Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Formale Verifikation von Heap-Algorithmen mit Frama-C

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von: Timon Lapawczyk
geboren am: 28.04.1993
geboren in: Berlin

Gutachter/innen: Prof. Dr. Holger Schlingloff
Dr. Jens Gerlach

eingereicht am: .....

verteidigt am: .....

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit entstand in Verbindung mit der Arbeit des Autors für die Verifikationsgruppe des Fraunhofer Instituts für Offene Kommunikationssysteme [1]. Im Rahmen seiner Tätigkeit schrieb der Autor unter anderem am Bericht *ACSL By Example* [1] mit.

In *ACSL By Example* werden Beispiele für die formale Spezifikation von C-Funktionen mit der ANSI/ISO-C Spezifikationssprache ACSL und die deduktive Verifikation mit Frama-C und dem Plug-in WP vorgestellt.

Bei der formalen Verifikation wird ein mathematischer Beweis erbracht, dass eine Funktion ihre Spezifikation erfüllt. Diese Form der Qualitätssicherung findet vor allem in sicherheitskritischen Bereichen Anwendung, in denen Fehler in der Software fatale Folgen haben können. Da die Programmiersprache C unter anderem in sicherheitskritischen eingebetteten Systemen eingesetzt wird, ist sie besonders interessant für die formale Verifikation.

Die Algorithmen in dieser Arbeit stammen aus der *Standard Template Library* [2]. Die *STL* ist eine Bibliothek für die C++ Programmiersprache, aus der viele Teile in die C++ Standard Bibliothek aufgenommen wurden. Für die Verifikation mit Frama-C werden die Funktionen aus der *STL* in der Sprache C implementiert.

Die formale Verifikation der Heap-Algorithmen in dieser Arbeit baut auf den bisherigen Ergebnissen aus *ACSL By Example* auf und soll auch Teil der nächsten Version von *ACSL By Example* sein.

1.1 Motivation und Ziele

Bei der Verifikation einer Funktion, die eine bereits formal verifizierte Funktion aufruft, wird die Spezifikation der aufgerufenen Funktion vorausgesetzt. So lassen sich komplexe Abläufe, wenn sie einmal formal verifiziert wurden, mit geringen Auswirkungen auf den Verifikationsaufwand in neue Implementierungen übernehmen.

Dadurch bildet die formale Verifikation von Algorithmen aus Standard Bibliotheken eine Grundlage für die einfache Verifikation vieler weiterer Funktionen.

In dieser Arbeit werden dafür die Algorithmen Push-Heap und Make-Heap formal verifiziert. Da die Algorithmen dicht zusammenhängen, lassen sich gut die Unterschiede in der Komplexität der Verifikation zwischen Funktionen, die sich hauptsächlich aus bereits verifizierten Funktionen aufbauen und Funktionen, die von Grund auf implementiert werden, ausmachen.

Während der Verifikation dieser Funktionen wird eine Methodik zur Quelltext-Annotation erarbeitet, die eine weitestgehend automatische Verifikation ermöglicht. Diese Methodik soll auch bei der Verifikation von anderen schwierigen Funktionen, unter anderem Funktionen mit Schleifen helfen, die richtigen Annotationen zu finden, um eine möglichst automatische Verifikation zu ermöglichen.

Diese Arbeit dient also nicht nur der Erweiterung der Bibliothek von formal verifizierten Funktionen, sondern bietet auch eine Hilfestellung für die formale Verifikation weiterer Funktionen.

[1] Die Verifikationsgruppe gehört zum System Quality Center des FOKUS Instituts. Siehe `https://www.fokus.fraunhofer.de/sqc`

1.2 Aufbau der Arbeit

Die Arbeit beginnt mit Kapitel 2, welches eine thematische Einführung darstellt. Die Heap-Datenstruktur wird informell eingeführt und es werden unterschiedliche Operationen auf Heaps vorgestellt. Bereits in diesem Kapitel wird eine Implementierung der beiden Funktionen gezeigt.

In Kapitel 3 werden die theoretischen Grundlagen für eine formale Betrachtung der in Kapitel 2 vorgestellten Konzepte gelegt. Dabei wird unter anderem die Spezifikationssprache ACSL vorgestellt und eine Übersicht über den Verifikationsprozess mit Frama-C und dem Plug-In WP gegeben.

Das Kapitel 4 bildet den Hauptteil dieser Arbeit. In diesem Kapitel werden die Inhalte des Kapitels 2 neu aufgearbeitet und formal spezifiziert. Den Großteil dieses Kapitels nehmen die Dokumentation der annotierten Implementierung der Heap-Algorithmen und die Vorstellung der Methodik für das Finden der Annotation ein.

In Kapitel 5 werden die Verifikationsergebnisse der Spezifikation aus Kapitel 4 vorgestellt.

Das Kapitel 6 beinhaltet eine Reflexion der Ergebnisse des Hauptteils. Es werden Erkenntnisse aus dem Verifikationsprozess und die Anwendbarkeit der Methoden auf ähnliche Funktionen diskutiert.

2 Heap-Algorithmen

Der C++ Standard definiert das Konzept von Heaps wie folgt [3, §25.4.6, eigene Übersetzung]:

1. Ein *Heap* ist eine besondere Organisierung von Elementen in einem Bereich zwischen zwei random access Iteratoren `[a,b)` mit den beiden Schlüsseleigenschaften:
   a) Es existiert kein Element größer als `*a` im Bereich und
   b) `*a` kann mit `pop_heap()` entfernt werden oder ein neues Element kann mit `push_heap()` eingefügt werden in $\mathcal{O}(\log(N))$.
2. Durch diese Eigenschaften bieten sich Heaps für die Verwendung als Prioritätswarteschlangen an.
3. `make_heap()` organisiert einen Bereich in einen Heap und `sort_heap()` wandelt einen Heap in eine sortierte Sequenz.

Diese Definition bildet einen guten Einstieg in die Thematik von Heaps und geht sogar auf den Zusammenhang der einzelnen Algorithmen ein. Für die formale Verifikation von Heaps und Heap-Algorithmen bedarf es aber einer genaueren Definition.

Der Apache C++ Standard Library User's Guide liefert die folgende Definition [4, §14.7, eigene Übersetzung]:

> Ein Heap ist ein Binärbaum, in dem jeder Knoten größer als die Werte aller seiner Kinderelemente ist. Ein Heap und ein Binärbaum können sehr effizient in einem Vektor gespeichert werden. Die Kinder für den Knoten i werden an die Positionen $2i + 1$ und $2i + 2$ geschrieben.

Die Definition von Heaps in dieser Arbeit bedient sich aller Eigenschaften der beiden zitierten Definitionen. In den folgenden Abschnitten werden dafür Schritt für Schritt die einzelnen Konzepte, die für ein genaues Verständnis der Eigenschaften nötig sind, erklärt.

Begonnen wird mit der strukturellen Eigenschaft für die Speicherung der Elemente. Für die Eigenschaft, die Elemente in einem Vektor speichern zu können, reicht die Restriktion, dass jeder Knoten maximal zwei Kinder haben darf, aus. Damit aber jedes Element im Bereich `[a,b)` zum Heap gehört, wie von der C++ Standard Definition gefordert, muss der Baum ein vollständiger Binärbaum sein. Nur so ist der Vektor lückenlos mit Elementen aus dem Heap gefüllt.

Vollständiger Binärbaum

In einem vollständigen Binärbaum gilt:

- Jeder Knoten besitzt maximal zwei Kinder,
- bis auf die letzte Ebene sind alle Knoten belegt,
- ein neuer Knoten wird an die erste freie Stelle von links eingefügt

Abbildung 2.1 zeigt einen vollständigen Baum für die Multimenge {3,4,4,5,5,7,9,11,14,16} aus ganzen Zahlen, der alle Voraussetzungen für einen vollständigen Binärbaum erfüllt. Der Baum erfüllt, wie im weiteren Verlauf des Kapitels geklärt wird, auch alle weiteren Eigenschaften eines Heaps.

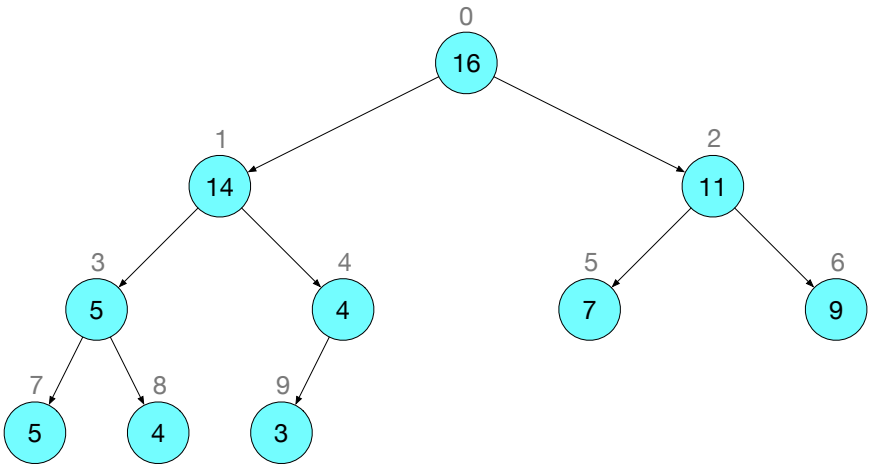

Abbildung 2.1: Beispiel Heap der Größe 10

Ordnung der Elemente

Beide Definitionen fordern eine gewisse Ordnung auf den Elementen. Für zwei Knoten, die in der Eltern-Kind Relation stehen, gilt, dass sie die totale Ordnung, größer-gleich auf den ganzen Zahlen, erfüllen. Für drei Elemente a, b, c vom Typ Integer gilt:

- Wenn $a \geq b$ und $b \geq a$, so gilt $a = b$ (*Antisymmetrie*),
- wenn $a \geq b$ und $b \geq c$, so gilt $a \geq c$ (*Transitivität*),
- es gilt $a \geq b$ oder $b \geq a$ (*Totalität*)

Angewendet auf Heaps bedeutet das, jedes Element ist größer-gleich zu seinen Kinder-Elementen. Daraus folgt, dass an der Wurzel des Heaps das größte Element steht. Beide Eigenschaften lassen sich am Beispiel Heap in Abbildung 2.1 beobachten.

Array-Schreibweise

Für die Speicherung von Heaps wird die in der Apache Definition vorgestellte Schreibweise implementiert. Ein Heap wird in einem C-Array gespeichert, indem das Wurzelelement an die erste Stelle des Arrays geschrieben wird und die Kinderelemente für jeden Knoten i an die Stellen:

`2·i+1` (linkes Kind) und

`2·i+2` (rechtes Kind)

geschrieben werden.

In Abbildung 2.1 stehen die Indizes für die Array Schreibweise in grau über den Knoten.

2.1 Vorteile der Heap-Datenstruktur

Da sich die Elemente eines Heaps in einem Array speichern lassen, wird nur die minimale Datenmenge benötigt. Es müssen nur die Werte der einzelnen Elemente gespeichert werden und es sind keine zusätzlichen Informationen, wie Zeiger auf die Kinderelemente, nötig. Außerdem reicht die Adresse des ersten Elements und die Länge des Heaps aus, um einen Heap zu referenzieren.

Abstrakte Heap-Operationen

Für Heaps sind die folgenden vier Operationen definiert. Sie werden auch in der C++ Standard Definition erwähnt und lassen sich auch im *worst-case* in logarithmischer Zeit ausführen.

- Push-Heap fügt ein neues Element zu einem Heap hinzu.
- Pop-Heap entfernt das Wurzelelement eines Heaps.
- Make-Heap formt ein beliebig sortiertes Array in einen Heap um.
- Sort-Heap formt einen Heap in ein sortiertes Array um.

Aufgrund der schnellen Einfüge- und Entfernoperationen finden Heaps zum Beispiel Anwendung in der Implementierung von Prioritätswarteschlangen [5, Seite 150-152]. Mit den Heap-Operationen lässt sich außerdem das Sortierverfahren Heapsort implementieren, das im *worst-case* eine Laufzeit von $O(n \cdot \log(N))$ hat [5, Seite 144-148]. Die folgende Tabelle zeigt die Laufzeiten für die vier vorgestellten Heap-Operationen.

Operation	Push-Heap	Pop-Heap	Make-Heap	Sort-Heap
Laufzeit	$\mathcal{O}(\log(N))$	$\mathcal{O}(\log(N))$	$\mathcal{O}(N \cdot \log(N))$	$\mathcal{O}(N \cdot \log(N))$

2.2 Heap-Algorithmen in der Standard Template Library

Für die vier Heap-Operationen existiert jeweils eine Implementierung in der Standard Template Library. Die Funktionen werden in der STL wie folgt spezifiziert:

- `push_heap` fügt ein Element zu einem Heap hinzu. Es wird vorausgesetzt, dass `[first, last-1)` schon ein Heap ist. Das Element, das hinzugefügt werden soll, ist `*(last-1)`. Als Nachbedingung gilt, dass `[first, last)` ein Heap ist.
- `pop_heap` entfernt das größte Element (`*first`) vom Heap `[first, last)`. Als Nachbedingungen gelten, dass `[first, last-1)` ein Heap ist und dass `*(last-1)` das entfernte Element ist.
- `make_heap` wandelt den Bereich `[first, last)` in einen Heap um. Als Nachbedingung gilt, dass `[first, last)` ein Heap ist.
- `sort_heap` wandelt einen Heap `[first, last)` in eine sortierte Folge um.

Diese Spezifikationen sind schon formaler und geben spezifische Eigenschaften für die Implementierungen der Funktionen vor. In dieser Arbeit werden die Algorithmen `push_heap` und `make_heap` genauer behandelt. Die Algorithmen hängen dicht zusammen, da die Implementierung von `make_heap` die Funktion `push_heap` aufruft. In den folgenden Abschnitten werden die beiden Funktionen, beginnend mit `push_heap`, näher vorgestellt und es wird jeweils eine Implementierung gezeigt.

2.3 Der Algorithmus `push_heap`

Die in Abschnitt 2.2 spezifizierte Funktion `push_heap` aus der STL ist für ein Paar aus generischen random access Iteratoren definiert [6]. Heaps werden in dieser Arbeit nur für ganze Zahlen betrachtet und so ist `push_heap` in dieser Arbeit als Funktion auf Integer-Arrays definiert. Das Array wird mit einem Zeiger auf das erste Element und der Länge des Arrays übergeben. Daraus ergibt sich die folgende Signatur:

```
void push_heap(int *a, unsigned int n);
```

Die Funktion `push_heap` erwartet ein Array der Länge n, dessen erste $(n-1)$ Elemente einen Heap bilden. Das einzufügende Element steht zu Funktionsbeginn an der letzten Stelle des Arrays. Diese Spezifikation hat den Vorteil, dass sichergestellt ist, dass im Array genug Platz für einen Heap der Größe n ist und in der Implementierung von `push_heap` keine Speicherverwaltung stattfinden muss. Nach dem Funktionsaufruf bildet das Array einen Heap der Länge n.

Die Abbildung 2.2 zeigt das Array aus Abbildung 2.1, ergänzt um das neue Element 15, im Vorzustand des Aufrufs von `push_heap`.

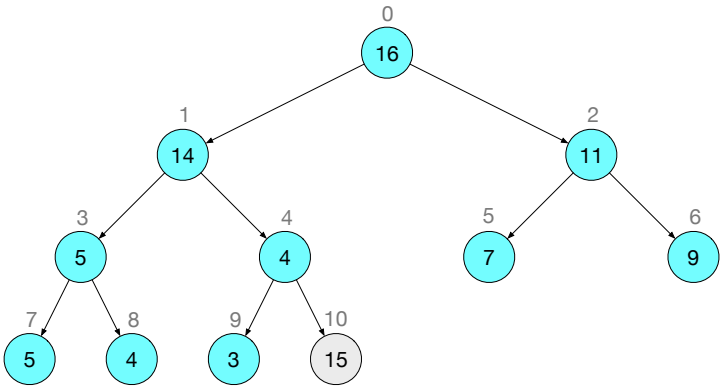

Abbildung 2.2: Beispiel-Aufruf von `push_heap` (Vorzustand)

Bis auf den letzten Knoten sind alle Knoten blau gefärbt und bilden gemeinsam einen Heap. Der letzte, grau gefärbte Knoten gehört noch nicht zum Heap. Abbildung 2.3 zeigt das Array im Nachzustand des Aufrufs von `push_heap`.

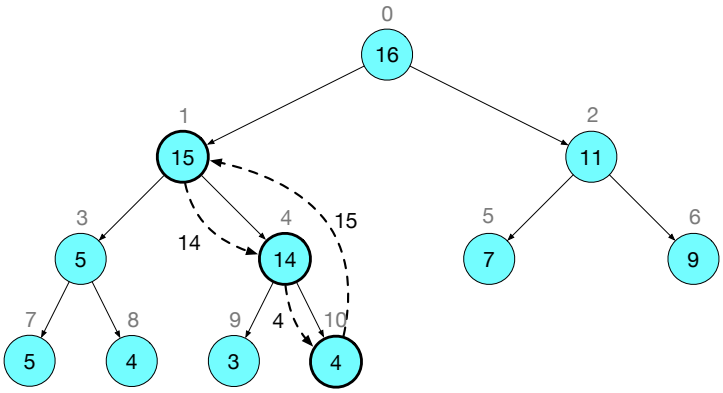

Abbildung 2.3: Beispiel-Aufruf von `push_heap` (Nachzustand)

Im Nachzustand sind alle Knoten blau gefärbt und bilden gemeinsam einen Heap.

Implementierung von `push_heap`

Zu Funktionsbeginn gilt die größer-gleich Beziehungen zwischen allen Elementen in der Eltern-Kind Beziehung bis zum vorletzten Element. Die Funktion darf erst beendet werden, wenn die größer-gleich Beziehung auch zwischen dem letzten Element und seinem Elternelement gilt.

Listing 2.1 zeigt eine Implementierung von `push_heap`.

```
void push_heap(int* a, unsigned int n)
{
  if (n == 1) {
    return;
  }
  const int val = a[n - 1];
  unsigned int hole = n-1;
  while (hole > 0) {
    const unsigned int par = (hole-1) / 2;
    if (a[par] < val) {
      a[hole] = a[par];
      hole = par;
    } else {
      break;
    }
  }
  a[hole] = val;
}
```

Listing 2.1: Implementierung von `push_heap`

Wenn der Heap im Nachzustand die Größe 1 haben soll, kann die Funktion abbrechen, denn es gibt keine 2 Heap-Elemente.

Für alle anderen Heapgrößen muss die größer-gleich Beziehung zwischen dem letzten Element und seinem Elternelement hergestellt werden. Dafür wird das neue Element in der Variable `val` zwischengespeichert. Anschließend wird in einer Schleife das Array mit dem Iterator `hole`, beginnend bei $(n-1)$, durchlaufen.

In der Schleife wird erst geprüft, ob das neue Element an der Stelle `a[hole]` eingefügt werden kann, so dass die Heap-Ordnung erfüllt ist.

- Wenn das der Fall ist, bricht die Schleife ab.
- Wenn das nicht der Fall ist, wird das Elternelement `a[par]` nach `a[hole]` geschrieben und die Variable `hole` wird mit dem Index des Elternelements aktualisiert.

  Die größer-gleich Beziehung zwischen dem alten Element `hole` und dem Elternelement ist nun hergestellt, da beide Elemente den gleichen Wert haben. Das Problem wird im nächsten Schleifendurchlauf eine Ebene weiter oben betrachtet.

Zum Schluss wird der neue Wert dem Element `a[hole]` zugewiesen. In der Schleife wurde sichergestellt, dass es kein Elternelement zu `a[hole]` gibt, welches kleiner als der neue Wert ist und aus dem letzten vollständig durchlaufenen Schleifendurchlauf ist klar, dass das neue Element größer als das alte Element in `a[hole]` ist. Die Heap-Ordnung ist hergestellt.

Auch wenn die Schleife schon im ersten Durchlauf abbricht, ist die Heap-Ordnung hergestellt.

2.4 Der Algorithmus `make_heap`

Auch die in Abschnitt 2.2 spezifizierte Funktion `make_heap` aus der STL [7] wird in dieser Arbeit als Funktion auf Integer-Arrays definiert. Die Signatur für `make_heap` sieht wie folgt aus:

```
void make_heap(int* a, unsigned int n);
```

Die Funktion `make_heap` ordnet die Elemente eines Arrays `a[0..n-1]` so um, dass sie einen Heap bilden. Die Abbildungen 2.4 und 2.5 zeigen die Umordnung in einem Array, durch den Aufruf von `make_heap`.

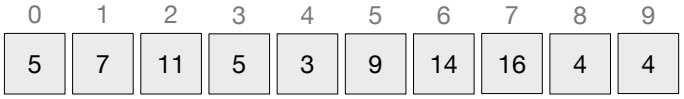

Abbildung 2.4: Das Array [5,7,11,5,3,9,14,16,4,4] vor dem Aufruf von `make_heap`

Im Vorzustand ist das Array der Länge 10 ungeordnet. Alle Elemente sind grau gefärbt und bilden zusammen keinen Heap.

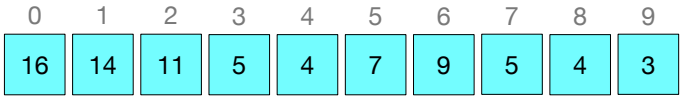

Abbildung 2.5: Das Array nach dem Aufruf von `make_heap`

Beim Aufruf von `make_heap` wird die Position einiger Elemente verändert. Alle Elemente sind blau gefärbt, um zu signalisieren, dass sie zusammen einen Heap bilden. Für jedes Array-Element gilt, dass es größer-gleich zu seinen Kinder-Elementen ist. Diese Beziehung wird besonders deutlich, wenn das Array im Nachzustand, wie in Abbildung 2.6 zu sehen, in der Baumschreibweise dargestellt wird.

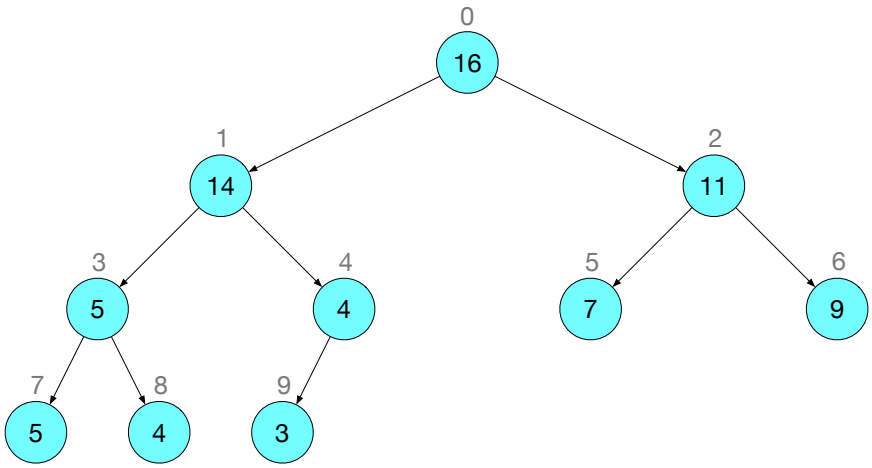

Abbildung 2.6: Die Baumstruktur des Arrays nach dem Aufruf von `make_heap`

Implementierung von `make_heap`

Mit der Funktion `push_heap` ist die Implementierung von `make_heap` relativ einfach. Um die Elemente des Arrays der Länge n in einen Heap umzuordnen, wird $(n-1)$ mal die Funktion `push_heap` auf dem Array aufgerufen. Dabei wächst die angegebene Länge für den Aufruf von `push_heap` mit jedem Aufruf um 1. Für ein Array der Länge 1 findet kein `push_heap` Aufruf statt, da jedes Element für sich alleine ein Heap ist. Listing 4.6 zeigt eine Implementierung von `make_heap`, die diesen Ansatz verfolgt.

```
void make_heap(int* a, unsigned int n)
{
  if (n == 0) {
    return;
  }

  for (unsigned int i = 2; i <= n; i++) {
    push_heap(a, i);
  }

}
```

Listing 2.2: Implementierung von `make_heap`

3 Die Werkzeugplattform Frama-C und die ACSL-Spezifikationssprache

Die Verifikation findet in dieser Arbeit mit der Verifikationsplattform Frama-C (Framework for Modular Analysis of C Programs) statt. Frama-C wird an den französischen Forschungseinrichtungen *CEA List (Commisariat à l'Énergie Atomique et aux Énergies Alternatives)* und *INRIA (Institut National de Recherche en Informatique et Automatique)* entwickelt und stellt Werkzeuge zur Analyse von C-Programmen bereit. Die Plattform hat zahlreiche Plug-ins, die sich hauptsächlich mit der statischen Analyse beschäftigen.

In dieser Arbeit wird das Plug-in WP [8] eingesetzt. Mit diesem Plug-in werden Beweisverpflichtungen aus einem C-Programm und der zugehörigen formalen Spezifikation generiert.

Dieses Kapitel behandelt die theoretischen Grundlagen und stellt die Umsetzung im Kontext der ACSL-Spezifikationssprache und der Werkzeugplattform Frama-C vor. Die Darstellung folgt Kapitel 2.2 der Diplomarbeit von Kim Völlinger [9].

3.1 Theoretische Grundlagen

Im Jahr 1969 stellte C. A. R. Hoare mit dem *Hoare Kalkül* eine Menge von logischen Regeln vor, welche es ermöglicht, Aussagen über Eigenschaften von imperativen Programmen zu formulieren [10].

Die Basis bildet das sogenannte Hoare-Tripel, das den Vor- und Nachzustand eines Programms beschreibt.

Für das Hoare-Tripel $\{V\}P\{N\}$ gilt, wenn vor der Ausführung des Programms P die Vorbedingung V gilt, dann gilt nach der Terminierung des Programms die Nachbedingung N.

Die Vor- und Nachbedingungen werden als prädikatenlogische Formeln definiert.

Beim einige Jahre später von E. W. Dijkstra vorgestellten *weakest precondition* Kalkül [11], auf Deutsch schwächste Vorbedingung Kalkül, wird durch die Rückwärtsanalyse des Programms die schwächste Vorbedingung gefunden, damit eine gegebene Nachbedingung gilt. Dafür definiert Dijkstra Regeln für einen Prädikatentransformator, welcher ausgehend von einer Nachbedingung N und einem Programm P ein Prädikat V berechnet.

Dabei wird für das Programm P die schwächste Vorbedingung V gesucht, damit nach Ausführung des Programms P die Nachbedingung N gilt. Diese Beziehung wird ausgedrückt durch $wp(P, N) = V$.

Für die schwächste Vorbedingung gilt unter anderem das Hoare-Tripel $\{wp(P, N)\}P\{N\}$.

3.2 Die Spezifikationssprache ACSL

Für die Spezifikation von C-Programmen haben die Entwickler von Frama-C die formale Spezifikationssprache ACSL (ANSI/ISO C Specification Language) entworfen [12].

In ACSL werden C-Programme durch Vor- und Nachbedingungen beschrieben. Spezifikationen in ACSL werden in C-Kommentaren in den Quelltext geschrieben, wobei ein „@“ zu Beginn eines Kommentars bedeutet, dass dieser ein ACSL-Kommentar ist (siehe Beispiel in Listing 3.1).

In dieser Arbeit werden lediglich die Elemente von ACSL vorgestellt, die auch Anwendung in der Verifikation von `push_heap` und `make_heap` finden. Für einen tieferen Einblick werden in *ACSL By Example* zahlreiche andere Eigenschaften von ACSL an praktischen Beispielen demonstriert.

Funktionsverträge

Die Vor- und Nachbedingungen der Spezifikation einer C-Funktion formen einen ACSL-Funktionsvertrag. Zusätzlich enthält ein Funktionsvertrag auch Aussagen über die Schreibzugriffe während der Funktion.

Der C-Kommentar in Listing 3.1 zeigt die Struktur eines Funktionsvertrags. Darin werden Vorbedingungen mit dem Schlüsselwort requires, Schreibzugriffe mit assigns und Nachbedingungen mit ensures gekennzeichnet.

Die Funktion `sqr` gibt das Quadrat des eingegebenen Integer `x` zurück. Im nächsten Abschnitt wird ein passender Funktionsvertrag für diese Funktion gezeigt, in dem `preconditions`, `memory locations` und `postconditions` durch passende Eigenschaften ersetzt werden.

```
/*@
   requires preconditions
   assigns  memory locations
   ensures  postconditions
*/
int sqr(int x);
```

Listing 3.1: Beispiel Funktionsvertrag

Alle weiteren in dieser Arbeit verwendeten ACSL-Konstrukte werden vorgestellt, wenn sie das erste Mal Anwendung finden.

Listing 3.3 zeigt einen Funktionsvertrag für `sqr` und die Implementierung der Funktion. In den Nachbedingungen kommt `\result` vor, womit in ACSL-Kommentaren der Rückgabewert der Funktion ausgedrückt wird. Die Nachbedingungen sind mit dem Label `result` versehen. So ist es später einfacher, in größeren Funktionsverträgen über bestimmte Eigenschaften zu sprechen. Da die Funktion keine Nebeneffekte hat, werden diese mit `\nothing` spezifiziert.

```
/*@
   assigns  \nothing;
   ensures  result: \result == x * x;
*/
int sqr(int x) {
  return x*x;
}
```

Listing 3.2: Funktionsvertrag für die Funktion `sqr`

Mit dieser Spezifikation kann Frama-C mit dem Plug-in WP gestartet werden.

3.3 Frama-C WP

In dieser Arbeit wird die Plattform Frama-C mit dem Plug-in WP für die deduktive Verifikation verwendet. Das WP (*weakest precondition*) Plug-in implementiert den *weakest precondition* Ansatz aus Abschnitt 3.1.

Für einen Funktionsvertrag und eine Funktion wird vom Plug-in WP die schwächste Vorbedingung berechnet, damit die Nachbedingung des Funktionsvertrags gilt.

Für eine Verifikation des Funktionsvertrags muss diese schwächste Vorbedingung aus den Vorbedingungen des Funktionsvertrags gefolgert werden können. Für diesen Schritt werden vom Plug-in WP Beweisverpflichtungen generiert, die von Theorembeweisern nachgewiesen werden.

Abbildung 3.1 zeigt den Ablauf der Verifikation mit Frama-C, wie bis zu diesem Punkt besprochen.

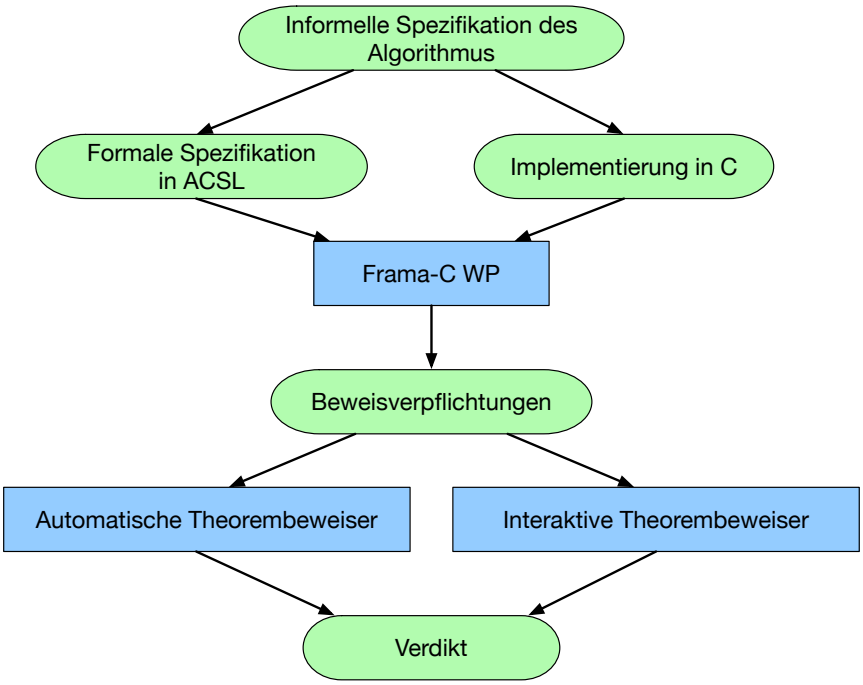

Abbildung 3.1: Verifikationsprozess mit Frama-C

In Kapitel 2 werden die Heap-Algorithmen informell spezifiziert und eine Implementierung vorgestellt. Mit diesen Spezifikationen findet im Kapitel 4, wie hier für die Funktion `sqr` geschehen, eine Spezifikation in ACSL statt. Aus der Spezifikation und der Implementierung werden Beweisverpflichtungen generiert, aus welchen mit Hilfe von Theorembeweisern das Verdikt der formalen Verifikation berechnet wird.

Der weitere Teil des Verifikationsablaufs für `sqr` wird auf den folgenden Seiten zu Ende geführt.

Für das Starten der Frama-C-Oberfläche wird der Befehl `frama-c-gui -wp -wp-rte [dateiname]` in der Konsole ausgeführt.

- frama-c-gui startet die grafische Oberfläche der Verifikationsplattform.
- -wp startet das Plug-in WP, welches die Beweisverpflichtungen aus den Vor- und Nachbedingungen generiert.
- -wp-rte ist eine Option für das Plug-in WP, die anzeigt, dass Quelltext-Annotationen in ACSL generiert werden sollen, um Laufzeitfehler, wie Überläufe, auszuschließen. Auch aus diesen Annotationen werden Beweisverpflichtungen generiert.

Der Screenshot 3.2 zeigt einen Ausschnitt aus der Frama-C-GUI, welcher den Quelltext, die Annotationen und das Verifikationsergebnis enthält.

Bei der Verarbeitung der C-Datei normalisiert Frama-C den Quelltext und die ACSL-Annotationen.

Zusätzlich zur Spezifikation aus Listing 3.2 zeigt der Screenshot zwei generierte Annotationen mit dem Label `rte`. Sie spezifizieren, dass es keinen Überlauf bei der Quadrierung von `x` gibt.

Integer in ACSL, wie hier `x`, sind mathematische Integer. Deshalb lassen sich auch Aussagen wie „der Wert ist größer als der maximale **`int`**-Wert“ treffen.

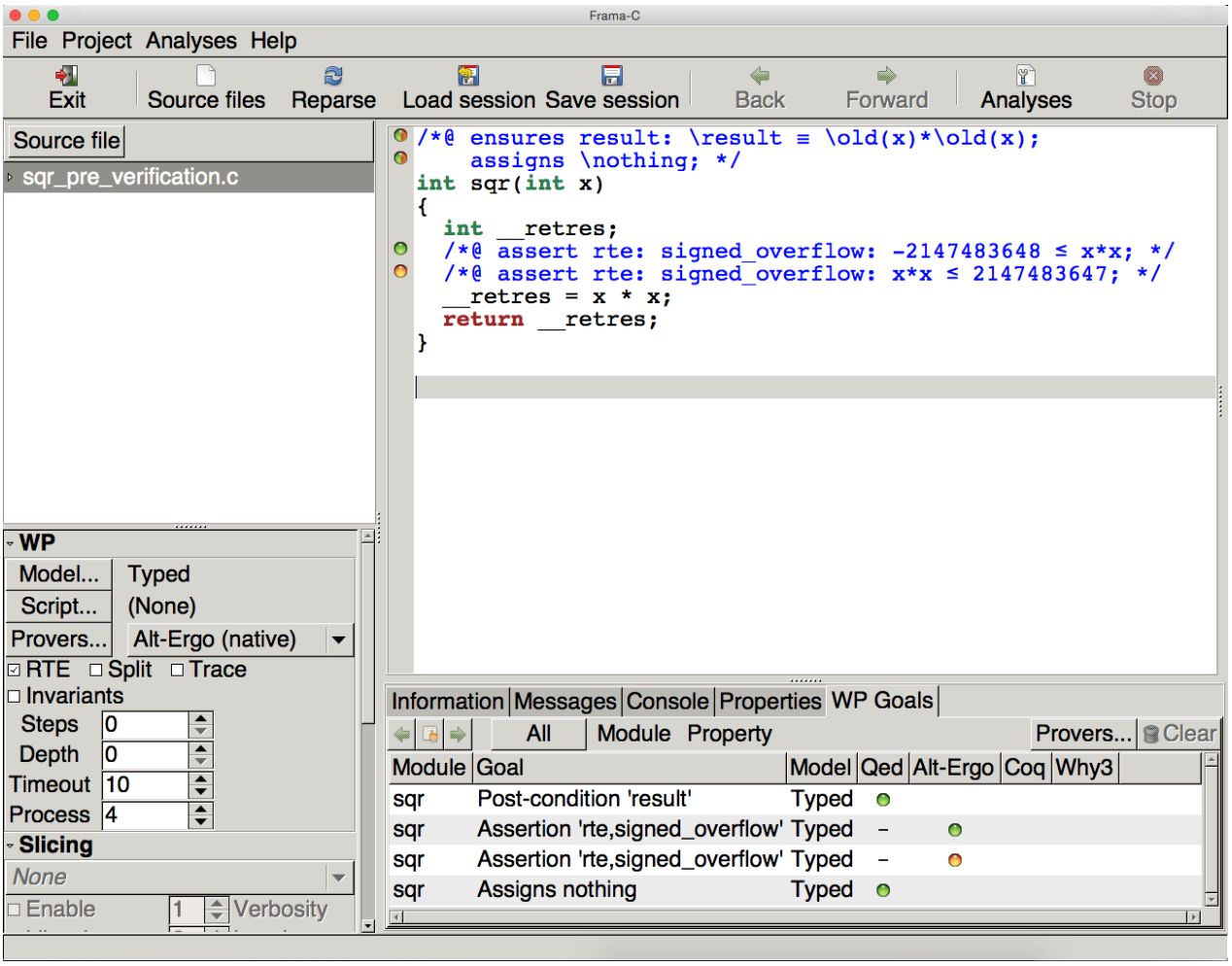

Abbildung 3.2: Normalisierte Form der Funktion `sqr` aus Listing 3.2 mit Verifikationsergebnis

Am linken Rand des Quelltextes wird das Verifikationsergebnis durch farbige Kreise angezeigt.

Ein grüner Kreis markiert eine bewiesene Annotation. Ein gelber Kreis markiert eine nicht bewiesene Annotation. Ein grün-oranger Kreis markiert eine Annotation, welche nur unter der Voraussetzung bewiesen wird, dass andere nicht bewiesene Annotationen wahr sind.

Nur eine der beiden generierten `rte` Annotationen wird bewiesen und die Nachbedingung `result` wird nur unter der Voraussetzung bewiesen, dass die unbewiesene `rte` Annotation wahr ist.

Es stellt sich heraus, dass `sqr` nur das spezifizierte Ergebnis zurückgibt, wenn bei der Quadrierung kein Überlauf stattfindet. Diese Eigenschaft muss als Vorbedingung in den Funktionsvertrag aufgenommen werden. Das führt zu der angepassten Spezifikation in Listing 3.3.

```
/*@
   requires limit: x * x <= INT_MAX;
   assigns  \nothing;
   ensures  result: \result == x * x;
*/
int sqr(int x) {
  return x*x;
}
```

Listing 3.3: Angepasster Funktionsvertrag für die Funktion `sqr`

Wird die Verifikation mit der angepassten Spezifikation gestartet, führt das zu dem in Screenshot 3.3 sichtbaren Ergebnis.

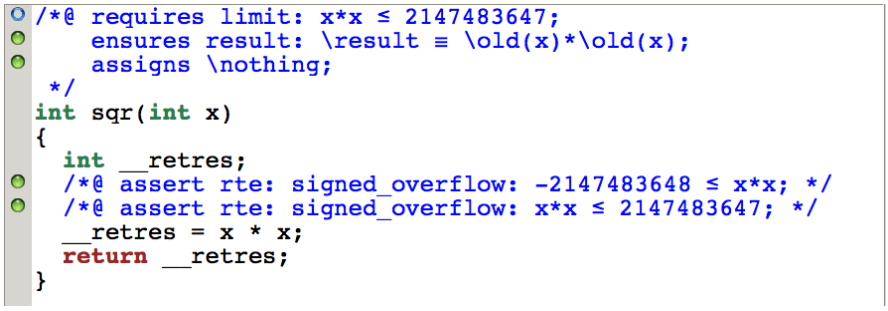

Abbildung 3.3: Normalisierte Form der Funktion `sqr` aus Listing 3.3 mit Verifikationsergebnis

Alle Kreise sind grün. Es konnten also alle von WP generierten Beweisverpflichtungen bewiesen werden.

3.4 Theorembeweiser

Damit die Beweisverpflichtungen nicht mühsam per Hand bewiesen werden müssen, werden Theorembeweiser eingesetzt. Ein Theorembeweiser ist ein Programm, welches die Beweissuche für ein Theorem übernimmt (automatische Theorembeweiser) oder unterstützt (interaktive Theorembeweiser).

Bei der Verifikation in dieser Arbeit werden die automatischen Theorembeweiser *Alt-Ergo*[13], *Z3* [14] und *CVC4* [15] verwendet

Bei der Verifikation von `push_heap` und `make_heap` wurde speziell darauf geachtet, dass möglichst viele Beweisverpflichtungen automatisch verifiziert werden können. Die Verifikation baut allerdings auf einigen früheren Spezifikationen aus *ACSL By Example* auf, die nicht im Rahmen dieser Arbeit entstanden sind und nicht alle automatisch bewiesen werden können. Das betrifft speziell die Beweise einiger Lemmas, die mit dem interaktiven Theorembeweiser Coq [16] bewiesen wurden. Die Verifikation mit Coq befindet sich allerdings außerhalb des Rahmens dieser Arbeit und wird in Kapitel 5 lediglich angeschnitten.

In ihrer Diplomarbeit hat sich Kim Völlinger intensiv mit dem Einsatz von Coq in Frama-C beschäftigt.

4 Formale Spezifikation ausgewählter Heap-Algorithmen

In diesem Kapitel wird die formale Spezifikation der Heap-Algorithmen `push_heap` und `make_heap` dokumentiert. Dazu gehören die formale Spezifikation der Funktionen in Form der in Kapitel 3 vorgestellten Funktionsverträge und die annotierte Implementierung der Funktionen.

Bei der Verifikation der Funktion `sqr` in Abschnitt 3.3 sind keine Annotationen im Quelltext notwendig, um den Funktionsvertrag vollständig zu verifizieren. Bei komplizierteren Implementierungen wie `push_heap` und `make_heap` ist eine automatische Verifikation ohne Annotationen nicht möglich. Es wird gezeigt, wie Annotationen im Quelltext die automatische Verifikation unterstützen können.

Zunächst wird, aufbauend auf der informellen Beschreibung von Heaps in Kapitel 2, eine mathematische Definition von Heaps und eine Spezifikation von Heaps in der ACSL-Syntax vorgestellt. Letztere ist für die Spezifikation der Funktionen notwendig.

Zwischen Elementen in einem Heap können die folgenden Beziehungen gelten:

$$\text{Linkes Kind des Index } i \qquad \text{child}_\text{l} : i \mapsto 2i + 1 \tag{4.1}$$

$$\text{Rechtes Kind des Index } i \qquad \text{child}_\text{r} : i \mapsto 2i + 2 \tag{4.2}$$

und

$$\text{Eltern-Index des Index } i \qquad \text{parent} : i \mapsto \frac{i-1}{2} \tag{4.3}$$

Ein Zusammenhang zwischen den Funktionen lässt sich mit den folgenden zwei Gleichungen herstellen. Die Gleichungen gelten für alle Integer i. Die ACSL-Integer-Division rundet stets auf Null (vgl. [12, §2.2.4]).

$$\text{parent}(\text{child}_\text{l}(i)) = i \tag{4.4}$$

$$\text{parent}(\text{child}_\text{r}(i)) = i \tag{4.5}$$

Listing 4.1 zeigt die Umsetzung dieser Funktionen und Gleichungen in ACSL.

```
/*@
   logic integer HeapParent(integer i) = (i - 1) / 2;

   logic integer HeapLeft(integer i) = 2*i + 1;

   logic integer HeapRight(integer i) = 2*i + 2;

   lemma HeapParentOfLeft:
     \forall integer i; 0 <= i ==> HeapParent(HeapLeft(i)) == i;

   lemma HeapParentOfRight:
     \forall integer i; 0 <= i ==> HeapParent(HeapRight(i)) == i;
*/
```

Listing 4.1: Die Beziehungen zwischen Elementen eines Heaps formal in ACSL beschrieben

Die logischen Funktionen `HeapParent`, `HeapLeft` und `HeapRight` operieren auf mathematischen Integer und geben einen mathematischen Integer zurück. Die Beziehungen zwischen den Funktionen `HeapParent` und `HeapLeft` und zwischen den Funktionen `HeapParent` und `HeapRight` werden jeweils durch die Lemmas `HeapParentOfLeft` und `HeapParentOfRight` ausgedrückt.

Lemmas werden von den automatischen Theorembeweisern verwendet und können so positive Auswirkungen auf die Beweisbarkeit haben. Damit der Beweis vollständig ist, bedarf jedes Lemma für sich aber auch einer Verifikation.

4.1 Das Prädikat `IsHeap`

Mit den vorangegangenen Definitionen aus Listing 4.1 lässt sich ein Prädikat formulieren, welches genau für solche Arrays wahr ist, die einen Heap nach der Definition in Kapitel 2 bilden. Listing 4.2 zeigt dieses Prädikat `IsHeap`. Mit Hilfe der `HeapParent` Funktion drückt es aus, dass jedes Element im Array kleiner-gleich zu seinem Elternelement ist.

```
/*@
  predicate IsHeap{L}(int* a, integer n) =
    \forall integer i; 0 < i < n  ==> a[i] <= a[HeapParent(i)];

  lemma HeapBounds:
    \forall integer a; 0 <= a  ==>  0 <= a/2 <= a;

  lemma HeapAndMaximum{L} :
    \forall int* a, integer n;
      0 <= n ==> IsHeap(a, n+1) ==> MaxElement(a, n+1, 0);
*/
```

Listing 4.2: ACSL-Definition eines Heaps mit dem `IsHeap` Prädikat

Das Listing zeigt auch die beiden Lemmas `HeapBounds` und `HeapAndMaximum`, mit denen für die Verifikation interessante Eigenschaften ausgedrückt werden.

Das Prädikat `MaxElement` ist genau dann wahr, wenn das Array-Element mit dem Index `max` eine obere Schranke für die Menge aller Array-Elemente von `0` bis `n` ist.

Auch dies ist eine Eigenschaft aus der Definition in Kapitel 2. Das Prädikat `MaxElement` wird in *ACSL By Example* definiert [1, §4.3].

4.2 Das Prädikat `MultisetUnchanged`

Das Einfügen (*pushen*) eines Elements in einen Heap erfordert in der Regel eine Umordnung der Elemente des Arrays (siehe Beispiel Abbildung 2.2 und 2.3). Deshalb bedarf es einer Möglichkeit, um auszudrücken, dass es sich bei den Schreiboperationen lediglich um eine Umordnung der vorhandenen Elemente handelt.

Sonst wäre zum Beispiel auch eine `push_heap` Implementierung zulässig, die jedem Element den gleichen Wert zuweist.

Die Multimenge der Array-Elemente nach dem Aufruf von `push_heap` muss also gleich der Multimenge der Array-Elemente aus dem Vorzustand sein.

Es wird das Prädikat `MultisetUnchanged` definiert, welches diese Eigenschaft, wie in Listing 4.3 zu sehen, formal beschreibt.

```
/*@
  predicate MultisetUnchanged{K,L}(int* a, integer n) =
     \forall int v; Count{L}(a, n, v) == Count{K}(a, n, v);
*/
```

Listing 4.3: Das Prädikat `MultisetUnchanged`

Das Prädikat ruft die logische Funktion `Count` auf, welche für ein Array und einen Wert die Anzahl der Vorkommnisse des Werts im Array zurückgibt. Ist diese Anzahl für alle Werte im Zustand `K` gleich der Anzahl im Zustand `L`, so ist die Multimenge unverändert.

4.3 ACSL-Funktionsvertrag von `make_heap`

Listing 4.4 zeigt die ACSL-Spezifikation von `make_heap`.

Der Funktionsvertrag enthält eine Nachbedingung, die das `MultisetUnchanged` Prädikat verwendet. Im Nachzustand der Funktion soll das Array mit der selben Multimenge von Integer gefüllt sein, wie im Vorzustand. Das Label `Old` bezeichnet hier den Vorzustand der Funktion und das Label `Here` den Nachzustand.

Die Nachbedingung `heap` spezifiziert, dass das Array im Nachzustand ein Heap der Größe n ist. Die Eigenschaft `max` ist eine zusätzliche Spezifikation, die direkt aus der Eigenschaft `heap` folgt. Der Zusammenhang ist durch das Lemma `HeapAndMaximum` in Listing 4.2 dargestellt.

```
/*@
   requires limit:  n < UINT_MAX;
   requires valid:  \valid(a + (0..n-1));

   assigns a[0..n-1];

   ensures heap:    IsHeap(a, n);
   ensures max:     n > 0 ==> MaxElement(a, n, 0);
   ensures reorder: MultisetUnchanged{Old,Here}(a, n);
*/
void make_heap(int* a, unsigned int n);
```

Listing 4.4: ACSL-Funktionsvertrag von `make_heap`

Der Funktionsvertrag enthält auch zwei Vorbedingungen, die den Eingabebereich einschränken, für den die Nachbedingungen die Funktion spezifizieren. Die angegebene Länge n muss ein gültiger Unsigned Integer sein (siehe Vorbedingung `limit`) und a muss ein gültiges Array der Länge n sein (siehe Vorbedingung `valid`).

4.4 ACSL-Funktionsvertrag von `push_heap`

Listing 4.5 zeigt die ACSL-Spezifikation von `push_heap`. Es werden die beiden in den Abschnitten 4.1 und 4.2 vorgestellten Prädikate `IsHeap` und `MultisetUnchanged` verwendet.

Die Eigenschaften `heap` spezifizieren, dass die Funktion einen Heap der Länge $(n-1)$ erwartet und sicher stellt, dass `a` nach Funktionsablauf ein Heap der Länge n ist.

Die Nachbedingung `reorder` sagt aus, dass vom Vor- zum Nachzustand eine Umordnung der Elemente des Arrays stattfindet.

Die assigns Klausel muss auf `a[0..n-1]` gesetzt sein, da in der Funktion in das gesamte Array geschrieben werden kann. Die Anforderungen `positive` und `valid` spezifizieren, wie gültige Eingaben für die Funktion aussehen.

Der Heap im Nachzustand muss mindestens die Größe `1` haben und `a` muss ein gültiges Array der Länge n sein, so dass die Funktion Zuweisungen auf die Array-Elemente ausführen kann.

```
/*@
   requires positive: n > 0;
   requires valid:    \valid(a + (0..n-1));
   requires heap:     IsHeap(a, n-1);

   assigns a[0..n-1];

   ensures heap:      IsHeap(a, n);
   ensures reorder:   MultisetUnchanged{Old,Here}(a, n);
*/
void push_heap(int* a, unsigned int n);
```

Listing 4.5: ACSL-Funktionsvertrag von `push_heap`

4.5 Annotierte Implementierung von `make_heap`

Der in Kapitel 3 gezeigte Funktionsvertrag von `sqr` lässt sich automatisch verifizieren. Für komplexere Vor- und Nachbedingungen und komplexere Implementierungen, wie `make_heap` ist die automatische Verifikation des Funktionsvertrags nicht so einfach möglich.

Die ACSL-Spezifikationssprache bietet unterschiedliche Möglichkeiten um Annotationen im Quelltext zu formulieren, welche die einzelnen Schritte der Funktion spezifizieren. Im Beispiel `sqr` wurden bereits einige Annotationen in Form von `rte` Zusicherungen generiert.

Wie für Vor- und Nachbedingungen des Funktionsvertrags, werden Beweisverpflichtungen aus den Annotationen generiert. Die Ergebnisse der Verifikation dieser werden bei der Verifikation von späteren Annotationen und den Nachbedingungen von den Theorembeweisern genutzt. So lassen sich auch komplexere Funktionen automatisch verifizieren.

Für die annotierte Implementierung von `make_heap` werden zwei Arten von Annotationen verwendet.

Zusicherungen

Eine Zusicherung ist die einfachste Art der Annotation. Sie formuliert eine Annahme, die zwischen zwei Anweisungen im Quelltext gelten soll. Es wird die Beweisverpflichtung generiert, dass die spezifizierte Eigenschaft an dieser Stelle der Implementierung gilt.

Um zum Beispiel zu zeigen, dass an einer bestimmten Stelle im Quelltext das Prädikat `IsHeap` für das Array `a` der Länge n gilt, wird die folgende Zusicherung eingefügt:

```
//@ assert heap: IsHeap(a, n);
```

Schleifen-Annotationen

Schleifen sind ein schwieriges Konstrukt für die formale Verifikation. Sie werden ähnlich wie die Funktionen mit einem Vertrag spezifiziert, welcher aus mehreren Komponenten besteht und vor die Schleife geschrieben wird.

- Analog zu der assigns Klausel für Funktionsverträge besitzen Schleifenspezifikationen eine loop assigns Klausel.
- In der loop variant wird ein Term formuliert, der von Schleifendurchlauf zu Schleifendurchlauf streng monoton fällt und stets positiv ist. Mit der loop variant wird die Terminierung der Schleife sichergestellt.
- Für die Spezifikation der funktionalen Eigenschaften einer Schleife werden Prädikate formuliert und als Schleifeninvariante hinzugefügt. Da eine Schleife nicht nur einen Vor- und Nachzustand besitzt, sondern auch Zustände zwischen den einzelnen Schleifendurchläufen, zeigt Abbildung 4.1, wann das Prädikat der Schleifeninvariante wahr sein muss, damit die Schleifeninvariante gilt.

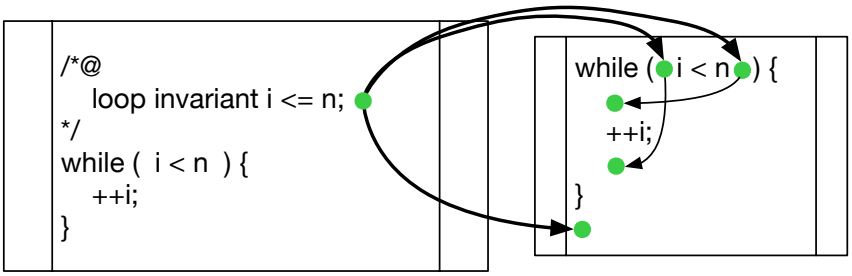

Abbildung 4.1: Der Geltungsbereich einer Schleifeninvariante

- Die Schleifeninvariante gilt vor jedem Test im Schleifenkopf und somit auch nach der Ausführung des Schleifenkörpers (bei einer For-Schleife inklusive Inkrementierung).
- Wenn der Test keine Seiteneffekte hat, gilt die Schleifeninvariante auch nach dem Test und somit auch vor dem Schleifenkörper und nach der Schleife.

Listing 4.6 zeigt die Annotierte Implementierung von `make_heap`. Vor der Schleife ist die Schleifen-Annotation zu sehen.

```
void make_heap(int* a, unsigned int n)
{
  if (n == 0) {
    return;
  }

  /*@
     loop invariant bounds:     2 <= i <= n+1;
     loop invariant heap:       IsHeap(a, i-1);
     loop invariant reorder:    MultisetUnchanged{Pre,Here}(a, i);
     loop invariant unchanged:  Unchanged{Pre,Here}(a, i, n);
     loop assigns   i, a[0..n-1];
     loop   variant n - i;
  */
  for (unsigned int i = 2; i <= n; i++) {
    push_heap(a, i);
  }

  //@ assert  heap: IsHeap(a, n);
}
```

Listing 4.6: Annotierte Implementation von `make_heap`

Die Schleifeninvariante `bounds` grenzt den Wertebereich ein, den `i` innerhalb der Schleife annimmt und erklärt sich ohne Weiteres mit einem Blick auf den Schleifenkopf.

Die Schleifeninvariante `2 <= i <= n+1` gilt auch wenn die Abfrage `i <= n` im Schleifenkopf fehlschlägt.

Die Schleifeninvariante `heap` nutzt das Prädikat `IsHeap`. Als zweiter Parameter wird `(i-1)` übergeben. So wächst die Länge des Heaps mit jedem Schleifendurchlauf, bis `IsHeap(a, i-1)` gleich der Nachbedingung `heap` des Funktionsvertrags ist (`IsHeap(a, n)`).

Auch für die Schleifeninvariante `reorder` wächst das betrachtete Array mit jedem Schleifendurchlauf um ein Element.

Die Schleifeninvarianten `heap` und `reorder` sind relativ einfach automatisch beweisbar, da sie Nachbedingungen des Funktionsvertrags von `push_heap` sind.

Da die Invariante `reorder` nur einen Teil des Arrays spezifiziert und das `loop assigns` aber spezifiziert, dass das gesamte Array geschrieben werden kann, muss zusätzlich sichergestellt werden, dass die Array-Elemente mit einem Index größer-gleich `i` auch eine Umordnung darstellen.

Da die `assigns` Klausel der Funktion `push_heap` sicherstellt, dass nur die Array-Elemente von 0 bis $(i-1)$ verändert werden, ändern sich die Werte die weiter hinten im Array stehen nicht und es kann die Schleifeninvariante `unchanged` definiert werden.

Das Lemma `MultisetUnchangedAndUnchanged` in Listing 4.7 kann benutzt werden, um aus den beiden Schleifeninvarianten `reorder` und `unchanged` zu folgern, dass das gesamte Array eine Umordnung ist.

```
/*@
   lemma
     MultisetUnchangedAndUnchanged{L1,L2}:
       \forall int *a, integer k,  n;
         (0 <= k <= n && MultisetUnchanged{L1,L2}(a, k) && Unchanged{L1,L2}(a, k, n))
         ==> MultisetUnchanged{L1,L2}(a, n);
*/
```

Listing 4.7: Das Lemma `MultisetUnchangedAndUnchanged`

Die Zusicherung `heap` am Funktionsende erscheint zunächst überflüssig, da sie den gleichen Zustand wie die Nachbedingungen beschreibt, in denen diese Eigenschaft bereits vorkommt.

Ohne diese Zusicherung kann allerdings die Nachbedingung `max` nicht bewiesen werden. Wie in Listing 4.2 mit dem Lemma `HeapAndMaximum` gezeigt wird, folgt die Eigenschaft `max` direkt aus der Eigenschaft `heap`. Da die Eigenschaften `heap` und `max` aber beide im selben Block (den Nachbedingungen des Funktionsvertrags) vorkommen, kann die Eigenschaft `max` nicht aus der Eigenschaft `heap` gefolgert werden.

Dies ist die einzige Eigenschaft des Funktionsvertrags, für die der Beweisassistent Coq eingesetzt werden muss. Aus der Zusicherung `heap` kann die Eigenschaft `max` leicht gefolgert werden. Für den Beweis wird Coq die Anweisung gegeben, das Lemma `HeapAndMaximum` anzuwenden.

4.6 Annotierte Implementierung von `push_heap`

Ebenso wie bei `make_heap` macht den größten Teil der Implementierung von `push_heap` eine Schleife aus. Da die korrekte Schleifen-Annotation für diese Schleife nicht so einfach zu formulieren ist, wie für `make_heap`, wird die Schleife gesondert in Verbindung mit den Nachbedingungen betrachtet.

Die zwei komplexen Nachbedingungen des Funktionsvertrags von `push_heap` sind `heap` und `reorder`. Diese Eigenschaften müssen über den Verlauf der Funktion im Fall von `heap` aufgebaut und im Fall von `reorder` erhalten bleiben.

Das folgende eingefärbte Listing wiederholt die nicht annotierte Implementierung von `push_heap`, wobei der Schleifenkörper ausgelassen wird.

```
void push_heap(int* a, unsigned int n)
{
  if (n == 1) {
    return;
  }
  const int val = a[n - 1];
  unsigned int hole = n-1;
  unsigned int parent = (hole-1) / 2;
  if (val > a[parent]) {
    a[hole] = a[parent];
    hole = parent;
  }
  if (hole == n-1) {
    return;
  }
  unsigned int parent = (hole-1) / 2;
  if (val > a[parent]) {
    a[hole] = a[parent];
    hole = parent;
  }
  if (hole == n-1) {
    return;
  }
  while (hole > 0) {
    ...
  }
  a[hole] = val;
}
```

```
void push_heap(int* a, unsigned int n)
{
  if (n == 1) {
    return;
  }
  const int val = a[n - 1];
  unsigned int hole = n-1;
  while (hole > 0) {
    const unsigned int par = (hole-1) / 2;
    if (a[par] < val) {
      a[hole] = a[par];
      hole = par;
    } else {
      break;
    }
  }
  a[hole] = val;
}
```

Die Implementierung ist mit unterschiedlichen Farben in 3 Teile aufgeteilt, wobei die Schleife einen eigenen Teil davon ausmacht. Der Teil vor der Schleife wird ab jetzt als *Prolog*, die Schleife selbst als *Hauptteil* und der Teil nach der Schleife als *Epilog* bezeichnet.

Im Prolog wird kein Element des Arrays verändert, und die Gültigkeit der Eigenschaften `heap` und `reorder` verändert sich zum Vorzustand von `push_heap` somit nicht.

Im Hauptteil und im Epilog können Elementen des Arrays neue Werte zugewiesen werden. In beiden Teilen besteht das Potential, dass die Gültigkeit der Eigenschaften beeinflusst wird. Durch die Nachbedingungen ist gegeben, dass nach dem Epilog sowohl die Eigenschaft `heap`, als auch die Eigenschaft `reorder` gelten muss. Was dazwischen passiert ist nicht spezifiziert.

Die Tabelle 4.1 zeigt die Gültigkeit der Eigenschaften zu den einzelnen Funktionsabschnitten.

Funktionsabschnitt	Eigenschaft	
	heap	reorder
Vorzustand	×	✓
Prolog	×→×	✓→✓
Hauptteil	×→?	✓→?
Epilog	?→✓	?→✓
Nachzustand	✓	✓

Tabelle 4.1: Die Gültigkeit der Nachbedingungen während des Funktionsablaufs von `push_heap`

Das Symbol „×“ bedeutet, dass die Eigenschaft nicht gilt, „✓“, dass die Eigenschaft gilt und „?“, dass nicht gewiss ist, ob die Eigenschaft gilt.

Beispielhafter Durchlauf von `push_heap` im Detail

Mit der aktuellen Spezifikation ist nicht eindeutig, welche Eigenschaften in der Schleife gelten und ob die geltenden Eigenschaften in jedem Schleifendurchlauf gleich sind. Für die Formulierung von Schleifeninvarianten ist es aber wichtig konstante Eigenschaften zu finden.

Vielleicht lassen sich mit einem beispielhaften Durchlauf gemeinsame Eigenschaften für die einzelnen Schleifendurchläufe finden.

Abbildung 4.2 zeigt den Beispiel-Aufruf von `push_heap` aus Abbildung 2.2 Schritt für Schritt.

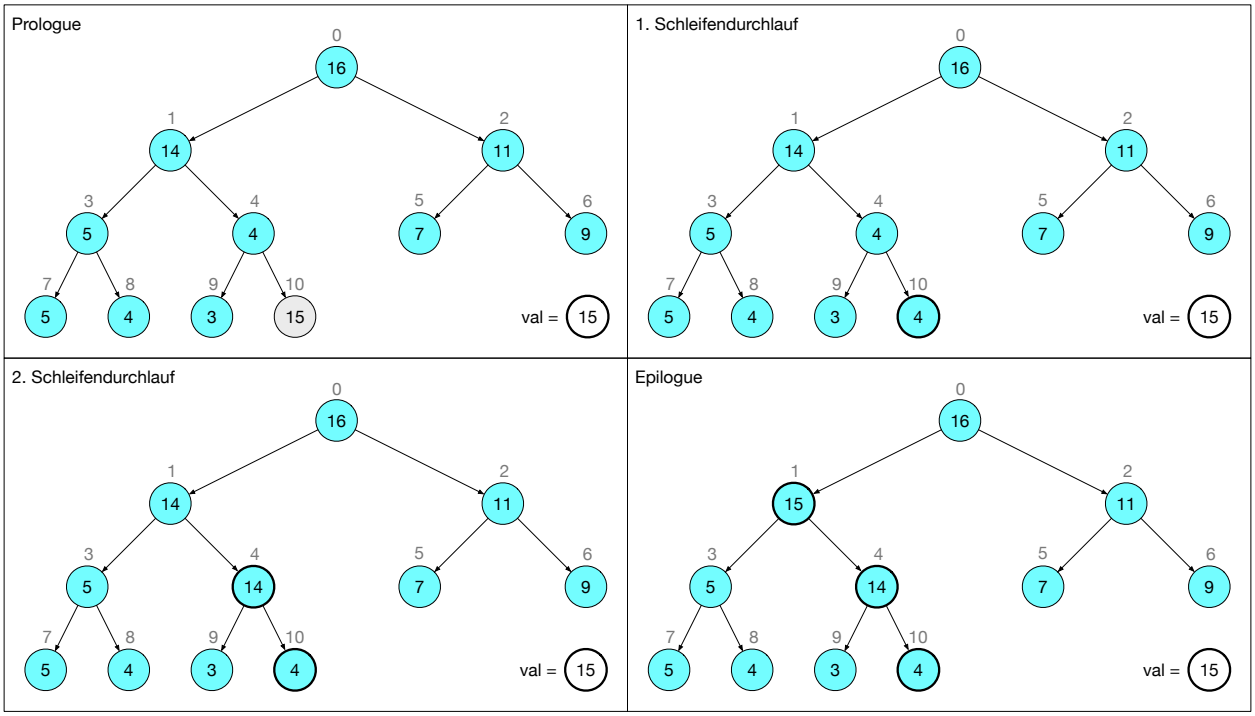

Abbildung 4.2: Darstellung der einzelnen Schritte für den Aufruf von `push_heap` aus Abbildung 2.2

Die Abbildung zeigt den Funktionsaufruf mit der Unterteilung in die 3 Funktionsabschnitte, wobei der Hauptteil in die beiden Schleifendurchläufe unterteilt ist, um Gemeinsamkeiten und Unterschiede während der einzelnen Schleifendurchläufe zu erkennen.

- Im Prolog wird der Wert des neuen Elements in der Variable `val` zwischengespeichert. Alle Elemente bis auf das letzte bilden einen Heap. Die Eigenschaft `heap` gilt also nicht. Da das Array noch unverändert ist, gilt die Eigenschaft `reorder` weiterhin.
- Im 1. Schleifendurchlauf wird dem Knoten mit dem Index 10 der Wert 4 zugewiesen. Bei dieser Zuweisung geht die Eigenschaft `reorder` verloren. Das Array enthält den Wert 4 einmal mehr als zu Funktionsbeginn und der Wert 15 kommt einmal weniger im Array vor als zu Funktionsbeginn. Dafür sind nun alle Knoten blau gefärbt und bilden gemeinsam einen Heap. Die Eigenschaft `heap` gilt, die Eigenschaft `reorder` aber nicht mehr.
- Im 2. Schleifendurchlauf wird dem Knoten mit dem Index 4 der Wert 14 zugewiesen. Der Wert 14 kommt einmal mehr im Array vor als zu Funktionsbeginn. Dafür kommt der Wert 4 wieder genauso oft im Array vor wie zu Funktionsbeginn. Der Wert 15 kommt weiterhin einmal weniger vor. Die Eigenschaft `reorder` bleibt nicht gegeben und die Eigenschaft `heap` bleibt erhalten.
- Im Epilog wird der Wert 15 an der richtigen Stelle im Array eingefügt. Die Eigenschaft `reorder` wird wieder hergestellt. Die Eigenschaft `heap` bleibt erhalten.

Nach der Betrachtung der Funktionsweise am dargestellten Beispiel wird klar, dass die Eigenschaft `heap` immer nach dem 1. Schleifendurchlauf gilt und im weiteren Funktionsablauf nicht mehr verloren gehen kann.

Auch im 1. Schleifendurchlauf geht die Eigenschaft `reorder` verloren, indem der Wert `a[hole]` einmal mehr vorkommt als zu Funktionsbeginn und der Wert der Variable `val` einmal weniger vorkommt.

In den weiteren Schleifendurchläufen bleibt gegeben, dass der Wert der Variable `val` einmal weniger im Array vorkommt, als zu Funktionsbeginn.

Der Wert, der einmal mehr vorkommt, kann sich von Schleifendurchlauf zu Schleifendurchlauf verändern, steht aber zu Beginn eines jeden Schleifendurchlaufs in `a[hole]`.

Daraus ergibt sich auch, dass die Eigenschaft `reorder` für den Rest der Schleife verletzt bleibt und erst im Epilog wieder hergestellt wird.

Umstrukturierung

Bei der Beobachtung sticht der 1. Schleifendurchlauf hervor. Für alle weiteren Schleifendurchläufe lassen sich gemeinsame Eigenschaften finden.

Wenn der 1. Schleifendurchlauf abgerollt wird, der Schleifenkörper also schon einmal vor der Schleife stattfindet, führt das zu den folgenden Funktionsabschnitten und der folgenden Gültigkeit der Eigenschaften.

Funktionsabschnitt	Eigenschaft	
	heap	reorder
Vorzustand	×	✓
Prolog	×→✓	✓→×
Hauptteil	✓→✓	×→×
Epilog	✓	×→✓
Nachzustand	✓	✓

Tabelle 4.2: Die Gültigkeit der Nachbedingungen während des Funktionsablaufs von `push_heap` in der umstrukturierten Implementierung

Für den Rest der Arbeit wird nur noch die umstrukturierte Implementierung von `push_heap` mit der abgerollten Schleife betrachtet. Die umstrukturierte Implementierung und die entsprechende Annotation wird in den folgenden Abschnitten für jeden Funktionsabschnitt einzeln betrachtet:

- *Prolog* (siehe Section 4.6.1)
- *Hauptteil* (siehe Section 4.6.2)
- *Epilog* (siehe Section 4.6.3)

Zu jedem Abschnitt kann zur Visualisierung auf das Beispiel die Abbildung 4.2 zurückgeblickt werden. Dabei ist zu beachten, dass der Nachzustand des Prolog nun durch den Baum „1. Schleifendurchlauf“ visualisiert wird.

4.6.1 Der Prolog

Listing 4.8 zeigt die annotierte Implementierung des Prolog. Wie im letzten Abschnitt besprochen, beinhaltet dieser Teil nun den 1. Schleifendurchlauf. Das Abrollen der Schleife erlaubt es, die kritische Iteration nach der das `MultisetUnchanged` Prädikat nicht mehr gilt, zu isolieren.

Diese Iteration ist zwangsläufig die erste Iteration. Wäre dem nicht so, dann wären die Werte in `a[hole]` und `a[parent]` gleich, es würde unter anderem gelten, dass `a[hole] <= a[parent]` und somit wäre die Heap Eigenschaft erfüllt.

```
void push_heap(int* a, unsigned int n)
{
  // start of prologue

  if (n == 1) {
    return;
  }

  const int val = a[n - 1];
  unsigned int hole = n-1;

  unsigned int parent = (hole-1) / 2;

  if (val > a[parent]) {
    /*@
       requires val:      a[hole] == val;
       assigns  a[hole];
       ensures  newhole:  Unchanged{Old, Here}(a, 0, hole);
       ensures  newhole:  Unchanged{Old, Here}(a, hole+1, n);
       ensures  val:      a[hole] == a[parent];
    */
    a[hole] = a[parent];
    hole = parent;
  }
  //@ assert heap:  IsHeap(a, n);

  if (hole == n-1) {
    return;
  }

  // end of prologue
```

Listing 4.8: Teil 1 der annotierten Implementierung von `push_heap`, der Prolog

In der Annotation des Prolog wird eine neue Möglichkeit der Quelltext-Annotation vorgestellt, der Anweisungsvertrag. Die Struktur eines Anweisungsvertrags ist aufgebaut wie ein Funktionsvertrag und muss daher nicht weiter erläutert werden. Ein Anweisungsvertrag spezifiziert einen Quelltext-Abschnitt, in diesem Fall eine einzelne Anweisung. Im Hauptteil ist auch ein Beispiel zu sehen, wie mit einem Anweisungsvertrag ein ganzer Quelltext Block spezifiziert werden kann.

Der Anweisungsvertrag in diesem Teil spezifiziert die Veränderungen im Array durch die Zuweisung. Das dafür verwendete Prädikat `Unchanged` ist genau dann wahr, wenn sich von einem Zustand in den nächsten kein Wert im Array verändert [1, §6.1].

Hier wird es für die Array-Abschnitte vor und nach dem Index `hole` genutzt.

Die Nachbedingungen dieses Anweisungsvertrags helfen bei der Verifikation der ersten Annotationen im Hauptteil.

Die Zusicherung `heap` zeigt an, dass nach diesem Teil der Implementierung die Eigenschaft `heap` gilt.

Prädikate für die `MultisetUnchanged` Störung

Im Abschnitt über die Umstrukturierung von `push_heap` wurden gemeinsame Eigenschaften für die Schleifendurchläufe nach dem 1. Schleifendurchlauf besprochen. Bisher wurde aber nur das Prädikat `MultisetUnchanged` vorgestellt, das genutzt wird, um zu spezifizieren, dass sich die Multimenge von einem Zustand in den anderen nicht ändert. Nun bedarf es Prädikaten um genau zu spezifizieren, wie sich die Multimenge von einem Zustand in den anderen ändert.

Bezüglich der Störung der Eigenschaft `reorder` kann der Zustand nach dem Prolog mit den Prädikaten in Listing 4.9 spezifiziert werden.

```
/*@
  predicate
    MultisetAdd{K,L}(int* a, integer n, int val) =
      Count{L}(a, n, val) == Count{K}(a, n, val) + 1;

  predicate
    MultisetMinus{K,L}(int* a, integer n, int val) =
      Count{L}(a, n, val) == Count{K}(a, n, val) - 1;

  predicate
    MultisetUnchangedExcept{K,L}(int* a, integer n, int val1, int val2) =
      \forall int i;
        i != val1 ==>
        i != val2 ==>
        Count{L}(a, n, i) == Count{K}(a, n, i);
*/
```

Listing 4.9: Prädikate zum Spezifizieren von Störungen von `MultisetUnchanged`

Die Prädikate bauen wie `MultisetUnchanged` auf der logischen Funktion `Count` auf. Mit den Prädikaten `MultisetAdd` und `MultisetMinus` kann spezifiziert werden, dass ein Wert einmal mehr beziehungsweise einmal weniger im Array vorkommt als im Vorzustand.

Nach dem Prolog tritt der Fall ein, dass das Prädikat `MultisetUnchanged` durch zwei Werte gestört ist. Für alle anderen Werte der Multimenge bleibt die Anzahl gleich. Das Prädikat `MultisetUnchangedExcept` beschreibt genau diese Eigenschaft. Es werden die beiden Werte als Parameter übergeben, deren Anzahl sich verändert hat. Für die Multimenge aus den anderen Werten gilt `MultisetUnchanged`.

Am Ende des Prolog gelten anstatt

```
  assert MultisetUnchanged{Pre,Here}(a, n);
```

die folgenden drei Prädikate

```
  assert MultisetAdd{Pre,Here}(a, n, a[hole]);
  assert MultisetMinus{Pre,Here}(a, n, val);
  assert MultisetUnchangedExcept{Pre,Here}(a, n, a[hole], val);
```

Für die Verifikation ist es nicht nötig diese am Ende des Prolog explizit hinzuschreiben, sie kommen aber als Schleifeninvarianten in den Annotationen des Hauptteil von `push_heap` vor (siehe Listing 4.10).

4.6.2 Der Hauptteil

Listing 4.10 zeigt die Annotierte Implementierung des Hauptteil, in dessen Schleifeninvarianten die im letzten Abschnitt vorgestellten Prädikate benutzt werden. Auch die Eigenschaft `heap` formt eine Schleifeninvariante.

```
// start of main act

/*@
   loop invariant bound:    0 <= hole < n-1;
   loop invariant heap:     IsHeap(a, n);
   loop invariant less:     a[hole] < val;
   loop invariant reorder:  MultisetMinus{Pre,Here}(a, n, val);
   loop invariant reorder:  MultisetAdd{Pre,Here}(a, n, a[hole]);
   loop invariant reorder:  MultisetUnchangedExcept{Pre,Here}(a, n, val, a[hole]);
   loop assigns   hole, a[0..n-1];
   loop variant   hole;
*/
while (hole > 0) {
  const unsigned int par = (hole-1) / 2;

  /*@
    requires heap:        IsHeap(a, n);
    assigns  hole, a[hole];
    ensures  heap:        IsHeap(a, n);
    ensures  reorder:     MultisetMinus{Pre,Here}(a, n, val);
    ensures  reorder:     MultisetAdd{Pre,Here}(a, n, a[hole]);
    ensures  reorder:     MultisetUnchangedExcept{Pre,Here}(a, n, val, a[par]);
    ensures  less:        a[hole] < val;
    ensures  decreasing:  hole < \old(hole);
  */
  if (a[par] < val) {
    /*@
       requires heap:       IsHeap(a, n);
       requires heap:       par == HeapParent(hole);
       requires heap:       HeapParent(hole) < hole < n-1;
       assigns  a[hole];
       ensures  heap:       a[hole] == a[HeapParent(hole)];
       ensures  unchanged:  Unchanged{Old,Here}(a, 0, hole);
       ensures  unchanged:  Unchanged{Old,Here}(a, hole+1, n);
       ensures  heap:       IsHeap(a, n);
    */
    a[hole] = a[par];
    /*@
       assigns hole;
       ensures bound:    hole < \old(hole);
       ensures less:     a[hole] < val;
       ensures reorder:  MultisetUnchangedExcept{Pre,Here}(a, n, val, a[hole]);
    */
    hole = par;
    //@ assert reorder:  MultisetUnchangedExcept{Pre,Here}(a, n, val, a[hole]);
  } else {
    break;
  }
  //@ assert heap:  IsHeap(a, n);
}

// end of main act
```

Listing 4.10: Teil 2 der annotierten Implementierung von `push_heap`, der Hauptteil

Bei der Formulierung der Schleifeninvarianten spielt der Iterator `hole` eine wichtige Rolle. Vor jedem Schleifendurchlauf steht im Array am Index `hole` der Wert, der im Schleifendurchlauf zuvor, beziehungsweise für den 1. Schleifendurchlauf im Prolog, einem der Kinderknoten von `hole` zugewiesen wurde. In `a[hole]` steht also der Wert der einmal mehr im Array vorkommt als zu Funktionsbeginn.

Der Schleifenkörper ist mit drei Anweisungsverträgen und zwei Zusicherungen annotiert. All diese Annotationen helfen bei der Verifikation der Schleifeninvarianten.

Die Eigenschaften aus den `heap` und `reorder` Schleifeninvarianten kommen auch im Anweisungsvertrag der Fallunterscheidung vor. Mit den zu verifizierenden Eigenschaften wird der Schleifenkörper bis hin zur Ebene der einzelnen Anweisungen annotiert. So wird der Umfang für die Verifikation der einzelnen Schritte möglichst gering gehalten und eine automatische Verifikation ermöglicht.

Das Finden der richtigen Annotationen ist ein mühsamer und anfangs oft überwältigender Prozess. Bei der Annotierung der Funktion `push_heap` wurde deshalb ein Flussdiagramm (siehe Abbildung 4.3) entworfen, welches die einzelnen Schritte, die zur Verifikation von `push_heap` geführt haben, veranschaulicht.

In Section 4.7 wird das Flussdiagramm erklärt und zur Veranschaulichung immer wieder auf die Annotationen aus dem Hauptteil zurückgegriffen.

4.6.3 Der Epilog

Der letzte Teil der Implementierung ist der Epilog. Listing 4.11 zeigt die annotierte Implementierung dieses Abschnitts.

Der Wert, der im Prolog aus dem Array entfernt wurde, wird im Epilog wieder in das Array eingefügt. Dazu wird ein Anweisungsvertrag formuliert, der genau die Veränderungen des Array Elements, dem der neue Wert zugewiesen wird, spezifiziert.

Der Anweisungsvertrag sichert auch zu, dass sich alle anderen Werte im Array nicht verändern. Zusammen mit den Nachbedingungen des Hauptteil lassen sich die beiden Annotationen `heap` und `reorder` ableiten.

```
  // start of epilogue

  /*@
     requires val:  a[hole] != val;
     assigns  a[hole];
     ensures  val:  a[hole] == val;
     ensures  val:  Unchanged{Old,Here}(a, 0, hole);
     ensures  val:  Unchanged{Old,Here}(a, hole+1, n);
  */
  a[hole] = val;
  //@ assert heap:     IsHeap(a, n);
  //@ assert reorder:  MultisetUnchanged{Pre,Here}(a, n);

  // end of epilogue
}
```

Listing 4.11: Teil 3 der annotierten Implementierung von `push_heap`, der Epilog

Die beiden Eigenschaften `heap` und `reorder` müssen am Ende der Funktion als Zusicherungen eingefügt werden. Obwohl der Zustand gleich dem Zustand ist der in den Nachbedingungen beschrieben wird, können die Eigenschaften `heap` und `reorder` hier bewiesen werden, nicht aber im Funktionsvertrag.

4.7 Methodik zum Finden der richtigen Quelltext-Annotationen

Für die automatische Verifikation von `push_heap` wurde im letzten Abschnitt eine reichlich annotierte Implementierung vorgestellt. Die richtigen Annotationen für eine solche Implementierung zu finden ist alles andere als trivial. Es wird eine gewisse Vertrautheit mit speziellen Eigenschaften der Verifikationsumgebung und der Beweiser vorausgesetzt.

Um diese Hürde für die automatische Verifikation weiterer Funktionen zu verkleinern, wurde die Suche nach den passenden Annotationen in Abbildung 4.3 dokumentiert.

Die Abbildung zeigt ein Flussdiagramm, welches primär als Hilfestellung für die Annotation von Schleifenkörpern dient, da diese auch mit aussagekräftige Schleifeninvarianten oft problematisch ist.

Schleifen stellen einen besonderen Fall dar, da mit den Schleifeninvarianten Eigenschaften existieren, die vor jedem Schleifendurchlauf gelten müssen. Für Funktionen ist das meist anders. Der Funktionsvertrag von `push_heap` enthält Eigenschaften, die als Vor- und Nachbedingungen auftauchen (`reorder`) und Eigenschaften, die sich von den Vor- zu den Nachbedingungen verändern (`heap`). Wenn diese Tatsache beachtet wird, können große Teile des Flussdiagramms auch für die Annotation ganzer Funktionen genutzt werden.

Für einen Schritt in dem neue Annotationen entstehen, enthält das Diagramm einen Quelltext-Kasten mit abstrakten Beispiel-Annotationen. Zwischen den einzelnen Schritten wird regelmäßig die automatische Verifikation gestartet, um die direkten Auswirkungen von einzelnen Zusicherungen und Lemmas auf das Verifikationsergebnis beobachten zu können.

Am Beispiel des Main Acts von `push_heap`

Für die Schleifeninvarianten des Main Act wird bewiesen, dass sie vor dem ersten Schleifendurchlauf gelten. Dass dies der Fall ist, wurde schon zum Ende des Prologue Abschnitts 4.6.1 besprochen. Die Schleifeninvarianten können aber nicht alle vollständig bewiesen werden.

Um herauszufinden, welche Anweisungen in der Schleife eine Herausforderung für die automatische Verifikation der Eigenschaften darstellen, wurden die Schleifeninvarianten als Zusicherungen zwischen die einzelnen Operationen in der Schleife geschrieben. Nach diesem Schritt ist genau zu sehen für welche Zuweisungen die Nachbedingungen genauer spezifiziert werden müssen.

Von diesen Zusicherungen bleiben in der finalen Version nur noch die Zusicherungen `heap` und `reorder` am Schleifenende übrig. Alle weiteren Zusicherungen wurden in den nächsten Schritten zu Anweisungsverträgen umgeformt oder entfernt.

Für jede Anweisung, die eine Zuweisung im Array tätigt, kommt die Eigenschaft `heap` im Anweisungsvertrag als Vor- und Nachbedingung vor. Diese Vor- und Nachbedingungen haben eine ähnliche Aussage wie eine Zusicherung vor und nach der Anweisung.

Um die automatische Verifikation auch über Anweisungen hinweg zu ermöglichen, nach denen bisher eine Eigenschaft nicht verifiziert werden konnte, muss untersucht werden, ob die Eigenschaft noch genau so gilt. Wenn die zu Grunde liegende Logik komplexer ist und von mehreren Eigenschaften abhängt, ist die Formulierung eines knappen Lemmas oft ein effektiver Schritt.

Vor allem für die `reorder` Eigenschaften halfen Lemmas die inneren Anweisungsverträge so zu gestalten, dass eine automatische Verifikation möglich ist. Denn wenn ein Lemma nicht direkt zu einer Verbesserung des Verifikationsergebnisses führt, können die einzelnen Bedingungen für das Lemma als Nachbedingungen eines Anweisungsvertrags formuliert werden. Durch diesen Schritt ist unter anderem zu sehen, ob überhaupt alle Bedingungen für das Lemma erfüllt werden.

Es sollten so viele Annotationen wie möglich wieder entfernt oder zusammengefasst werden, damit die Struktur der Implementierung weiterhin erkenntlich bleibt.

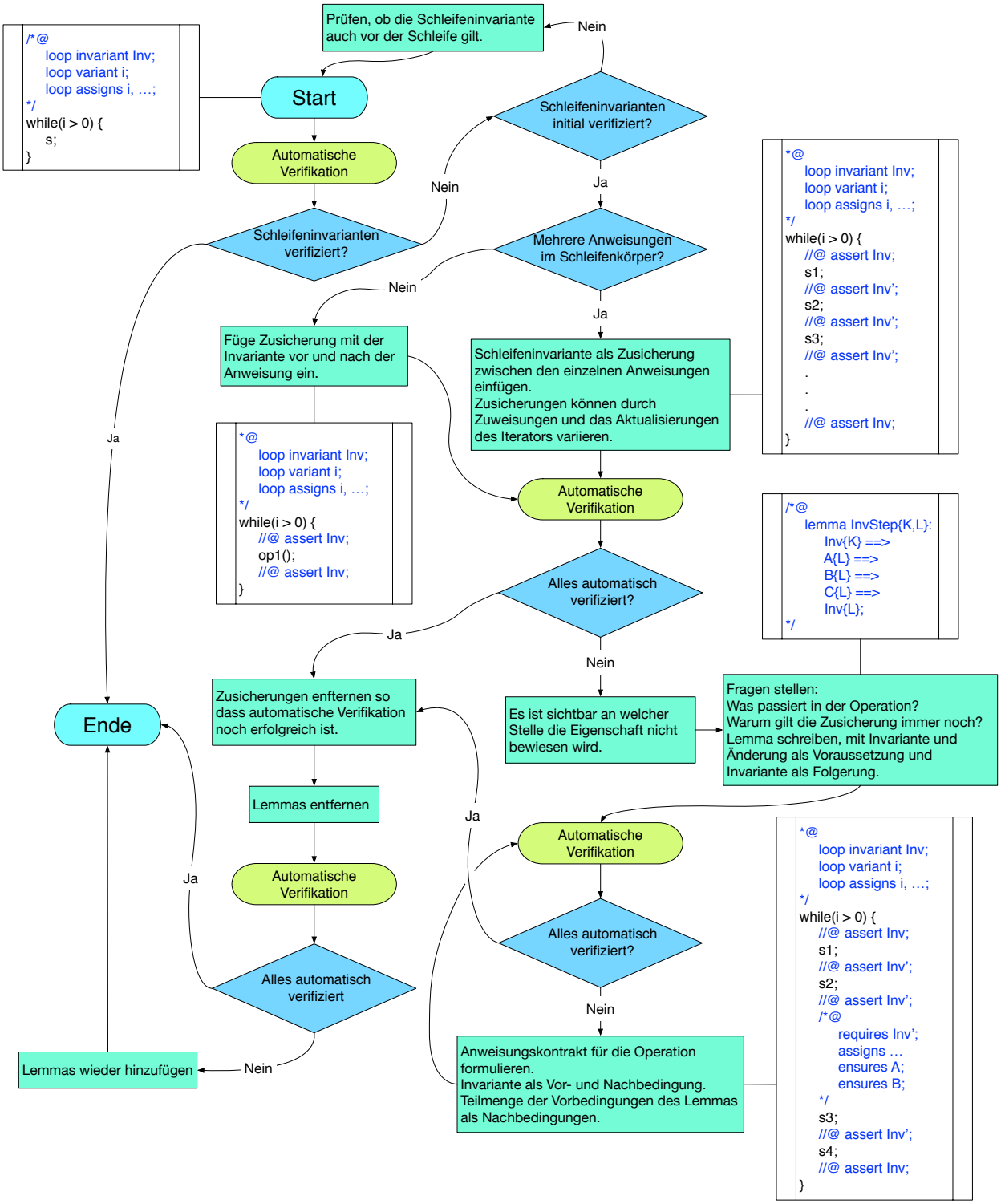

Abbildung 4.3: Vorgehensweise zum finden der richtigen Annotationen in einer Schleife

5 Ergebnisse der formalen Verifikation mit Frama-C

In diesem Kapitel wird das Ergebnis der formalen Verifikation vorgestellt. Zuerst werden die Spezifikationen des Systems genannt, auf dem die Verifikation stattgefunden hat. Anschließend werden die Ergebnisse der Verifikation mit den einzelnen Theorembeweisern aufgelistet.

Die Verifikation in dieser Arbeit fand auf einem Xubuntu 15.10 System statt, welches mit der Software Virtual Box 5.0.20. auf einem Mac OS X El Capitan System lief.

Frama-C ist auf dem System in der Version Aluminium-20160501 installiert und wird mit den folgenden Optionen gestartet:

```
-pp-annot -no-unicode -wp -wp-rte -wp-driver ../../external.driver
-wp-script '../../wp0.script'  -wp-model Typed+ref -wp-timeout 30
-wp-coq-timeout 20 -wp-prover cvc4 -wp-prover z3 -wp-prover alt-ergo
-wp-prover coq  -wp-par 1 -wp-out push_heap.wp-wp-prover coq
```

Die Theorembeweiser sind in den folgenden Versionen installiert.

Theorembeweiser	**Version**	**Quelle**
CVC4	1.4	`https://cvc4.cs.nyu.edu/web`
Z3	4.4.2	`https://github.com/Z3Prover/z3`
Alt-Ergo	0.99.1	`http://alt-ergo.lri.fr`
Coq	8.4.6	`https://coq.inria.fr`

Für die Validierung der Verifikationsergebnisse ist dieser Arbeit ist die `tar` Datei

`StandardAlgorithms.tar` beigelegt. In diesem Verzeichnis sind alle weiteren Dateien aus *ACSL By Example* enthalten, die für die Verifikation der Funktionen `make_heap` und `push_heap` nötig sind, an denen nicht in dieser Arbeit gearbeitet wurde.

Die Verifikation von `make_heap` und `push_heap` kann mit den Kommandos `make make_heap.wp` und `make push_heap.wp` in den Verzeichnissen

`StandardAlgorithms/heap/make_heap` und

`StandardAlgorithms/heap/push_heap` ausgeführt werden.

Für die Verifikation einiger der darin enthaltenen Lemmas wurde der interaktive Theorembeweiser eingesetzt. Diese Lemmas werden nach dem Verifikationsergebnis aufgelistet.

Die Tabelle 5.1 zeigt das Ergebnis der formalen Verifikation. Für die Funktionen `make_heap` und `push_heap` ist je angegeben, wie viele Beweisverpflichtungen für die annotierte Implementierung generiert werden, wie viele davon bewiesen werden und von welchem Theorembeweiser sie bewiesen werden.

Algorithmus	Section	Beweisverpflichtungen			Theorembeweiser				
		Alle	Bewiesen	(%)	Qed	Alt-Ergo	CVC4	Z3	Coq
`make_heap`	4.3	36	36	100	8	0	24	2	2
`push_heap`	4.4	94	94	100	35	2	52	2	3

Tabelle 5.1: Ergebnis der Verifikation der Heap-Algorithmen

Für beide Funktionen werden alle Beweisverpflichtungen nachgewiesen. Die meisten Beweisverpflichtungen werden von automatischen Theorembeweisern gezeigt. Einige wenige durch den interaktiven Theorembeweiser Coq.

Bei `make_heap` wird das verwendete Lemma `HeapAndMaximum` nur mit Coq bewiesen. Für die Nachbedingung `max` musste in Coq ein kurzer Beweis formuliert werden, damit das Lemma `HeapAndMaximum` angewendet wird.

Bei `push_heap` werden die Lemmas `CountSectionBounds`, `CountSectionMonotonic` und ebenfalls das Lemma `HeapAndMaximum` mit Coq bewiesen.

6 Diskussion

In diesem Kapitel findet eine Reflexion der Arbeit aus Kapitel 4 und der Ergebnisse aus Kapitel 5 statt. Die Ergebnisse der Verifikationsarbeit werden zusammengefasst und diskutiert.

Die Arbeit hatte unter anderem die vollständige Verifikation der Algorithmen `push_heap` und `make_heap` zum Ziel. Im Kapitel 5 wurde dokumentiert, dass beide Funktionen vollständig und weitestgehend automatisch verifiziert werden.

Den Großteil der Verifikation machte die Erstellung der annotierten Implementierung aus. Es wurden die Schleifen-Annotationen für die beiden Funktionen formuliert und im Fall von `push_heap` fand eine Umstrukturierung der Implementierung statt und es wurden viele Annotationen hinzugefügt.

Nach der Umstrukturierung und Annotierung von `push_heap` wurde klar, dass die annotierte Implementierung einer Funktion sich nicht nur durch hinzugekommene Kommentare von der ursprünglichen Implementierung unterscheidet, sondern auch Unterschiede in der Implementierung selbst existieren können. Diese Veränderungen können zum einen durch die formale Spezifikation und zum anderen durch die Ergebnisse von fehlgeschlagenen Verifikationsversuchen initiiert werden.

Die in Kapitel 3 vorgestellte Abbildung 3.1 muss angepasst werden und enthält die annotierte Implementierung als einzelnes Dokument. Außerdem müssen die Interaktionen zwischen den einzelnen Prozessen und Dokumenten angepasst werden. So muss die annotierte Implementierung an das Verhalten von Frama-C und den Theorembeweisern angepasst werden um das Verifikationsergebnis zu verbessern.

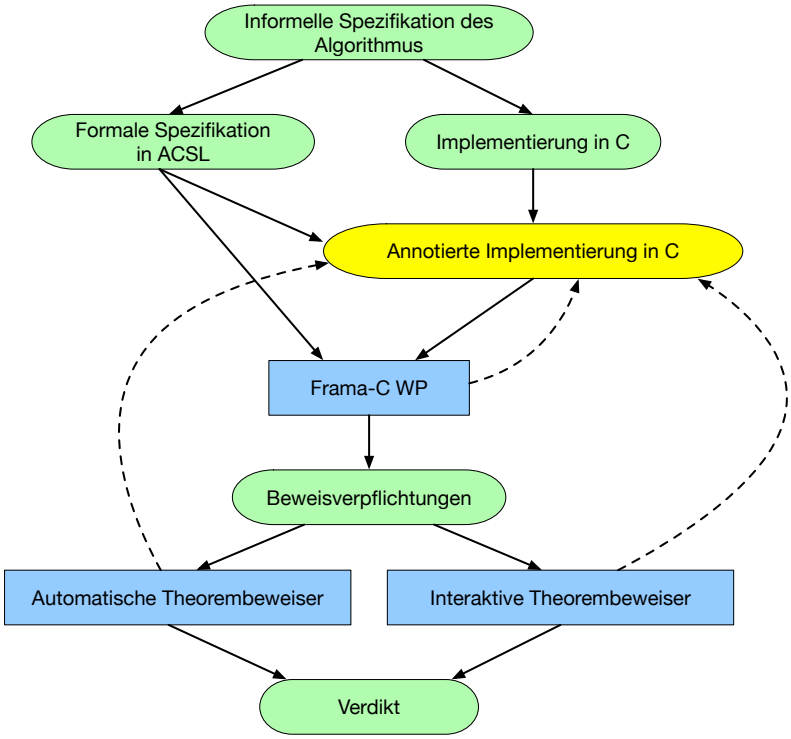

Abbildung 6.1: Angepasste Darstellung des Verifikationsprozesses mit Frama-C

Die Umstrukturierung von `push_heap` war besonders umfangreich und für `make_heap` war eine Umstrukturierung nicht nötig. Dieser Arbeitsschritt kann für verschiedene Funktionen unterschiedliche Ausmaße annehmen. Es ist aber oft hilfreich die einzelnen Operationen in der Funktion zumindest im Hinblick auf die Spezifikation zu hinterfragen.

Ein weiteres Ziel war es, zu zeigen, dass die Verifikation von Funktionen, die bereits verifizierte Funktionen aufrufen, einfacher sein kann, als die Verifikation von Funktionen die ohne solche Aufrufe implementiert werden.

Am unterschiedlichen Umfang der Abschnitte 4.5 und 4.6 ist zu erkennen, dass die Funktion `make_heap`, obwohl sie insgesamt eine höhere Komplexität besitzt, entschieden weniger Aufwand bei der Verifikation mit sich bringt. Dafür sorgt der Aufruf der Funktion `push_heap`, welcher im Gegensatz zu einer erneuten Implementierung und Annotation der Funktionalität viel weniger Arbeit verursacht.

Mit der in Kapitel 4.7 vorgestellten Methode für das Finden der richtigen Annotationen, bietet die Arbeit auch eine Hilfestellung für die Verifikation von Funktionen, die nicht auf bereits verifizierten Funktionen aufbauen. Dabei stellte der Arbeitsschritt, so viele einzelne Operationen wie möglich zu spezifizieren, den großen Vorteil dieser Herangehensweise dar.

Das Problem wurde in viele kleine Probleme aufgeteilt. Mit dieser Methode wurde schnell ersichtlich, welche Operationen besonders schwierig für die Verifikation der Nachbedingungen sind. Mit dieser Erkenntnis ließen sich dann entsprechende Spezifikationen für die einzelnen Operationen finden.

Speziell in Schleifen muss aber darauf geachtet werden, die richtigen Zusicherungen einzufügen. Die Schleifeninvarianten enthalten oft Variablen, die im Verlauf der Schleife überschrieben werden und nach diesen Zuweisungen nicht mehr gültig sind, bis eine weitere Zuweisung stattfinden, oder der Iterator aktualisiert wird.

Da die Verifikation von einzelnen Zusicherungen, Schleifeninvarianten und Nachbedingungen meist auf den Eigenschaften anderer Annotationen aufbaut, können flüchtig gewählte, falsche Annotationen viel unnötige Arbeit verursachen.

Beweise einer Annotationen, die auf anderen Zusicherungen aufbauen, sind nämlich nur dann wahr, wenn auch die Zusicherungen wahr sind. Da eine falsche Zusicherung nicht wahr werden kann, kann keine erfolgreiche Verifikation stattfinden.

Literaturverzeichnis

[1] Jochen Burghardt und Jens Gerlach und Timon Lapawczyk. ACSL By Example. http://www.fokus.fraunhofer.de/download/acsl_by_example, eingesehen am 27.06.2016, Februar 2016.

[2] Homepage der Standard Template Library. http://www.sgi.com/tech/stl/, eingesehen am 20.06.2016.

[3] The C++ Standards Committee. Working Draft, Standard for Programming Langugage C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf, eingesehen am 19.06.2016.

[4] Apache. Apache C++ Standard Library User's Guide. http://stdcxx.apache.org/doc/stdlibug, eingesehen am 19.06.2016.

[5] Donald E. Knuth. *The art of computer programming*, volume 3. Addison-Wesley Longman Publishing Co., Boston, MA, USA, 2nd edition, 1998.

[6] Der push_heap Algorithmus in der Standard Template Library. http://www.sgi.com/tech/stl/push_heap.html, eingesehen am 19.06.2016.

[7] Der make_heap Algorithmus in der Standard Template Library. http://www.sgi.com/tech/stl/make_heap.html, eingesehen am 19.06.2016.

[8] Homepage vom Frama-C WP Plug-in. http://frama-c.com/wp.html, eingesehen am 28.06.2016.

[9] Kim Völlinger. Einsatz des Beweisassistenten Coq zur deduktiven Programmverifikation. Diplomarbeit, Humboldt-Universität zu Berlin, August 2013.

[10] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.

[11] Edsger W. Dtra. Guarded commands, nondeterminacy and formal derivation of program. *Communications of the ACM*, 18:453–457, 1975.

[12] Patrick Baudin und Pascal Cuoq und Jean-Christophe Filliâtre und Claude Marché und Benjamin Monate und Yannick Moy und Virgile Prevosto. ACSL: ANSI/ISO C Specification Language. http://frama-c.com/download/acsl.pdf, eingesehen am 26.06.2016.

[13] Homepage von Alt-Ergo. http://alt-ergo.lri.fr/, eingesehen am 26.06.2016.

[14] Github Seite von Z3. https://github.com/Z3Prover/z3, eingesehen am 26.06.2016.

[15] Homepage von CVC4. https://cvc4.cs.nyu.edu/web/, eingesehen am 26.06.2016.

[16] Homepage von Coq. https://coq.inria.fr/, eingesehen am 26.06.2016.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 20. Juli 2016 .....