, Michael Balser, Alexander Knapp, Wolfgang Reif, and Andreas Thums. Interactive verification of uml state machines. In *Formal Methods and Software Engineering*, number 3308 in LNCS. Springer, 2004.

[7] Koen Claessen. Equinox, a new theorem prover for full first-order logic with equality. Presentation at Dagstuhl Seminar 05431 on Deduction and Applications, October 2005.

[8] Koen Claessen and Niklas Sörensson. New techniques that improve mace-style model finding. In *Proc. of Workshop on Model Computation (MODEL)*, 2003.

[9] John Derrick, Gerhard Schellhorn, and Heike Wehrheim. Mechanizing a correctness proof for a lock-free concurrent stack. In *FMOODS*, pages 78–95, 2008.

[10] Andriy Dunets, Gerhard Schellhorn, and Wolfgang Reif. Automating Algebraic Specifications of Non-freely Generated Data Types. In Cha et al., editor, *ATVA*, volume 5311 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 2008.

[11] Andriy Dunets, Gerhard Schellhorn, and Wolfgang Reif. Bounded Relational Analysis of Free Data Types. In Bernhard Beckert and Reiner Hähnle, editors, *TAP*, volume 4966 of *Lecture Notes in Computer Science*, pages 99–115. Springer, 2008.

[12] Andriy Dunets, Gerhard Schellhorn, and Wolfgang Reif. Automated Flaw Detection in Algebraic Specifications. *J. Autom. Reasoning*, 2009.

[13] Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into sat. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.

[14] Holger Grandy, Markus Bischof, Kurt Stenzel, Gerhard Schellhorn, and Wolfgang Reif. Verification of Mondex Electronic Purses with KIV: From a Security Protocol to Verified Code . In *Proceedings of Formal Methods 2008 (FM08), (to appear)*, LNCS, Turku, Finland, 2008. Springer.

[15] Reiner Hähnle, Maritta Heisel, Wolfgang Reif, and Werner Stephan. An Interactive Verification System Based on Dynamic Logic. In J. Siekmann, editor, *8th International Conference on Automated Deduction. Proceedings*. Springer LNCS 230, 1986.

[16] Dominik Haneberg, Simon Bäumler, Michael Balser, Holger Grandy, Frank Ortmeier, Wolfgang Reif, Gerhard Schellhorn, Jonathan Schmitt, and Kurt Stenzel. The User Interface of the KIV Verification System — A System Description. *Electronic Notes in Theoretical Computer Science UITP special issue*, 2006.

[17] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[18] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Lecture Notes in Computer Science*, pages 300–314. Springer-Verlag, 2001.

[19] John Harrison. Inductive definitions: Automation and application. In *TPHOLs*, pages 200–213, 1995.

[20] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*, pages 3–16, 2005.

[21] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[22] The Alloy Project. http://alloy.mit.edu.

[23] Daniel Jackson. Automating first-order relational logic. In *Proceedings of the 8th ACM SIGSOFT Symposium*, pages 130–139. ACM Press, 2000.

[24] Web Präsentation der Fallstudie über die Linearisierbarkeit. URL: `http://www.informatik.uni-augsburg.de/swt/projects/linearizability.html`.

[25] Web Präsentation der KIV Projekte. URL: `http://www.informatik.uni-augsburg.de/swt/projects/`.

[26] Viktor Kuncak and Daniel Jackson. Relational analysis of algebraic datatypes. In *Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2005.

[27] Jacques Loeckx, Hans-Dieter Ehrich, and Markus Wolf. *Specification of Abstract Data Types (Wiley Teubner on Applicable Theory in Computer Science)*. John Wiley & Sons, 1 edition, November 1996.

[28] William McCune. Mace4 reference manual and guide, 2003.

[29] William McCune. Otter 3.3 reference manual, 2003.

[30] William McCune. Prover9 manual, April 2008.

[31] Jia Meng, Claire Quigley, and Lawrence C. Paulson. Automation for interactive proof: First prototype. *Inf. Comput.*, 204(10):1575–1596, 2006.

[32] Frank Ortmeier, Wolfgang Reif, Gerhard Schellhorn, Andreas Thums, Bernhard Hering, and Helmut Trappschuh. Safety analysis of the height control system for the Elbtunnel. In *SafeComp 2002*, pages 296 – 308, Catania, Italy, 2002. Springer LNCS 2434.

[33] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A Prototype Verification System. In *CADE*, pages 748–752, 1992.

[34] Lawrence C. Paulson. *Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow)*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[35] Lee Pike, Paul Miner, and Wilfredo Torres-Pomales. Diagnosing a failed proof in fault-tolerance: A disproving challenge problem. In *DISPROVING 2006 Participants' Proceedings*, pages 24–33, 2006.

[36] Wolfgang 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-Verlag, Berlin, 1995.

[37] Wolfgang Reif and Gerhard Schellhorn. Theorem Proving in Large Theories. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications*, volume III, 2. Kluwer Academic Publishers, Dordrecht, 1998.

[38] Wolfgang Reif, Gerhard Schellhorn, and Kurt Stenzel. Proving System Correctness with KIV. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development. Proceedings*. Springer LNCS 1214, 1997.

[39] Wolfgang Reif, Gerhard Schellhorn, and Andreas Thums. Flaw detection in formal specifications. In *IJCAR*, pages 642–657, 2001.

[40] Wolfgang Reif, Gerhard Schellhorn, and Andreas Thums. Flaw detection in formal specifications. In *IJCAR*, pages 642–657, 2001.

[41] John Rushby. Noninterference, transitivity, and channel-control security policies. Technical report, dec 1992.

[42] Gerhard Schellhorn. *Verifikation abstrakter Zustandsmaschinen*. PhD thesis, Universität Ulm, Fakultät für Informatik, 1999. (in German).

[43] Gerhard Schellhorn and Wolfgang Ahrendt. The WAM Case Study: Verifying Compiler Correctness for Prolog with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume III: Applications, chapter 3: Automated Theorem Proving in Software Engineering, pages 165 – 194. Kluwer Academic Publishers, 1998.

[44] Gerhard Schellhorn and Axel Burandt. Specification and Verification of Distributed Technical Systems with Central Control. In C. Lewerentz and T. Lindner, editors, *Formal Development of Reactive Systems*. Springer LNCS 891, 1994.

[45] Gerhard Schellhorn, Wolfgang Reif, Axel Schairer, Paul A. Karger, Vernon Austel, and David C. Toll. Verification of a formal security model for multiapplicative smart cards. In *ESORICS*, pages 17–36, 2000.

[46] Gerhard Schellhorn, Wolfgang Reif, Axel Schairer, Paul A. Karger, Vernon Austel, and David C. Toll. Verification of a formal security model for multiapplicative smart cards. In *Proc. of the* 6^{th} *European Symposium on Research in Computer Security (ESORICS)*, 2000. (to appear).

[47] Ilya Shlyakhter. Generating effective symmetry-breaking predicates for search problems. *Discrete Applied Mathematics*, 155(12):1539–1548, 2007.

[48] John K. Slaney. Finder: Finite domain enumerator - system description. In *CADE*, pages 798–801, 1994.

[49] The SMV System. URL: `http://www.cs.cmu.edu/ modelcheck/smv.html`.

[50] Kurt Stenzel. A formally verified calculus for full Java Card. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology and Software Technology (AMAST) 2004, Proceedings*, Stirling Scotland, July 2004. Springer LNCS 3116.

[51] Kurt Stenzel. *Verification of Java Card Programs*. PhD thesis, Universität Augsburg, Fakultät für Angewandte Informatik, URL: `http://www.opus-bayern.de/uni-augsburg/volltexte/2005/122/`, or `http://www.informatik.uni-augsburg.de/forschung/dissertations/`, 2005.

[52] Andreas Thums. Fehlersuche in Formalen Spezifikationen (in German). Master's thesis, Universität Ulm, Germany, 1998.

[53] Emina Torlak, Felix Sheng-Ho Chang, and Daniel Jackson. Finding minimal unsatisfiable cores of declarative specifications. In *FM*, pages 326–341, 2008.

[54] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *TACAS*, pages 632–647, 2007.

[55] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *TACAS*, pages 632–647, 2007.

[56] Engin Uzuncaova and Sarfraz Khurshid. Constraint prioritization for efficient analysis of declarative models. In *FM*, pages 310–325, 2008.

[57] Tjark Weber. *SAT-based Finite Model Generation for Higher-Order Logic*. PhD thesis, Institut für Informatik, Technische Universität München, Germany, April 2008.

[58] zChaff SAT solver. http://www.princeton.edu/ chaff/zchaff.html.

[59] Jian Zhang and Hantao Zhang. Sem: a system for enumerating models. In *IJCAI*, pages 298–303, 1995.

Curriculum Vitae

Name:	**Andriy Dunets**
Geburtstag:	30. Juli 1981
Geburtsort:	Ternopil, Ukraine
Staatsangehörigkeit:	ukrainisch
Familienstand:	ledig

Schulische Ausbildung:

09/1987 - 06/1996	**Allgemeinschule Nr. 75 in Lwiw**
09/1996 - 06/1998	**Mathematische Schule Lwiw** Abitur mit Gesamtnote *sehr gut*
09/1998 - 06/2000	**Universität Lwiw** Studium der Angewandten Mathematik Grundstudium mit Gesamtnote *sehr gut*
10/2000 - 04/2005	**Universität Augsburg** Studium der Angewandten Informatik Diplom mit Gesamtnote *sehr gut*
05/2005 - 02/2010	**Universität Augsburg** Promotion zum Dr. rer. nat.

Beruflicher Werdegang:

03/2002 - 10/2002	**Siemens Corporate Technology, München** Werkstudent
10/2002 - 07/2003	**Universität Augsburg** Lehrstuhl für Systemnahe Informatik und Kommunikationssysteme wissenschaftliche Hilfskraft, Übungsgruppenleiter
04/2004 - 04/2005	**Universität Augsburg** Lehrstuhl für Softwaretechnik und Programmiersprachen wissenschaftliche Hilfskraft, Übungsgruppenleiter
05/2005 - 03/2010	**Universität Augsburg** Lehrstuhl für Softwaretechnik und Programmiersprachen wissenschaftlicher Mitarbeiter