Softwaretechnik
2002
Prof. Dr. H. Klaeren und M E Leypold

Eberhard Karls
Universität
Tübingen

Lösungsskizze 4 : Saalbestuhlung, Makefile

Inhaltsverzeichnis

1 Saalbestuhlung

1.1 Nachbarschaft

types

1.0 $stuhl = \mathsf{token};$

Über (einzelne) Stühle benötigen wir keine weiteren Informationen. Alle Ausführungen in der Aufgabenstellung sind insoweit irreführend, als Nummerierung oder Zeichen für einen Stuhl uns nicht bei der Frage helfen, ob Stühle benachbart sind oder nicht. Stattdessen benötigen wir eine Abstraktion, die uns sagt, ob *jeweils 2 Stühle* benachbart sind. Dies ist eine symmetrische Relation, die wir hier als boolesche Funktion formulieren:

2.0 $bestuhlung = stuhl \times stuhl \xrightarrow{m} \mathbb{B}$

.1 $\mathsf{inv}\ r \triangleq \mathsf{let}\ ss =$
.2 $\quad \{s.\#1 \mid s \in (\mathsf{dom}\ r)\} \cup$
.3 $\quad \{s.\#2 \mid s \in (\mathsf{dom}\ r)\}\ \mathsf{in}$
.4 $\quad (\mathsf{dom}\ r) = \{\mathsf{mk\text{-}}(s1, s2) \mid s1, s2 \in ss\} \wedge$
.5 $\quad \forall s \in ss, s' \in ss \cdot$
.6 $\quad\quad r(\mathsf{mk\text{-}}(s, s')) = r(\mathsf{mk\text{-}}(s', s))$

Die Invariante beschreibt im Wesentlichen die Symmetrie. Wir können den Typ *bestuhlung* als eine Beschreibung der Aufstellung von Stühlen interpretieren.

functions

3.0 $stuehle : bestuhlung \to stuhl\text{-}\mathsf{set}$

.1 $stuehle(b) \triangleq$
.2 $\quad \{s.\#1 \mid s \in (\mathsf{dom}\ b)\}$

Die in einer *bestuhlung* enthaltenen Stühle erhalten wir als Definitionsmenge der Relation. Damit packen wir zwei unterschiedliche Konzepte des Problems („Welche Stühle stehen im Saal?“ und „Welche Stühle haben welche Nachbarn?“) in eine Struktur.

Man kann sich solchen Modellierungsproblemen („Was modelliere ich wie?“) übrigens auch sachte nähern, indem man für jedes Konzept des Problembereichs (z. B. Stühle in einem Saal, Belegungsliste, Nachbarschaftsrelation) eigene Strukturen (VDM-SL-Typen) definiert. Man fügt dann beim Zusammensetzen (Dinge, die wir über einen Theatersaal wissen) geeignete Invarianten hinzu. Auf diese Art entsteht eine gewisse Redundanz (Komponenten des Modells sind durch Invarianten so stark miteinander verkoppelt, dass man aus einer Komponente die andere berechnen kann), die man durch eine Transformation der Spezifikation „wegschaffen“ kann. Natürlich spielen auch hier wieder Retrieve-Funktionen eine wichtige Rolle, um alte und neue Spezifikationen miteinander zu verbinden.

1.2 Belegung

types

4.0 $belegung = stuhl\text{-}\mathsf{set}$

Der Belegungszustand einer Veranstaltung wird am einfachsten durch die Menge der belegten Stühle charakterisiert.

Die beiden folgenden Prädikate werden sich im Weiteren als nützlich erweisen: Sie sagen aus, ob alle Stühle einer bestimmten Menge frei bzw. belegt sind.

functions

5.0 $sind\text{-}frei : belegung \times stuhl\text{-set} \to \mathbb{B}$
.1 $sind\text{-}frei\,(belegt, ss) \triangleq$
.2 $\quad \forall s \in ss \cdot$
.3 $\quad\quad s \notin belegt;$

6.0 $sind\text{-}belegt : belegung \times stuhl\text{-set} \to \mathbb{B}$
.1 $sind\text{-}belegt\,(belegt, ss) \triangleq$
.2 $\quad \forall s \in ss \cdot$
.3 $\quad\quad s \in belegt$

Übrigens ist das eine Prädikat nicht einfach eine Verneinung des anderen, weswegen wir auch beide Prädikate benötigen.

1.3 Veranstaltung

types

7.0 $veranstaltung :: s : bestuhlung$
.1 $\quad\quad\quad\quad\quad\quad bz : belegung$
.2 inv $v \triangleq v.bz \subseteq stuehle\,(v.s)$

Der Typ *veranstaltung* enthält alles, was für die Sitzebelegung über eine Veranstaltung bekannt sein muss: Die Ausstellung der Stühle *s*, und die momentane Belegung *bz*. Dabei enthält *s* indirekt die Angabe, welche Stühle überhaupt im Saal stehen, und natürlich dürfen nur Stühle belegt sein, die auch im Saal stehen. Letzteres wird in der Invariante ausgesagt.

1.4 Zusammenhänge

Wir sind nun soweit, dass wir definieren können, wann eine Menge von Stühlen zusammenhängend ist (der Aufgabenstellung folgend nennen wir die Funktion etwas irreführend *sind-nebeneinander*).

functions

8.0 $sind\text{-}nebeneinander : bestuhlung \times stuhl\text{-set} \to \mathbb{B}$
.1 $sind\text{-}nebeneinander\,(istNachbar, ss) \triangleq$
.2 $\quad \text{card } ss = 1 \vee$
.3 $\quad \exists s \in ss \cdot$
.4 $\quad\quad \text{let } rest = (ss \setminus \{s\}) \text{ in}$
.5 $\quad\quad (sind\text{-}nebeneinander\,(istNachbar, rest) \wedge$
.6 $\quad\quad\quad \exists s' \in rest \cdot istNachbar\,(\text{mk-}(s, s')))$
.7 pre $\neg\, ss = \{\}$

Eine Menge von Stühlen ist demnach zusammenhängend, wenn wir einen Stuhl finden können, der einen Nachbarn im Rest der Menge hat, und dieser Rest der Menge seinerseits zusammenhängend ist.

Einige alternative Formulierungen sind vorgeschlagen worden[1]. Beispielsweise ist eine Stuhlmenge dann zusammenhängend, wenn sich für je zwei Stühle einer Menge ein Pfad

[1] Sie sind aber meist nur unbefriedigend ausformuliert worden.

(von Nachbar zu Nachbar) finden lässt, der vollständig durch die infragestehende Menge läuft. Lösungen, die versuchen, die Randstühle zu identifizieren (maximal 2 Randstühle) verlassen sich meist auf besondere Eigenschaften der Nachbarschaftsrelation (maximal 2 Nachbarn) und würden in einem fünfdimensionalen Theater sicher nicht funktionieren[2].

Auf zwei Fallstricke möchte ich noch besonders hinweisen. Einige Teilnehmer haben versucht etwas in der folgenden Art zu formulieren:

```
types

9.0  stuhl' :: rechter-nachbar : stuhl'
 .1            linker-nachbar : stuhl'
```

Aber die durch diesen Ausdruck beschriebene Menge – die Wertemenge des Typs *stuhl'*) ist nicht wirklich wohldefiniert, auch wenn z. B. die VDMTools an dieser Stelle bei der Typprüfung keinen Fehler melden. Man kann sich eine Darstellung der Wertemenge eines Typs als eine Menge von Termen vorstellen, deren induktive Definition durch die Typdefinition gegeben ist. Aber Terme von denen *jeder* einen anderen aus derselben Menge enthält sind unendlich – das ist sicher Unsinn, hier fehlt die Abbruchbedingung für die Rekursion.

Ich habe den Eindruck, bei dem gerade zitierten Fehler wäre eine gewisser Anteil objektorientierten Gedankenguts involviert gewesen. Aber die Komponenten eines *Composite Type* in VDM-SL sind keine *Referenzen*, sondern schlicht *Bestandteile* der betreffenden Elemente[3].

Zudem bin ich der Ansicht, dass hier eine Vermischung von Konzepten vorliegt, nämlich zwischen der *Identität* eines Stuhles, und der Frage, welche *Nachbarn* ein Stuhl hat. Beides zur gleichen Zeit behandeln und im selben Konstrukt modellieren zu wollen, ist nicht nur schlechter Stil, sondern macht hier auch die Spezifikation sinnlos, auch wenn der Text gewisse Assoziationen beim Leser erzeugt: Darum allerdings geht es bei formalen Methoden nicht allein.

Ein weiterer Stolperstein scheint gewesen zu sein, dass viele Teilnehmer in *sind-verfuegbar* und *suche-plaetze* versucht haben, statt *Mengen* von Stühlen *Listen* von Stühlen zu behandeln. In der Folge konstruierten sie dann natürlich rekursive Funktionen über Listen, um Elemente aus diesen zu entfernen, um Elemente zu finden, und so weiter. Mit Mengen wäre das nicht nötig gewesen: Es sei darauf hingewiesen, dass der Existenzquantor beispielsweise eine Suche nach einem Element mit bestimmten Eigenschaften in einem gewissen Sinn ersetzt. Listen dagegen implizieren eine Reihenfolge, daher ist es nicht verwunderlich, wenn man bei ihrer Verwendung schließlich mit der Konstruktion von Programmschleifen endet (wozu sich der Teil von VDM-SL, den wir kennengelernt haben, nur sehr bedingt eignet).

In der Aufgabenstellung war jedoch keine Reihenfolge impliziert (im Gegenteil, es ist ausdrücklich von Mengen von Stühlen die Rede). Listen waren also eine zu spezielle Darstellung der Ergebnisse und Eingaben von *sind-verfuegbar* und *such-plaetze*. Ich bin auch offen gestanden schlichtweg fasziniert, wie kompliziert die Lösung dann werden kann.

[2] :-)

[3] Der Unterschied ist wirklich schwer zu erklären, wenn er nicht bereits geläufig ist, vor allem in dieser Richtung. Ich halte nämlich *Bestandteil* für den fundamentaleren Begriff in allen mathematischen oder formalen Systemen, dagegen muss der Begriff der *Referenz* erst konstruiert werden. Wer hier verwirrt ist, den möchte ich bitten, ein Weilchen über den Unterschied zwischen „bestehen aus“ und „verweisen auf“ zu meditieren.

1.5 Buchung von Plätzen

functions

10.0 $sind\text{-}verfuegbar : veranstaltung \times \mathbb{N} \to \mathbb{B}$

.1 $sind\text{-}verfuegbar\,(v,n) \triangleq$

.2 $\quad \exists\, ss : stuhl\text{-set} \cdot$

.3 $\qquad sind\text{-}frei\,(v.bz, ss) \wedge$

.4 $\qquad sind\text{-}nebeneinander\,(v.s, ss) \wedge$

.5 $\qquad \mathsf{card}\ ss = n;$

Eine Menge von freien Plätzen in einer Veranstaltung ist verfügbar, wenn eine Menge von Plätzen mit den gewünschten Eigenschaften (siehe Nachbedingung) existiert.

11.0 $suche\text{-}plaetze\,(v : veranstaltung, n : \mathbb{N})\ v' : veranstaltung, ss : stuhl\text{-set}$

.1 pre $sind\text{-}verfuegbar\,(v,n)$

.2 post $ss \subseteq stuehle\,(v.s) \wedge$

.3 $\quad \mathsf{card}\ ss = n \wedge$

.4 $\quad sind\text{-}frei\,(v.bz, ss) \wedge$

.5 $\quad sind\text{-}nebeneinander\,(v.s, ss) \wedge$

.6 $\quad sind\text{-}belegt\,(v'.bz, ss) \wedge$

.7 $\quad v.s = v'.s \wedge$

.8 $\quad unchanged\text{-}but\,(v.bz, v'.bz, ss)\ ;$

Die Spezifikation von *suche-plaetze* ergibt sich nun geradezu trivial unter Verwendung der vorangegangenen Definitionen. Vorbedingung ist natürlich, dass genügend Sitze verfügbar sein müssen. Die Nachbedingungen sagen das Folgende aus:

- Wir wollen nur Stühle belegen, die auch Plätze der Veranstaltung sind.
- Wenn wir n Stühle anfordern, wollen wir auch n Stühle als Ergebnis bekommen.
- Die Plätze, die wir als Ergebnis bekommen sollen *vorher* alle frei gewesen sein.
- Die Plätze sollen *nachher* belegt sein.
- Die Plätze sollen zusammenhängen.
- An der Bestuhlung der Veranstaltung ändert sich nichts.
- An der Belegung aller anderen Plätze der Veranstaltung ändert sich nichts.

Das Prädikat *unchanged-but* müssen wir noch nachreichen:

12.0 $unchanged\text{-}but : belegung \times belegung \times stuhl\text{-set} \to \mathbb{B}$

.1 $unchanged\text{-}but\,(b, b', ss) \triangleq$

.2 $\quad b \setminus ss = b' \setminus ss$

1.6 Totale Operation

Die spezifizierte Operation *suche-plaetze* ist geeignet für den Einsatz in einer durch Mutexe geschützten Region des Programms auf dem Server, der die Datenbank verwaltet.

Wenn wir kein exklusives Locking des Datenbestandes auf dem Server durch den Client (Terminal) zulassen wollen[4], dann ist die Operation in der gegebenen Spezifikation nicht für die Verwendung auf dem Terminal geeignet. Es ist nämlich so, dass der Buchungszustand der Veranstaltung auf dem Server gespeichert ist. Weil die Terminals nebenläufig auf diesem Zustand operieren, können aus dem Daten- und Kontrollfluss auf dem Terminal keine Aussagen über den Zustand auf dem Server gemacht werden, es kann also auch keine Vorbedingung garantiert werden.

Selbst eine Abfrage des Belegungszustandes unmittelbar vor dem Aufruf der Operation *suche-plaetze*, etwa so

```
if (server->sindverfuegbar(v,n)){
   server->suche-plaetze(v,n,&result); }
else {
   display("not available");
   ...                                 }
```

ist zu nichts nütze, da zwischen der Abfrage und der Operation Zeit vergeht, in der möglicherweise ein anderes Terminal den Serverzustand ändert (Race-Condition).

Ein Ausweg sind, wie erwähnt, Locks (Mutexe), aber das ist bei Systemen, die räumlich entfernt voneinander sind und unvermutet getrennt werden könnten, immer schwierig. Besser ist hier der Ausweg, die Operation einfach in der bekannten Manier *total* zu machen und um ein Fehlerflag zu ergänzen:

types

13.0 $failure\text{-}flag = \mathbb{B}$

functions

14.0 $suche\text{-}t\,(v\!:\!veranstaltung, n\!:\!\mathbb{N})\ v'\!:\!veranstaltung, ss\!:\!stuhl\text{-set}, FAILED\!:\!failure\text{-}flag$
.1 post if $sind\text{-}verfuegbar\,(v,n)$
.2 then $\neg\, FAILED \wedge$ post-$suche\text{-}plaetze\,(v,n,$ mk-$(v',ss))$
.3 else $FAILED \wedge v = v'$

Auf dem Server würden dann wieder Mutexe zum Einsatz kommen, dort ist das allerdings viel weniger kritisch.

[4] Und darin wären wir gut beraten. Was würde beispielsweise geschehen, wenn der Client durch einen Crash offline geht, nachdem er das Lock gesetzt hat?

2 Einfache Makefiles

Make ist ein Tool, mit dem das inkrementelle Update einer Kollektion voneinander abhängiger Dateien gesteuert werden kann. Das ursprüngliche Make hatte seinen ersten Auftritt (soweit ich weiß) Ende der 70er Jahre im Bell Unix V7. Seitdem gab es sehr viele Erweiterungen des ursprünglichen Konzepts und sehr viele vom ursprünglichen Make inspirierte Klone und Substitute. Im Folgenden werde ich vor allem von *GNU Make* sprechen, auch wenn der Anfang meiner Ausführungen ebenso auf andere Makevarianten (etwa *BSD Make*) zutrifft.

GNU Make bietet viele Möglichkeiten, vom expliziten und offenkundigen, bis zum abstrakten, die Abhängigkeit von Dateien und die Befehle, welche zur Regenerierung von Produkten ausgeführt werden müssen, zum Ausdruck zu bringen. Ich werde im Folgenden schrittweise von einfacheren zu mehr abstrahierenderen Arten der Formulierung von Makefiles hinführen, indem ich Stück für Stück redundante Strukturen wegabstrahiere.

Jedes Makefile dieses Abschnittes wäre eine gültige Lösung gewesen. Alle Makefiles dieses Abschnitts sind in der Archivdatei *FiFo-2002-09-25.tar.gz* enthalten, die von der Webseite zur Vorlesung heruntergeladen werden kann[5].

2.1 *Makefile*

Dies ist das einfachste Makefile, das die Aufgabenstellung erfüllt. Es gibt 2 Pseudo-Targets (*all* in Zeile 5, *clean* in den Zeile 19+20), welche nicht Produkten des Übersetzungsprozesses entsprechen, sondern eher administrative Funktion haben. Die Produkte (*Targets*) sind explizit gelistet, sowie die Edukte (*Dependencies*) aus denen sie gefertigt werden, und ein Befehl (*Action*), mit dem aus den Edukten jeweils das Produkt erzeugt werden kann.

```
 5  all: testschedule

 7  jobs.o          : jobs.cc jobs.hh
 8          g++ -Wall -g -c jobs.cc

10  queues.o        : queues.cc queues.hh jobs.hh
11          g++ -Wall -g -c queues.cc

13  testschedule.o  : testschedule.cc queues.hh jobs.hh
14          g++ -Wall -g -c testschedule.cc

16  testschedule     : testschedule.o queues.o jobs.o
17          g++ -Wall -g -o testschedule \
                            testschedule.o queues.o jobs.o

19  clean:
20          rm -f *.o testschedule *~
```

Eine gewisse Redundanz fällt hier auf: Produkte und Edukte werden sowohl in der Abhängigkeitsdeklaration (z. B. Zeile 16), als auch im dazugehörigen Übersetzungsprozess (z. B. Zeile 17) explizit genannt.

[5] Der C++-Quelltext aus dieser Datei (ohne die Makefiles) war Arbeitsmaterial zur dieser Aufgabe und gleichzeitig Lösung zu Aufgabe 2.

2.2 *Makefile.2*

Die erwähnte Redundanz kann durch die Verwendung der sogenannten *automatischen Variablen* in den Aktionen vermieden werden. Vor dem Ausführen des Übersetzungsbefehls werden die automatischen Variablen durch Teil der Abhängigkeitsregeln ersetzt, und zwar:

$@ durch den Namen des Produktes,

$< durch den Namen des linken Eduktes,

$^ durch die Liste der Edukte.

Wie man sich mit diesen Regeln leicht überzeugen kann, ist das folgende Makefile äquivalent zum vorangegangenen:

```
 5  all: testschedule

 7  jobs.o          : jobs.cc jobs.hh
 8          g++ -Wall -g -c $<

10  queues.o        : queues.cc queues.hh jobs.hh
11          g++ -Wall -g -c $<

13  testschedule.o  : testschedule.cc queues.hh jobs.hh
14          g++ -Wall -g -c $<

16  testschedule     : testschedule.o queues.o jobs.o
17          g++ -Wall -g -o $@ $^

19  clean:
20          rm -f *.o testschedule *~
```

Jetzt wird sichtbar, dass die Struktur der Befehle zum Übersetzen in Objektdateien vollkommen gleich ist. Diese Redundanz nutzen und beseitigen wir in den nächsten beiden Schritten

2.3 *Makefile.3*

Wir können die Befehle zum Übersetzen in eine eigene Variable abstrahieren. Es sei hier angemerkt, dass die Expansion von Variablen so spät als möglich erfolgt, so dass die Erwähnung der automatischen Variablen in der *Definition* von *CXX-COMPILE* und *CXX-LINK* auch außerhalb einer Übersetzungsregel Sinn macht, sofern diese Variablen in einer Übersetzungsregel *verwendet* werden.

```
 5  all: testschedule

 7  CXX-COMPILE = g++ -Wall -g -c $<
 8  CXX-LINK    = g++ -Wall -g -o $@ $^

11  jobs.o          : jobs.cc jobs.hh
12          $(CXX-COMPILE)

14  queues.o        : queues.cc queues.hh jobs.hh
15          $(CXX-COMPILE)

17  testschedule.o  : testschedule.cc queues.hh jobs.hh
18          $(CXX-COMPILE)
```

```
20  testschedule     : testschedule.o queues.o jobs.o
21          $(CXX-LINK)

23  clean:
24          rm -f *.o testschedule *~
```

An sich bringt das nicht so viel, hat aber den Vorteil, dass Änderungen an den bei der Übersetzung verwendeten Compileroptionen nun an einer zentralen Stelle der Objektdatei vorgenommen werden können.

2.4 *Makefile.4*

Die ähnlichen Übersetzungsregeln für die Objektdateien lassen sich nun in eine *Patternregel* (Zeilen 10-11) separieren:

```
 5  all: testschedule

 7  CXX-COMPILE = g++ -Wall -g -c $<
 8  CXX-LINK    = g++ -Wall -g -o $@ $^

10  %.o: %.cc
11          $(CXX-COMPILE)

13  jobs.o          : jobs.hh
14  queues.o        : queues.hh jobs.hh
15  testschedule.o  : queues.hh jobs.hh

17  testschedule     : testschedule.o queues.o jobs.o
18          $(CXX-LINK)

20  clean:
21          rm -f *.o testschedule *~
```

Diese bedeutet, dass, wo immer eine auf „.o“ endende Datei benötigt wird, diese durch Ausführung der Befehle in *CXX-COMPILE* aus einer auf „.cc“ endenden Datei gewonnen werden könnte. In den Zeilen 13-15 werden durch Abhängigkeitsregeln *ohne* Übersetzungsbefehle (Action) weitere Abhängigkeiten hinzugefügt, über diejenigen hinaus, die sich aus der Anwendung der Patternregel ergeben.

Make muss sich nun auch die intermediären Produkte, die es benötigt, um die endgültigen Produkte fertigen zu können, selbst zusammensuchen. Für *jobs.o* sieht der Ablauf also so aus:

- Um *testschedule* zu erzeugen, wird u. a. *jobs.o* benötigt.
- Für *jobs.o* existiert keine explizite Regel. Aber die gegebene Patternregel (Zeilen 10-11) besagt, dass sich *jobs.o* aus *jobs.cc* mittels *$(CXX-COMPILE)* erstellen ließe.
- Die Datei *jobs.cc* existiert, und hat ihrerseits keine Produktionsregel. Auf dieser Seite ist der Vorgang, die notwendigen Aktionen zur Fertigung aller nötigen Produkte zu bestimmen, also abgeschlossen.
- In Zeile 13 ist *jobs.hh* als weiteres Edukt von *jobs.o* genannt, die vollständige Liste der Edukte ist also: *jobs.cc*, *jobs.hh*.
- Wenn nun *jobs.o* älter ist als *jobs.cc* oder *jobs.hh*, muss *CXX-COMPILE* expandiert – man erhält *g++ -Wall -g -c jobs.cc* – und ausgeführt werden.

3 Trickreiche Makefiles

Da ich Makefiles über alles liebe, möchte ich noch drei kunstvollere Exemplare aus der Menagerie vorführen.

Makefile.tricky separiert die projektspezifischen Details in wenige Zeilen, während der Rest des Makefiles generisch für eine große Klasse kleiner C++-Projekte ist.

Makefile.tricky2 generiert zusätzlich die Abhängigkeiten der einzelnen Module automatisch.

Makefile.special schließlich ist so gestaltet, dass es für jedes Modul das Vorhandensein einer Headerdatei (einer Schnittstellendefinition) erzwingt.

3.1 *Makefile.tricky*

Das folgende Exemplar abstrahiert noch einige Variablen ab, um dem Programmierer in den Zeilen 2-11 die Möglichkeit zu geben, die in den Übersetzungsprozessen verwendeten Werkzeuge zu beeinflussen. Die Variablennamen entsprechen gängiger Praxis.

Darüberhinaus existieren nun eine Linkregel (Zeilen 22-23) und eine separate Übersetzungsregel für Module (Zeilen 16-17), was die Erwähnung der Modulheader in den Abhängigkeiten erspart. Die Abhängigkeitenliste für *jobs.o* ist deshalb weggefallen.

Zeilen 13-25 und 31-32 sind vollkommen generisch. Nur noch die Zeilen 5, 27-29 enthalten projektspezifische Angaben. Zeile 25 ist nötig, um eine eingebaute Regel (siehe Abschnitt 4) zu deaktivieren.

```
 5  all: testschedule

 7  CXX       = g++
 8  CPPFLAGS  =
 9  CXXFLAGS  = -Wall -g
10  LDFLAGS   =
11  LOADLIBES =

13  CXX-COMPILE = $(CXX) $(CPPFLAGS) $(CXXFLAGS) -c -o $@ $<
14  CXX-LINK    = $(CXX) $(CXXFLAGS) $(LDFLAGS) -o $@ $^ $(LOADLIBES)

16  %.o: %.cc %.hh
17          $(CXX-COMPILE)

19  %.o: %.cc
20          $(CXX-COMPILE)

22  %: %.o
23          $(CXX-LINK)

25  %: %.cc                             # cancel this rule

27  queues.o        : jobs.hh
28  testschedule.o  : queues.hh jobs.hh
29  testschedule    : queues.o jobs.o

31  clean:
32          rm -f *.o testschedule *~
```

3.2 *Makefile.tricky.2*

Das folgende Exemplar ist besonders interessant: Zum einen wurden alle projektspezifischen Details (Zeilen 17-25) und alle Parameter der Übersetzungsprozesse (Zeilen 9-15) in eigenen Abschnitten gesammelt. Zum anderen schreibt das Makefile seine Abhängigkeiten selbst. Die prinzipielle Idee für dieses Verfahren, sowie die Begründung, warum so etwas sinnvoll ist, wird soweit ich weiss zum ersten Mal im Text *Recursive Make Considered Harmful* (Miller, 1997) angesprochen.

```
 9  default: all

11  CXX       = g++
12  CPPFLAGS  =
13  CXXFLAGS  = -Wall -g
14  LDFLAGS   =
15  LOADLIBES =

17  MODULES         = jobs queues
18  PROGRAMS        = testschedule

20  testschedule    : $(MODULES:%=%.o)

22  clean:
23          rm -f *.o testschedule *~

25  all: $(PROGRAMS)

29  CXX-COMPILE = $(CXX) $(CPPFLAGS) $(CXXFLAGS) -c -o $@ $<
30  CXX-LINK    = $(CXX) $(CXXFLAGS) $(LDFLAGS) -o $@ $^ $(LOADLIBES)

32  CXX-MKDEPS  = {  $(CXX) $(CPPFLAGS) -MM $< >TMP.$@ && \
33                   cp TMP.$@ $@                       && \
34                   sed 's|.o:|.deps:|' < TMP.$@ >>$@ && \
35                   rm TMP.$@                          ;} \
36                || { rm -f $@ ; false ; }

38  %.o: %.cc %.hh
39          $(CXX-COMPILE)

41  %.o: %.cc
42          $(CXX-COMPILE)

44  %: %.o
45          $(CXX-LINK)

47  %: %.cc # cancel this rule / could be replaced by COMPILE-LINK rule

49  %.deps: %.cc
50          $(CXX-MKDEPS)

52  ifneq ($(MAKECMDGOALS),veryclean)
53    include $(PROGRAMS:%=%.deps) $(MODULES:%=%.deps)
54  endif

56  veryclean: clean
57          rm -f *.deps
```

In den meisten Varianten von *Make* ist es möglich, Makefilefragmente durch eine *include*-

Anweisung einzuschliessen. *GNU Make* hat dabei die Eigentümlichkeit, dass es die bereits bekannten Regeln durchgeht, um festzustellen, ob Informationen vorliegen, wie die einzuschließende Datei erzeugt werden kann: Ist diese Datei nicht vorhanden oder nach den bekannten Regeln nicht auf dem neuesten Stand, so wird die Datei erzeugt bzw. regeneriert und der ganze Ablauf neu gestartet.

Wir müssen also für jedes Modul und jedes Programm (Hauptmodul) ein entsprechendes Makefilefragment einschließen, in dem die nötigen Abhängigkeiten stehen. Dies geschieht mittels eines Patternersetzungsmechanismus in Zeile 53. Dann müssen wir noch die Regeln liefern, wie diese Dateien (mit der Endung „.deps“) durch Untersuchung der Dateien mit der Endung „.cc“ erzeugt werden können. Dies geschieht in den Zeilen 49-50 (Patternregel) und 32-36.

Der Kern des Mechanismus in Zeile 32-36 ist, dass die meisten C-Compiler (hier *gcc* mit dem Schalter *-MM*) die Abhängigkeiten der Objektdateien von Headerdateien aus den Quellen ermitteln und ausgegeben können [6]. Beispielsweise steht nach einem Makelauf in *queue.deps*:

```
queues.o: queues.cc queues.hh jobs.hh
queues.deps: queues.cc queues.hh jobs.hh
```

Die Abhängigkeitsliste für *queues.deps* ist nötig, damit diese Datei auch regeneriert wird, wenn sich eine der indirekt eingeschlossenen Headerdateien ändert: Schließlich könnte in dieser eine neue *include*-Direktive hinzugekommen sein, oder eine vorhandene weggefallen sein. Diese Abhängigkeitenliste wird natürlich nicht vom Compiler ausgegeben (die für *queues.o* dagegen schon), sondern durch Nachbearbeitung die Ausgabe des Compilers mit dem Shellskript-Fragment in Zeile 34 erzeugt.

3.3 *Makefile.special*

```
 5  all: testschedule
 6
 7  CXX       = g++
 8  CPPFLAGS  =
 9  CXXFLAGS  = -Wall -g
10  LDFLAGS   =
11  LOADLIBES =

13  CXX-COMPILE = $(CXX) $(CPPFLAGS) $(CXXFLAGS) -c -o $@ $<
14  CXX-LINK    = $(CXX) $(CXXFLAGS) $(LDFLAGS) -o $@ $^ $(LOADLIBES)

16  %.oo: %.cc %.hh
17          $(CXX-COMPILE)

19  %.OO: %.cc
20          $(CXX-COMPILE)

22  %: %.OO
23          $(CXX-LINK)

25  %: %.cc                             # cancel this rule

27  queues.oo        : jobs.hh
28  testschedule.OO  : queues.hh jobs.hh
```

[6] Das ist nicht so überraschend. Wer, wenn nicht der C-Compiler, sollte sonst über Informationen verfügen, welche Dateien im Laufe einer Übersetzung gelesen werden

```
29  testschedule     : queues.oo jobs.oo

31  clean:
32          rm -f *.oo *.OO testschedule *~
```

Dieses Makefile ist wieder von *Makefile.tricky* abgeleitet. Hier wird versucht, anhand der Endung einer Objektdatei eine unterschiedliche Behandlung für gewöhnliche Module, für die Schnittstellendefinitionen in einer Headerdatei vorliegen müssen, und Hauptmodulen, für die dies nicht so sein muss, zu implementieren. Diese Unterscheidung wird hier an der Endung der erzeugten Objektdatei festgemacht: Gewöhnlich Module enden auf „.oo“ während Hauptmodule auf „.OO“ enden. Die Regeln sind so gestaltet, dass für die Erzeugung einer auf „.oo“ endenden Datei sowohl eine auf „.cc“ endende Datei *als auch* eine entsprechende auf „.hh“ endende Datei nötig ist (Zeilen 16-17). Liegt für ein gewöhnliches Modul keine Headerdatei vor, wird der Übersetzungsprozess immer scheitern.

4 Eingebaute Make-Regeln und Konventionen

Zum Abschluss möchte ich noch darauf hinweisen, dass die meisten Make-Varianten bereits eingebaute Regeln haben, welche sie kennen, ehe sie beginnen, das vom Benutzer definierte Makefile zu lesen. Beispielsweise hat *GNU Make* die folgende Regel bereits eingebaut:

```
%.o: %.cc
        $(CXX) -c $(CPPFLAGS) $(CXXFLAGS)
```

Die Struktur der eingebauten Regel und die verwendeten Namen (*CXX*, *CPPFLAGS*, *CXXFLAGS*) entsprechen dabei gewissen Konventionen, die auch für alle anderen eingebauten Regeln gelten: z. B. ist *CC* immer der C-Compiler (und der Linker), *CFLAGS* die an diesen Compiler übergebenen Flags.

Die eingebauten Regeln sparen zum einen Arbeit, beispielsweise hätten wir statt *Makefile.4* nur das Folgende schreiben müssen:

```
 5  all: testschedule

 7  CXX      = g++
 8  CXXFLAGS = -Wall -g
 9  CC       = g++          # enforce linking with g++

11  jobs.o          : jobs.hh
12  queues.o        : queues.hh jobs.hh
13  testschedule.o  : queues.hh jobs.hh

15  testschedule    : queues.o jobs.o

17  clean:
18          rm -f *.o testschedule *~
```

Andererseits kann das Vorhandensein der eingebauten Regeln natürlich eigene Regeln behindern oder zu ungeplanten Resultaten (Abkürzungen) führen. Aus diesem Grund wird in den *trickreichen Makefiles* die Regel „%.o: %.cc“ explizit abgeschaltet, indem sie *ohne* Befehlsteil aufgeführt wird. Will man, dass *GNU Make* alle eingebauten Befehle vergisst, muss man es mit der Option *-r* aufrufen.

(Übung für den Leser: Warum die Zuweisung in Zeile 9? Was passiert, wenn diese weggelassen wird?)

Sehr viel mehr Informationen über die angesprochenen Themen finden sich im *GNU Make Manual* (Stallman und McGrath, 2000) und in *Recursive Make Considered Harmful* (Miller, 1997).

Literatur

[Miller 1997] MILLER, Peter: *Recursive Make Considered Harmful*. 1997. – URL `http://www.tip.net.au/~millerp/rmch/recu-make-cons-harm.html`

[Stallman und McGrath 2000] STALLMAN, Richard M. ; MCGRATH, Roland: *GNU Make: A Program for Directing Recompilation: GNU make Version 3.79*. 59 Temple Place - Suite 330, Boston, MA 02111, USA: Free Software Foundation, Inc. (Veranst.), 2000. – URL `http://www.gnu.org/manual/make-3.79.1/make.html`