Methoden und Werkzeuge zur Verifikation von Software

Larch

Steffen Hartmann
steffen.hartmann@gmx.de

Frankfurt am Main
Mai 2000

Dieser Vortrag wurde im Rahmen des Seminars *Methoden und Werkzeuge zur Verifikation von Software* im Sommersemester 2000 an der Johann Wolfgang Goethe - Universität Frankfurt am Main von mir angefertigt und gehalten.

Inhaltsverzeichnis

1 Einführung

Larch ist eine Sprache, in der man logische Konstrukte beschreiben kann und zu der es Werkeuge gibt, mit deren Hilfe man Beweise überprüfen kann. Es ist jedoch nicht möglich, komplexere Beweise automatisch zu führen.

Zuerst soll ein kleiner Eindruck gegeben werden, wie in Larch spezifiziert wird. Es werden die verschiedenen Möglichkeiten vorgestellt. Dann folgt eine genauere Beschreibung der zwei wichtigsten Varianten. Es folgen weitere Larch Spezifikationsdetails, bis zuletzt ein Werkzeug zur automatischen Beweisprüfung vorgestellt wird.

2 Eine zweigleisige Spezifikation

Larch unterstützt eine zweigleisige Art der Spezifikation. Jede Spezifikation hat Komponenten, die in zwei verschiedenen Sprachen geschrieben sind: Die eine Sprache ist unabhängig von jeder Programmiersprache, die andere ist speziell für eine Programmiersprache entworfen. Die erste Art ist die Larch Shared Language (LSL), die anderen sind die Larch interface languages.

2.1 Interfacesprachen

Die Interfacesprachen werden benutzt, um die Interfaces zwischen Programmkomponenten zu beschreiben. Dabei wird sich sehr nah an der entsprechenden Programmiersprache orientiert. Deshalb existieren verschiedene Interfacesprachen. Die zwei bekanntesten sind LCL (für C; nicht für C++) und LM3 (für Modula-3). Andere Interfacesprachen wurden für Ada, CLU, C++, ML, und Smalltalk geschrieben. Es gibt ebenfalls allgemeine Larch Interfacesprachen, die für spezielle Programmiersprachen spezialisiert werden können, oder benutzt werden, um Interfaces zwischen Programmen in verschiedenen Programmiersprachen zu spezifizieren.

Spezifikationen, die in solch einer Interfacesprache geschrieben sind, sind im allgemeinen kürzer, als die, die in einer universellen Sprache (LSL) geschrieben wurden. Außerdem sind sie verständlicher für Programmierer, die diese Komponenten benutzen und implementieren.

Abbildung 1: (LCL Spezifikation `getTask`)

```
uses TaskQueue;
mutable type queue;
immutable type task;

task *getTask(queue q) {
      modifies q;
      ensures
         if isEmpty(q^)
            then result = NIL ∧ unchanged(q)
            else (*result)' = first(q^) ∧ q' = tail(q^);
}
```

Abbildung 1 zeigt ein Beispiel, das ein kleines Fragment eines Schedulers eines Betriebssystems darstellt. Die Spezifikation ist in der Interfacesprache LCL geschrieben. Dieses Fragment führt zwei abstrakte Typen und eine Prozedur ein, um einen Task aus der Taskqueue zu wählen. Hier die wichtigsten Notationen in Kürze: `*` meint einen Pointer auf (wie in C), `result` zeigt auf den Wert, den eine Prozedur zurückgegeben hat, das Symbol ^wird benutzt, um auf den Wert zum Zeitpunkt des Prozeduraufrufes zu zeigen, und `'`, um auf den Wert am Prozedurende zu zeigen.

Die Spezifikation von getTask ist keinesfalls abgeschlossen. Denn, aus der Spezifikation geht nicht hervor, welcher Task aus getTask als nächstes gewählt wird. Ist es der, der am längsten in q gewesen ist? Ist es der in q, mit der höchsten Priorität?

2.2 LSL, Larch Shared Language

Interfacespezifikationen vertrauen auf Behelfsspezifikationen, die in LSL geschrieben sind, damit die Semantik für die primitiven Terme, die sie benutzen, unterstützt wird. Es ist aber auch möglich, eine Spezifikation nur in LCL zu schreiben (aber keinesfalls empfehlenswert).

Abbildung 2: (LSL Spezifikation, die von getTask benutzt wird)

```
TaskQueue: trait
   includes Nat
   task tuple of id: Nat, important: Bool
   introduces
      new: → queue
      __⊣__: task, queue → queue
      isEmpty, hasImportant: queue → Bool
      first: queue → task
      tail: queue → queue
   asserts
      queue generated by new, ⊣
      ∀ t: task, q: queue
         isEmpty(new);
         ¬isEmpty(t⊣q);
         ¬hasImportant(new);
         hasImportant(t⊣) ==
            t.important ∨ hasImportant(q);
         first(t⊣q) ==
            if t.important ∨ ¬hasImportant(q)
               then t else first(q);
         tail(t⊣q) ==
            if first(t⊣q) == t then q else t ⊣ tail(q)
```

Abbildung 2 enthält einen Teil einer LSL Spezifikation, die die Operatoren spezifiziert, die in der Interfacespezifikation von getTask benutzt werden. Aufgrund der Informationen in dieser LSL Spezifikation kann man folgern, dass der Task, der auf das Ergebnis von getTask zeigt, der zuletzt eingefügte important Task ist, wenn ein solcher Task denn existiert. Andernfalls ist es der zuletzt eingefügte Task.

Viele Spezifikationen besitzen eine Struktur wie diese. Sie vertrauen implizit auf die Behelfsspezifikationen, in dem sie ein Interface beschreiben und annehmen, dass der Leser diese Strukturen kennt. (z.B. Mengen, Listen, ...) Man kann solche Spezifikationen natürlich mißverstehen. Nur wenn man die gleiche Intuition wie der Spezifizierer hat, wird das Programm das selbe tun. Es gibt aber keinen Weg sicherzustellen, dass die Intuition passt. LSL Spezifikationen ermöglichen eindeutige mathematische Definitionen für Terme, die in Interfacespezifikationen erscheinen.

Larch unterstützt deshalb eine Trennung mit grundlegenden Konstrukten in der LSL-Ebene und Programmierdetails in der Interface-Ebene. Aus folgenden Gründen wird dem Spezifizierer empfohlen, das meiste der Komplexität von Spezifikationen in der LSL-Ebene zu halten:

- LSL Spezifikationen sind besser wiederverwendbar als Interfacespezifikationen.

- LSL liegt eine einfachere Semantik zugrunde, als den meisten Programmiersprachen (und daher als den meisten Interfacesprachen), daher neigen Spezifizierer weniger dazu Fehler zu machen, und Fehler sind einfacher zu finden.
- Es ist einfacher, Behauptungen zu machen und sie über die semantischen Eigenschaften von LSL zu prüfen, als über die semantischen Eigenschaften der Interfacespezifikation.

Viele Programmierfehler werden beim Ausführen des Programmes gefunden, also beim Testen. Einige Larch Spezifikationen sind ausführbar, die meisten aber nicht, denn Larch legt mehr Wert auf Kürze und Klarheit, als auf Ausführbarkeit. Um es möglich zu machen, Spezifikationen zu überprüfen, bevor man sie implementiert, erlaubt Larch Spezifizierern redundante Behauptungen über Spezifikationen zu machen. Diese Behauptungen können mechanisch überprüft werden. Es gibt einige Werzeuge, die Spezifizierern helfen, diese Behaupungen mechanisch zu überprüfen und debuggen.

3 LSL, Larch Shared Language

3.1 Operatoren & Sorten

LSL Spezifikationen definieren zwei Arten von Symbolen: Operatoren und Sorten. Das Konzept von Operatoren und Sorten ist das gleiche, wie es in der konventionellen mathematischen Logik benutzt wird. Es ist dem Programmiersprachenkonzept von Prozeduren und Typen ähnlich, aber es ist wichtig diese zwei Konzepte nicht zu verwechseln. Beim Besprechen der LSL Spezifikation wird also über Operatoren und Sorten gesprochen und bei Programmkonstrukten über Prozeduren und Typen. Operatoren stehen für totale Funktionen auf Tupeln von Werten nach Werten. Sorten stehen für disjunkte nicht leere Mengen von Werten, und werden benutzt, um den Definitionsbereich und Bildbereich von Operatoren anzuzeigen. In jeder Interfacesprache müssen Prozeduren und Typen das bedeuten, was sie in der jeweiligen Programmiersprache bedeuten.

3.2 Der Trait

Der Trait (Charakterzug) ist die Grundeinheit von Spezifikationen in LSL. Er führt Operatoren ein und spezifiziert ihre Eigenschaften, manchmal definiert er auch einen abstrakten Typ. Oft ist es nützlich eine Menge von Eigenschaften zu definieren, die einen Typ nicht vollständig charakterisiert, dies ist ebenfalls möglich.

Abbildung 3 zeigt einen Trait, der eine Klasse von Tabellen spezifiziert, die Werte in indizierten Plätzen speichert. Die Spezifikation beginnt mit dem Einbinden eines anderen Traits, `Integer`. Diese Spezifikation liefert Informationen über die Operatoren $+$, 0, und 1, die zur Definition der Operatoren in `Table`, benutzt werden.

Die Einführungsklausel deklariert eine Menge von Operatoren, jeder mit seiner Signatur (die Sorten des Definitions- und Bildbereiches). Die Signaturen werden zum Sortencheck von Termen benutzt, in der gleichen Weise wie Prozeduraufrufe per Typcheck in Programmiersprachen überprüft werden.

Abbildung 3: (Table.lsl)

```
Table: trait
      include Integer
      introduces
         new: → Tab
         add: Tab, Ind, Val → Tab
         __ ∈ __ : Ind, Tab → Bool
         lookup: Tab, Ind → Val
         size: Tab → Int
      asserts ∀ i, i1: Ind, v: Val, t: Tab
         ¬(i ∈ new);
         i ∈ add(t, i1, v) == i = i1 ∨ i ∈ t;
         lookup(add(t, i, v), i1) ==
            if i = i1 then v else lookup(t, i1);
         size(new) == 0;
         size(add(t, i, v)) ==
            if i ∈ t then size(t) else size(t) + 1
```

Der Rumpf der Spezifikation folgt dem reservierten Wort `asserts`. Er enthält Gleichungen zwischen Termen, die Operatoren und Variablen enthalten. Operatoren müssen auf diese Weise nicht immer vollständig definiert sein, so wie `lookup` in Abbildung 3 nicht vollständig definiert wurde. Es wird nämlich nichts über den Wert des Terms `lookup(new, i)` ausgesagt.

Mathematisch gut durchdachte Traits definieren eine Theorie (eine Menge von Sätzen, die logisch abgeschlossen sind) in mehrsortiger Prädikatenlogik erster Ordnung mit Gleichheit. Eine solche Theorie enthält die Trait Behauptungen, die üblichen Axiome der Prädikatenlogik erster Ordnung und alles was daraus folgt, aber keinesfalls mehr. Dies garantiert, dass Formeln der Theorie nur in der Gegenwart von Behauptungen dieses Traits folgen, niemals in der Abwesenheit. Ganz im Gegensatz zu algebraischen Spezifikationssprachen, die auf initialen oder finalen Algebren basieren. Der Vorteil: Es ist sichergestellt, dass alle Sätze, die über einer unvollständigen Spezifikation bewiesen wurden, gültig bleiben, wenn die Spezifikation erweitert wird.

3.3 generated by

Oft sind stärkere Theorien als obige notwendig. Zum Beispiel, wenn man einen abstrakten Typ definiert. Mit Hilfe der Konstrukte `generated by` und `partitioned by` gibt es zwei Möglichkeiten, die Spezifikationen stärker zu machen.

Eine `generated by` Klausel behauptet, dass eine Liste von Operatoren eine vollständige Menge von Generatoren für eine Sorte ist. Jeder Wert der Sorte ist also gleich einem, der als eine endliche Anzahl von Anwendungen dieser Operatoren und Variablen anderer Sorten, geschrieben werden kann. Dies rechtfertigt ein Generator-Induktions-Schema, um Spezifikationen über diese Sorte zu beweisen. Zum Beispiel die natürlichen Zahlen, die von 0 und `succ` generiert werden, und die Integer die durch 0, `succ`, und `pred` generiert werden. Die Behauptung

```
Tab generated by new, add
```

kann, wenn sie zu `Table` hinzugefügt wird, dazu benutzt werden, Sätze per Induktion über `new` und `add` zu beweisen. Ebenso kann jeder Wert der Sorte `Tab` aus `new`, durch eine endliche Anzahl von Anwendungen von `add`, konstruiert werden.

3.4 partitioned by

Eine `partitioned by` Klausel behauptet, dass eine Liste von Operatoren eine vollständige Menge von Beobachtern für eine Sorte erzeugt. Das heißt, dass alle verschiedenen Werte der Sorte ausgezeichnet werden können, wobei nur Operatoren dieser Sorte benutzt werden. Terme, die nicht unterscheidbar sind und jeden Beobachter benutzen, sind deshalb gleich. Beispiel: Mengen sind ausgezeichnet durch $\in$ und Mengen, die die gleichen Elemente enthalten, sind gleich. Jede `partitioned by` Klausel ist ein neues Axiom, das eine neue Ableitungsregel für Werte dieser Sorte rechtfertigt. Zum Beispiel fügt die Behauptung

$$\texttt{Tab partitioned by } \in, \text{lookup}$$

die folgende Ableitungsregel zu der Theorie von `Table` hinzu:

$$\frac{\forall i1 : ind(i1 \in t1 = i1 \in t2), \forall i1 : ind(lookup(t1, i1) = lookup(t2, i1)))}{t1 = t2}$$

Es ist aufschlussreich, einige Sachen, die `Table` nicht spezifiziert, aufzuführen:

- Es wird nicht gesagt, wie Tabellen repräsentiert werden.
- Es gibt keinen Algorithmus, um Tabellen zu verändern.
- Es wird nicht gesagt, welche Prozeduren auf Tabellen zu implementieren sind.
- Es wird nicht gesagt, was passiert, wenn jemand ein `Ind` sucht das nicht in `Tab` ist.

Die ersten zwei Entscheidungen liegen im Bereich der Implementierung. Die dritte und vierte liegen im Bereich der Interfacespezifikation.

4 LCL, Larch Interface Language for C

LCL ist eine Larch Interfacesprache für standard C, aber kein C Dialekt. Programme, die in LCL spezifiziert werden, sind C Programme, die von jedem C Compiler übersetzt werden.

Um einen kleinen Einblick in LCL zu bekommen, wird im Folgenden die Relation zwischen LCL Spezifikationen und C Programmen und die Struktur einer LCL Spezifikation erläutert.

4.1 Die Relation zwischen LCL und C

C ist eine allgemeine und flexible Sprache, die oft verwendet wird. Üblich ist die Programmierung mit Modulen, also besteht solch ein C Programm aus einer Menge von Programmeinheiten. Ein Modul besteht aus einem *Interface* und einer *Implementation*. Das Interface ist eine Sammlung von Typen, Funktionen, Variablen und Konstanten, die in anderen Modulen verwendet werden.

Ein C Modul M besteht üblicherweise aus 3 Dateien:

- M.c enthält den größten Teil der Implementierung, einschließlich Funktionsdefinitionen und privaten Daten.
- M.h enthält eine Beschreibung des Interfaces und Teile der Implementierung.

- M.o enthält die compilierte Form. Diese Datein werden gelinkt um das ausführbare Programm zu erhalten.

C Module, die mit Hilfe von LCL spezifiziert wurden, haben zwei weitere Dateien:

- M.lcl enhält die LCL Interfacespezifikation — eine formale Beschreibung der Typen, Funktionen, Variablen und Konstanten — zusammen mit Kommentaren für die Dokumentation. Es ersetzt M.h als Dokumentation für Programmierer, die dieses Modul benutzen.
- M.lh ist eine Header - Datei, die automatisch aus M.lcl generiert wird und in M.h eingebunden wird. Die automatische Generierung erspart es dem User, Informatioen in dem Headerfile zu wiederholen. Das spart Arbeit und vermeidet Fehler.

M.lcl kann ebenfalls auf eine andere Dateiart zeigen:

- .lsl Dateien enthalten zusätzliche Spezifikationen in der Form von LSL Traits, um die Operatoren aus den .lcl Dateien genauer zu definieren.

4.1.1 Das LCL Speichermodell

Da LCL, wie C, statisch getypt ist, ist die Art der Werte, an die ein Objekt in einem Zustand gebunden werden kann, fest. Analog hat jeder LSL Wert eine einmalige Sorte. Um die zwei Sprachen zu verbinden, gibt es eine Abbildung von LCL Typen auf LSL Sorten. Jeder „build-in-type"von C, jeder von Typkonstruktoren gebildeter Typ von C (z.B. `int *`), und jeder in LCL definierte abstrakte Typ basiert auf einer LSL Sorte. Wenn ein Ausdruck e, ein Objekt von Typ T bezeichnet und T auf eine Sorte S basiert, dann sind die Werte von $e\hat{}$ und e' von der Sorte S. Die Sorte, auf der ein Typ basiert, erscheint in LCL Spezifikationen nicht explizit. Statdessen steht der LCL Type Spezifizierer für seine assoziierte Sorte.

Ein standard Trait definiert Operatoren auf Sorten, auf denen C „build-in-types"basieren. Programmierer mit C Erfahrung wissen schon was diese Operatoren bedeuten. Vom Spezifizierer gelieferte Traits, werden benutzt, um Applikationsspezifische Operatoren einzuführen. eine `uses` Klausel verkörpert von Spezifizierern gelieferte Traits und verbindet Typen mit Sorten. Hier ein Ausschnitt einer Spezifikation:

```
uses Vector(int for elem, int[] for vec);

void vMult(int i, int a[]) {
    modifies a;
    ensures a' = i * a^;
}
```

Die `uses` Klausel verkörpert den Trait Vektor (der hier nicht gezeigt wird) mit zwei Umbenennungen: Die Sorte der Werte, die in Objekten vom Typ int enthalten, steht für elem und die Sorte der Werte, die von Objekten vom Typ int[] enthalten, steht für vec. Der Operator *, der in der `ensures` Klausel benutzt wird, ist definiert in diesem Trait. Die Gleichung, die diese Operatorsorte enthält, überprüft, ob i für einen int, a für ein array und $a\hat{}$ und a' für Vektoren von Integers stehen.

4.1.2 Variablen, Typen, Objecte und Zustände

Jedem Bereich eines C Programms ist eine Umgebung zugeordnet, die Variablen auf getypte Objekte abbildet. Ein Typ ist eine Menge von Werten und eine Menge von Operationen, die man auf diese Werte anwenden kann.

LCL liefert zwei verschiedene Arten von Typen: *exposed types* die genau den C Typen entsprechen, und *abstract types* die es in C gar nicht gibt. Die Typüberprüfung für *exposed types* folgt den Regeln in C. Für *abstract types* wird die Typüberprüfung strikt auf der Basis von Namen durchgeführt.

Ein Objekt ist ein Container für Werte von einem besonderem Typ. Genauer, kann man es sich als Speicherregion vorstellen. Die Hauptarten von Werten sind:

- *basic values* Dies sind mathematische Konstanten, wie der Integer 3, der Buchstabe A und die Menge $\{3\}$. Solche Werte sind unabhängig von dem Zustand der Berechnung.

- *structs* Das sind Sammlungen von Objekten, jedes Objekt bezeichnet durch einen Feldnamen. Zum Beispiel in der Variablendeklaration

  ```
  struct {int fieldA; char fieldB;} s;
  ```

  bezeichnet `s.fieldA` ein Object von Typ intObj und `s.fieldB` ein Object vom Typ charObj.

- *unions* Die sind wie *structs*, mit der Ausnahme, dass ihre Objekte im Speicher überlappen.

- *pointer* Diese zeigen auf ein Objekt in einer homogenen Sequenz von Objekten. Wenn `p` ein Pointer ist, bezeichnet `* (p + minIndex(p))` das erste Object der Sequenz und `* (p + maxIndex(p))` das letzte Objekt, wobei `minIndex(p)` $\leq 0 \leq$ `maxIndex(p)`.

  Der Infixoperator `->` dereferenziert einen *pointer* zum *struct* und wählt dann ein Mitglied aus.

- *arrays* Dies sind homogene Sequenzen von Objecten. Wenn `a` ein Array bezeichnet, dann bezeichnet `a[0]` das erste und `a[maxIndex(a)]` das letzte Objekt der Sequenz.

  Während C nur eine kleine Unterscheidung zwischen Pointer und Array macht, ist diese bei LCL größer. In C sind Funktionsprototypen, wie `char *s` und `char s[]` equivalent. In LCL erlaubt der Prototyp `char *s` Zugriff auf Buchstaben von `* (s + minIndex(s))` bis `* (s + maxIndex(s))`, während `char a[]` Zugriff auf Buchstaben von `s[0]` bis `s[maxIndex(s)]` gewährt.

- *objects* Der Wert eines Objektes kann auch ein Objekt sein. Zum Beispiel, der Wert eines Feldes einer Struktur, kann wieder eine Struktur sein.

Ein Funktionsaufruf kann die Werte eines Objektes verändern, auf die der Aufrufer Zugriff hat, nicht aber die Umgebung des Aufrufers. Für die hiesigen Betrachtungen, kann man sich den Zustand eines C Programmes als Abbildung von Objekten auf Werte denken.

In LCL zeigen die Postfixoperatoren ˆ und ′ auf die Werte von Objekten in den pre- und post-Zuständen einer Funktion. Man kann sie anwenden auf Objekte, die Ausdrücke bezeichnen, Sammlungen von Objeken und Sequenzen von Objekten.

- Wenn man ˆ oder ′ auf ein Objekt anwendet, erhält man den Wert, der in dem Objekt gespeichert ist.

- Wenn man ˆ oder ′ auf ein Array anwendet, erhält man einen Vektor, der die Werte des Arrays, in der Reihenfolge der Arraynummerierung, enthält.

- Wenn man ˆ oder ′ auf eine Struktur anwendet, erhält man ein Tupel von Objekten / Werten zurück. Auch hier gilt: enthält die Struktur ein Array, so wird ein Vektor zurückgeliefert, enthält sie ein Pointer, so wird auch nur ein Pointer zurückgegeben.

4.2 Funktionsspezifikationen

Eine C Funktion kommuniziert mit dem Aufrufer, indem sie ein Ergebnis zurückliefert, dem Aufrufer Zugriff auf das Objekt gewährt, ein solches Objekt verändert oder die Kontrolle an einen anderen Platz zurückgibt. Eine solche C Funktion kann man verwenden, ohne sie oder die von ihr benutzten Funktionen zu verstehen. Hier die wichtigsten Teile einer LCL Funktion:

- Die `requires` Klausel (precondition) beschränkt den Zustand und die Argumente, mit denen der Client die Funktion aufrufen darf. Sie bezieht sich nur auf Werte des pre-states. Eine fehlende `requires` Klausel ist äquivalent zu der schwächsten möglichen Klausel, `requires true`.
- Die `modifies` Klausel sagt, was eine Funktion ändern darf. Wenn keine `modifies` exsistiert, darf nichts geändert werden.
- Die `ensures` Klausel (postcondition) beschränkt das Funktionsverhalten, wenn es entsprechend aufgerufen wird. Es berichtet über den Zustand, wenn die Funktion aufgerufen wird (dem prestate) und den Zustand, wenn die Funktion beendet wird (dem poststate). Das Objekt `result` enthält den Wert, der durch die Funktion zurückgegeben wird und das Objekt `control` enthält das Objekt, an das die Kontrolle zurückgegeben wird.

Der Client erwartet, dass die Bedingungen der Precondition vor jedem Aufruf hergestellt werden. Dafür garantiert der Client, dass die Funktion terminiert und die Postcondition, bis auf Änderung der `modifies list`, erfüllt.

Der Implementierer setzt möglicherweise die Precondition bei jedem Eintritt voraus. Unter dieser Vermutung, muss die Implementierung in einem Zustand terminieren, der die Postcondition erfüllt, ohne Veränderungen an Objekten, die für den Client sichtbar sind oder nicht in der `modifies list` sind, vorzunehmen. Wenn die Precondition nicht erfüllt wird, ist das Funktionsverhalten unbeschränkt. Deshalb sollten die Klauseln der Precondition schwach gehalten werden.

5 Interfacespezifikation

Eine Interfacespezifikation definiert ein Interface zwischen Programmkomponenten und ist in einer Programmiersprachen spezifischen Interfacesprache geschrieben. Jede Spezifikation muss die Informationen enthalten, die benötigt werden, um das Interface zu benutzen und um Programme zu schreiben, die es implementieren. Der Kern jeder Larch Interfacesprache ist ein Modell der Zustände, die durch die assoziierte Programmiersprache verändert werden können.

5.1 Programmzustände / locs

Zustände sind Abbildungen von locs (abstract storage locations, auch Objekte genannt) auf Werte. Jeder Variablenbezeichner hat einen Typ und ist verknüpft mit einem loc dieses Typs. Die Hauptarten der Werte, die in loc gespeichert werden können, sind:

- *basic values* Das sind mathematische Konstanten, wie der Integer 3 und der Buchstabe A. Solche Werte sind unabhängig vom Status der Berechnung.
- *exposed types* Diese sind Datenstrukturen, die vollständig durch einen Typkonstruktor der Programmiersprache beschrieben werden. (z.B. `int *` in C oder `ARRAY [1..10] OF INTEGER` im Modula-3) Die Repräsentation ist für Clients sichtbar.

- *abstract typs* Datentypen, die man sich am besten als Sammlungen von verwandten Operationen auf Sammlungen von verwandten Werten vorstellt. Abstrakte Typen werden benutzt, um Repräsentationsinformationen vor Clients zu verbergen.

Jede Interfacesprache unterstützt Operatoren, die auf locs angewendet werden können, um ihre Werte in relevanten Zuständen zu extrahieren. (üblicherweise die pre- und post-states einer Prozedur)

Jeder locs Typ definiert die Art der Werte, die er auf einen Zustand abbilden kann. Ebenso wie jeder loc einen einmaligen Typ und jeder LSL Term eine einmalige Sorte hat. Um die zwei Reihen in einer Larch Spezifikation zu verbinden, gibt es eine Abbildung von Interfacesprach Typen (auch abstrakte Typen) auf LSL Sorten. Dies ist einfach möglich, da jede Art von *basic values*, *exposed types*, und *abstract types* auf LSL Sorten basiert. Interfacespezifikation benutzen Typen und Sorten. Die Eigenschaften dieser Werte sind in LSL definiert, wobei der Sorte entsprechende Operatoren benutzt werden.

Für jede Interfacesprache definiert ein standard Trait Operatoren, die auf Werte dieser Art (Programmiersprachen Grundtypen und andere ungeschützte Typen) angewendet werden können. Benutzer, die schon mit der Programmiersprache vertraut sind, werden ein intuitives Verständnis dieser Operatoren haben. Abstrakte Typen sind typischerweise auf Sorten basiert, die in Traits von Spezifizierer geliefert werden.

5.2 Ein Interfacesprachen Beispiel

Abbildung 4:

```
mutable type table;
uses Table(table for Tab, char for Ind,
      char for Val, int for Int);
constant int maxTabSize;

table table_create(void) {
      ensures result' = new ∧ fresh(result);
}
bool table_add(table t, char i, char c) {
      modifies t;
      ensures result = (size(t^) < maxTabSize ∨ i ∈ t^)
         ∧ (if result then t' = add(t^, i, c) else t' = t^);
}
char table_read(table t, char i) {
      requires i ∈ t^;
      ensures result = lookup(t^, i);
}
```

Abbildung 4 enthält ein Fragment einer Spezifikation, die in LCL geschrieben ist. Es wird das `Table` Trait aus Abbildung 3 verwendet.

6 Implementierungen im Bezug auf Spezifikationen

Einer der Vorteile von Larch's zweireihigem Ansatz der Spezifikation ist, dass die Beziehung von Implementierung und Spezifikation relativ einfach ist. Das sieht man am einfachsten Anhand der Abbildung 5 und 6. Abbildung 5 ist eine Spezifkation in LCL und Abbildung 6 eine entsprechende C Implementierung.

Abbildung 5:

```
void choose(int x, int y) int z; {
      modifies z;
      ensures z' = x ∨ z' = y;
}
```

Abbildung 6:

```
void choose(int x, int y) int z; {
      if(x > y) z = x;
      else z = y;
}
```

Die Spezifikation definiert eine Relation zwischen dem Programmzustand, wenn `choose` aufgerufen wird, und dem Zustand, wenn die Prozedur beendet wird. Diese Relation enthält alle Paare von Zuständen $< pre, post >$, in denen

- der Zustand sich nur in einem Wert von der globalen Variable z unterscheidet, und
- in *post* der Wert von z einer der beiden Argumente, die `choose` übergeben wurden, ist.

Die Implementierung definiert ebenfalls eine Relation auf Programmzuständen. Diese Relation enthält alle Paare von Zuständen $< pre, post >$, in denen

- der Zustand sich nur in einem Wert von der Variable z unterscheidet, und
- in *post* der Wert von z das Maximum der beiden Argumente, die `choose` übergeben wurden, ist.

Man sagt, dass die Implementierung von `choose` in Abbildung 6 die Spezifikation von Abbildung 5 erfüllt — oder eine korrekte Implementierung von Abbildung 5 ist — weil die Relation, die durch die Implementierung definiert ist, eine Teilmenge der Relation ist, die durch die Spezifikation definiert ist. Der Begriff der Erfüllung für die Implementierung von abstrakten Typen ist ein wenig komplizierter, da der Implementierer eines abstrakten Typs simultan auf zwei Abstraktionsebenen arbeitet und wird daher hier nicht weiter ausgeführt.

Abbildung 7:

```
#include "bool.h"
#define maxTabSize(10)

typedef struct { char ind[maxTabSize]; char val[maxTabSize];
                 int next; } tableRep;
typedef tableRep * table;
```

```
table table_create(void) {
    table t;
    t = (table)malloc(sizeof(tableRep));
    if (t == 0) {
        printf("Malloc returned null in table_create\n");
        exit(1);
    }
    t->next = 0;
    return t;
}

bool table_add(table t, char i, char c) {
    int j;
    for (j = 0; j < t->next; j++)
        if (t->ind[j] == i) {
            t->val[j] = c;
            return TRUE;
        }
    if (t->next == maxTabSize) return FALSE;
    t->ind[t->next] = i;
    t->val[t->next++] = c;
    return TRUE;
}

char table_read(table t, char i) {
    int j;
    for (j = 0; TRUE; j++)
        if (t->ind[j] == i) return t->val[j];
}
```

Abbildung 7 zeigt eine Implementierung der LCL Spezifikation aus Abbildung 4. Ein Wert vom abstrakten Typ `table` wird hier durch einen Pointer zu einer Struktur, die zwei Arrays und einen Integer enthält, repräsentiert.

Die Abstraktionsfunktion ist die Brücke zwischen der in der Implementierung eines abstrakten Typs benutzten Datenstruktur, und den abstrakten Werten, die implementiert werden. Sie stellt jeden Wert des Repräsentationstyps als einen Wert des abstraken Typs da. Hier repräsentieren wir `table` durch einen Pointer `t` auf eine Struktur `tableRep`.

Wenn das Tripel $< ind, val, next >$ die Werte des Feldes dieser Struktur in einigen Zuständen `s` enthält, dann definieren wir den abstrakten Wert, der durch `t` im Zustand `s` repräsentiert wird, als `toTab(<ind, val, next>)`, wobei

```
toTab(<next, ind, val>) ==
  if next = 0 then empty
  else insert(toTab(<next - 1, ind, val>),
              ind[next], val[next])
```

Abstraktionsfunktionen bilden oft mehrere Parameter auf einen Wert ab. Hier zum Beispiel, wenn `t` den empty `table` repräsentiert, ist `if t->next = 0`, ganz gleich was der Inhalt von `t->ind` und `t->val` ist.

Die Typdefinitionen in Abbildung 7 erzeugen eine Datenstruktur mit der man jeden Wert des Typs `table` repräsentieren kann. Aber nicht jeder Wert der Datenstruktur kann einen Wert des Typs `table` repräsentieren. Beim Definieren der Abstraktionsfunktion nimmt man implizit an, dass gewisse Datenstrukturen gültige Repräsentationen sind. Zum Beispiel ist `toTab` nicht definiert, wenn `t->next` negativ ist. Um solche Annahmen explizit zu machen, benutzt man Repräsentationsinvarianten. Für diese Implementierung, ist die Repräsentationsinvariante:

- Der Wert von `next` liegt zwischen 0 und `maxTabSize`

  $0 \leq$ `t->next` $\wedge$ `t->next` $<$ `maxTabSize`

- und kein Index darf mehr als einmal in `ind` erscheinen (im Bereich 0 bis `next`):
  $\forall i, j : int$
  $(0 \leq i \wedge i < j \wedge j < t- > next)$
  $\Rightarrow (t- > ind[i]) \neq (t- > ind[j])$

Man beweist, dass diese Repräsentationsinvariante gültig ist, mit Hilfe von Datentypinduktion. Weil `tabel` ein abstrakter Typ ist, wissen wir, dass Clienten nicht direkt auf die Datenstruktur, die benutzt wird um `table` zu repräsentieren, zugreifen können. Deshalb sind alle Werte von Typ `table`, die während der Programmausführung vorkommen, von der Funktion, die im Interface spezifiziert wurde, generiert. Um zu beweisen, dass dies die Invariante erfüllt, reicht es, wegen der Implementierung der Funktionen `table`, zu zeigen:

- Der Wert, der von `table_create` zurückgegeben wird, erfüllt die Invariante (Grundschritt der Induktion).

- Immer wenn `table_add` aufgerufen wird und die Invariante für `t^` gilt, wird sie ebenfalls für `t'` gelten.

- Immer wenn `table_read` aufgerufen wird und die Invariante für `t^` gilt, wird sie ebenfalls für `t'` gelten.

Ein geringfügig anderes Datentyp-Induktions-Prinzip kann benutzt werden, um über Clients von abstrakten Datentypen zu urteilen. Um zu zeigen, dass eine Eigenschaft für alle Instanzen eines Typs gilt, induziert man über alle möglichen Sequenzen von Funktionsaufrufen, die locs eines Typs erzeugen oder ändern. Das ist ein Grund, warum man Spezifikationen der Funktionen eher als ihre Implementierungen benutzt. Zum Beispiel, um zu beweisen, dass `size(t)` niemals größer als `maxTabSize` ist, zeigt man:

- Die Spezifikation von `table_create` impliziert, dass die Größe, die von `table` zurückgegeben wird, nicht gößer als `maxTabSize` ist.

- Die Spezifikation von `table_add` impliziert zusammen mit der Hypothese $t\hat{} \leq$ `maxTabSize`, dass $t' \leq$ `maxTabSize`.

Wenn die Abstraktionsfunktion gegeben ist, ist es relativ einfach zu definieren, wie sie bei der Prozedurimplementierung in Abbildung 7 sein muss, um die Spezifikation in Bild 4 zu erfüllen. Zum Beispiel, sagt man, dass die Implementierung von `table_read` ihre Spezifikation erfüllt, weil das Bild, was man bekommt, wenn man die Abstraktionsfunktion auf allen Werten des Typs `table` in der durch die Implementierung definieren Relation anwendet, eine Teilmenge der durch die Spezifikation definierten Relation ist. Bemerke, dass jedes Argument, bei dem die Implementierung von `table_read` ihre Spezifikation erfüllt, auf die `requires` Klausel der Spezifikation und auf die Repräsentationsinvariante vertraut.

7 LP, Larch Proof Assistant

In der obigen Beschreibung von LSL und LCL wurde schon erwänt, dass Werkzeuge existieren, die diese Sprachen unterstützen. LP ist ein solches Werkzeug, es unterstützt LSL, LCL und LM3.

LP ist ein Beweisassistent für eine Teilmenge von mehrsortiger Prädikatenlogik erster Ordnung mit Gleichheit, also der Logik, auf der die Sprache Larch basiert. LP wurde entworfen, um große Probleme effizient zu bearbeiten und um Spezifizierern mit wenig Erfahrung beim Satz-Beweisen zu helfen.

LP ist hauptsächlich ein interaktiver Beweisassistent oder Beweisdebugger, also kein vollautomatischer Satz-Beweiser. So wurde LP entworfen um standard (aber möglischerweise langatmige) Beweis-Schritte automatisch durchzuführen und nützliche Informationen, für den Fall, dass der Beweis schiefgeht, zu sammeln. LP basiert nicht auf komplizierten Heuristiken, um Beweise Automatisch zu finden. Es macht es Benutzern dafür einfacher, standard Techniken wie Beweis durch Fälle, Induktion oder Widerspruch anzuwenden, die Wahl zwischen den Strategien bleibt aber beim Benutzer.

7.1 Der Lebenszyklus eines Beweises

Beweisen ist wie Programmieren: Beweise werden entworfen, kodiert, gedebugged, und (manchmal) dokumentiert. Bevor man einen neuen Beweis entwirft, ist es notwendig die Dinge, über die argumentiert wird, und die Vermutung zu formalisieren. Der Entwurf des Beweises beginnt mit einem Umriss der Struktur, einschliesslich Schlüssel Lemmas und Beweismethoden. Der Beweis selber muss in hinreichender Genauigkeit gegebein sein, um überzeugen zu können. Menschen sind häufig auch durch unzuverlässige Beweise überzeugt, deshalb ist es gut einen mechanischen Skeptiker zu haben, der genau prüft.

Wurde ein Teil eines Beweises schon kodiert, kann LP zu debuggen verwendet werden. Die wenigsten Beweise sind beim ersten mal erfolgreich. Manchmal ist die Vermutung falsch, manchmal die Formalisierung oder der Beweis ist noch unvollständig. Es kann aber auch sein, dass die Beweisstrategie fehlerhaft oder nicht detailliert genug ist. LP unterstützt einige Möglichkeiten, die dazu beitragen, dass Problem zu finden und zu verstehen, wenn der versuchte Beweis versagt.

Während dem Debuggen von Beweisen, formulieren Benutzer häufig die Axiome und Vermutungen neu. Nach jeder Änderung in der Axiomatisierung ist es notwendig, nicht nur die Vermutungen, die der Beweis grade versuchte zu beweisen, wieder zu überprüfen, sondern ebenfalls die Vermutungen, die mit den alten Axiomen schon bewiesen wurden. LP unterstützt einen bei dieser Arbeit.

Man kann ein Session-Skript aufnehmen, was man wieder abspielen kann. LP *prettyprints* die Skript Dateien, indem er so einrückt, dass die Strukur des Beweises wiedergegeben wird. Er kommentiert gegebenenfalls Skript Dateien mit Informationen, die anzeigen, wann ein Teilziel eingeführt wurde (im Allgemeinen bei Beweisen durch Induktion), und wann Teilziele und Sätze bewiesen sind. Auf Wunsch hält LP, wenn er ein Skript wieder abspielt, am ersten Punkt, wo die Anmerkung und die neuen Beweise divergieren. Das macht es einfacher den Beweis davor zu bewahren, „out of sync“mit dem eigentlichen Gedanken des Autors zu werden.

7.2 Ein kleiner Beweis

Abbildung 8:

```
Nat: trait
      includes AC(+, Nat)
      introduces
         0: → Nat
         s: Nat → Nat
         __ < __ : Nat, Nat → Bool
      assets
         Nat generated by 0, s
         ∀ i, j, k: Nat
            i + 0 == i;
            i + s(j) == s(i + j);
            ¬(i < 0);
            0 < s(i);
            s(i) < s(j) == i < j
         implies ∀ i, j, k: Nat
            i < j ⇒ i < (j + k)
```

Abbildung 8 enthält eine kurze LSL Spezifikation, einschließlich einer kleinen Vermutung (die folgt dem reservierten Wort `implies`), die vermutlich aus den Axiomen folgt. Abbildung 9 zeigt ein Skript von einem LP Beweis dieser Vermutung.

Abbildung 9:

```
set name nat
declare sort Nat
declare variables i, j, k: Nat
declar operators
      0: → Nat
      s: Nat → Nat
      +: Nat, Nat → Nat
      <: Nat, Nat → Bool
      ..
assert Nat generated by 0, s
assert ac +
assert
      i + 0 == i
      i + s(j) == s(i + j)
      ¬(i < 0)
      0 < s(i)
      s(i) < s(j) == i < j
      ..
set name lemma
prove i < j ⇒ i < (j + k) by induction on j
       <> 2 subgoals for proof by induction on j
         [] basis subgoal
         resume by induction on i
         <> 2 subgoals for proof by induction on i
            [] basis subgoal
            [] induction subgoal
         [] induction subgoal
      [] conjecture
qed
```

Das `declare` Kommando führt Variablen und Operatoren in die LSL Spezifikation ein. Das `assert` Kommando liefert die LSL Axiome im Bezug auf die Operatoren; die `Nat generated by` Behauptung ist ein Induktionsschema für `Nat`. Das `prove` Kommando beginnt einen Beweis durch Induktion. Der Diamant (`<>`) sind Bemerkungen, die von LP unterstützt werden; sie zeigen die Ausgabe von Teilergebnissen der Induktion an. Die Box (`[ ]`) Bemerkungen werden ebenfalls von LP unterstützt; sie zeigen die Ausgabe von Teilergebnissen und schließlich die Ausgabe des Hauptbeweises an. Das `resume` Kommando beginnt eine verschachtelte Induktion. Keine weitere Benutzerinteraktion ist nötig, um diesen Beweis zu vervollständigen. Das `qed` Komando in der letzten Zeile fragt LP nach einer Bestätigung, dass keine ausstehenden Vermutungen mehr vorhanden sind.

7.3 Beweismechanismen in LP

Hier noch ein Überblick über die Beweismechanismen in LP: LP unterstützt zwei Mechanismen um Sätze zu beweisen, die *forward* und die *backward inference* (Schlussfolgerungen). Die *forward inferences* folgern Konsequenzen aus einem logischem System, die *backward inferences* hingegen erzeugen Teilziele, deren Beweis ausreicht, um die Vermutung zu begründen. Für die *forward inference* existieren in LP vier Methoden:

1. Die automatische **Normalisierung** erzeugt neue Konsequenzen, wenn eine neue Umschreiberegel in ein System hinzugefügt wird. LP hält Umschreiberegeln, Gleichungen und Folgerungsregeln in der normalen Form.

   Wenn eine Gleichung oder Umschreiberegel zu einer Identität normalisiert, wird sie ausrangiert, da sie logisch und operational überflüssig ist. Wenn alle Hypothesen einer Folgerungsregel zu Identiäten normalisieren, wird die Folgerungsregel durch die Gleichungen ihrer Schlussgfolgerung ersetzt. Wenn alle Folgerungen einer Folgerungsregel zu Identitäten normalisieren, wird die Folgerungsregel ausrangiert.

   Benutzer können Gleichungen, Umschreiberegeln und Folgerungsregeln vor der automatischen Normalisierung schützen, um die Performance von LP zu verbessern und um einzelne Formen davor zu bewaren, in einem Beweis benutzt zu werden. Ebenfalls ist es möglich, Umschreiberegeln und Folgerungsregeln auszuschalten, um sie davor zu schützen, automatisch angewendet zu werden.

2. Die automatische **Anwendung von Folgerungsregeln** erzeugt neue Konsequenzen, nachdem Gleichungen und Umschreiberegeln eines Systems normalisiert wurden. Folgerungsregeln können ebenso durch einen expliziten Befehl angewendet werden. Zum Beispiel um Gleichungen zu schützen.

3. Die Berechnung von **Kritischen-Paar Gleichungen** und der Knuth-Bendix Vervollständigungsfunktion erzeugt Gleichungs-Konsequenzen (so wie `i(e) == e`) von unvollständigen Umschreibesystemen.

4. Explizite **Instanziierung** von Variablen in Formeln, Umschreiberegeln und Folgerungsregeln erzeugt ebenfalls Konsequenzen. Zum Beispiel kann man in einem System, das die Folgerungsregel

   $$when\ \ (\forall e)\ \ e \in x == e \in y\ \ yield\ \ x == y$$

   und die Umschreiberegel $e \in (x \cup y) \rightarrow e \in x \vee e \in y$ enthält, y in der Folgerungsregel durch $x \cup x$ instanziiren, um die Schlußfolgerung $x == x \cup x$ zu erzeugen.

Für die *backward inferences* existieren sieben Beweismethoden. Diese werden durch die `prove` und `resume` Befehle aufgerufen. In jeder Methode generiert LP eine Menge von Teilzielen, die zu beweisen sind. Diese Teilziele sind Lemmas, die zusammen ausreichen, um die Schlussfolgerung zu implizieren. Für manche Methoden erzeugt LP zusätzliche Hypothesen, die dazu benutzt werden können, besondere Teilziele zu beweisen.

1. **Normalisierung** umschreibt Vermutungen. Wenn eine Vermutung zu einer Identität normalisiert, dann ist sie ein Satz. Sonst wird die normalisierte Vermutung ein Teilziel, das noch zu beweisen ist.

2. **Beweis durch Fallunterscheidung** kann auch eine Vermutung normalisieren. Der Befehl `prove e by cases t1, ..., tn`, wobei `t1, ..., tn` Prädikate sind, zwingt LP dazu die Gleichung `e` für den Beweis in die Fälle `t1, ..., tn` zu unterscheiden. Oder für $n = 1$ in die Fälle: `t1` und ¬`t1`. Für $n > 1$ ist ein zu beweisendes Teilziel, dass die Fälle `t1` ∨ ... ∨ `tn` ausreichen. Zusätzlich werden in jedem Fall `ti` die Variablen von `ti` in `ti` und `e` durch neue Konstanten substituiert. Dadurch werden `ti'` und `ei'` gebildet, welche dann benutzt werden, um zusammen mit der Hypothese `ti'` → `true` das neue Teilziel `ei'` zu bilden. Wenn durch das Hinzunehmen der Hypothese `ti'` zu dem Fall `ti` eine Inkonsistenz folgt, dann ist der Fall unmöglich und `ei'` nichtssagend wahr. Sonst muss gezeigt werden, dass das Teilziel `ei'` aus den durch die Hinzunahme der Hypothese hinzugekommenen Axiomen folgt.

   Die Fallunterscheidung hat zwei wichtige Anwendungen. Wenn die Vermutung ein Satz ist, dann kann die Fallunterscheidung ein Mangel an Vollständigkeit in der Umschreiberegel umgehen. Wenn die Vermutung kein Satz ist, dann kann ein versuchter Beweis durch Fallunterscheidung die Vermutung vereinfachen und macht es leichter zu verstehen, warum der Beweis nicht folgt.

3. **Beweise durch Induktion** basieren auf Induktionsregeln. Zum Beispiel ein Induktionsbeweis von

   $$isEmpty(c) \Rightarrow \neg(e \in c)$$

   aus den Axiomen von `LinearContainer` benötigt zwei Schritte. Im *Grundschritt* muss gezeigt werden, dass

   $$isEmpty(empty) \Rightarrow \neg(e \in empty)$$

   Dies folgt aus den Axiomen durch Normalisierung. Im *Induktionsschritt* wählt man eine neue Konstante `cc` und nimmt an, dass gilt: (Induktionsannahme)

   $$isEmpty(cc) \Rightarrow \neg(e \in cc)$$

   Man zeigt dann, dass:

   $$isEmpty(insert(e1, cc)) \Rightarrow \neg(e \in insert(e1, cc))$$

   Dies folgt durch Normalisierung aus den durch die Induktionsannahme hinzugekommenen Aximonen.

4. **Beweise durch Wiederspruch** liefern eine indirekte Beweismethode. Wenn aus dem Hinzunehmen der Negation der Vermutung im logischen System von LP eine Inkonsistenz folgt, dann ist die Vermutung ein Satz.

5. **Beweise von Implikationen** kann man führen, indem man eine vereinfachte Form der Beweise durch Fallunterscheidung benutzt. Der Befehl `prove t1` ⇒ `t2 by` ⇒ zwingt LP dazu, das Teilziel `t2'` mit Hilfe der Hypothese `t1'` →

`true` zu beweisen. `t1'` und `t2'` erhält man wie beim Beweis durch Fallunterscheidung. Dies reicht aus, weil die Implikation nichtssagend wahr ist, wenn `t1'` falsch ist.

6. **Beweise von Bedingungen** kann man führen indem man eine vereinfachte Form der Beweise durch Fallunterscheidung benutzt. Der Befehl

   $$prove\ \ if(t1, t2, t3) == t4\ \ by\ \ if$$

   zwingt LP dazu, das Teilziel `t2' == t4'` mit Hilfe der Hypothese `t1'` zu beweisen, und das Teilziel `t3' == t4'` mit Hilfe der Hypothese ¬`t1'`. `t1'`, `...`, `t4'` erhält man wie beim Beweis durch Fallunterscheidung.

7. **Beweise von Konjunktionen** liefern einen Weg, um den Aufwand der Umschreibung modulo der Assoziativität und Kommutativität von ∧ zu reduzieren. Der Befehl `prove t1` ∧...∧ `tn by` ∧ zwingt LP dazu jeden `t1, ...,` `tn` als ein getrenntes Teilziel zu beweisen.

LP erlaubt dem Benutzer zu spezifizieren, welche Methoden der *backward inference* automatisch angewendet werden, und in welcher Reihenfolge. Das kann man mit Hilfe des Befehls `proof-methods` machen. Zum Beispiel der LP Befehl

```
set proof-methods if, ⇒,normalisation
```

sagt LP, dass immer, wenn eine Vermutung bewiesen werden muss, eine dieser drei Methoden automatisch angewendet werden soll, in der gegebenen Reihenfolge.

8 Quellen

John V. Guttag, James J. Horning Larch: Languages And Tools For Formal Specification, Springer-Verlag New York 1993