Institut für Softwaretechnologie
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3345

SCR-Spracherweiterung für Enterprise Architect

Wolfgang Fellger

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. Erhard Plödereder
Betreuer:	Dipl.-Inf. Bernd Holzmüller
begonnen am:	12. Juni 2012
beendet am:	12. Dezember 2012
CR-Klassifikation:	C.3, D.2.1, D.3.1, D.3.4

Abstract

„Software Cost Reduction“ is a method developed by David Parnas in 1978 to develop both formal and compact specifications for software systems. After several uses in the general vicinity of the US military, this approach has become mostly forgotten.

This thesis picks up the SCR specification language again and adapts it for use with the modern UML modelling tool Enterprise Architect. Along with the UML language profile, a compiler and code generators for C# and ANSI C are being developed.

Zusammenfassung

„Software Cost Reduction“ ist ein 1978 von David Parnas entwickelter Ansatz, um Spezifikationen für Steuergeräte formal prüfbar und gleichzeitig kompakt zu formulieren. Nach diversen Einsätzen im Umfeld des US-Militärs ist der Ansatz inzwischen weitgehend in Vergessenheit geraten.

Diese Arbeit greift die SCR-Spezifikationssprache auf und passt sie für den Einsatz mit dem modernen UML-Werkzeug Enterprise Architect an. Neben dem Sprachprofil werden hierzu ein Compiler und Codegeneratoren für C# und ANSI-C entwickelt.

Inhaltsverzeichnis

Kapitel 1

Einführung

Motivation

Seit über dreißig Jahren[1] ist bekannt, dass Fehler in frühen Phasen eines Softwareprojekts überpropertionale Kosten verursachen – das Phänomen lässt sich in der ikonischen „Badewannenkurve“ illustrieren [LL07, Kapitel 4]. Auch intuitiv ist verständlich, dass ein vergessenes und vom Compiler sofort bemerktes Semikolon in wenigen Sekunden korrigiert ist, während eine vergessene Anforderung unter Umständen die gesamte Architektur der Software ins Wanken bringt – das Projekt wird auf null zurückgesetzt.

Seit ähnlich langer Zeit werden daher Ansätze entwickelt, die Qualität der Anforderungsanalyse zu erhöhen. Dies gestaltet sich dadurch schwierig, dass Menschen nur ein begrenztes Auffassungsvermögen haben, auf Abstraktionen und Weglassen unnötiger Details angewiesen sind – tatsächlich muss also bei jeder Spezifikationstechnik ein *Kompromiss* zwischen Formalität, Ausdrucksstärke und kompakter Notation gefunden werden. (Auch Programmcode ist eine formale Formulierung des Systemverhaltens – der Kunde wird sich ein solches „Anforderungsdokument“ aber zurecht nicht verkaufen lassen!) Gesucht sind also formale Notationen hoher Ausdrucksstärke, die begrenzt auf die entscheidenden Teile der Software – also die eigentliche Programmlogik – so überschaubar sind, dass sie mit dem Kunden diskutiert werden können.

Trotz all dieser Forschungsarbeit ist in der allgemeinen Softwareentwicklung wenig von den Ergebnissen angekommen; im Embedded-Bereich konnten sich kommerzielle Werkzeuge wie *SCADE* und *Esterel* verbreiten. Die zusätzlich vertriebenen Codegeneratoren dieser Systeme sind noch für ein mittelständiges Unternehmen eine größere Investition. Andere Ansätze, wie die Weiterentwicklung der SCR-Methodik durch Heitmeyer [Heitmeyer07], unterliegen Ausfuhrbeschränkungen.

Diese Arbeit greift die SCR-Methodik von Parnas und Heninger auf und modifiziert sie für den Einsatz mit UML. Anschließend werden ein Compiler und Codegeneratorframework entworfen und beispielhaft für die Ausgabesprachen C# und C implementiert.

Die SCR-Methodik

Die ursprüngliche SCR-Methodik wurde 1978 von David Parnas und dem US Navy Research Laboratory entwickelt [Heninger] und fußt auf zwei Pfeilern:

[1] Barry Boehm beschrieb diesen Zusammenhang spätestens 1981; die Erkenntnis ist vermutlich noch wesentlich älter.

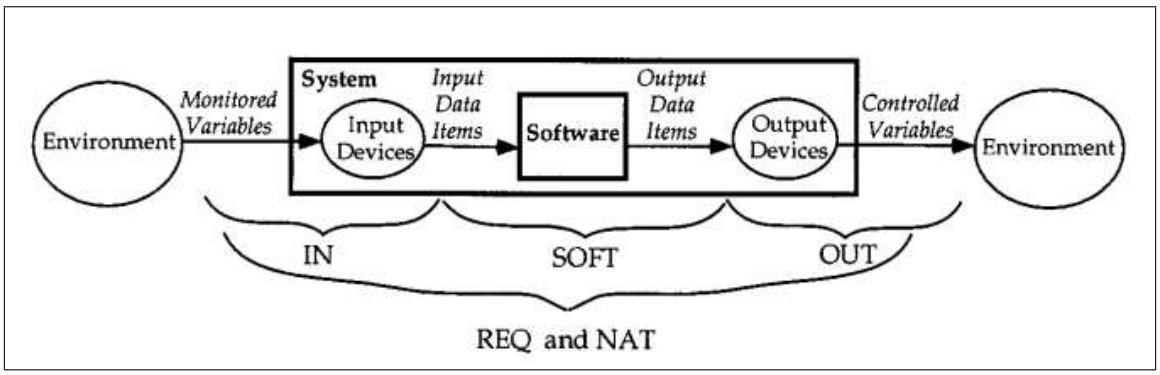


Abbildung 1.1: Vier Variablen-Modell. [Santillan]

- Eine formale Spezifikationsmethode, die sich auf die mathematische Abbildung zwischen (idealisierten) Eingaben und Ausgaben konzentriert
- Richtlinien, um die Qualität einer Spezifikation zu beurteilen.

Diese Methode wurde knapp zwei Jahrzehnte später von Parnas und Madey zum „Vier-Variablen-Modell“ generalisiert [ParnasMadey]. Dieses unterscheidet zwischen vier Mengen von Variablen und vier Relationen zwischen ihnen:

monitored Eine Umgebungsbedingung, die das System überwacht (z.B. Temperatur).

controlled Eine Umgebungsbedingung oder ein Aktuator, auf den das System Einfluss hat (z.B. Heizungsventil).

input Eine (idealisierte) Repräsentation einer *monitored variable* (z.B. Zahl zwischen -20 und +50 in ℃).

output Eine Repräsentation des gewünschten Wertes bzw. der Aktion, um eine *controlled variable* zu steuern (z.B. „Ventil öffnen“, „Ventil schließen“).

NAT „Natürliche“ Beziehungen (tatsächlich Annahmen) zwischen monitored- und controlled variables (z.B. Öffnen des Heizungsventils führt zu Temperaturanstieg).

REQ Gewünschte Beziehungen zwischen monitored- und controlled variables, also welchen Zustand das System *herstellen* soll (z.B. Temperatur auf 20℃ halten).

IN Abbildung zwischen monitored- und *input variables* mit Angabe der Toleranzgrenzen.

OUT Abbildung zwischen *output*- und controlled variables mit Angabe der Toleranzgrenzen.

Wichtigste Erkenntnis ist, dass in diesem Ansatz *idealisierte* Eingaben und Ausgaben betrachtet werden, also nur die Steuerlogik beschrieben wird. Zwei zusätzliche Relationen, von und zu den tatsächlichen Sensoren bzw. Aktuatoren, sind erforderlich, um ein komplettes Softwaresystem zu beschreiben. **NAT** folgt primär aus dem Entwicklungsprozess der Hardware (*welche* Sensoren und Aktuatoren sind auf Basis der Naturgesetze geeignet), während **REQ** die Spezifikation der Steuereinheit darstellt. Die folgenden Teile dieser Arbeit konzentrieren sich auf diese letzte Relation.

Um das Systemverhalten formal und in einer Form zu erfassen, die gut maschinenprüfbar ist, wird interner Zustand durch endliche Automaten und boolesche oder arithmetische *Terme* ausgedrückt. Dabei kommen drei Arten von Tabellen zum Einsatz:

Condition Tables beschreiben eine Systemvariable als Funktion eines Zustandsautomaten und Bedingungen. Siehe dazu das folgende Beispiel:

Mode *Pressure*	**Conditions**	
High, Permitted	True	False
TooLow	Overridden	Not Overridden
Value =	Off	On

Der aktuelle Zustand des angegebenen Automaten wählt die Zeile; die erste zutreffende Bedingung wählt die Spalte. Trifft keine Bedingung zu, behält die Variable implizit ihre vorherige Belegung.

Obige Tabelle definiert also, dass der Wert der Variablen in den Zuständen „High“ und „Permitted“ immer `Off` ist, im Zustand „TooLow“ hingegen `On`, außer das Prädikat `Overridden` (das ein weiterer Term oder eine Eingabe sein kann) ist wahr.

Event Tables beschreiben eine Systemvariable als Funktion eines Zustandsautomaten und eines *Ereignisses*, d.h. der Flanke einer Bedingung. Hierzu wird eine temporale Funktion @T für den Übergang zu *wahr* bzw. eine Funktion @F für den Übergang zu *falsch* eingeführt.

Mode Transition Tables sind ein Spezialfall von Event Tables, die die Übergangsfunktion eines endlichen Automaten darstellen.

Die zweite wichtige Eigenschaft liegt in den Ausdrücken selbst: Neben den aus Programmiersprachen bekannten booleschen und arithmetischen Konstrukten gibt es temporale Primitive, mit denen zeitliche Zusammenhänge kompakt spezifiziert werden können. So wird etwa `Duration(a) > 3s` wahr, wenn die Bedingung a in den letzten drei Sekunden durchgehend erfüllt war.

In der praktischen Anwendung hängt dieser Ansatz stark von der unterstützenden Software ab; die einzigen dem Autor bekannten Implementierungen sind *CoRE* [FaulkBracket] und das *SCR-Toolset* von Heitmeyer [Heitmeyer07]. Beide unterliegen Ausfuhrbeschränkungen der amerikanischen Streitkräfte und sind nicht kommerziell verfügbar.

Problemstellung

Die SCR-Methodik wurde ursprünglich für das US-Militär entwickelt; nachdem sie in den 1980er-Jahren mehrfach erfolgreich eingesetzt wurde, ist sie inzwischen in Vergessenheit geraten und außerhalb der USA praktisch unbekannt.

Die Arbeit geht zurück auf den Wunsch der ICS AG, die SCR-Methodik ergänzend zu SCADE für ein Projekt einzusetzen. Dies war aufgrund von Ausfuhrbeschränkungen für das am Naval Research Lab entwickelte Toolset zunächst nicht möglich. Bei den weiteren Vorbereitungen wurde dann ersichtlich, dass eine grafische Darstellung insbesonders der Mode Transition Tables die Kommunikation mit dem Kunden erleichtert. Für diesen Zweck bietet sich UML an, da sowohl die Darstellung auf Seiten des Kunden und die Modellierungswerkzeuge auf Seiten des Entwicklers bekannt sind.

Diese Arbeit verfolgt daher das Ziel, die ursprüngliche SCR-Notation nach UML zu portieren, und die Grundzüge eines neuen SCR-Toolsets zu entwickeln. Bei diesem Prozess bietet sich eine vorsichtige Modernisierung der Sprache an, die anschließend im praktischen Einsatz auf ihren Nutzen geprüft werden kann.

Konkret sollten neben einem UML-Profil ein Compiler mit Frontend für das UML-Modellierungswerkzeug Enterprise Architect sowie Backends für die Sprachen C# und optional ANSI-C entwickelt werden. Dabei wurde eine zumindest rudimentäre Integration des Compilers mit Enterprise Architect gewünscht, so dass aus dem Werkzeug heraus auf Fehler geprüft und Code generiert werden kann.

Als Implementierungssprache wurde C# 4.0 aufgrund des Sprachumfangs und der guten Integration von COM gewählt.

Enterprise Architect

Enterprise Architect ist ein von Sparx Systems vertriebenes kommerzielles UML-Werkzeug. Es unterstützt die UML 2.0, Round-Trip-Engineering und benutzerdefinierte UML-Profile. Außerdem kann es durch Plugins erweitert werden.

Modellierungsprojekte werden in sogenannten „Repositories“ organisiert, die eine Hierarchie von Paketen und UML-Elementen sowie verschiedenste Diagrammtypen enthalten können. Wie in der UML üblich stellen Diagramme eine Sicht auf die modellierte Struktur dar, es können also N:M-Beziehungen zwischen Diagrammen und Elementen bestehen.

Component Object Model (COM)

Das Component Object Model ist eine von Microsoft entwickelte Technik, die objektorientierte Kommunikation über Programm- und Programmiersprachengrenzen hinweg ermöglicht. Ihr bekanntester Einsatz ist in der sogenannten Automatisierungs-Schnittstelle der Microsoft-Office-Pakete, so dass Skripten und Plugins sowohl innerhalb wie außerhalb des Prozesses dasselbe Datenmodell und die selbe Funktionalität angeboten werden kann.

Einen ähnlichen Weg geht Enterprise Architect, der sein Datenmodell ebenfalls sowohl Plugins wie auch externen Prozessen zur Verfügung stellt.

Das .NET-Framework von Microsoft ist eine Weiterentwicklung des COM, weshalb alle .NET-Sprachen eine hervorragende Integration mit COM-Schnittstellen ermöglichen.

C#

C# ist eine objektorientierte Programmiersprache für die .NET-Umgebung. Sie greift Konzepte aus Java, C++ und Object Pascal auf; in den Versionen 3 und 4 kamen schließlich funktionale Anteile entlehnt aus Haskell dazu.

C# ist stark typisiert, unterstützt generische Typen sowie Lambda-Funktionen mit Closures, und verfügt mit LINQ über ein Framework, das die funktionalen Anteile unter anderem für komfortable Listenverarbeitung nutzt.

Harel-Diagramme / State Charts

State Charts formalisieren und erweitern die in der theoretischen Informatik gebräuchliche Darstellung von regulären Automaten um verschachtelte, parallele und diverse Pseudo-Zustände. Die Zustandsübergänge werden mit *Transitionen*, gerichteten Kanten zwischen zwei Zuständen, dargestellt. Sie können (und im Falle dieser Arbeit müssen) über einen *Wächterausdruck* (Guard) verfügen; eine Transition kann nur verfolgt werden, wenn dieser Ausdruck wahr ist.

Verschachtelte Zustände erlauben es, einen Zustand in einen weiteren Zustand einzubetten. Dieser wird dadurch ein sogenannter *Composite State*. Eine Maschine muss sich

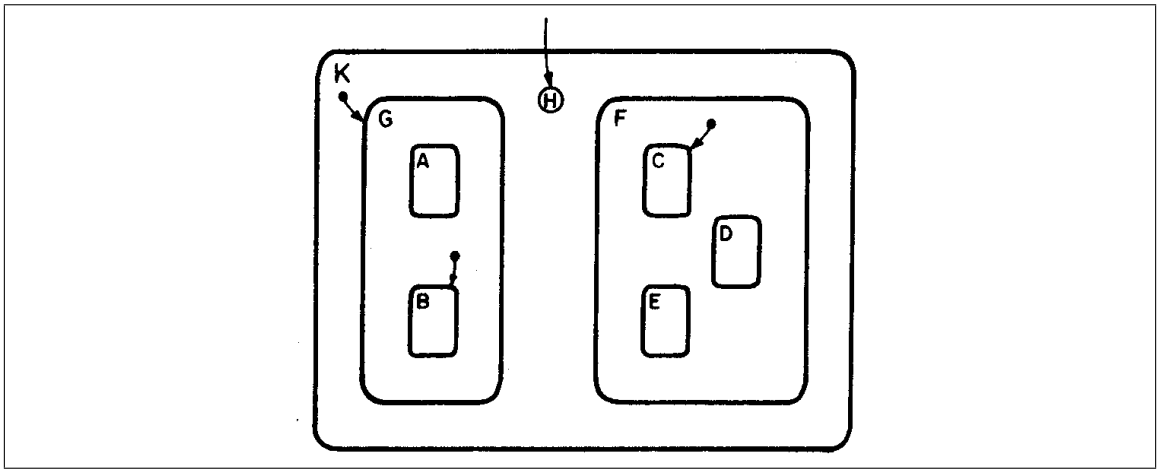


Abbildung 1.2: Beispiel für ein Harel-Diagramm (aus [Harel86])

jedoch immer in genau einem „einfachen“ Zustand (ohne weitere Kindzustände) befinden; dies impliziert dann, dass sie sich auch in allen übergeordneten Zuständen befindet. In obigem Diagramm befindet sich eine Maschine im Zustand D gleichzeitig auch in F und K.

Pseudozustände werden von Harel eingeführt, um das Modell um Funktionen zu erweitern, die in einem Zustandsdiagramm schlecht auszudrücken sind. Für diese Arbeit relevant sind dabei nur zwei Typen: Der *Initial*-Zustand und die *History*-Zustände H und H*. Der Initialzustand ermöglich es, auch Kompositzustände mit einer Transition anzuspringen. Dabei erfolgt dann automatisch ein Wechsel in den mit dem Initialzustand markierten Unterzustand. Im Beispiel würde mit einer Transition nach K in G und rekursiv in B gewechselt werden.

Die Zustände H und H* speichern den Zustand beim Verlassen des sie umgebenden Kompositzustands. Werden sie angesprungen, stellen sie diesen Zustand wieder her. Hierbei stellt H nur den unmittelbaren Kindzustand wieder her (*flache Historie*), während H* die gesamte Konfiguration wieder herstellt (*tiefe Historie*). Ist im Beispiel etwa D aktiv, daraufhin wird K verlassen und anschließend H angesprungen, wird (nur) der Zustand F wiederhergestellt. Das System befindet sich darauf im Zustand C, da dieser der Initialzustand von F ist. Wenn der H-Knoten im Beispiel H* wäre, würde hingegen die gesamte Konfiguration {K, F, D} wiederhergestellt.

Für die anderen Erweiterungen, die mit Harel-Diagrammen eingeführt wurden, wird auf [Harel86] und [SMD] verwiesen.

Anforderungen für eingebettete Systeme

Eingebettete Systeme müssen häufig auch zuverlässige Systeme sein – sei es, weil die Komponente eine sicherheitskritische Funktion übernehmen soll (etwa die Steuerung für einen Bahnübergang) oder einfach nur das Ausbringen von Fehlerkorrekturen erheblich teurer ist als bei üblicher Desktopsoftware (z.B. die Bedienoberfläche eines Autoradios).

Neben den häufig ebenfalls mitschwingenden Anforderungen für garantierte Reaktionszeiten (Echtzeitsysteme), ist eine der Charakteristika für zuverlässige Software der Dauerbetrieb. Im Gegensatz zu Desktopsoftware (und häufig dem Computer) wird die Software also nicht gelegentlich neu gestartet, sondern muss im Extremfall über Jahrzehnte soweit funktionieren, dass sie sich aus jeder Fehlersituation erholen kann. Einer der kritischen Punkte in dieser Beziehung ist die dynamische Allokation von Speicher. Nicht nur ist es schwer, eine feste Obergrenze zu beweisen, bei Allokation mit unterschiedlichen

Blockgrößen ist das Fragmentieren des Hauptspeichers ohne sogenannte Kompaktifizierung unvermeidbar[2], so dass Laufzeiteigenschaften und Speicherverbrauch mit steigender Ausführungszeit schlechter werden.

Idealerweise ist also ein System mit konstantem Speicherverbrauch zu konstruieren. Praktisch bedeutet dies etwa in C#, dass der `new`-Operator nach Abschluss der Initialisierungsphase verboten ist – eine für Desktopsoftware völlig ungewohnte Anforderung.

Für unseren Fall steht diese Anforderung im Konflikt mit dem Wunsch nach Unendlichkeitsarithmetik – das System muss für jede Variable und für jedes Zwischenergebnis einen Speicherblock fester Größe allokieren, der bei einem Überlauf nicht dynamisch „wachsen“ kann, wie dies etwa in libgmp[3] oder Python realisiert ist. Auch die Zähler, mit denen ein `Duration`-Ausdruck implementiert wird und nicht zuletzt der Speicher, der vergangene Werte einer Variable hält, muss statisch zu begrenzen sein. Dies fordert Kompromisse im Sprachdesign, die im nächsten Kapitel diskutiert werden.

Abgrenzung

In dieser Arbeit kann nur der Grundstein eines neuen SCR-Toolsets geschaffen werden. Dies umfasst die Definition der Sprache und Entwicklung eines vollständigen Compilers. Im Zweifelsfall liegt der Fokus auf einem benutzbaren (und dazu hinreichend fehlerfreien) Ganzen, nicht auf möglichst vielen Funktionen.

Nicht Bestandteil der vorliegenden Arbeit sind daher assistierende Werkzeuge für die Modellierung oder Validierung der Programme. Ebenfalls verzichtet wird auf arbeitsintensive Maßnahmen, die hauptsächlich für eine Zertifizierung des Codegenerators erforderlich sind – vor allem der formale Korrektheitsbeweis für jedes Sprachkonstrukt in beiden Codegeneratoren. Stattdessen wird die Semantik einfach gehalten, um eine solche Zertifizierung später nicht zu sabotieren.

Ebenfalls verzichtet wird auf einen Interpreter zur interaktiven Ausführung im UML-Werkzeug oder einen Debugger. Beide Ziele werden näherungsweise durch den C#-Codegenerator und strukturerhaltende Codegeneration erreicht, so dass das Modell übersetzt und anschließend mit den gewohnten Werkzeugen debuggt werden kann.

Wir erlauben uns auch einige Freiheiten bei der Umsetzung der SCR-Methodik, so dass diese Umsetzung keine sogenannte „getreue“ ist. Wie bei Problemen des Sprachentwurfs üblich gibt es hier keine absolute Wahrheit, sondern höchstens eine bessere Eignung für bestimmte Zwecke. Eine kurze Stellungnahme dazu, ob sich diese Änderungen aus Sicht des Autors bewährt haben, ist im Fazit zu finden.

Ähnliche Arbeiten

Wie bereits beschrieben sind alternative Umsetzungen der SCR-Methodik die Suiten *CoRE* [FaulkBracket] und das *SCR-Toolset* von Heitmeyer [Heitmeyer07], die wieder selbst auf den ursprüngliche Artikel von Parnas und Heninger [Heninger] zurückgehen.

Mangels Verfügbarkeit dieser beiden Suiten beschränken sich die Anleihen dieser Arbeit auf die Dokumentation dazu. Weitere Anleihen werden bei SCADE und Simulink gezogen.

---

[2] Siehe etwa Kapitel 3 der Vorlesung „Real-Time Programming“ an der Universität Stuttgart; `http://www.iste.uni-stuttgart.de/ps/lehre/ueberblick/real-time-programming.html`.

[3] `http://gmplib.org/manual/Memory-Management.html`

Ideen für die Semantik, insbesonders die in Kapitel 3 diskutierten Alternativen bezüglich starker oder schwacher Semantik und Auswertungszeitpunkt, sowie der Einsatz von unendlicher Arithmetik, basieren auf einem im Vorfeld der vorliegenden Arbeit angefertigten internen Papier der ICS AG von Martin Sulzmann [Sulzmann].

Kapitel 2

Eine Anpassung des SCR-Modells

Das SCR-Entwicklungsmodell erreichte seinen Höhepunkt in den 1980er Jahren. In neuerer Zeit wird es im akademischen Umfeld vor allem durch die Arbeiten von Heitmeyer ([Heitmeyer02], [Heitmeyer07], [Santillan]) am Leben gehalten und weiter gelehrt; im praktischen Einsatz ist es jedoch weitgehend in Vergessenheit geraten. Die vorliegende Arbeit wagt eine Anpassung auf moderne Technologien. Hierzu werden praktische Erfahrungen von zwei Mitarbeitern der ICS AG genutzt, die Methodik anderen Ingenieuren zu vermitteln, Bernd Holzmüller und Martin Sulzmann. Kernpunkte der Anpassung sind die Portierung in ein UML-Profil, Einsatz von weniger, aber mächtigeren Konstrukten sowie der weitgehende Verzicht auf tabellarischer Darstellung von Berechnungen.

Der letzte Punkt mag provokativ sein; zumindest für Fallunterscheidungen ist die Tabellenform im Vergleich zu Aktivitätsdiagrammen auch einfacher zu handhaben. Die Umsetzung nach UML erforderte hier einen Kompromiss zugunsten der grafischen Darstellung, der unserer Ansicht nach aber unschädlich für die Kernpunkte der SCR ist.

Aus der ursprünglichen SCR unverändert übernommen wird hingegen

- das Taktungsmodell (diskreter Takt mit äquidistanter Schrittweite)
- die temporalen Operatoren `@T` und `Duration`
- der Verzicht auf Sprachmittel, die Schleifen ermöglichen; die Sprache ist also nicht turingmächtig, ein Takt terminiert in konstanter Zeit
- die Möglichkeit, Zeitangaben unabhängig von der Taktung zu treffen.

Ersatzkonstrukte für SCR-Tabellen

Der einfachste Schritt – und die initiale Idee für die UML-Syntax – ist es, die *Mode Transition Tables* durch verschachtelbare Automaten im Sinne von Harel [Harel86] zu ersetzen, die als *State Chart* auch Einzug in den Standardsprachumfang der UML gefunden haben. Die Übergangstabellen mögen kompakter sein, allerdings können die Automaten in der Übersichtlichkeit gut konkurrieren (vgl. Abbildung 2.1).

Condition Tables als kontrollierende Struktur für Ausgabevariablen werden durch das vertraute Konstrukt der if-then-else-Kette ersetzt. Wir heben die Anforderung auf, von genau einem Mode abzuhängen. Anstatt die Tabelle implizit zu einer totalen Funktion zu ergänzen, erzwingen wir einen unbedingten Zweig; in diesem kann dann explizit angegeben werden, dass der Wert aus dem letzten Takt übernommen werden soll.

Old Mode	Event	New Mode
TooLow	@T(WaterPres $\geq$ Low)	Permitted
Permitted	@T(WaterPres $\geq$ Permit)	High
Permitted	@T(WaterPres $<$ Low)	TooLow
High	@T(WaterPres $<$ Permit)	Permitted

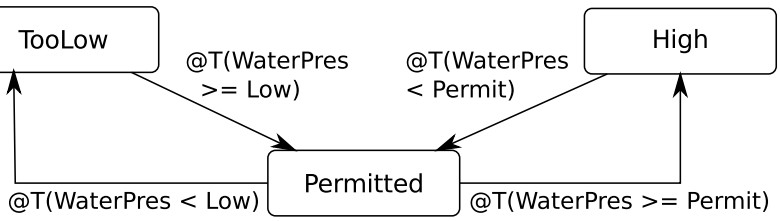


Abbildung 2.1: Automat dargestellt als Übergangstabelle (oben) und State Chart (unten)

Event Tables werden abhängig von ihrem Anwendungsfall durch eines der folgenden Konstrukte ersetzt:

- Dient sie nur zur Definition eines *Terms*, also einem zustandslosen Funktionsmakro, kommen **Definitions** zum Einsatz, die zusätzlich eine sinnvolle Gruppierung von Funktionen und Konstanten ermöglichen.
- Ist der Ausgabetyp eine Aufzählung, können sie dank der gelockerten Bedingungen für Abhängigkeiten durch einen Mode abgelöst werden.
- Bei anderen Ausgabetypen muss die Event Table durch einen Output realisiert werden. Dies hat den Nachteil, dass der Wert so nach außen sichtbar wird. Aus diesem Grund wurde gegen Ende der Arbeit kurzfristig noch das Element «Variable» eingeführt, das einem Output entspricht, aber nicht in der Schnittstelle veröffentlicht wird.

Abgesehen von ihrer gruppierenden Funktion werden Variablen für Zwischenergebnisse immer dann nötig, wenn sich das Datenflussmodel nicht als Baum darstellen lässt, also Schleifen enthält (siehe den in Abbildung 2.2 dargestellen Ausschnitt aus einem SCADE-Diagramm).

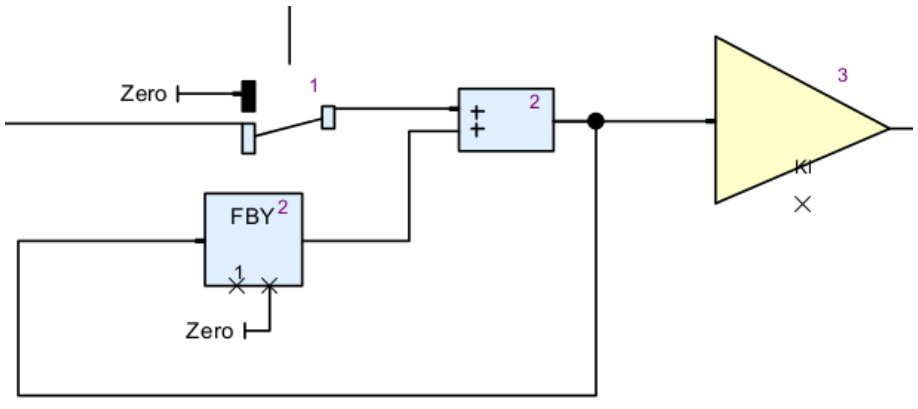


Abbildung 2.2: Komplexer Datenfluss, der eine Zwischenvariable erfordert

Erweiterungen des Sprachumfangs

Einige Bestimmungen der SCR heben wir auf; für die Erweiterung gilt:

- Beliebig viele Eingabewerte, Modes und Outputs dürfen ihren Wert in einem Takt ändern.
- Modes dürfen von allen Variablen des Systems abhängen, also auch von anderen Modes und Outputs.
- Outputs sind von allen anderen Variablen lesbar, insbesondere von sich selbst; das in der SCR immer implizite Halten eines Zustands wird so explizit möglich, wenn der Nutzer es wünscht.

Wie oben angedeutet werden die SCR-*Terms* in Form von Definitions umgesetzt und erweitert. Eine **Definition** ist eine Sammlung zusammengehöriger Konstanten und Funktionen, wobei die Funktionen im Gegensatz zur ursprünglichen SCR parametrisiert sein können.

Die temporalen Konstrukte werden erweitert; zusätzlich zu `@T` und `Duration` erlauben wir den Zugriff auf vergangene Werte jeder Variable mit den `Pre` und `PreN`-Konstrukten.

Zur Spezifikation von Zeitangaben wird ein spezielles Literal (`1.7s`) eingeführt. Rechenoperationen auf diesem Typ sind erlaubt, solange der Wert zur Compilezeit in Takte aufgelöst werden kann.

Automatische Typwahl

Für alle arithmetischen Ausdrücke wird der mögliche Wertebereich von den Eingabevariablen bis zu einem Vergleich oder einer Zuweisung als Intervall verfolgt (Bereichsinferenz). Dies erlaubt es, geeignete Typen für die Implementierung automatisch zu wählen; der Nutzer verwendet nur einen uniformen Integer-Typ mit optionaler Wertebereichsangabe.

Einschränkungen und Abgrenzung

Wie auch die ursprüngliche SCR definieren wir nur das reine Systemverhalten, also die Relationen **NAT** und **REQ** gemäß dem Vier-Variablen-Modell. Wir vernachlässigen die Abbildung von tatsächlichen Sensorwerten auf Eingabewerte (**IN**) ebenso wie den umgekehrten Weg für die Ausgabe (**OUT**). Daraus folgt, dass auch jegliche IO-Kommunikation und dazugehörige Fehlerbehandlung Aufgabe des Nutzers bleibt; der generierte Code ist eine Bibliothek, kein lauffähiges System.

Aus dieser Idealisierung folgt die Annahme, dass ein diskreter Takt und diskrete Werte zugrunde liegen, der Systemzustand ist also weder in der Zeit noch in der Belegung kontinuierlich. Gleitkommaarithmik schafft unserer Ansicht hier mehr Probleme als sie löst und wird daher nicht angeboten (abgesehen von den zur Compilezeit aufgelösten Zeitangaben).

Die eingesetzte Bereichsinferenz berücksichtigt keine Bedingungen und liefert daher unter Umständen eine sehr schlechte Abschätzung des möglichen Wertebereichs:

```
x: range 0..10
y: range 0..5

if (x <= 2)
    y = x^2        → inferierter Bereich 0..100, tatsächlich 0..4
else
    y = x / 2
```

Die Qualität der statischen Analyse ist also nicht hoch genug, um Bereichsfehler im Allgemeinen zu erkennen; sie liefert eine zu große („konservative“) Abschätzung des möglichen Wertebereichs eines Ausdrucks. Die Entscheidung fiel daher darauf, Zuweisungen mit einer nur teilweisen Überlappung der Wertebereiche zuzulassen. Lediglich wenn der mögliche Bereich eines Ausdrucks und der Zielbereich gar nicht überlappen, kann ein Fehler sicher gemeldet werden (in obigem Beispiel etwa `y = x + 6`). Um die Allokation von großen Speichermengen zu vermeiden, wird zur Laufzeit außerdem mit einem festen, größten möglichen Typ für Zwischenergebnisse gearbeitet. Die Konsequenz aus diesen beiden Entscheidungen ist, dass sowohl Bereichsverletzungen bei der Zuweisung als auch Überlauffehler bei den Zwischenergebnissen nicht ausgeschlossen werden können.

Um das System mit konstantem Speicheraufwand zu implementieren, sind noch einige weitere Einschränkungen nötig, die im folgenden Kapitel diskutiert werden.

Kapitel 3

Sprachbeschreibung

In diesem Kapitel wird beschrieben, wie die im ersten Kapitel dargestellten Aufgaben mit den im vorherigen Kapitel beschriebenen Änderungen als Sprache umgesetzt werden.

Anschließend werden die getroffenen Entscheidungen bezüglich der dynamischen Semantik der Sprache näher diskutiert.

Syntax und Notation

Dieser Abschnitt gibt einen Überblick über den Aufbau eines SCR-EA-Modells sowie der verfügbaren temporalen Ausdrücke. Eine detaillierte Definition der Syntax und Semantik findet sich in den Projektdokumenten.

Strukturelemente

Ein SCR-Modell besteht aus beliebig vielen der folgenden Komponenten:

Input repräsentiert eine Eingabe ins System, z.B. einen Sensorwert. Eingabevariablen sind typisiert (siehe Typsystem).

Definition gruppiert nutzerdefinierte Konstanten und Funktionen.

Mode Zustandsmaschine, die auch verschachtelt werden kann. Die History-Zustände `H` und `H*` werden unterstützt (siehe Einführung); *Events*, *Aktionen* und parallele Zustände gibt es nicht.

Ein Mode besteht aus Zuständen und Transitionen. *Transitionen* sind Bedingungen, die durch Ausdrücke formuliert werden. Der Automat ist deterministisch; besitzt ein Knoten mehrere Transitionen, werden diese nummeriert und die erste Transition mit wahrer Bedingung verfolgt.

Output berechnt einen Ausgabewert z.B. aus Inputs und Zuständen. Dies entspricht den Condition Tables im SCR-Ansatz, wobei auch hier die im Abschnitt Ausdrücke beschriebenen temporalen Beziehungen genutzt werden können (in der SCR Event Tables). Auch Ausgabewerte sind typisiert.

Variable für Zwischenergebnisse; entspricht strukturell einem Output, wird aber nicht in der Schnittstelle des generierten Moduls veröffentlicht.

Enumeration Ein nutzerdefinierter Aufzählungstyp.

Es ergibt sich die folgende Struktur als Pseudo-EBNF:

```
Modell = {EnumerationType} {Definition} {Variable}
         {Input} {Mode} {Output}
EnumerationType = Name: Name {, Name}
Input = Name: Type = Default_Value
Definition = {Constant} {Function}
Variable = Output
Mode = Name: Initial_State {State} {Transition}
Output = Name: Type = Default_Value, Output_Table

Constant = Name: Type = Expression
Function = Name: Type = {Parameter} -> Expression
State = Name [Initial_State {<Unterzustand>} {Transition}]
Transition = State [Number] Condition -> State
Output_Table = Expression |
               if Condition then Expression
               {else if Condition then Expression}
               else Expression

Parameter = Name: Type
Type = <Enum-Name> | "range" <min>".."<max> |
       "boolean" | "int" | "short" | "long"
```

Condition und *Expression* werden textuell notiert und im Abschnitt Ausdrücke behandelt, die übrigen Nichtterminale werden als UML-Diagramm umgesetzt.

Notation

Dieser Abschnitt beschreibt, wie die oben angegebene Syntax in UML abgebildet wird. Hierzu wird für alle oben definierten Nichtterminale jeweils die entsprechende UML-Struktur angegeben.

Für die Kompilierung werden alle Elemente unterhalb eines vom Nutzer bestimmten UML-Package (inklusive aller Unterpakete) herangezogen. Eventuell modellierte Beziehungen zwischen den Elementen sind rein informativ und werden nicht berücksichtigt. Elemente mit Stereotypen, die nicht den hier aufgeführten entsprechen, werden ignoriert.

EnumerationType

Modelliert als Klasse mit Stereotyp «enumeration» (UML-Enumeration). Attributnamen entsprechen Elementen der Aufzählung.

Input

Modelliert als Klasse mit Stereotyp «Input».

Name Klassenname

Type Typ des Attributs „Value“; Einheit aus Attribut „Unit“

Default_Value Initialwert des Attributs „Value“

Definition

Modelliert als Klasse mit Stereotyp «Definition».

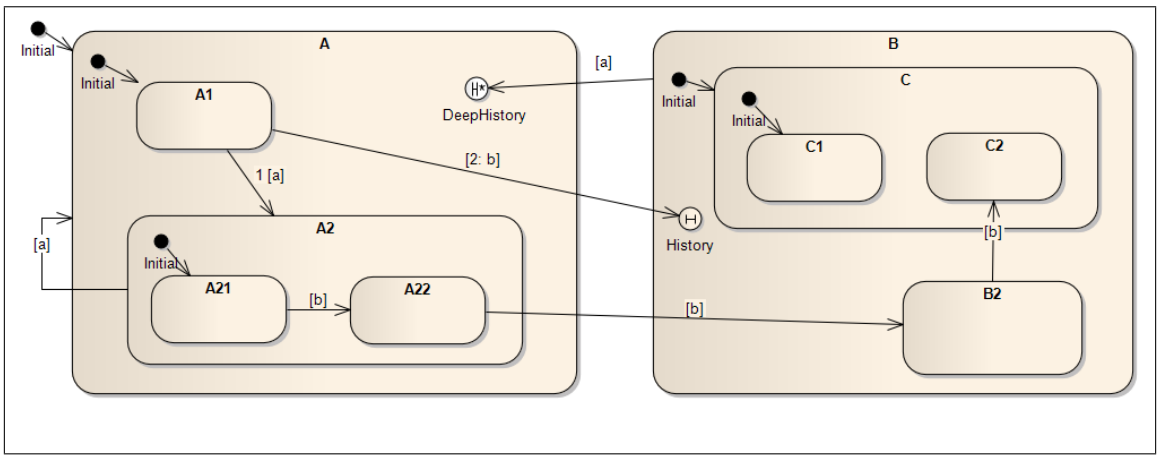


Abbildung 3.1: Modellierung eines Zustandsautomaten

Constant 'Const'-Attribut mit *Type* und Initialwert als *Value*

Function 'Operation' mit Ausdruck in 'Behavior', Rückgabetyp und Parameter wie üblich spezifiziert

Mode

Modelliert als Klasse mit Stereotyp «Mode» und verbundenem Zustandsdiagramm („composite“).

Das Klassenelement selbst trägt hier keine Semantik außer dem Elementnamen.

Es werden alle Elemente unterhalb des verbundenen Zustandsdiagramms herangezogen. Dabei ist unerheblich, ob sie in diesem Diagramm auch angezeigt werden.

Für die Struktur gelten rekursiv die folgenden Bedingungen (wobei die oberste Ebene wie ein **StateMachine**-Knoten behandelt wird):

StateMachine Enthält 1 bis n untergeordnete *State*- oder *StateMachine*-Knoten, genau einen *InitialState*-Knoten und jeweils höchstens einen *History*- bzw. *DeepHistory*-Knoten. Der Name des Zustands muss gesetzt sein.

Der Knoten darf beliebig viele ausgehende Transitionen (*StateTransition*) haben; andere Verbindungen oder Verbindungen zu Elementen außerhalb der Struktur werden ignoriert. Transitionen müssen eine Übergangsbedingung als Guard enthalten. Wenn mehr als eine ausgehende Transition vorhanden ist, müssen die Transitionen nummeriert sein. Verbindungen zu Elementen außerhalb der Struktur werden ignoriert.

Die Nummerierung kann wahlweise im Feld „Name“ der Transition oder mit dem Präfix `<Zahl>:` im Guard erfolgen. Die Transitionen werden in der Reihenfolge vom niedrigsten zum höchsten Wert ausgewertet.

State Enthält keine untergeordneten Elemente. Für Transitionen gelten dieselben Regeln wie bei *StateMaschine*.

InitialState Genau eine unbedingte Transition, die vom *InitialState*-Knoten zu einem Geschwisterelement vom Typ *State* oder *StateMachine* führt.

History Keine ausgehenden Transitionen.

Parallele Zustände sowie Eintritts- und Austrittsaktionen werden nicht unterstützt.

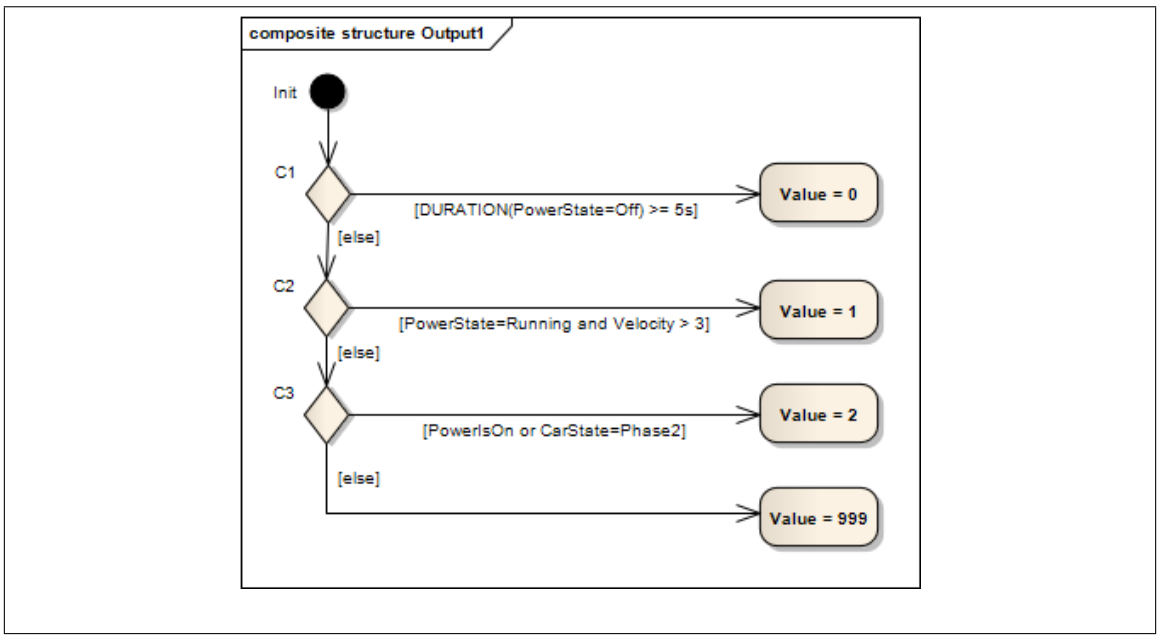


Abbildung 3.2: Modellierung einer if-then-else-Kette als Aktivitätsdiagramm

Output / Variable

Modelliert als Klasse mit Stereotyp «Output» und verbundenem Aktivitätsdiagramm („composite“).

Für dieses Konstrukt kann auch der Stereotyp «Variable» verwendet werden. Das Element wird dann nicht in die Schnittstelle des generierten Codes übernommen, sondern dient für interne Zwischenergebnisse. Aus Sicht des Programms ist das Verhalten identisch.

Name Klassenname

Type, Default_Value Typ und Initialwert des Attributs „Value“; siehe Input

Output_Table Modellierung der if-then-else-Kette als Aktivitätsdiagramm mit Struktur gemäß obiger Abbildung:

- Genau ein *ActivityInitial*-Knoten mit genau einer unbedingten Transition auf
  - ein einzelnes *Action*-Element **oder**
  - eine Kette von *Decision*-Knoten mit jeweils genau zwei ausgehenden *ControlFlow*-Elementen
- Jeder *ControlFlow* hat als Ziel entweder eine *Action* oder einen *Decisision*-Knoten. Eine Verzweigung zu zwei *Decision*-Knoten ist dabei nicht erlaubt.
- Der Name jedes *Action*-Knoten beginnt mit der Pseudozuweisung „Value =“; der Ausdruck kann wahlweise als Effekt oder direkt hinter dem Namen notiert werden.
- Die *ControlFlow*-Elemente notieren als Guard entweder `else` oder einen booleschen Ausdruck.

Ausdrücke

Ausdrücke haben die folgende Syntax (EBNF, Ausdrücke in `//` sind regulär):

```
Condition  = <Expression mit Typ boolean>
Expression = "if" Condition "then" Expression "else" Expression |
             SimpleExpr
SimpleExpr = SimpleExpr Binop SimpleExpr |
             Unop SimpleExpr |
             "(" Expression ")" |
             Zahlliteral | Sekundenliteral |
             Name | Name "(" Expression {"," Expression} ")"
Binop = "+" | "-" | "*" | "/" | "^" |
        ">" | "<" | "=" | "<=" | ">=" | "!=" |
        "and" | "or"
Unop = "-" | "+" | "not"
Zahlliteral = /[0-9]+/
Sekundenliteral = /[0-9]+(\.[0-9]+)?s/
Name = /[@a-zA-Z][_a-zA-Z0-9]*/
```

Operatorenpräzedenz und Semantik werden, soweit im Folgenden nicht anders beschrieben, an Pascal angelehnt. `^` ist ein Potenzoperator.

Als Bezeichner sind in Ausdrücken mit ihrem jeweiligen Namen zugänglich:

- Inputs, Modes, Outputs
- Nutzerdefinierte Konstanten und Funktionen
- Aufzählungswerte
- Die unten beschriebenen vordefinierten Funktionen
- In Funktionen die jeweiligen Parameter.

Die vordefinierten Funktionen sind `Abs`, `Max`, `Min` mit der üblichen Semantik sowie die folgenden temporalen Primitive:

`PreN(` *expr*, *const* `)` Wertet *expr* im Kontext des *const*-ten vorherigen Takts aus. Beispiele:

`PreN(B, 2)` der Wert, den der Ausdruck B im vorletzten Takt hatte

`PreN(KeyPressed, 1) and not KeyPressed` gdw. der Schalter in diesem Takt losgelassen wurde (wobei `KeyPressed` ein Input vom Typ `bool` ist).

Ergibt den Initialwert des Ausdrucks (d.h. unter Initialwerten der beteiligten Variablen), wenn das System noch keine *const* Takte läuft (entsprechend der „stationären Semantik“, siehe Abschnitt „Negative Zeit“).

`Duration(` *cond* `)` Gibt die Anzahl der Takte zurück, die *cond* bereits gilt. Beispiele:

`Duration(KeyPressed) = 0` gdw. der Schalter gerade nicht gedrückt ist

`Duration(KeyPressed) = 1` gdw. der Schalter in diesem Takt gedrückt wurde

`Duration(KeyPressed) > 4s` gdw. der Schalter seit mehr als vier Sekunden gedrückt gehalten wird

Kann einen Wert „unendlich“ zurückgeben, wenn die Bedingung seit Systemstart gilt.

Aus diesen werden die folgenden zusätzlichen temporalen Operatoren abgeleitet:

`Pre(` *expr* `)` Kurzform für `PreN(`*expr*`, 1)`.

`@T(` *cond* `)` reagiert auf die „steigende Flanke“ von *cond*, d.h. die Bedingung wurde in diesem Takt wahr (entspricht `not Pre(cond) and cond` bzw. `Duration(cond) = 1`).

`Since(` *cond* `)` Anzahl Takte, seit der die Bedingung das letze Mal galt (entspricht `Duration(not` *cond* `)`).

Die folgende Tabelle enthält Beispiele über den zeitlichen Verlauf von Ausdrücken. „Input“ sei eine Eingabevariable vom Typ `int` mit dem Standardwert von 0.

Takt	**1**	**2**	**3**	**4**	**5**	**6**
Input	4	5	6	4	2	2
Input >= 5	F	T	T	F	F	F
Pre(Input)	0	4	5	6	4	2
Duration(Input >= 5)	0	1	2	0	0	0
Duration(Input < 5)	inf	0	0	1	2	3
@T(Input >= 5)	F	T	F	F	F	F

Statische Prüfungen

Typsystem

Die folgenden Typen sind verfügbar:

1. boolesche Werte
2. Ganzzahlen mit Bereichsober- und Untergrenzen und einer optionalen Einheitenbezeichnung.
3. Aufzählungen
4. Zustandsmenge (Ergebnistyp von Mode)
5. Taktzahlen. Beginnen bei 0 und verfügen über den speziellen Wert „unendlich“

In Ausdrücken verwendete Bezeichner und deren Typkompatibilität werden statisch geprüft. Grundsätzlich sind nur namensgleiche Typen zuweisungskompatibel. Vom Nutzer können lediglich die Typen 1, 2 und 3 deklariert werden. Abgesehen von den Zustandsmengen (siehe unten) sind die erlaubten Operationen in der folgenden Tabelle aufgeführt. Alle Beziehungen sind symmetrisch.

Typ	boolesch	Aufzählung	Ganzzahlen	Taktzahlen
boolesch	`not`, `and`, `or`			
Aufzählung	keine	Vergleich		
Ganzzahlen	keine	keine	Vergleich, `+`, `-`, `*`, `/`, `^`	
Taktzahlen	keine	keine	Vergleich	Vergleich, `+`

Vergleichsoperatoren sind `<`, `>`, `=`, `>=`, `<=` und `!=`; der Rückgabetyp ist boolesch. Bei allen anderen Operatoren entspricht der Eingabe- dem Ausgabetyp.

Jede Variablendeklaration mit einer Aufzählung definiert einen eigenen Typ. Sie sind untereinander nicht zuweisungskompatibel. Jeder `Mode` deklariert implizit einen anonymen Aufzählungstyp, der die Namen aller Zustände enthält. Sein Ergebnistyp ist eine Menge über diesem Aufzählungstyp. Einzige erlaubte Operationen sind `=` und `!=` mit dem zugehörigen Aufzählungstyp, die als „ist Element von“ interpretiert werden.

Zur Compilezeit existiert außerdem der Hilfstyp „Zeitangabe“, der wie eine rationale Zahl (in Takten) behandelt wird. Er kann nur durch Sekundenliterale eingeleitet werden und wird vor der Codegenerierung aufgelöst. Dieser Typ kann nicht zugewiesen werden (also nicht als Rückgabewert, Funktionsparameter etc. verwendet werden). Alle arithmetischen und Vergleichsoperatoren außer `^` sind mit Zeitangaben und Ganzzahlen erlaubt; Rückgabetyp ist wieder eine Zeitangabe.

Ein Sekundenliteral wird mit der rationalen Taktanzahl ersetzt. Aus `2.05s` wird so bei zehn Takten pro Sekunde `20.5`. Der Ausdruck `2s + 2` besitzt somit die Semantik „2 Sekunden und zwei Takte“. Da Zuweisungen nicht erlaubt sind, muss ein Ausdruck vom Typ Zeitangabe letztlich verglichen werden. Hierbei wird der Wert des Ausdrucks abhängig von der Vergleichsoperation gerundet und durch die entsprechende Ganzzahl ersetzt. Ausdrücke vom Typ Zeitangabe müssen immer statisch bestimmbar sein, sonst ist ein Übersetzungsfehler auszugeben.

Der mögliche Bereich eines ganzzahligen Ausdrucks bildet einen Subtyp und verändert sich durch arithmetische Operationen (Details siehe Bereichsinferenz). Hierdurch ergibt sich für jeden ganzzahligen Ausdruck eine Menge möglicher Werte. Sind beide Operanden ganzzahlig, gelten folgende Regeln für Vergleich und Zuweisung (schwache Bereichsinferenz):

- Ist die rechte Seite einer Zuweisung keine Teilmenge der linken, wird eine Laufzeitüberprüfung eingefügt
- Ist der Schnitt von linker und rechter Seite einer Zuweisung oder eines Vergleichs leer, ist dies ein Übersetzungsfehler.

Taktzahlen werden nur vom Primitiv `Duration()` zurückgegeben. Sie haben eine Untergrenze von 0. Die Implementierung definiert eine Obergrenze sowie einen maximal darstellbaren Wert. Im Gegensatz zu den Ganzzahlen wird strikte Bereichsinferenz angewendet, d.h.:

- für Addition muss der mögliche Bereich im darstellbaren Bereich liegen.
- Beim Vergleich mit Ganzzahlen muss das Maximum des ganzzahligen Ausdrucks kleiner sein als die Obergrenze.

Vergleiche von Taktzahlen sind nur zwischen zwei Duration-Primitiven erlaubt. Sie werden so übersetzt, dass nur die Relation verfolgt wird und kein Überlauf möglich ist.

Bereichsinferenz

Jedem ganzzahligen Ausdruck wird ein Bereich möglicher Werte zugeordnet. Für Variablen gibt es wahlweise die anonyme Deklaration in der Form:

```
range <min>..<max>
```

oder die folgenden vordefinierten Typen:

	short	**int**	**long**
min	-2^{15}	-2^{31}	-2^{63}
max	$2^{15}-1$	$2^{31}-1$	$2^{63}-1$

Für Literale und Konstanten ist `min` = `max` = Wert des Ausdrucks.

Der neue Bereich nach Operationen ergibt sich gemäß den üblichen Regeln der Intervallarithmetik, so ist etwa `0..5 + 2..3 = 2..8`. Für die exakte Definition sei auf das Entwurfsdokument verwiesen.

Dynamische Semantik

Dieser Abschnitt beschreibt und begründet die getroffenen Entscheidungen für die dynamische Semantik der Sprache.

Arithmetik

Die Entscheidung gegen die übliche Wrap-Around-Semantik der C-ähnlichen Sprachen, also ein stiller Überlauf vom höchsten zum niedrigsten darstellbaren Wert eines Typs, war schnell getroffen (wobei C selbst das Verhalten nur für vorzeichenlose Typen festlegt).

Alle arithmetischen Operationen arbeiten nach der Unenedlichkeitssemantik, also dem üblichen Verhalten von Zahlen in $\mathbb{N}$. Ein passender Maschinentyp wird basierend auf den Ergebnissen der Bereichsinferenz pro Operation gewählt. Eine Implementierung darf einen maximalen Typ für Zwischenergebnisse definieren, dieser muss mindestens den Wertebereich von `long` haben. Bei einer Verletzung ist ein Laufzeitfehler zu generieren.

Negative Zeit

Grundsätzlich kann der Systemzustand als ein *Trace* beschrieben werden, also eine endliche Abfolge der Belegungen $B_1, B_2, \ldots, B_n$ für alle Variablen. Eine solche Belegung kann aus den aktuellen Eingaben und dem bisherigen Trace berechnet werden (ein *Takt*).

Allerdings ist es mit dem *Pre*-Primitiv möglich, über den Beginn des Trace hinauszublicken, also einen negativen Traceindex zu erreichen. Hierfür muss eine geeignete Semantik gefunden werden. Zur Wahl stehen grundsätzlich die *starke*, *schwache* und *stationäre* Semantik:

- Die schwache bzw. starke Semantik definieren eine Bedingung, in der negative Traceindizes vorkommen, pauschal als wahr bzw. falsch.
- Die stationäre Semantik fordert eine Ausgangsbelegung B_0, die sich in die Vergangenheit unendlich oft wiederholt.

Da wir `Pre()` in unserem Fall auf beliebige Ausdrücke, nicht nur Bedingungen, anwenden möchten ist die Auslegung von starker und schwacher Semantik allerdings unklar. Wir

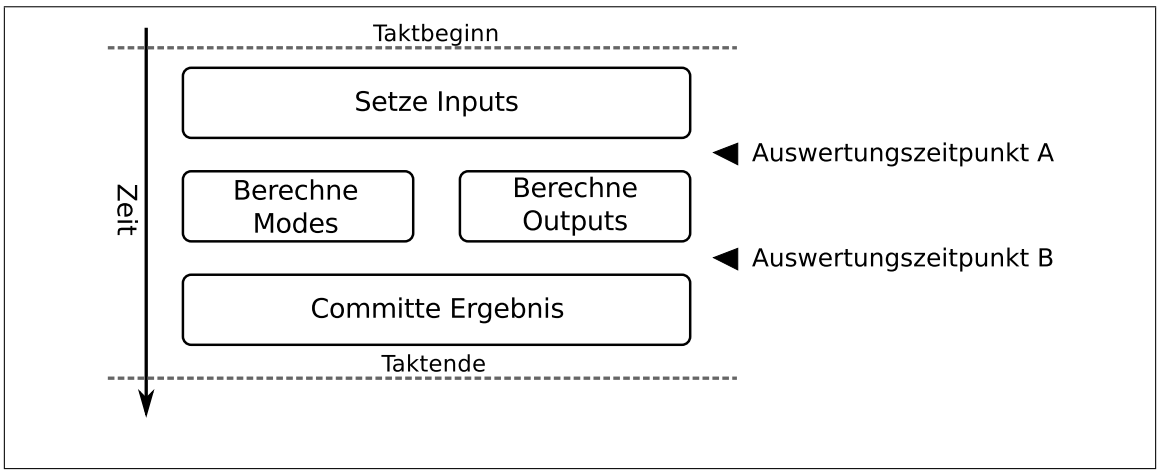


Abbildung 3.3: Zeitliche Abarbeitung eines Takts und mögliche Auswertungszeitpunkte

entscheiden uns daher für die stationäre Semantik, womit pro *Variable* ein Standardwert definiert werden muss. In vielen Fällen lassen sich die starke und schwache Semantik mit geeigneten Standardwerten simulieren. Von einer Erweiterung wie in SCADE, die es erlaubt, einen Standardwert pro *Ausdruck* anzugeben, wird für diese Version abgesehen.

Auswertungszeitpunkt

Betrachtet man den Systemzustand wie oben angegeben als Trace, so ergibt sich für die Variablenbelegung eine datenflussähnliche Semantik. Da im Gegensatz zur ursprünglichen SCR aber sowohl mehr als eine Variable pro Takt verändert werden und Variablen von anderen Variablen abhängen sollen dürfen, wird eine Entscheidung über den Auswertungszeitpunkt nötig (siehe Abbildung).

1. Datenflusssemantik (Zeitpunkt A): Alle Variablen erhalten als Eingabe den Systemzustand, bestehend aus dem bisherigen Trace und den neuen Eingaben. Von den anderen Variablen ist daher der Zustand aus dem letzten Takt (oder der Standardwert) sichtbar. Die Variablen können parallel oder in beliebiger Reihenfolge berechnet werden; allerdings verzögern sich Reaktionen um einen Takt pro Abhängigkeit.
2. SCADE-Semantik (Zeitpunkt B): Der neu berechnete Wert einer Variablen ist für andere Variablen unmittelbar, d.h. noch im selben Takt, sichtbar. Im Gegensatz zu obigem Ansatz ist dies nicht in allen Situationen möglich; der Übersetzer muss die Abhängigkeiten zwischen den Variablen zu einer korrekten Auswertungsreihenfolge auflösen und auf Zyklen prüfen. Dies lässt sich durch eine topologische Sortierung erreichen.

Tatsächlich besteht eine Abhängigkeit im Sinne der zweiten Semantik nur, wenn mit einem Zeitversatz von 0 auf die Variable zugegriffen wird; ansonsten ist der Wert bereits errechnet und die Auswertungsreihenfolge unerheblich. Damit kann die erste Semantik in der zweiten simuliert werden, indem alle Zugriffe auf eine Variable mit Pre() umgeben werden; sie ist also echt mächtiger.

Weiterhin stellte sich heraus, dass eine topologische Sortierung ohnehin implementiert werden musste, da Funktionen und Konstanten ebenfalls auf zyklische Abhängigkeiten zu prüfen sind. Daher fiel etwa zur Halbzeit der Arbeit die Entscheidung, auf die zweite Semantik umzustellen. Neben dem eigentlichen Ermitteln der Auswertungsreihenfolge waren dazu nur kleine Änderungen in den Codegeneratoren und der Laufzeitbibliothek

erforderlich (der berechnete Wert war vor der Änderung vor Lesezugriffen durch das Restsystem verborgen). Der größte Aufwand ergab sich durch Anpassung der Testfälle und schon existierender Dokumentation, insgesamt nahm die Umstellung aber nur etwa einen Arbeitstag in Anspruch.

Automatenmodell

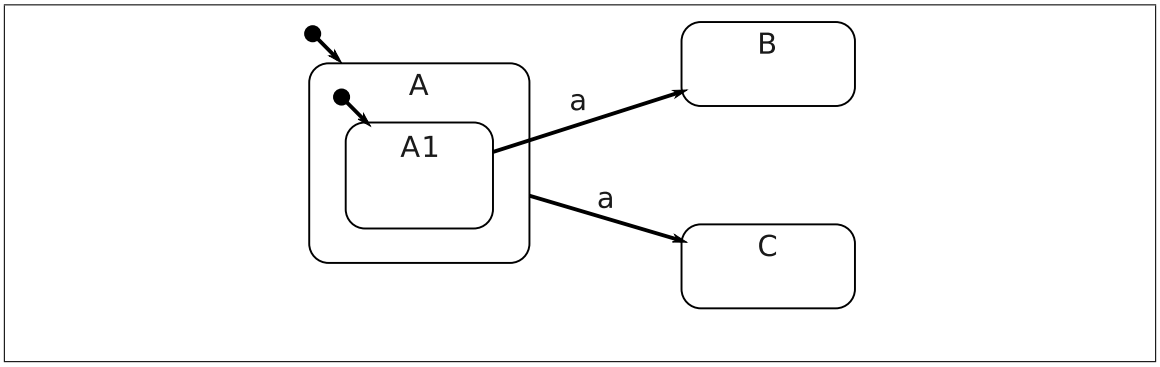


Abbildung 3.4: Einfacher Harel-Automat mit unverträglichen Zustandsübergängen

Die Semantik für verschachtelte Zustandsautomaten erwies sich als nicht so standardisiert wie erhofft, auch weil sich bei diesen Modellen unweigerlich Mehrdeutigkeiten ergeben, so dass eine „natürliche“ Semantik nur für die grundlegenden Konstrukte existiert und auch dann nichtdeterministisch ist. Der einfachste Fall ist eine Eingabe, die zwei unverträgliche Zustandsübergänge ermöglicht, siehe Abbildung 3.4. Im Allgemeinen ist nicht statisch zu erkennen, ob derartige Übergänge existieren. Diese Mehrdeutigkeiten werden durch parallele und History-Zustände noch weiter verschärft. Die UML versäumt es leider, eine formale Semantik für die Zustandsdiagramme anzugeben, und konzentriert sich nur auf die Syntax.

Die formalste Abhandlung zu diesem Thema ist immer noch das ursprüngliche Paper von Harel [Harel86]. Die Semantik ist allerdings auch hier nur kurz angerissen, es wird allerdings auf die Mehrdeutigkeiten und die Implementierung `STATEMATE` hingewiesen.

In Absprache mit dem Betreuer wurde daher entschieden, dass nur folgende vereinfachte Semantik implementiert wird, um die Mehrdeutigkeiten aufzulösen:

- Pro Takt findet deterministisch höchstens ein Übergang statt. Zu diesem Zweck werden alle Transitionen aufsteigend nummeriert. Die Bedingungen der Transitionen werden von den äußeren Zuständen nach innen und in Reihenfolge der Nummerierung ausgewertet; der erste Übergang mit wahrer Bedingung wird verfolgt. Ist keine Bedingung wahr, verbleibt der Automat im vorherigen Zustand. Obiger Automat würde also unter Eingabe `a` von A1 nach C übergehen.
- Ein History-Zustand innerhalb von Zustand *S* speichert den aktuellen Zustand (bei `H`) oder einen Snapshot aller Unterzustände (bei `H*`) in jedem Takt, in dem S aktiv ist. Wird der History-Zustand angesprungen, wird dieser gespeicherte Zustand wiederhergestellt (*S* ist danach in jedem Fall aktiv). Der Sprung in den History-Zustand zählt als Übergang im Sinne der obigen Definition, d.h. die Wiederherstellung erfolgt unmittelbar vor dem Ende eines Takts.
- Parallele Zustände sowie Eintritts- und Austrittsaktionen werden nicht unterstützt.

Kapitel 4

Umsetzung

Architektur

Die Architektur der Software folgt dem klassischen Compilerbau, das heißt es findet eine Zerlegung in *Frontend* (Einlesen und Parsen des Programms), *semantische Analyse* (Überprüfung und Transformation der Datenstruktur aus dem Frontend) sowie ein bis mehrere *Backends* (Code-Generierung) statt.

Eine Besonderheit der Sprache ist, dass sie in zwei große syntaktische Blöcke zerfällt:

a. Die grafisch notierten Strukturelemente sowie die
b. textuell notierten Ausdrücke.

Die beiden Teile können grob als Spezifikations- und Implementierungsteil einer Kompilationseinheit betrachtet werden, wie dies auch in anderen Sprachen, z.B. Ada, gebräuchlich ist. Für diesen Fall wird allerdings die Konstruktion von zwei separaten Parsern erforderlich.

Da Enterprise Architect als UML-Werkzeug vorgegeben ist, bietet sich für den strukturellen Teil eine Integration mit dessen Automatisierungs-Schnittstelle (COM) an. Dies ermöglicht es, die gleiche Schnittstelle von außerhalb wie innerhalb des Prozesses zu verwenden und macht es so einfach, sowohl ein Plugin als auch ein Kommandozeilenwerkzeug anzubieten.

Die Abhängigkeit von Enterprise Architect wird soweit möglich gekapselt, d.h. die Programmstruktur in eine eigene Struktur überführt. Das Datenmodell von Enterprise Architect wird also nur lesend genutzt und könnte durch ein anderes UML-Werkzeug oder Dateiformat ergänzt werden.

Für die Belange dieser Arbeit wird vor allem die Konstruktion von Testfällen leichter; da für jedes Sprachfeature ein Testfall vorgesehen ist, wären hier sonst Dutzende von separaten UML-Dateien zu erstellen. Auf diese Weise kann ein Programm komplett im Code erstellt oder verschiedene Varianten aus einer einzigen Vorlage generiert werden. Die von Enterprise Architect erzeugten Repository-Dateien sind für diesen Zweck relativ unhandlich (etwa 2 MB für ein neu angelegtes Dokument).

Für den textuellen Teil wird auf einen Parsergenerator zurückgegriffen. Die Entscheidung fällt auf das verbreitete Werkzeug [ANTLR], das mehrere Plattformen und Sprachen unterstützt, einen modernen grafischen Grammatikeditor enthält und direkt abstrakte Syntaxbäume (AST) generieren kann.

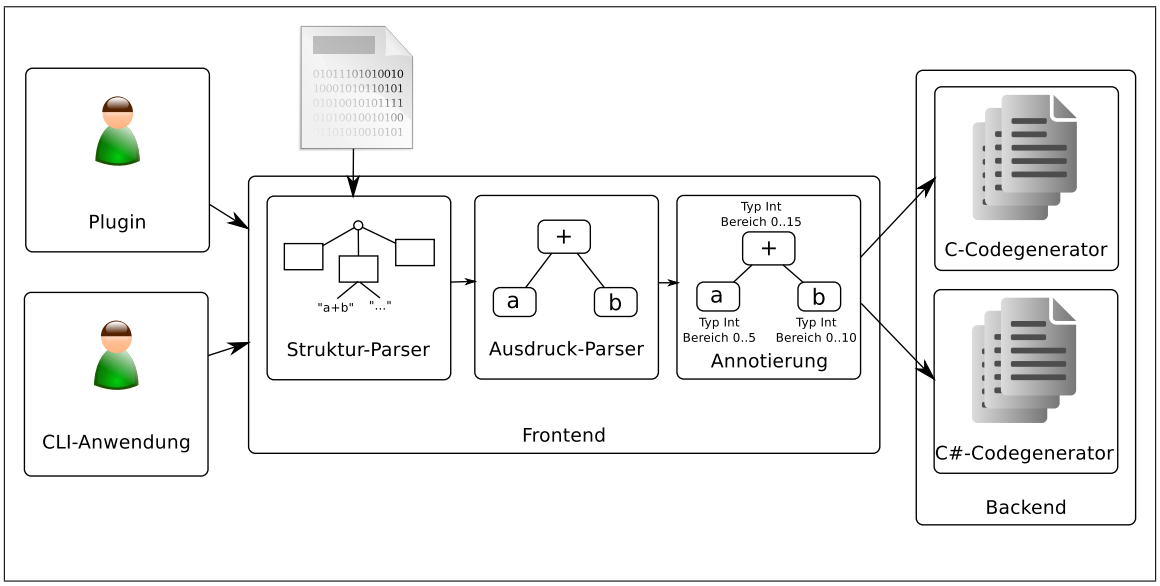


Abbildung 4.1: Komponenten und Module von SCR-EA

Für die semantische Analyse wird nach Erfahrungen aus dem Prototyp ein objektorientierter Ansatz gewählt, d.h. die einzelnen Knoten des AST können sich basierend auf ihrem Typ selbst annotieren.

Eine Übersicht und detailliertere Beschreibung folgt in den nächsten Abschnitten.

Komponenten

Das Projekt besteht aus den folgenden einzelnen Komponenten bzw. Programmen:

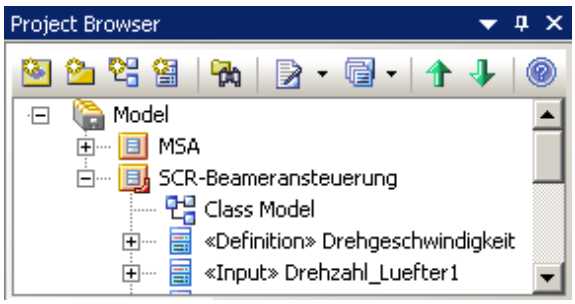
- Implementierung der SCR-EA-Sprache (Frontend)
- Codegenerator für ANSI-C (C90)
- Codegenerator für C# 4.0
- Compiler-Plugin für Enterprise Architect 7.5+ (`EASCRCompilerPlugin.dll`)
- Kommandozeilen-Compiler (`Compiler.exe`)

Nutzerschnittstellen

Die Compilerbibliothek benötigt die folgenden Schnittstellen und Informationen:

- eine Instanz von EA.Repository
- eine Diagrammdatei und darin den Bezeichner eines Wurzelpakets (siehe Abbildung 4.2)
- Ausgabemöglichkeit für Meldungen, die Kommunikation eines bestimmten EA-Elements erlaubt
- das Ausgabeverzeichnis für generierten Code
- ggf. Optionen, z.B. gewünschter Codegenerator

Abgesehen von den Meldungstexten muss die Bibliothek während der Kompilation keine Daten mit der Benutzerschnittstelle austauschen. Aufgrund dieser überschaubaren Interaktionen wird die Entwicklung von zwei verschiedenen Benutzerschnittstellen angestrebt:



Unter dem Wurzelknoten des Repositories („Model") befinden sich ein oder mehrere Top-Level-Pakete, welche das eigentliche Programm (Elemente und Diagramme) enthalten.

Abbildung 4.2: Grobstruktur eines Repositories in Enterprise Architect

Kommandozeilen-Anwendung Erhält Dateinamen, optional Wurzelpaket, Ausgabeverzeichnis und Optionen als Parameter; instanziiert Enterprise Architect über die angebotene COM-Schnittstelle. Ausgabe von Fehlermeldungen erfolgt auf die Standardausgabe.

Plugin für Enterprise Architect Startet die Kompilation auf einem vom Nutzer gewählten Wurzelpaket; Ausgabeverzeichnis und Optionen werden über einen Windows.Forms-Dialog abgefragt. Die Schnittstelle für Plugins ist eine Obermenge der externen COM-Schnittstelle und erfordert daher keine Änderungen im Compiler. Meldungen werden in eine Konsole von Enterprise Architect ausgegeben, dabei wird die Möglichkeit genutzt, sie mit den betreffenden Elementen zu verlinken.

Frontend

Die strukturellen Teile der Sprache werden gemäß der Beschreibung in Kapitel 3.1 in Datenklassen umgesetzt. Dem Autor ist kein Parsergenerator für diese Art Aufgabenstellung bekannt, daher erfolgt die Implementierung von Hand. Trotz der einfachen, vorhersehbaren Struktur liegt der Hauptaufwand in der Fehlerbehandlung. Eine Schwierigkeit besteht darin, dass sich ähnliche Modellierung in recht unterschiedlichen Datenstrukturen auswirken kann; beispielsweise unterstützt Enterprise Architect sowohl sogenannte Composite- als auch „eingebettete" Diagramme, die grafisch fast nicht unterscheidbar sind.

Als andere wesentliche Ursache für Verwirrung stellte sich heraus, dass in der UML Diagramme nur als einfache Views definiert werden (also ohne eigene Semantik), die eigentlich maßgebliche Elementhierarchie in den Werkzeugen aber meist wenig prominent präsentiert wird. In Enterprise Architect erfolgt dies standardmäßig etwa in einer einklappbaren Seitenleiste am Rande des Sichtbereiches. Hierdurch entsteht ein Konflikt zwischen dem Nutzer, der in der Regel in Diagrammen denkt, und der tatsächlich modellierten Struktur der Elemente untereinander. Die Werkzeuge versuchen zwar mit diversen Heuristiken, diese Verbindung synchron zu halten (so wird etwa bei der Modellierung eines Zustandsautomaten ein Zustand automatisch dem Zustand untergeordnet, der ihn im Diagramm grafisch umschließt); allerdings versagte diese Automatik im Test gelegentlich – es entsteht ein Problem, das vergleichbar mit vergessenen Blockzeichen in klassischen Programmiersprachen ist (und ähnlich schwer zu erkennen!). Zusätzlicher Code wurde in den Parser eingefügt, um derartigen Fällen so gut wie möglich zu begegnen.

Die texturellen Teile wurden in [ANTLR] umgesetzt. Hierbei wurde mäßiger Gebrauch von den angebotenen Baumtransformationen gemacht; so kann etwa eine Argumentliste am Kontext erkannt und noch im Parser aufgeteilt werden; die folgende Schritte benötigen

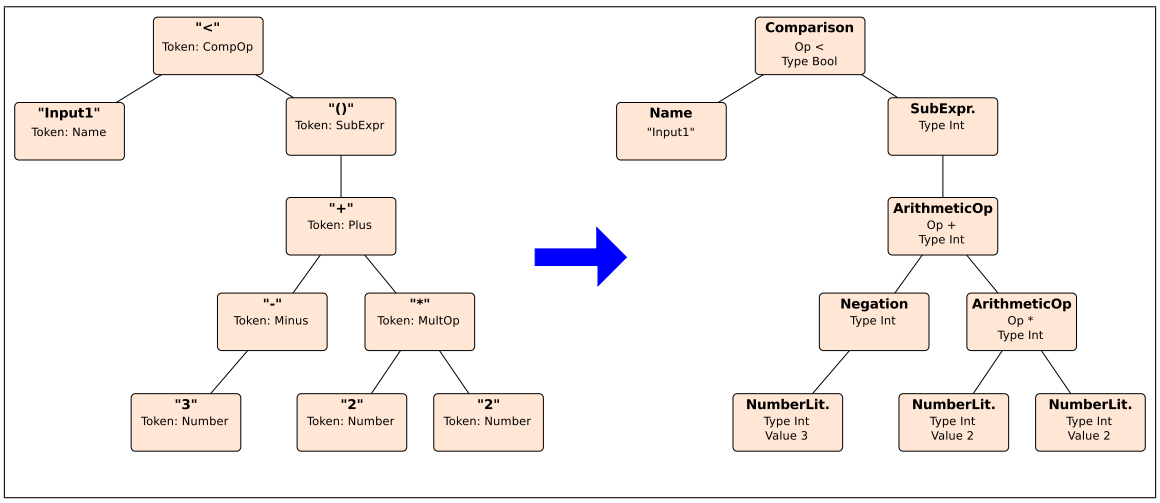


Abbildung 4.3: Der Ausdruck `Input1 < (-3 + 2*2)` als Syntaxbaum und nach der Transformation

daher keine Kenntnis über Klammern und Kommas mehr. Auch die Operatorrangfolge wird bereits durch die Grammatik gehandhabt. Der von ANTLR erzeugte AST kann anschließend ohne strukturellen Änderungen nur abhängig von Tokentyp und Kindanzahl auf eine eigene Baumstruktur abgebildet werden:

Token	Kindanzahl	Eigene Klasse
Or, And	2	BooleanOp
Not	1	BooleanNot
CompOp	2	Comparison
Plus, Minus	1 2	SubExpression (+) oder Negation (-) ArithmeticOp
MultOp	2	ArithmeticOp
Power	2	PowerOp
Name	beliebig	Name
NumberLiteral	keine	NumberLiteral
TimeLiteral	keine	TimeLiteral
SubExpr	1	SubExpression
If	3	Conditional

Ein Beispiel zeigt Abbildung 4.3.

Semantische Analyse

Mit Rücksicht auf den möglichen Umfang der Arbeit kommen statische Analyseverfahren und Optimierungen nur begrenzt zum Einsatz. Neben der Typprüfung mit einem aus Ada entlehnten Modell, um mehrdeutige Aufzählungen aufzulösen[1] ist dies die Bereichsinferenz über alle arithmetischen Ausdrücke, so dass jedem Ausdruck vom Typ Integer ein

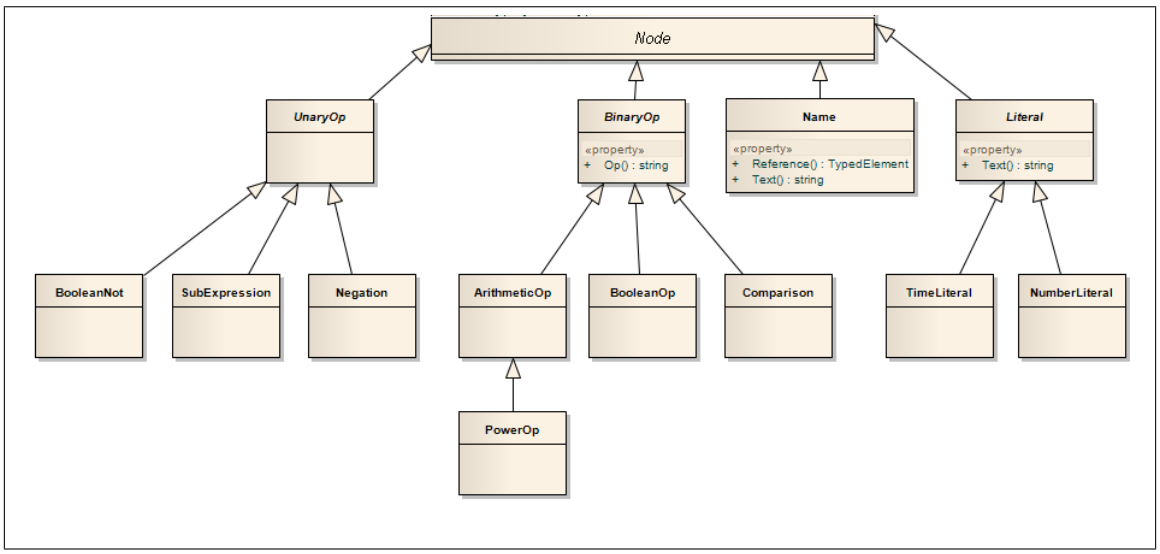


Abbildung 4.4: Hierarchie der Ausdrucksknoten

Bereich zugeordnet ist. Als Seiteneffekt daraus ist *Constant Folding*[2] eine triviale Optimierung.

Im Prototyp wurde die semantische Analyse direkt auf dem von ANTLR zurückgegebenen monomorphen Ausdrucksbaum aufgebaut, d.h. alle Knoten waren von derselben, passiven Datenklasse. Dies führte jedoch zu einer sehr prozeduralen Implementierung, bei der zu Beginn fast jeder Methode eine Fallunterscheidung über alle Ausdruckselemente stand.

Im Endsystem wird diesen Problemen mit einer stärker objektorientierten polymorphen Baumstruktur begegnet, wobei eine zweistufige Hierarchie mit einer Entscheidung nach Kindanzahl (unäre / binäre Operation, Literal oder benannter Ausdruck) und einer Klasse pro Ausdruckselement eingesetzt wird, die in Abbildung 4.4 dargestellt ist. Der von ANTLR produzierte AST wird in diesem Ansatz unmittelbar in diese Struktur transformiert, wodurch auch eine weitere Entkapselung von der Laufzeitbibliothek des Parsergenerators erreicht wird.

Der wesentliche Unterschied für die semantische Analyse ist, dass ein solcher Ausdrucksbaum sich selbst annotieren kann; auf der Basisklasse sind Methoden für die jeweilige Analysephasen definiert, die polymorph und rekursiv auf den Baum angewendet werden. Das `SemanticProgram`, die Einstiegsklasse für die semantische Analyse, wird dadurch entlastet und muss nur noch die Semantik am Rande und oberhalb des Ausdruckslevels implementieren. Hierzu gehört beispielsweise, ob der Typ eines Ausdrucks zu seinem Kontext passt, die Bestimmung der Auswertungsreihenfolge und Aufruf der Analysephasen auf den einzelnen Ausdrücken. Ein weiterer Vorteil ist, dass die Sprache leichter um neue Elemente erweitert werden kann. Das if-then-else-Konstrukt beispielsweise wurde erst im letzten Drittel der Arbeit eingefügt; der größte Teil des Aufwands war das Nachziehen der Dokumentation.

[1] In Ada ist es erlaubt, zwei Aufzählungen mit dem selben Wert, z.B. ROT, GRUEN, BLAU, zu definieren und die Werte ohne Qualifikation des Aufzählungstyps zu verwenden. Der Compiler ermittelt diesen am Kontext.

[2] Das Zusammenfassen von Operationen auf Konstanten, statt die Berechnung bei jedem Zugriff erneut auszuführen; beispielsweise kann `2 * Pi` zur Übersetzungszeit durch `6.2832` ersetzt werden.

Ausführungsmodell

Da Laufzeitfehler nicht zu vermeiden sind, kommt ein einfaches transaktionales Modell zum Einsatz. Dieses implementiert im Fehlerfall ein „Rollback“ auf den letzten erfolgreich berechneten Takt, so dass das System in einem definierten Zustand bleibt und sich verhält, als hätte es die zum Fehler führende Eingabe nie gegeben. Dies lässt dem Nutzer Spielraum, für die Fehlerbehandlung den Takt entweder auszulassen, mit „sicheren“ Werten erneut zu berechnen oder das Modell zurückzusetzen.

Die Berechnung eines Taktes wird hierzu in drei Phasen unterteilt:

1. Setzen der Input-Variablen aus den neu übergebenen Werten
2. Berechnen des nächsten Wertes aller anderen Variablen („Step“)
3. wenn erfolgreich: Commit aller Variablen

Um die Fehlersemantik umzusetzen, wird in *flüchtige* und *persistente* Felder aller Klassen unterschieden. In *Step* werden persistente Felder als Nur-Lesend, flüchtige als Nur-Schreibend betrachtet. Ein zweiter Aufruf von *Step* hat damit keine Wirkung; erst in *Commit* werden die flüchtigen Felder dann persistiert (entweder durch einfaches Überschreiben, oder im Falle der Variablenwerte, durch Pushen in den FIFO-Stack.)

Immer wenn andere Berechnungen von flüchtigen Werten abhängen, ist die korrekte Ausführungsreihenfolge zu beachten. Die folgenden Fälle treten auf:

Abhängigkeit von Inputs Werden immer zu Beginn von „Step“ gesetzt. Die Reihenfolge innerhalb der Gruppe ist beliebig, da Inputs keine Abhängigkeiten haben können.

Zählerobjekte für „Duration“-Ausdrücke Durch das obige Fehlerverhalten muss nur „at least once“ (pro Takt) und nicht „exactly once“ garantiert werden. Die „DurationExpression“ kann sich daher beim Zugriff auf ihren Wert selbst aktualisieren. Die dadurch eventuell mehrfache Auswertung des Ausdrucks wird für die erste Version hingenommen.

Abhängigkeiten von Mode und Output Da Funktionen eingesetzt werden, kann diese Abhängigkeit nur wieder innerhalb der Menge Modes$\cup$Outputs auftreten. Eine Abhängigkeit besteht genau dann, wenn ein beliebiger Ausdruck des Elements einen Bezeichner enthält, der ein Element obiger Menge mit Zeitversatz 0 referenziert. Der entstehende Abhängigkeitsgraph kann mit topologischer Sortierung gelöst werden.

Codegenerierung

Eine Aufgabenstellung war, eine strukturelle Ähnlichkeit zwischen dem eingegebenen Modell und dem generierten Code zu bewahren, weshalb sich ein einfaches Übersetzungsschema anbietet.

Jeder Output und Mode wird daher in einer eigenen Quelltextdatei abgelegt, die von einer zentralen Programmdatei aus verwaltet werden. In C# werden jeweils Klassen eingesetzt, in C ein klassenähnliches Codeschema. Für Inputs ist kein nutzerspezifischer Code (außer dem Standardwert) zu erzeugen; dies wird an die Hauptklasse delegiert. Für benutzerdefinierte Typen wurde die Entscheidung getroffen, sie in einer weiteren Datei zu bündeln, da deren Deklaration jeweils kurz ausfällt.

Da zwischen C und C# erhebliche syntaktische Ähnlichkeiten bestehen, wurde die Codeformatierung und bestimmte gemeinsame Konstrukte wie if-then-else-Ketten, Kommentare und Funktionsdeklarationen in ein gemeinsam genutztes Modul ausgelagert. Dieses Modul hält den generierten Code im Speicher, um die Codegenerierung logisch von der Reihenfolge der Statements im Code zu entkoppeln; dies erwies sich insbesondere für ANSI-C als Vorteil. Beispielsweise muss für die Aufgabe, die Duration-Zähler für ein bestimmtes Element zu erzeugen, an bis zu sechs Stellen Code bearbeitet werden:

- Beim Initialiseren des Elements ist der Zähler mit zu initialisieren.
- Der Code innerhalb des `Duration`-Konstrukts muss als Hilfsfunktion erzeugt werden (in C# kommt eine anoynme Funktion zum Einsatz).
- Die Dimensionierung des Arrays für alle `DurationCounter`-Elemente muss angepasst werden.
- In die Methoden `bind`, `step` und `commit` des umgebenden Elements muss Code eingebunden werden, der die Aufrufe an die `DurationCounter`-Objekte durchreicht.

Abgesehen von der gemeinsamen Bibliothek bestehen beide Codegeneratoren aus einem Verbund von jeweils vier Klassen:

Generator-Hauptklasse delegiert an die untergeordneten Generatoren und erzeugt das Hauptprogramm.

Output- und Mode-Generator implementieren das Erzeugungsschema für Output- oder Mode-Elemente.

ExpressionGenerator wandelt SCR-EA-Ausdrücke in die Zielsprache. Dies ist keine strenge 1:1-Beziehung, da z.B. `Duration`-Ausdrücke in ein Zählerobjekt umgewandelt und getrennt erzeugt werden. Diese Klasse wird von den anderen Generatoren aggregiert.

Dieser Ansatz mag simplistisch erscheinen, erwies sich aber für die Zwecke dieser Arbeit als völlig ausreichend (da die Vorarbeiten in die semantische Analyse verlagert sind, liegt etwa der Umfang des C#-Codegenerators nur noch bei etwa 500 SLOC).

Implementierung

Zu Ende der Spezifikationsphase wurde klar, dass das durch den ursprüngliche Zeitplan vorgegebene Wasserfall-Entwicklungsmodell nicht hilfreich war. Zum einen fehlte mir die Übung, um alle semantischen Feinheiten nur mit Stift und Papier zu klären; zum anderen war bei einigen Features nicht offensichtlich, ob sie mit der Forderung nach konstantem Speicherbedarf überhaupt umsetzbar waren. Ich entschloss mich daher, die auf drei Wochen angelegte Entwurfsphase aufzuteilen und etwa zwei Wochen für die Anfertigung eines Prototyps mit reduziertem Sprachumfang einzuschieben. In Abstimmung mit dem Betreuer wurden die folgenden Merkmale festgelegt:

- Unterstützte Strukturelemente: Input, Output, Mode Charts
- Mode Charts ohne Verschachtelung und History-Zustände
- Unterstützung von `Duration()` und `Pre()`

- Keine Unterstützung von Zeitangaben, Vergleich von Duration nur mit Ganzzahlen
- Minimale statische Semantik: Zuordnen von Bezeichnern, Typannotation
- Entsprechend reduzierte dynamische Semantik, aber mit spezifiziertem Fehlerverhalten.

Dies konnte mit dem bestehenden Architekturentwurf in der sechsten und siebten Woche erfolgreich umgesetzt werden. Dabei fiel das Auslesen der UML-Diagramme über die von Enterprise Architect angebotene Automatisierungs-Schnittstelle als mühevoll und zeitraubend auf, da die Schnittstelle der Präsentation im Programm teils nur lose folgt und eine andere Terminologie verwendet. Das Einlesen von Mode Charts wurde daher zunächst verschoben; zum Testen wurden die Datenstrukturen, die Ausgabe des Strukturparsers sind, manuell erzeugt (siehe Abschnitt „Komponenten“, detaillierte Beschreibung im Entwurfsdokument) und kamen sowohl für die Unit-Tests der statische Semantik als auch der Codegenerierung zum Einsatz.

Aus dem Prototyp konnte für den weiteren Entwurf die Lehre gezogen werden, dass sowohl die Codegenerierung als auch die semantische Analyse den sinnvollen Umfang für eine einzelne Klasse deutlich sprengten. Mit diesen Erkenntnissen wurde der Entwurf nachgebessert; der passive Syntaxbaum aus ANTLR wird in ein objektorientiertes Modell umgewandelt, bei dem die einzelnen Knoten des Ausdrucks sich selbst überprüfen können und damit jeweils auf einer gut wartbaren Größe bleiben. Für den Codegenerator wurde eine ähnliche Aufspaltung anhand der Strukturelemente (Hauptprogramm, Mode Chart, Output) vorgenommen und die Generierung eines einzelnen Ausdrucks per Aggregation ausgelagert. Die Module zum Einlesen konnten hingegen weitgehend unverändert übernommen werden.

Umsetzung von Pre und Funktionen

Die Definition lässt zwei alternative Interpretationen von Pre zu: a) es verschiebt den betrachteten Zeitpunkt oder b) es wirkt als ein Verzögerungsgatter im Sinne der Elektrotechnik. Letzteres ließe sich ähnlich wie Duration durch ein pro Takt aktualisiertes Objekt mit dem Unterausdruck als Parameter übersetzen. Dieses Objekt evaluiert den Ausdruck in jedem Takt und legt das Ergebnis mit entsprechender Verzögerung als Rückgabewert an. Dies ist einfach umzusetzen (und wird in SCADE auch so gehandhabt), hat aber einige Nachteile:

- der Ausdruck wird in jedem Takt ausgewertet, unabhängig davon ob das Ergebnis benötigt wird.
- der Ausdruck im generierten Code zerfleddert, da der Unterausdruck in den Konstruktor verlegt werden muss. Der Ausführungspfad wird schwerer nachzuvollziehen.

Weiterhin muss in Ansatz b) für jeden *Pre-Ausdruck* eine FIFO-Schlange angelegt werden.

Der Ausdruck `Pre(a + Pre(b*3))` würde so etwa beispielsweise nach `pre1` übersetzt, mit den Deklarationen:

```
pre1 = new Pre(a + pre2);
pre2 = new Pre(b * 3);
```

Betrachten wir einen Ausdruck ohne Funktionen im Sinne von a). Jeder Knoten des Ausdrucksbaums kann um das Attribut *Zeitversatz* annotiert werden. Rekursiv kann nun folgende Transformation angewandt werden:

- Übernehme den Zeitversatz vom Elternknoten (0, wenn Wurzelknoten).
- Wenn der aktuelle Knoten von der Form `PreN( expr, N )` ist, erhöhe den Zeitversatz um N, und ersetze den Knoten durch `( expr )`.
- Annotiere für Variablen das Maximum aus allen Zeitversätzen, mit denen zugegriffen wird.
- Rekursion über Unterausdrücke.

Wird diese Transformation für alle Ausdrücke durchgeführt, so ist in jedem Variablenzugriff der nötige Zeitversatz und in den Variablendeklarationen der programmweit maximale Zeitversatz annotiert. Dies kann dann durch eine FIFO-Schlange pro *Variable* mit minimaler Länge übersetzt werden. Dies verbessert den Speicherbedarf, wenn programmweit verschiedene Pre-Ausdrücke auf dieselben Variablen zugreifen. Hauptvorteil ist die verbesserte Lesbarkeit, da der Ausdruck im Gegensatz zu b) nicht geteilt werden muss; so wird aus `Pre(a + Pre(b*3))` der Ausdruck `a[-1] + (b[-2]*3)`, der keine weiteren Deklarationen erfordert.

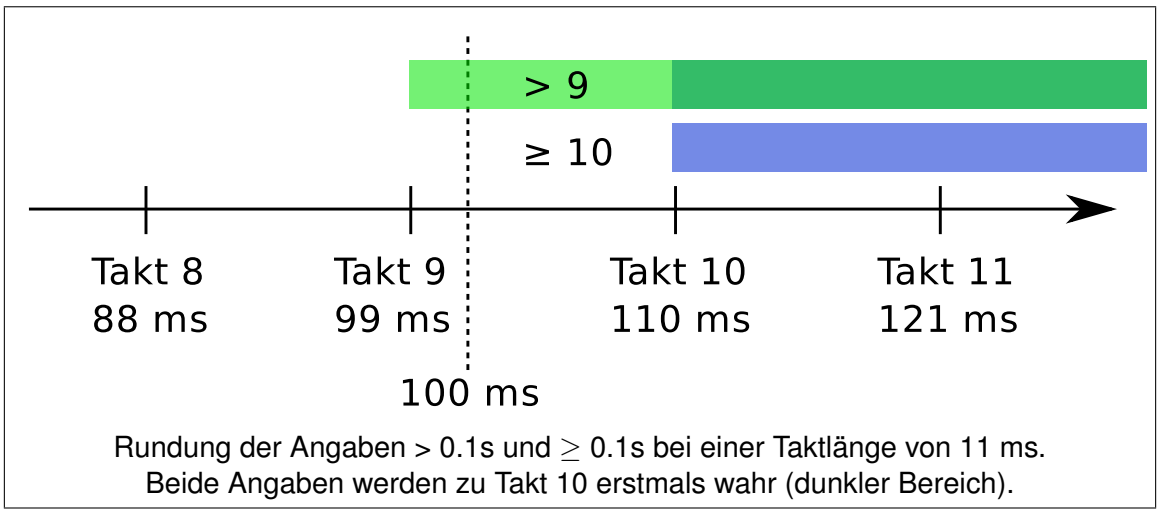
Nachteil dieses Ansatzes ist allerdings die Behandlung von Funktionen. Hängt eine Funktion nicht nur von ihren Parametern ab, was wir zulassen wollen, so erhält man mit dem oben beschriebenen Ansatz ein falsches Ergebnis. Drei Lösungsansätze sind denkbar:

1. Setze die Funktionen an der Stelle des Aufrufs ein.
2. Übergebe den Zeitversatz als zusätzlichen Parameter. Die Funktion muss diesen bei allen Variablenzugriffen zusätzlich berücksichtigen.
3. Übersetze die Funktion einmal für jeden auftretenden Zeitversatz.

Der zweite Ansatz erweist sich als schwierig, sobald Funktionen wieder auf Funktionen zugreifen. Hier ist die Länge der erforderlichen Rückschau nur noch über einen Abhängigkeitsbaum aller Funktionen zu beantworten, anstatt einfach das Maximum aus allen Ausdrücken gebildet werden kann. Der dritte Ansatz dupliziert bereits Code, verschiebt das Problem aber nur: Für jede Funktion wäre zu annotieren, mit welchen Zeitversätzen auf sie zugegriffen wird. Auch hier ergibt sich eine Abhängigkeit zwischen Funktionen. Ist z.B. `a() = Pre(b())` und `a` wird nun mit dem bisher nicht vorhandenen Zeitversatz N aufgerufen, so muss auch von `b` eine neue Variante erstellt werden.

Mit dem Hintergedanken, im Zweifelsfall eher korrekten statt eleganten Code zu generieren, fiel die Entscheidung damit auf die erste Variante; Funktionen werden immer eingesetzt, nicht als eigene Einheiten übersetzt.

Zu beachten ist, dass dieses Problem letzlich nur aus der gewählten Implementierung von `Pre` folgte. In den Fallstudien ergab sich dann leider, dass Funktionen erheblich häufiger zum Einsatz kommen als `Pre`, und die Nachteile des Inlining daher die Lesbarkeits- und Laufzeitvorteile von Ansatz a deutlich überwiegen. Rückblickend hätte eine Umsetzung wie in SCADE insgesamt zu einem besseren Ergebnis (im Sinne von lesbarem Code) geführt. Ideal wäre eine Kombination aus beiden Varianten, beispielsweise Ansatz a für Ausdrücke ohne Funktionen und b sonst.



Rundung der Angaben > 0.1s und $\geq$ 0.1s bei einer Taktlänge von 11 ms.
Beide Angaben werden zu Takt 10 erstmals wahr (dunkler Bereich).

Abbildung 4.5: Rundung von Zeitangaben zu Takten

Ablauf der semantischen Analyse

Die semantische Analyse zerfällt in mehrere Phasen, die im Folgenden beschrieben sind. Innerhalb einer Phase werden alle Fehler gesammelt, aber die weitere Verarbeitung abgebrochen, wenn die vorherige Phase nicht erfolgreich beendet wurde.

Symboltabelle erstellen Die Symboltabelle wird mit den globalen Funktionen und Konstanten initialisiert; anschließend werden alle benutzerdefinierten Bezeichner von Typen und Variablen gesammelt.

Konstanten annotieren Die Konstanten werden auf zyklische Beziehungen geprüft und ihr Wert errechnet. Diese Trennung ist erforderlich, da für Konstanten abweichend zu anderen Ganzzahlen kein Subtyp vom Nutzer definiert wird (dieser entspricht einer Bereichsdefinition mit genau dem Wert der Konstante).

Andere Ausdrücke annotieren Ermittelt für alle Ausdrucksbäume rekursiv Typ- und Bereichsinformationen, prüft, ob diese zulässig sind und mit den Deklarationen übereinstimmen.

Zeitausdrücke auflösen Alle Zeitangaben werden unter Rücksicht auf die eingestellte Taktzeit und die übergeordnete Vergleichsoperationen in ganze Takte umgewandelt und gerundet. Ergibt sich eine Zeitangabe nicht genau auf eine Taktgrenze, so ist für `>` ab-, für `>=` aber aufzurunden, um die Semantik zu erhalten (siehe Abbildung 4.5). Um Rundungsfehler zu vermeiden kommt für die Berechnungen der C#-Datentyp `decimal` zum Einsatz (ein Gleitkommatyp zur Basis 10).

Funktionen einsetzen Funktionen und Konstanten werden in die Ausdrücke eingesetzt. Sind Optimierungen eingeschaltet, werden die ermittelten Bereichsgrenzen anschließend mit dem lokalen Kontext verbessert (so kann etwa auch `f(const)` als konstanter Wert erkannt und bei der Codegenerierung ersetzt werden).

Zeitversatz ermitteln Das Zeitversatz-Attribut jedes Ausdrucksknoten wird aus der Schachtelung der `Pre`-Funktionen ermittelt. Dies ist äquivalent zu einem „Durchschieben“ der `Pre`-Funktionen auf die Variablenzugriffe. Hierbei wird auch das globale Maximum der nötigen Rückschau pro Variable ermittelt.

Abschließend wird noch durch topologische Sortierung die Auswertungsreihenfolge der Elemente bestimmt (und ob sich zyklische Abhängigkeiten ergeben, die durch `Pre` aufgelöst werden müssen).

Laufzeitbibliothek

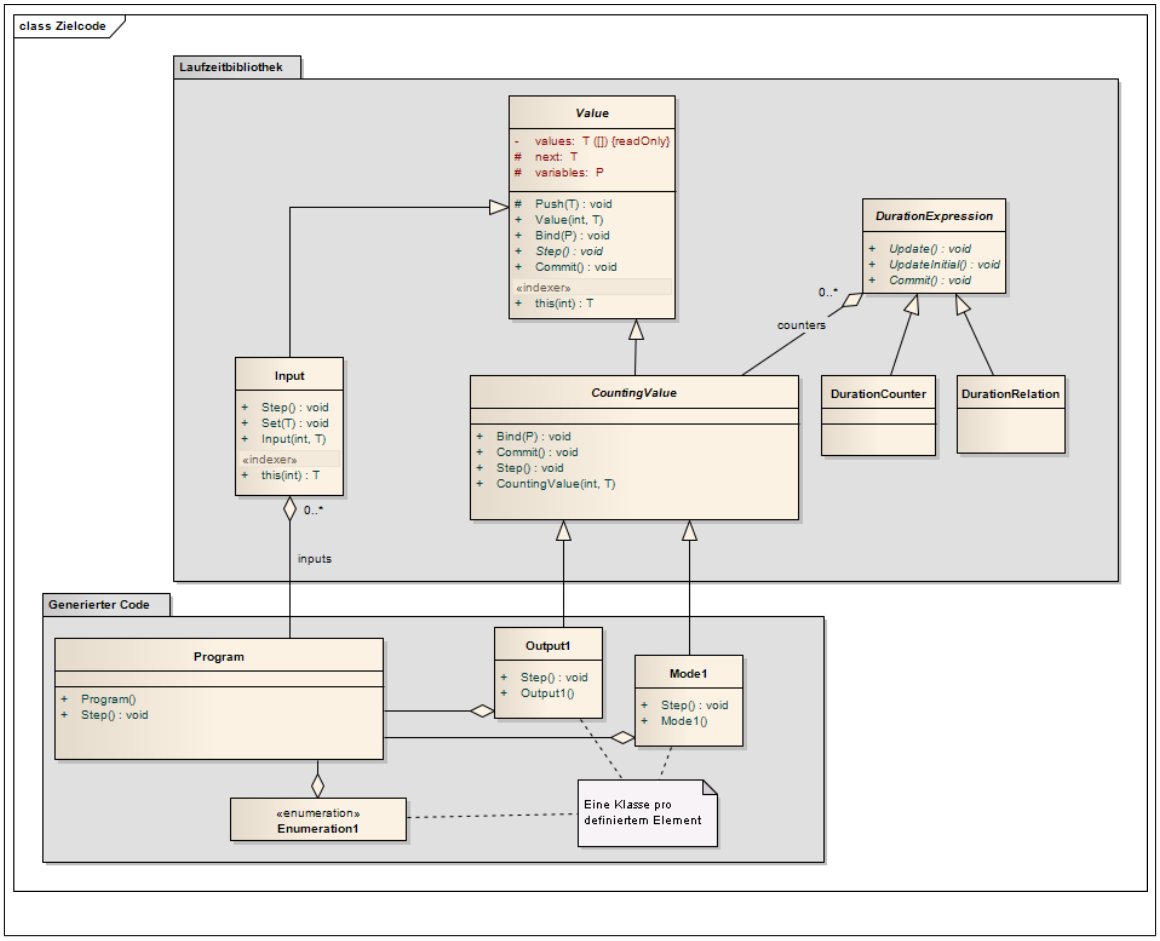


Abbildung 4.6: Übersicht über die Laufzeitbibliothek (C#)

Die Laufzeitbibliothek für diese Sprache übernimmt überschaubare Funktionen. Wichtigste Aufgabe ist die Bereitstellung der Zählerobjekte (siehe Abschnitt „Umsetzung von Pre und Funktionen“ oder die Übersichtsgrafik), sowie einige globale Deklarationen.

Für C# ist es mit dem `checked`-Block sehr einfach möglich, Compilerprüfungen für Bereichsüberläufe einzufügen. Die Ergebnisse der Bereichsinferenz müssen nur noch genutzt werden, um Variablen und Zwischenergebnisse geeignet zu typisieren. Falls das Ergebnis des Ausdrucks numerisch ist, muss anschließend noch auf die deklarierten Bereichsgrenzen der Variable geprüft werden.

Für ANSI-C existiert kein derartiges Sprachmittel. Natürlich bieten die meisten Compiler eine entsprechende Option an, allerdings wollten wir eine Abhängigkeit von Compilereinstellungen vermeiden. Zumindest bei GCC scheint `-ftrapv` auch eine sehr wenig getestete Funktion zu sein, so dass Fehler immer wieder in Release-Versionen durchdringen (siehe z.B. [gcc52478] und die dort dokumentierten früheren „Duplikate“).

Die erforderlichen Prüfungen selbst zu implementieren erweist sich als erstaunlich schwierig ([Seacord, Kapitel 5], [Leitner]), auch da der ANSI-C-Standard das Verhalten für überlaufende Operationen auf vorzeichenbehafteten Ganzzahlen völlig undefiniert lässt ([C90, Anhang A]), so dass der Fehler bereits aufwendig *vor* der Operation diagno-

stiziert werden muss. Nicht einmal Breiten und Wertebereiche der verfügbaren Integertypen sind standardisiert. Aus der von modernen Sprachen verwöhnten Sicht des Autors stellte sich daher mehr als einmal die Frage, ob ANSI-C denn nun eine Sprache oder nur ihre Syntax definieren soll.

Um den Aufwand an dieser Stelle nicht explodieren zu lassen, musste auf die erst ab C99 standardisierten Typen `int16_t`, `int32_t` und `int64_t` zurückgegriffen und die Unterstützung auf die Compiler GCC, Clang und MSVC eingeschränkt werden.

Kapitel 5

Test und Auswertung

Automatisierte Tests stellen bei einem Compiler eine besondere Herausforderung dar. Kurioserweise gibt es hierzu fast keine Forschungsliteratur, bestenfalls finden sich Praxisberichte für spezifische Compiler (z.B. [CTesting]). Das „Drachenbuch“ [AhoUllman86], sonst verlässlicher Ratgeber, behandelt das Thema auf einer einzigen Seite. Der folgende Abschnitt stützt sich daher weitgehend auf persönliche Erfahrungen, die mit meinem Betreuer und Professor Plödereder abgestimmt wurden.

Unit-Tests

Alle Belange des Frontends, also Einlesen und Transformieren des Programms, lassen sich hervorragend durch Unit-Tests abdecken. Die Tests für den Strukturparser wurden von der der semantischen Analyse getrennt. Der Parser für Ausdrücke ist wie besprochen automatisch generiert. Auf einen separaten Test wurde verzichtet, auch da er ohnehin indirekt an fast jedem Testfall beteiligt war.

Aufgrund der Dateigröße und langen Startzeiten von Enterprise Architect wurde die Testsuite für den Strukturparser in ein einzelnes Repository gebündelt. Auf der Ausgabe des Parsers laufen dann mehrere Unit-Tests hintereinander, die einzelne Teile der erzeugten Programmdefinition testen.

Für den Test der semantischen Analyse werden diese Programmdefinitionen primär zur Laufzeit erzeugt. Dies hat zum einen den Vorteil, dass sie sehr viel besser zu parametrisieren sind, um alle Sonderfälle im Test zu erreichen; zum anderen, dass beim Erstellen der Testfälle Spezifikationslücken im Zwischenformat entdeckt werden konnten.

Beim Auftreten eines Fehlers wurde dieser konsequent zuerst durch einen Testfall repliziert und erst danach eine Korrektur durchgeführt. Diese Anleihe aus der testgetriebenen Entwicklung führt dazu, dass die Unit-Test-Bibliothek Regressionen besonders gut erkennen kann.

Ab der zweiten Projekthälfte wurde die Codeüberdeckung der Unit-Tests mit dem Werkzeug OpenCover gemessen. Unit-Tests für neue Funktionalität wurden dabei nach größeren Etappen oder ab dem Schwellwert von 90% nachgezogen. Zur Abgabe des Projekts waren für das Frontend und die Tests des Codegenerators (siehe nächster Abschnitt) 93,5% Überdeckung erreicht.

Codegenerator- und Systemtest

Die Korrektheitsanforderungen an einen Codegenerator lassen sich grob in die folgenden Stufen einteilen:

1. Die Übersetzung ist erfolgreich (Compiler stürzt nicht ab)
2. Die Ausgabe ist syntaktisch gültig (ein weiterer Compiler oder die Zielmaschine kann sie interpretieren)
3. Der generierte Code zeigt auf der Zielplattform *genau ein* Verhalten, das durch die Sprachreferenz erlaubt ist
4. Formaler Beweis, dass 3. für alle Konstrukte der Sprache gilt.

Jeder Unit- oder Systemtest ist darauf angewiesen, dass aus einer Eingabe eine als korrekt überprüfbare Ausgabe erfolgt. Für Stufe 1 ist dies kein Problem; Unit-Test-Bibliotheken haben hierfür spezielle Funktionalität. Für Stufe 2 ist es bereits etwas schwieriger; die entsprechenden Compiler der Ausgabesprache müssen auf dem System installiert und aus den Unit-Tests heraus aufgerufen werden. Die Tests weisen damit bereits nichttriviale Vorbedingungen auf.

Für Stufe 3 wird die Anforderung erheblich „weicher“ und schon daher schwerer zu prüfen. Aber ganz abgesehen von den Freiheitsgraden, die der Sprachstandard selbst lässt, stoßen wir hier an eine Grenze. Nach dem üblichen Ansatz wäre hier für jeden Testfall von Hand ein Programm mit dem korrekten Verhalten zu schreiben, und dieses mit dem generierten Programm zu vergleichen. Leider ist die Äquivalenz von Programmen das klassische Beispiel der theoretischen Informatik, dass noch schwierigere Probleme als das Halteproblem existieren; und schon dieses ist für alle turingmächtigen Sprachen nicht allgemein zu lösen. Natürlich ist es dies in Spezialfällen möglich (beispielsweise sind zwei Programme, die sich nur durch ihre Kommentierung unterscheiden, äquivalent), uns ist aber kein Produkt bekannt, das für unsere Zwecke mächtig genug wäre. Es ist also praktisch nicht möglich, die Soll-Ausgabe eines Codegenerators in einer automatisch prüfbaren Form zu definieren; ein Regressionstest kann zwar feststellen, dass sich die Ausgabe *geändert* hat, diese Aussage ist aber ohne manuelle Inspektion nutzlos.

In der Praxis – und damit auch für diese Arbeit – bleibt somit nur, die Korrektheit der Ausgabe in einem weiteren Test zu überprüfen. Hierbei wird der generierte Code zuerst übersetzt, anschließend mit vordefinierten Eingaben aufgerufen und das korrekte Verhalten anhand der Ausgaben überprüft. Jeder Unit-Test des Codegenerators enthält also (unter Umständen mehrere) untergeordnete automatisierte Systemtests für den generierten Code.

Da in unserem Fall die Ausgabe noch kein ausführbares Programm ist, musste unterhalb des Unit-Test-System ein weiteres Testsystem konstruiert werden:

- Ein Programmfragment, mit dem der Bibliothekscode zu einer ausführbaren Datei gelinkt werden kann, muss geschrieben oder zusätzlich generiert werden.
- Von den Unit-Tests aus müssen die Unterprogramme zur Laufzeit übersetzt und mit vorgefertigten Ein-/Ausgabesequenzen aufgerufen werden.

Nach der Konstruktion dieses automatisierten Testsystems ist zumindest Stufe 2 der obigen Definition erreicht. Für Stufe 3 bleibt das Problem, einen Fehler im Blackbox-Test zu

erkennen; [LL07, Seite 454], benennt die üblichen Erfolgsaussichten mit etwa 50%. Hier konnte der Erfahrungsschatz der ICS AG genutzt werden, um für die beiden Fallstudien systematisch eine Suite von Testfällen zu entwickeln.

Ergebnisse des Systemtests

Die Ergebnisse des Tests waren durchweg erfreulich: Etwa ein Dutzend Ungenauigkeiten in den Spezifikationen der Fallstudien (Anhänge 1 und 3) wurden aufgedeckt. Mit den endgültigen Testsequenzen (Anhänge 2 und 4) konnten insgesamt fünf Modellierungsfehler nachgewiesen werden, die aber keine Änderung am Compiler selbst erforderten:

- **MSA**: Vergleich mit Bremsdruck ist als $>=$ spezifiziert, aber als $>$ implementiert
- **MSA**: Zeitliche Bedingung fehlt (erst nach 1,5 Sekunden „Motor aus“ kommandieren)
  $\rightarrow$ Dank Definitionen und Duration() war nur eine kleine Änderung erforderlich.
- **Beamer**: Während dem Nachlaufen kann kein Wechsel in den Zustand „beide defekt“ erfolgen
  $\rightarrow$ Ein weiterer Metazustand „Normalbetrieb“ wurde eingeführt, der „Eingeschaltet“ und „Ausgeschaltet“ umfasst.
- **Beamer**: Wenn A defekt ist, wieder anläuft und dann B defekt ist wird NICHT nach „beide defekt“ geschaltet
  $\rightarrow$ Der Output muss auch von seinem vorherigen Zustand abhängen und darf Fehlerzustände nicht löschen.
- **Beamer**: Die Taktgrenzen sind ungenau modelliert (bis zu $\pm$ 2 Takte)

Lediglich ein einziges Problem mit dem generierten Code wurde gefunden: Aufgrund eines Missverständnis des EA-Datenmodells wurde die spezifizierte Reihenfolge verschiedener Übergänge eines Mode nicht beachtet.

Alle Testsequenzen wurden unmodifiziert mit dem C#- und C-Testrunner ausgeführt, die durchweg dasselbe Verhalten zeigten. Zusammen mit den Ergebnissen aus den Unit-Tests, die dank dem automatisierten Testsystem ebenfalls systematisch übersetzt und ausgeführt wurden, kann daher mit hoher Sicherheit von Stufe 3 ausgegangen werden.

Typwahl und Optimierung

Zusätzlich wurde für den Systemtest eine Instrumentierung eingebaut, um die Effizienz der automatischen Typwahl zu prüfen. Diese zählt die Anzahl der arithmetischen Operationen (ohne Vergleiche) und die Typen der Zielsprache, auf der diese Operationen ausgeführt wurden. Der Typ ergibt sich aus den Ergebnissen der Bereichsinferenz, also einer konservativen Schätzung, welcher Wertebereich für die Operation erforderlich ist. Der Einfachheit halber werden vom Compiler dabei in dieser Version lediglich vorzeichenbehaftete Integer mit 16 (`short`), 32 (`int`) und 64 (`long`) Bit eingesetzt.

Die Ergebnisse dieser Instrumentierung sind in der folgenden Tabelle dargestellt.

Programm	**Beamer**		**CruiseControl**	
Optimierung	ohne	mit	ohne	mit
arithm. Operationen	48	0	132	132
davon `short`	24	0	0	0
davon `int`	12	0	0	6
davon `long`	12	0	132	126

CruiseControl ist eine erste ICS-interne Anwendung, die durch einen Mitarbeiter aus einem SCADE-Modell portiert wurde. Sie ist arithmetisch wesentlich komplexer als die beiden Fallstudien. Die Motor-Start-/-Stopp-Automatik enthält nur Vergleiche mit Konstanten und ist daher für diese Statistik uninteressant.

Auch die Beamersteuerung enthält lediglich parametrisierte Konstanten (in Form von Funktionen), die durch die Optimierung nach dem Inlining vollständig eliminiert werden konnten. Dies war auch der Grund, die Funktionalität einzubauen.

Da der von der Intervallarithmetik geschätzte Wertebereich einer Operation vor allem bei Multiplikation und Potenzoperationen sehr schnell groß wird, ist sie auf vom Nutzer möglichst eng eingegrenzte Wertebereiche angewiesen. Wird dies wie im Fall der Beameransteuerung konsequent durchgeführt, ist die Abschätzung für Zwecke der Typwahl ausreichend genau: der überwiegende Teil der Operationen kann sicher auf dem `int`-Datentyp durchgeführt werden.

Im Falle von *CruiseControl* wurden hingegen alle Eingabevariablen als `int` deklariert, was erwartungsgemäß dazu führt, dass für *alle* arithmetischen Operationen ein Cast nach `long` erforderlich ist. Überraschend konnte die Optimierung aber selbst hier sechs Operationen auf `int` verkleinern: die Ursache sind Funktionsaufrufe mit konstanter Null, die in einigen Fällen durchgeführt wurden. Tatsächlich hätte man also die gesamte Operation streichen können, diese Optimierung wurde aber bei der Entwicklung – fälschlich – noch als überflüssig angenommen.

Kapitel 6

Fallstudien

In diesem Kapitel werden mit der Software zwei konkrete, wenn auch vereinfachte, Steuerungssysteme von der Spezifikation bis zum testbaren Programm umgesetzt. Besonderes Augenmerk liegt auf den Arbeitsabläufen und verbleibenden Schwachpunkten in der Sprache.

Motorstart-Stopp-Automatik (MSA)

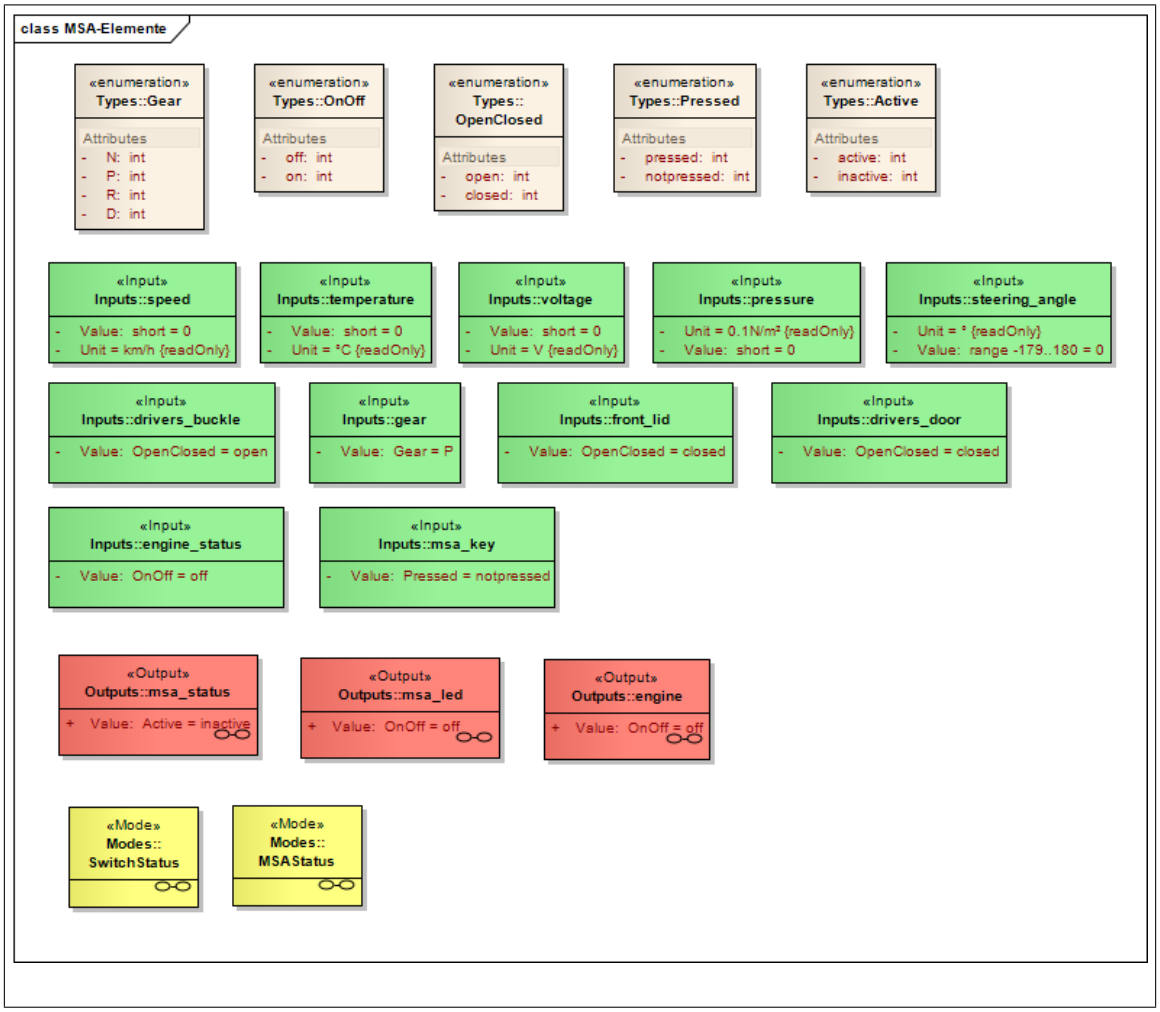


Abbildung 6.1: Strukturelemente für die Motorstart-Stopp-Automatik

Die MSA ist ein arithmetisch einfacher Testfall; diverse Sensorwerte werden auf Schwellenwerte verglichen und nicht weiter verarbeitet, Ausgabe ist im wesentlichen ein boolescher Aufzählungswert (Kommandierung Motor an oder aus). Sichtbar wird hier vor allem, ob der Ansatz mit Anforderungen typischer Länge noch gut lesbar ist, sowie der Umgang mit Aufzählungen.

Kurzbeschreibung

Die MSA kann vom Nutzer an- und abgeschaltet werden; der Zustandswechsel erfolgt über einen Knopfdruck, wenn bestimmte Bedingungen erfüllt sind. Ist die MSA aktiv, vergleicht sie u.a. Temperatur, Batteriespannung und Geschwindigkeit mit vorgebenenen Schwellenwerten. „Motor aus“ wird nur kommandiert, wenn alle diese Bedingungen gelten. Zusätzlich gibt es einige Kriterien wie „Fahrertür geöffnet“, bei der die MSA sich automatisch selbst deaktiviert.

Die vollständige Spezifikation findet sich in Anhang 1.

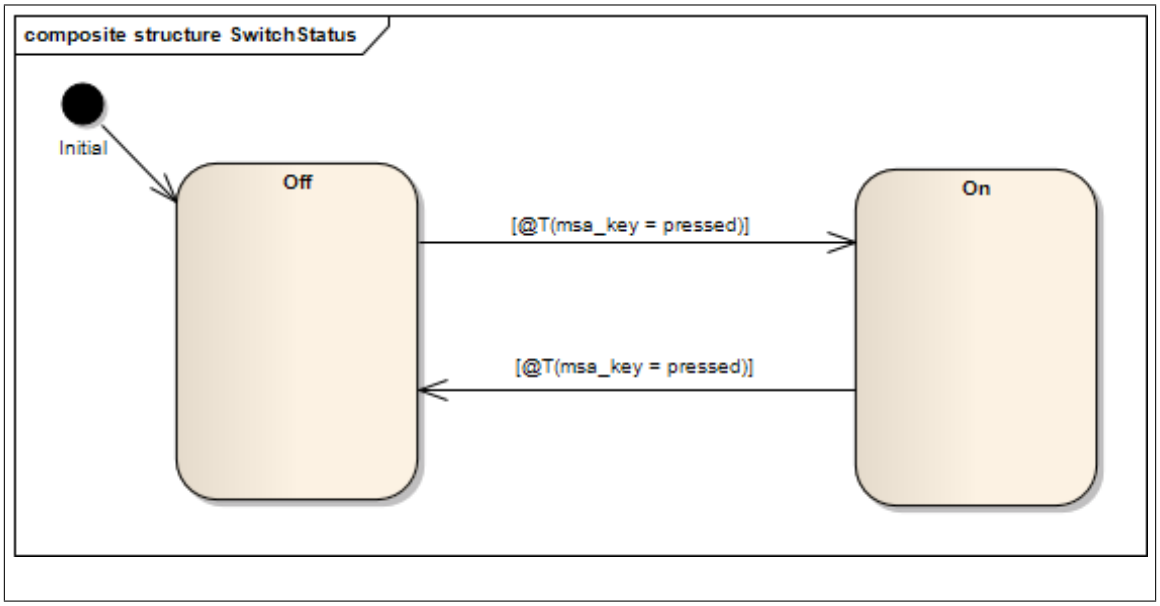


Abbildung 6.2: Ist die MSA eingeschaltet?

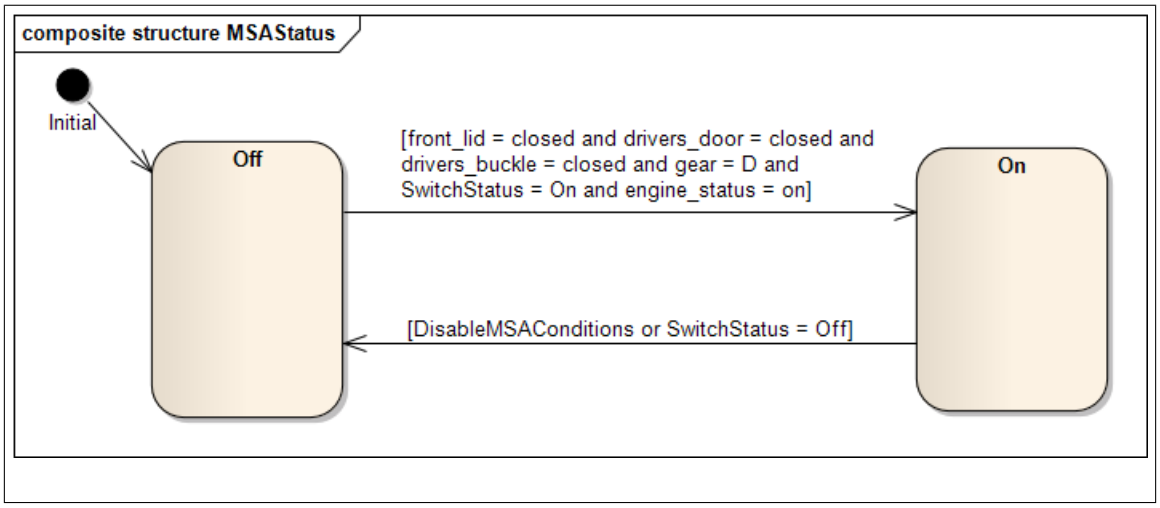


Abbildung 6.3: Sind die Vorbedingungen erfüllt?

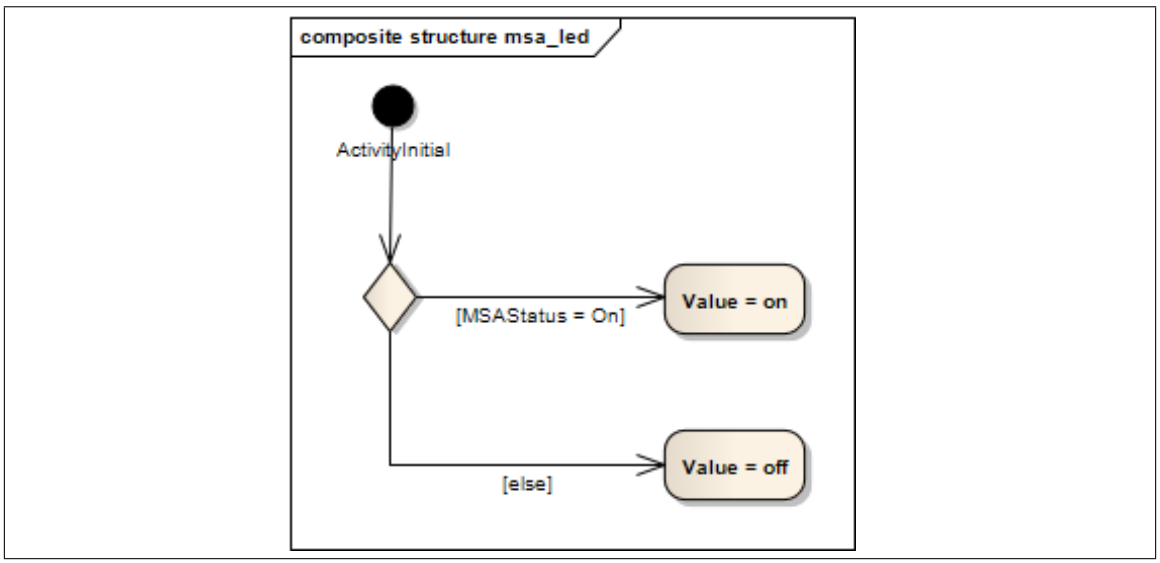


Abbildung 6.4: Kontrolleuchte der MSA

Umsetzung

Die MSA erfordert recht umfangreiche Definitionen von Eingabewerten und Aufzählungen (siehe Abbildung 6.1), die aber direkt aus der Spezifikation entnommen werden können. Neben den drei Ausgabewerten sind noch zwei einfache Mode Charts mit je zwei Zuständen erforderlich, um den Systemzustand abzubilden: Ist das System angeschaltet (`SwitchStatus`, Abbildung 6.2) und seine Vorbedingungen erfüllt (`MSAStatus`, Abbildung 6.3).

Die Kontrollleuchte kann daraus unmittelbar gesetzt werden (Abbildung 6.4), die eigentliche Kommandierungsentscheidung kann direkt im Output `engine` getroffen werden. In der ersten Modellierung (Abbildung 6.5, links) wird ersichtlich, dass Wiederholungen enthalten sind und die Ausdrücke daher unübersichtlich werden. Außerdem werden direkt einige undefinierte Randfälle (größer gleich vs. echt größer) ersichtlich. Beides konnte im zweiten Ansatz (rechts und unten) durch den Einsatz von nutzerdefinierten Funktionen gelöst werden.

Fazit

Dieses Beispiel konnte in unter vier Stunden umgesetzt werden und damit sogar schneller als das ursprüngliche Anforderungsdokument in „Prosa“. Die Eingabe von langen Bedingungen ist in Enterprise Architect manchmal mühsam, da für die Übergangsbedingungen teilweise nur einzeilige Felder zur Verfügung stehen. Hier kann aber mit dem Sprachmittel der Definitionen geholfen werden.

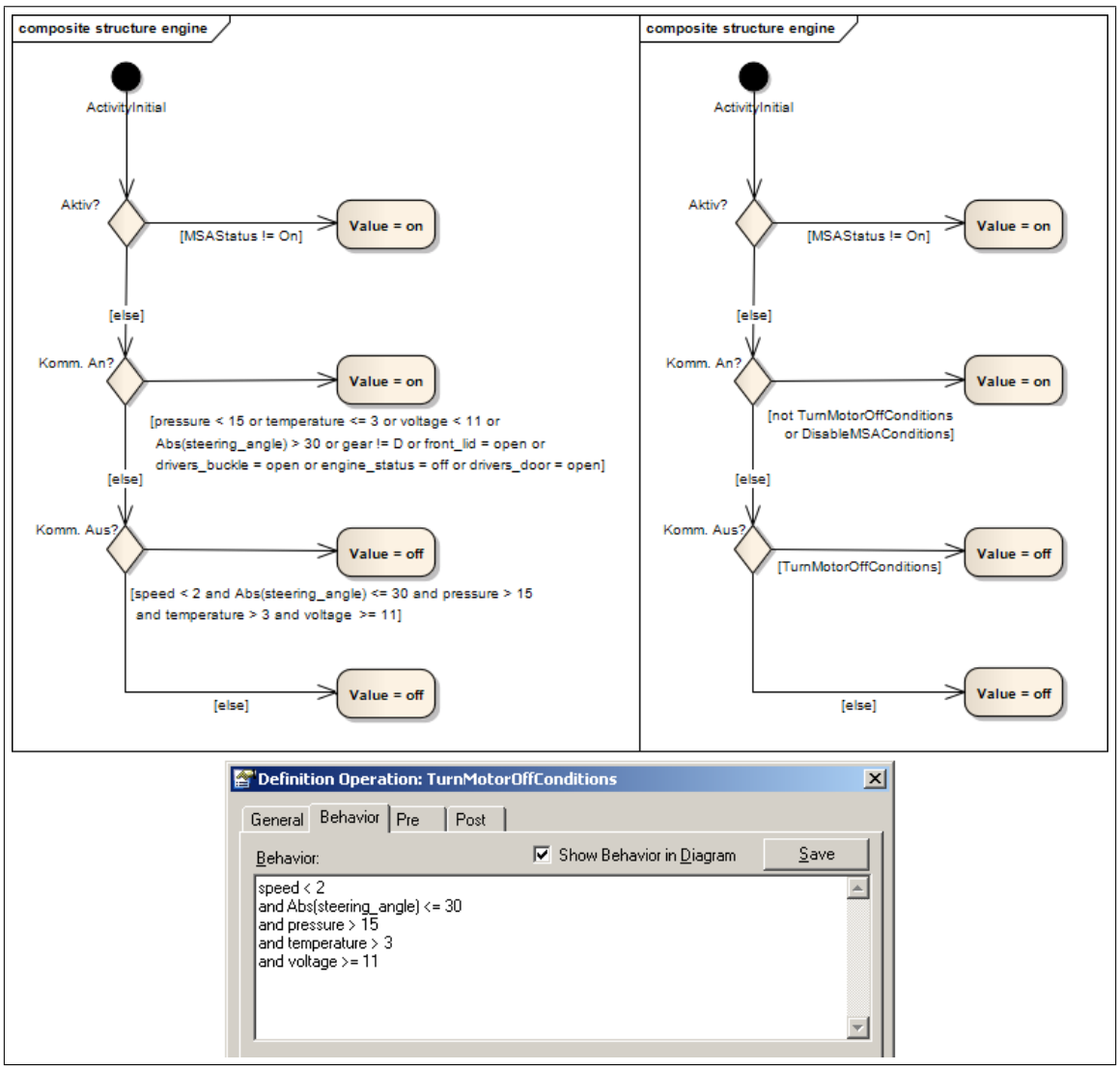


Abbildung 6.5: Modellierung der Start-/Stoppkommandierung

Beamersteuerung

Basierend auf einem echten Anwendungsfall soll die Steuerung eines Beamers mit zwei überwachten Lüftern nachgebildet werden. Interessant ist hierbei, dass der Eingabewert der Lüfter zunächst umgerechnet werden muss und der damit einhergehende Einsatz von Definitionen.

Kurzbeschreibung

Das Gerät soll den Ausfall eines Lüfters melden, aber erst bei Ausfall beider Lüfter abschalten. Die Lüfterdrehzahl muss umgerechnet werden, bevor sie mit dem spezifizierten Schwellwert verglichen werden kann; ein Lüfter gilt als defekt, wenn er diesen Schwellwert nicht innerhalb einer gewissen Zeitspanne erreicht (Einschalten kommandiert) bzw. unterschreitet (Ausschalten kommandiert). Hinzu kommt der übliche Nachlauf sowie weitere Restriktionen für das An- und Ausschalten des Geräts (das Gerät muss mindestens 60 Sekunden ausgeschaltet bleiben, zum Ausschalten ist der Knopf zweimal zu drücken).

Die vollständige Spezifikation findet sich in Anhang 3.

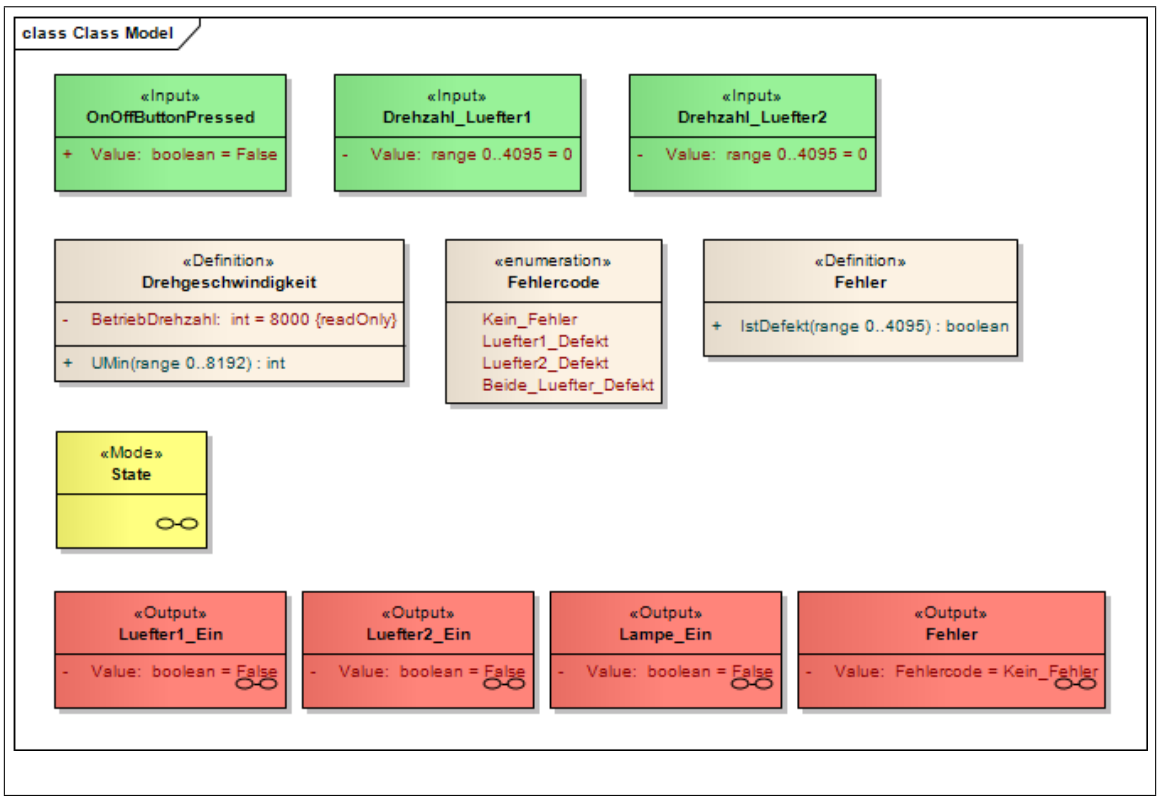


Abbildung 6.6: Strukturelemente der Beamersteuerung

Umsetzung

Die beiden Definitionen für einen Defekt und die Umrechnung der Lüfterdrehzahlen sind offensichtliche Kandidaten für eine Funktionsdefinition in SCR-EA. Zu beachten ist, dass Parameter nicht nur mit ihrem aktuellen Wert übergeben werden, sondern der komplette Trace zur Verfügung steht; es ist daher kein Problem, Parameter mit Pre() und Duration() zu verwenden. Ohne die if-Ausdrücke hätte IstDefekt nicht als Funktion umgesetzt werden können und wäre mit einem Mode-Element pro Lüfter zu simulieren gewesen. (Abbildung 6.6 und Abbildung 6.7, links)

Der gegebene Ausdruck für die Umrechnung muss etwas umgestellt werden, um für Integerarithmetik geeignet zu sein (Abbildung 6.7, rechts).

Für den Gerätezustand werden verschachtelte Zustände zur Gruppierung genutzt, so dass die Logik für das Einschalten der Lampe lediglich auf „Eingeschaltet“ prüfen muss (Abbildungen 6.9 und 6.8).

Fazit

Auch dieses Beispiel konnte in unter einem Arbeitstag von der Spezifikation zu einem testbaren C-Programm überführt werden. Es wurden aber auch verbleibende Schwachpunkte der bestehenden Implementierung aufgezeigt.

Ein Fallstrick war die nötige Umstellung des Ausdrucks für die Umrechnung der Lüfterdrehzahl. Hier wurde prompt übersehen, dass die mittlere Konstante unter Integerarithmetik zu 0 auswertet. Zwar kann dies in diesem Fall mangelndem Kaffee angehängt werden, aber es handelt sich um eine eigentlich unnötige Transformationsleistung beim Umsetzen der Spezifikation, die relativ leicht zu Fehlern führen kann. Für zukünftige Erweiterungen ist zu überlegen, ob und wie Arithmetik auf rationalen Zahlen zugelassen werden soll (dies muss nicht zwangsläufig Gleitkommaarithmetik sein).

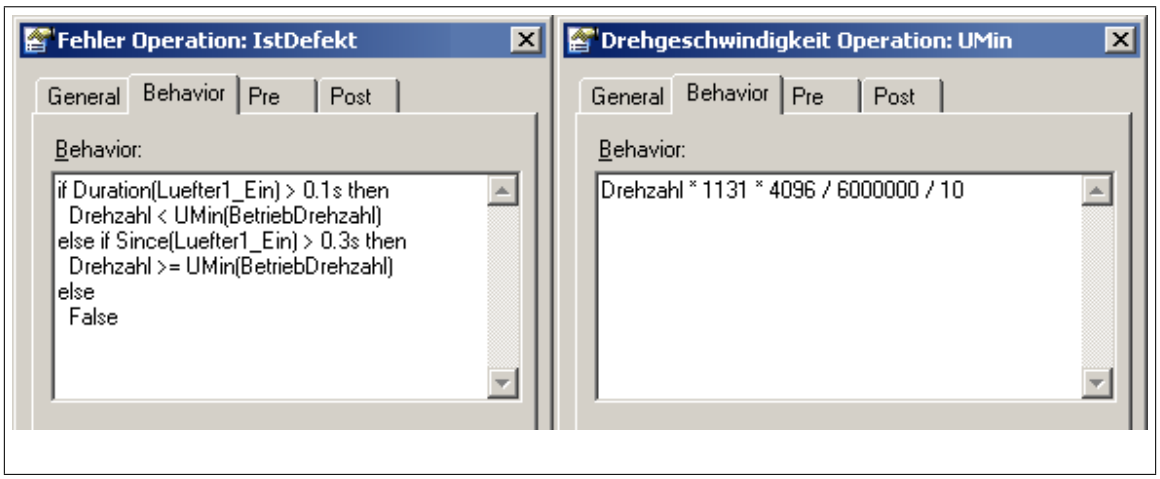


Abbildung 6.7: Definition von `IstDefekt()` und `UMin()`

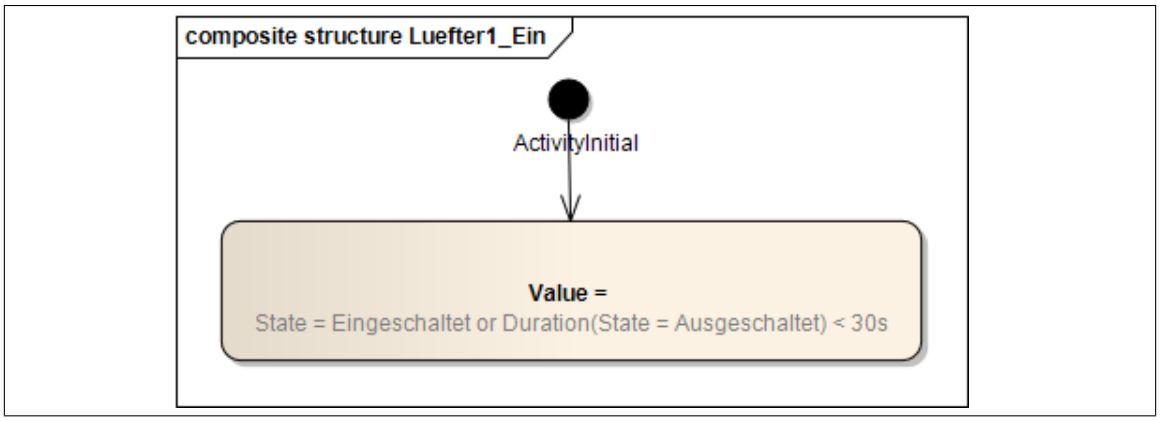


Abbildung 6.8: Einschaltbedingung für Lüfter

Die Übergangsbedingung „Automat war seit gewisser Zeit in diesem Zustand“ erwies sich als erstaunlich umständlich zu spezifizieren (6.9, hervorgehobener Übergang). Nicht nur muss der Name des Automats wiederholt werden, besonders unintuitiv ist, dass ein `Pre()` einzufügen ist, da sonst eine zyklische Abhängigkeit entsteht – eine Folge der in Kapitel 3 diskutierten Umstellung auf die SCADE-Semantik. Dies könnte mit Syntaxzucker wie etwa einem zusätzlichen Makro `HasBeenIn(state, time)` gelöst werden.

Die Verschachtelung von Definitionen erwies sich als elegant bei der Modellierung; da Funktionen aber momentan durch reines Einsetzen implementiert sind, ist das Ziel, lesbaren Code zu generieren, dabei schnell verfehlt (Abbildung 6.10, oben).

Als subtiles Problem trat dabei zunächst auf, dass auch die Vorteile von eingesetztem Code, nämlich bessere Optimierung, nicht genutzt wurden – die Bereichsinferenz endete weiterhin an Funktionsgrenzen und konnte daher nicht erkennen, dass der Ausdruck `UMin(8000)` eine Konstante ist. Die Optimierung wurde kurzfristig nachgerüstet und führt in diesem Fall zu einer deutlichen Vereinfachung (Abbildung 6.10, unten). Für eine Weiterentwicklung stellt sich allerdings dennoch die Frage, wie und in welchen Situationen Funktionen als eigene Einheiten übersetzt werden sollten.

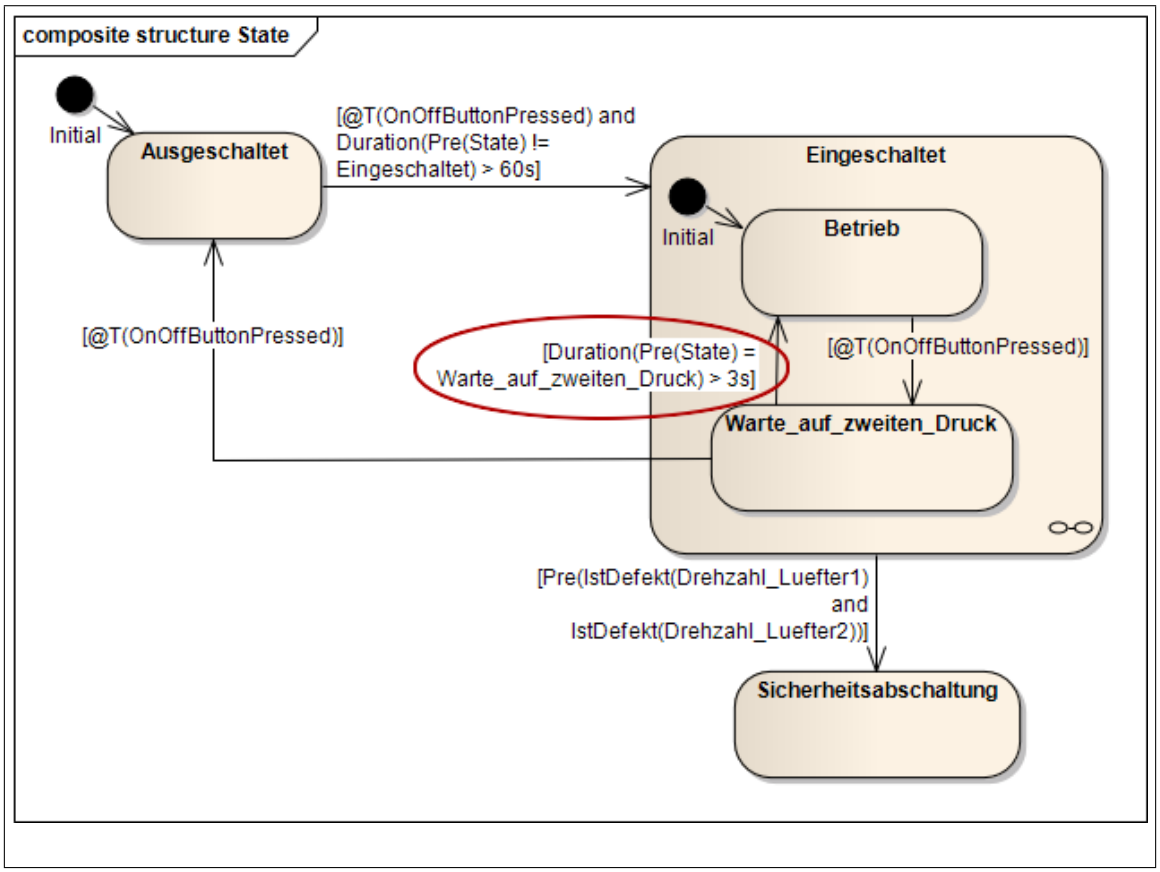


Abbildung 6.9: Zustandsmaschine für die Lüftersteuerung

```
119     /* Pre(IstDefekt(Drehzahl_Luefter1) and IstDefekt(Drehzahl_Luefter2)) */
120     if (((/*IstDefekt()*/((((dc_value(&scrb_State.durationCounters[2]) > 2) ? (scrb_Drehzahl_Luefter1.
        values[1] < /*UMin()*/(int32_t)checkRange(safeDiv(safeDiv(((int64_t)(8000 * 1131) * (int64_t)4096
        ), 6000000), 10), -2147483648, 2147483647)) : ((dc_value(&scrb_State.durationCounters[3]) > 6) ? (
        scrb_Drehzahl_Luefter1.values[1] >= /*UMin()*/(int32_t)checkRange(safeDiv(safeDiv(((int64_t)(8000
        * 1131) * (int64_t)4096), 6000000), 10), -2147483648, 2147483647)) : /*False=*/((1 != 1))))) &&
        /*IstDefekt()*/(((dc_value(&scrb_State.durationCounters[4]) > 2) ? (scrb_Drehzahl_Luefter2.values[
        1] < /*UMin()*/(int32_t)checkRange(safeDiv(safeDiv(((int64_t)(8000 * 1131) * (int64_t)4096),
        6000000), 10), -2147483648, 2147483647)) : ((dc_value(&scrb_State.durationCounters[5]) > 6) ? (
        scrb_Drehzahl_Luefter2.values[1] >= /*UMin()*/(int32_t)checkRange(safeDiv(safeDiv(((int64_t)(8000
        * 1131) * (int64_t)4096), 6000000), 10), -2147483648, 2147483647)) : /*False=*/((1 != 1)))))))) {
121         scrb_State_enter(State_STATES_Sicherheitsabschaltung, EM_NORMAL);
```

```
119     /* Pre(IstDefekt(Drehzahl_Luefter1) and IstDefekt(Drehzahl_Luefter2)) */
120     if (((/*IstDefekt()*/((((dc_value(&scrb_State.durationCounters[2]) > 2) ? (scrb_Drehzahl_Luefter1.
        values[1] < 617) : ((dc_value(&scrb_State.durationCounters[3]) > 6) ? (scrb_Drehzahl_Luefter1.
        values[1] >= 617) : /*False=*/((1 != 1))))) && /*IstDefekt()*/(((dc_value(&scrb_State.
        durationCounters[4]) > 2) ? (scrb_Drehzahl_Luefter2.values[1] < 617) : ((dc_value(&scrb_State.
        durationCounters[5]) > 6) ? (scrb_Drehzahl_Luefter2.values[1] >= 617) : /*False=*/((1 != 1)))))))
        {
121         scrb_State_enter(State_STATES_Sicherheitsabschaltung, EM_NORMAL);
```

Abbildung 6.10: Erzeugter Code vor und nach der Optimierung von `UMin(8000)`

Kapitel 7

Fazit

Mögliche Erweiterungen

Da es sich bei dieser Arbeit um eine Neuentwicklung handelt, ergaben sich zahlreiche Ideen, wie das Projekt weiter geführt werden kann. Die folgenden Bemerkungen sind nach Einschätzung des Autors priorisiert.

Funktionen als Einheit erhalten
Funktionen werden in der aktuellen Implementierung immer eingesetzt (Inlining). Dies hat zwar Vorteile für die Optimierung, widerspricht aber dem Ziel, möglichst lesbaren Code zu erzeugen. Wie in Kapitel 4, „Umsetzung von Pre und Funktionen" erläutert, müssten Funktionen mit ihrem Zeitversatz parametrisiert oder Pre anders implementiert werden.

Als Teillösung ist denkbar, zumindest Funktionen, die nur von ihren Parametern abhängen, als Funktionen zu übersetzen. Diese wurde nicht umgesetzt, da in der Fallstudie nur eine einzige solche Funktion auftrat, die dann nur mit konstanten Argumenten aufgerufen wurde.

Abschätzungen verbessern
Die Programmstruktur von SCR ist relativ eingeschränkt und die Constraints für einen Ausdruck daher erheblich leichter zu verfolgen als in üblichen Programmiersprachen. Durch das Einbinden eines Solvers könnten die Bereichsabschätzungen vermutlich so weit verbessert werden, dass der Compiler Bereichsverletzungen als Fehler statt als Warnung werten kann und Laufzeitfehler damit ausgeschlossen werden können.

Automatische Validierung, Testfallerzeugung
In eine ähnliche Richtung schlagen auch diese beiden Erweiterungen; alle erfordern ein besseres Programmverständnis, das etwa mit einem SMT[1]-Solver erreicht werden könnte.

Hiermit wäre es beispielsweise möglich, zu jedem Zustand eines Automaten einen Testfall zu generieren, der in diesen Zustand führt. Insbesonders kann aus unerreichbaren Zuständen auf einen Fehler bei der Modellierung oder in der Spezifikation geschlossen werden.

Als weiteres Beispiel könnte ein Beweis geführt werden, in welchen Fällen der unbedingte else-Zweig einer Entscheidungskette ausgeführt wird. Der Übersetzer würde ihn dann genau fordern, wenn die realisierte Funktion nicht total ist.

[1] Satisfiability Modulo Theories. Die Erweiterung eines SAT-Solvers um Theorien (im mathematischen Sinn), also etwa Integerarithmetik oder Funktionen.

Bessere Integration in EA, Unterstützung der Modellierung
Die Modellierung etwa einer Entscheidungskette ist derzeit relativ mühsam und könnte etwa in einer tabellarischen Darstellung ähnlich der ursprünglichen SCR schneller erfolgen. Dies liegt deutlich außerhalb dem Rahmen dieser Arbeit und kann als unabhängiges Erweiterungsprojekt gesehen werden.

Tatsächlich wurde in der ICS AG bei den Vorarbeiten für diese Arbeit bereits ein Prototyp eines solchen Plugins erstellt.

Anbindung an weitere UML-Werkzeuge
war nicht im Fokus der Aufgabenstellung, durch konsequente Kapselung (siehe Entwurf) wurde dieser Weg aber nicht verbaut. Ein Frontend, das etwa generische XMI[2]-Dateien verarbeitet, ist problemlos denkbar. Lediglich ein Mechanismus zur Auswahl des geeigneten Parsers muss noch ergänzt werden.

Gleitkommaarithmetik
Obwohl Gleitkommaarithmetik sowohl den Beweis der Korrektheit (aufgrund der bei Operationen auftretenden Rundungsfehler) sowie der Laufzeit (wegen schlechter Unterstützung auf vielen Embedded-CPUs) erschwert, ist sie für viele physikalische Prozesse natürlicher zu verwenden als Ganzzahlarithmetik. Die Umstellung eines gegebenen Terms auf Ganzzahlarithmetik erwies sich bei den Fallstudien als potenzielle Fehlerquelle. Es wäre wünschenswert, dies zumindest für Konstanten zu unterstützen, zumal dies für die ebenfalls rationalen Zeitangaben bereits in sehr ähnlicher Form implementiert ist.

Erweiterung um Verbundtypen (Structs)
Da die Sprache nur implizite Zuweisungen und keine Statements definiert, war lange unklar, wie eine sinnvolle Syntax für den Umgang mit Verbundtypen aussehen kann. Ein nur lesender Zugriff wurde als zu eingeschränkt erachtet. Daher wurde auf eine Implementierung in dieser Arbeit verzichtet.

Für Verbundtypen denkbar wäre etwa eine Compound-Zuweisung in der Form `Value = {a, b, c}`. Diese wäre vor allem dann sinnvoll, wenn eine komplexe Entscheidungskette zu dieser Zuweisung führt. Derzeit müsste diese Zuweisung als drei Output-Variablen modelliert und die Entscheidungskette damit repliziert werden.

Zusammenfassung und Ausblick

Mit dieser Arbeit wurde der Grundstein für ein neues SCR-Toolset, basierend auf grafischer Notation in UML gelegt. Der Compiler ist mit dem C-Codegenerator bereits für den praktischen Einsatz geeignet, während der C#-Codegenerator bequemes Debugging und Rapid Prototyping erlaubt (wenn etwa ein Mockup der Bedienoberfläche sofort mit Funktion unterlegt werden kann).

Der Sprachumfang wurde bewusst klein gehalten und der Fokus auf korrekte Codegenerierung gelegt. Von Anfang an wurden Unit-Tests eingesetzt, gegen Ende des Projekts auch automatische Übersetzung und Tests der generierten Programme mit vorgefertigten Eingabesequenzen. Insbesondere Regressionen konnten damit während der Entwicklung unverzüglich erkannt werden. Die durch die ICS AG durchgeführten Systemtests konnten dann auch nur ein einziges Generierungsproblem aufzeigen, aber etliche Unklarheiten an den informellen Anforderungsdokumenten.

[2]XML Metadata Interchange, ein häufig für UML-Modelle verwendetes Austauschformat.

Nicht in dieser Arbeit umgesetzt wurden Bestandteile, die über dieses Kernsystem hinausgehen: so ist zum einen eine bessere automatische Validierung des Modells und Unterstützung bei der Modellierung wünschenswert.

Wieder einmal zeigte sich, dass Sprachdesign keine dankbare Aufgabe ist und viel Erfahrung fordert. Einige Entscheidungen des Sprachdesigns erwiesen sich in den Fallstudien als der schlechtere Weg:

Die gewählte Umsetzung von Pre durch Durchschieben des Zeitversatzes erwies sich durch das Einsetzen der Funktionen als unelegant. Grundsätzlich ist aber nichts daran auszusetzen, Pre auf beliebige Ausdrücke anwenden zu können.

Der Verzicht auf tabellarische Darstellung erwies sich zumindest für die Modellierung als hinderlich. Dies liegt auch daran, dass UML-Werkzeuge hier zu viele Freiheitsgrade lassen und UML-Profile keine Möglichkeit enthalten, die Zusammenhänge zwischen den Elementen im Sinne einer BNF zu spezifizieren.

Die Unterstützung von hierarchischen Automaten hingegen ist aus Sicht des Autors äußerst hilfreich, und sei es nur zur Gruppierung von Zuständen. So kann in der Beameransteuerung etwa der Zustand innerhalb „Angeschaltet“ beliebig modelliert werden, ohne dass die Ausgabeelemente für Lampen und Lüfter von den Änderungen betroffen sind.

Mehrere frühe Entwurfsentscheidungen konnten bereits mit dem Prototyp getestet (und als verbesserungswürdig erkannt) werden. Die Entscheidung, den ursprünglichen Zeitplan für diesen Prototyp anzupassen, erwies sich damit als überaus nützlich.

Das Ziel, lesbaren Code zu generieren, wurde nicht vollständig erreicht. Dies liegt zum einen daran, dass Funktionen nur inline verwendet und damit gegebenfalls unnötig repliziert werden; zum anderen daran, dass die eingesetzte Intervallabschätzung zu schwach war, um Laufzeitprüfungen überflüssig zu machen. Letzteres wirkt sich vor allem auf den C-Code negativ aus, bei dem viele arithmetische Operationen durch Funktionsaufrufe ersetzt werden müssen.

Kapitel A

Anhänge

Hinweis: Die folgenden Dokumente sind nur auf der CD vorhanden:

- Ausführliche Sprachreferenz
- Entwicklungsdokumente (Spezifikation, Entwurf, Code, Codedokumentation, Wartungshinweise)
- Überdeckungsberichte
- Benutzerdokumentation.

A.1 Spezifikation der Motor-Start-Stopp-Automatik

Motor Start-Stop-Automatik Software Requirements

Funktionsübersicht

Die Motor Start-Stop-Automatik zeigt die Funktionalität eines Steuergerätes im Automobil, das in Situationen, in denen das Automobil eine gewisse Zeit stehen muss (z.B. an Ampeln, im Stau …) kurzzeitig den Motor abschaltet, um den Benzinverbrauch, Lärmemissionen etc. zu reduzieren. Im Gegensatz zum normalen Abschalten des Motos mithilfe der Zündung soll der Motor aber bei der Abschaltung durch die Start-Stop-Automatik „in Bereitschaft“bleiben, d.h. er soll beim Lösen der Bremse sofort wieder anspringen und anfahren. Die Funktionalität des hier modellierten Steuergeräts betrifft nur die Steuerung der Ab- und wieder Anschaltung des Motors an sich, nicht sonstige bei Nutzung einer Start-Stop-Automatik zusätzlich notwendige Vorgänge wie der permanente Aufbau einer hinreichenden Spannung, Wärme etc. am Anlasser des Motors, um das schnelle Anlassen physikalisch gewährleisten zu können.

Die Steuerung der Ab- und Anschaltung soll in Abhängigkeit von einer Reihe von Eingängen geschehen und hauptsächlich einen binären Ausgang steuern, nämlich das Signal „Motor ausschalten“ bzw. „Motor wieder anschalten“.

Das Fahrzeugsystem kann dann das Signal weiterverarbeiten, so kann es natürlich sein, dass von Seiten der Start-Stop-Automatik der Motor angeschaltet bleiben sollte, aber aus anderen Gründen der Motor doch abgeschaltet wird. Das bisher beschriebene boolesche Haupt-Ausgangssignal der Start-Stop-Automatik wird also noch mit anderen Signalen mit einem logischen „und“ verknüpft.

Requirements

1. Das System arbeitet nur in einem Fahrzeug mit Automatik-Getriebe. Die Gänge heißen „P“, „R“, „N“ und „D“.
2. Der Tastendruck des Start-Stop Schalters wird auf einen Zustand (aktiv/inaktiv) gelegt. Zustandswechsel geschehen bei Übergang von Off nach On des Schalters. Der Zustandswechsel wird blockiert, falls die Haube, der Gurt oder die Türe auf sind. Ein Vorwählen ist nicht erlaubt.

3. Die Start-Stop-Automatik wird aktiviert falls alle folgenden Bedingungen gelten:
    - Die Motorhaube ist zu
    - Die Fahrertür ist zu
    - Der Fahrer ist angeschnallt (Fahrergurtschloss ist geschlossen)
    - Der Fahrgang („D“) ist eingelegt
    - Der Start-Stop Schalter ist aktiv
    - Der Motor läuft

4. Die Start-Stop Automatik wird deaktiviert, falls eine der folgenden Bedingungen gilt:
    - Die Motorhaube ist auf, oder
    - Die Fahrertür ist auf, oder
    - Der Fahrer ist nicht angeschnallt, oder
    - Der Fahrgang ist nicht D, oder
    - Motor ist aus, oder
    - Der Start-Stop-Schalter ist nicht aktiv
5. We die Start-Stop-Automatik inaktiv ist kommandiert die Start-Stop-Automatik Motor an.
6. Wenn die Start-Stop-Automatik aktiv ist und alle folgenden Bedingungengelten, kommandiert die Start-Stop-Automatik Motor aus:
    - Das Fahrzeug steht (Geschwindigkeit < 2 km/h)
    - Der Einschlagwinkel des Lenkrads ist zwischen 30 ° oder -30 ° (jeweils absolut)
    - Der Fahrer steht länger als 1,5s auf der Bremse (der Bremsdruck beträgt => 1.5 N/m^2)
    - Die Außentemperatur beträgt über 3° Celsius
    - Die Batteriespannung beträgt mindestens 11 Volt.

7. Wenn die Start-Stop-Automatik aktiv ist und eine der folgenden Bedingungen gilt, kommandiert die Start-Stop-Automatik Motor an:
    - Der Fahrer geht von der Bremse (der Bremsdruck beträgt < 1.5 N/m^2)
    - Die Außentemperatur fäll tauf oder unter 3° Celsius
    - Die Batteriespannung fällt unter 11 Volt
    - Der Einschlagwinkel des Lenkrads wird auf über 30 ° oder unter -30 ° (jeweils absolut) erhöht
    - Der Fahrgang ist nicht „D“
    - Die Motorhaube wird geöffnet
    - Das Fahrergurtschloss wird geöffnet
    - Der Motor wird manuell ausgeschaltet, z.B. über die Zündung
    - Die Fahrertür wird geöffnet

Schnittstellen

Monitored:

- **Int** speed: Geschwindigkeit des Wagens in km/h
- **Int** temperature: Außentemperatur in Grad Celsius
- **Int** voltage: Batteriespannung in Volt
- **Int** pressure: Bremsdruck in 0,1 N/m^2
- **Int** steering_angle: Einschlagwinkel des Lenkrads in Grad (-179 bis 180)
- **Enum {open, closed}** drivers_buckle: Fahrergurtschloss offen o. geschlossen (=Gurt angelegt)
- **Enum {P, R, N, D}** gear: Aktuell eingelegter Gang
- **Enum {open, closed}** front_lid: Motorhaube

- **Enum {open, closed}** drivers_door: Fahrertür
- **Enum {on, off}** engine_status: Läuft der Motor oder nicht
- **Enum {pressed, notpressed}** msa_key: Start-Stop-Taste gedrückt oder nicht gedrückt

Controlled:

- **Enum {active, inactive}**: msa_status: Zustand ob MSA aktiv oder inaktiv ist (Zustand wir auf einen Ausgang gelegt).
- **Enum {on, off}**: msa_led: Anzeige, ob MSA aktiv oder inaktiv ist
- **Enum {on, off}** engine: Motor an oder ausschalten

A.2 Testsequenz MSA

Test Step	Wait Cycles	Input 1 drivers_buckle	Input 2 drivers_door	Input 3 engine_status	Input 4 front_lid	Input 5 gear	Input 6 msa_key	Input 7 pressure	Input 8 speed	Input 9 steering_angle	Input 10 temperature	Input 11 voltage	Output 1 engine	Output 2 msa_led	Output 3 msa_status	Description	Testcase Requirement
1		closed	closed	off	closed	P	notpressed	0	0	0	0	0	on	off	inactive	initial state	Req 4 Req 5
2		closed	closed	off	closed	P	pressed	0	0	0	0	0	on	off	inactive	set SwitchStatus to on	Req 2
3		open	closed	on	closed	D	notpressed	0	0	0	0	0	on	off	inactive	all conditions are fullfilled except drivers_buckle = closed	Req 3
4		closed	open	on	closed	D	notpressed	0	0	0	0	0	on	off	inactive	all conditions are fullfilled except drivers_door = closed	Req 3
5		closed	closed	on	open	D	notpressed	0	0	0	0	0	on	off	inactive	all conditions are fullfilled except front_lid = closed	Req 3
6		closed	closed	on	closed	P	notpressed	0	0	0	0	0	on	off	inactive	all conditions are fullfilled except gear = D (set to P)	Req 3
7		closed	closed	on	closed	D	pressed	0	0	0	0	0	on	off	inactive	set SwitchStatus to off	Req 2
8		closed	closed	on	closed	D	notpressed	0	0	0	0	0	on	off	inactive	all conditions are fullfilled except SwitchStatus = on	Req 3
9		closed	closed	off	closed	D	pressed	0	0	0	0	0	on	off	inactive	set SwitchStatus to on again all conditions are fulfilled except engine_status = on	Req 2 Req 3
10		closed	closed	on	closed	D	notpressed	0	0	0	0	0	on	on	active	all conditions are fulfilled msa_status set to active in addition msa_led set to on	Req 3
11		open	closed	on	closed	D	notpressed	0	0	0	0	0	on	off	inactive	stop msa by setting drivers_buckle = True	Req 4
12		closed	closed	on	closed	D	notpressed	0	0	0	0	0	on	on	active	set msa_status back to active	Req 3
13		closed	open	on	closed	D	notpressed	0	0	0	0	0	on	off	inactive	stop msa by setting drivers_door = True	Req 4
14		closed	closed	on	closed	D	notpressed	0	0	0	0	0	on	on	active	set msa_status back to active	Req 3
15		closed	closed	on	open	D	notpressed	0	0	0	0	0	on	off	inactive	stop msa by setting front_lid = True	Req 4
16		closed	closed	on	closed	D	notpressed	0	0	0	0	0	on	on	active	set msa_status back to active	Req 3
17		closed	closed	on	closed	N	notpressed	0	0	0	0	0	on	off	inactive	stop msa by setting gear = N	Req 4
18		closed	closed	on	closed	D	notpressed	0	0	0	0	0	on	on	active	set msa_status back to active	Req 3
19		closed	closed	off	closed	D	notpressed	0	0	0	0	0	on	off	inactive	stop msa by setting engine_status = off	Req 4
20		closed	closed	on	closed	D	notpressed	0	0	0	0	0	on	on	active	set msa_status back to active	Req 3
21		closed	closed	on	closed	D	pressed	0	0	0	0	0	on	off	inactive	stop msa by setting SwitchStatus = True	Req 2 Req 4
22		closed	closed	on	closed	D	notpressed	0	0	0	0	0	on	off	inactive	release msa_key nothing happens	Req 2
23		closed	closed	on	closed	D	pressed	0	0	0	0	0	on	on	active	set msa_status back to active by pressing msa_key again (on)	Req 2 Req 3

24	151	closed	closed	on	closed	D	pressed	15	2	0	4	11	on	on	active	all conditions of Req 6 fullfilled (wait 150 steps due to break condition) except speed < 2 km/h	Req 6
175		closed	closed	on	closed	D	pressed	15	0	-31	4	11	on	on	active	all conditions of Req 6 fullfilled except -30 <= steering_angle <= 30 (=-31)	Req 6
176		closed	closed	on	closed	D	pressed	15	0	31	4	11	on	on	active	all conditions of Req 6 fullfilled except -30 <= steering_angle <= 30 (=31)	Req 6
177		closed	closed	on	closed	D	pressed	15	0	0	3	11	on	on	active	all conditions of Req 6 fullfilled except temperature > 3 (=3)	Req 6
178		closed	closed	on	closed	D	pressed	15	0	0	4	10	on	on	active	all conditions of Req 6 fullfilled except voltage => 11 (=10)	Req 6
179		closed	closed	on	closed	D	pressed	14	0	0	4	11	on	on	active	all conditions of Req 6 fullfilled except pressure >= 15 (=14)	Req 6
180	150	closed	closed	on	closed	D	pressed	15	0	0	4	11	on	on	active	all conditions of Req 6 fullfilled for exactly 1,5s	Req 6
330		closed	closed	on	closed	D	pressed	14	0	0	4	11	on	on	active	all conditions of Req 6 fullfilled except pressure >= 15 (=14)	Req 6
331	151	closed	closed	on	closed	D	pressed	15	0	0	4	11	off	on	active	all conditions of Req 6 fullfilled for longer than 1,5s	Req 6
482		closed	closed	on	closed	D	pressed	15	0	0	4	11	off	on	active	wait one step all conditins of Req 7 = False	Req 6 Req 7
483		closed	closed	on	closed	D	pressed	14	0	0	4	11	on	on	active	restart engine by setting pressure = 14	Req 7
484	151	closed	closed	on	closed	D	pressed	15	0	0	4	11	off	on	active	stop engine again	Req 6
635		closed	closed	on	closed	D	pressed	15	0	0	3	11	on	on	active	restart engine by setting temperature = 3	Req 7
636		closed	closed	on	closed	D	pressed	15	0	0	4	11	off	on	active	stop engine again	Req 6
637		closed	closed	on	closed	D	pressed	15	0	0	4	10	on	on	active	restart engine by setting voltage = 10	Req 7
638		closed	closed	on	closed	D	pressed	15	0	0	4	11	off	on	active	stop engine again	Req 6
639		closed	closed	on	closed	D	pressed	15	0	31	4	11	on	on	active	restart engine by setting steering_angle = 31	Req 7
640		closed	closed	on	closed	D	pressed	15	0	0	4	11	off	on	active	stop engine again	Req 6
641		closed	closed	on	closed	D	pressed	15	0	-31	4	11	on	on	active	restart engine by setting steering_angle = -31	Req 7
642		closed	closed	on	closed	D	pressed	15	0	0	4	11	off	on	active	stop engine again	Req 6
643		closed	closed	on	closed	R	pressed	15	0	0	4	11	on	off	inactive	restart engine by setting gear = R	Req 7
644		closed	closed	on	closed	D	pressed	15	0	0	4	11	off	on	active	stop engine again	Req 6
645		open	closed	on	closed	D	pressed	15	0	0	4	11	on	off	inactive	restart engine by setting drivers_buckle = open	Req 7
646		closed	closed	on	closed	D	pressed	15	0	0	4	11	off	on	active	stop engine again	Req 6

A.3 Spezifikation der Beamersteuerung

SCR-Fallstudie: Beamer-Ansteuerung

Hier geht es um die (Software-) Ansteuerung eines Beamers. Inputs und Outputs beziehen sich auf die Steuerungssoftware, die im Beamer integriert ist.

Inputs

Button zum intelligenten Ein- und Ausschalten des Beamers, d.h. der Lampe des Beamers. Es wird vorausgesetzt, dass ein Ein-/Ausschalter zur Stromversorgung existiert, der bereits eingeschaltet sein muss, damit die Steuerung überhaupt aktiv werden kann.

```
OnOffButtonPressed: Bool := False
```

Rückmeldung der Drehzahlen der beiden im Beamer integrierten Lüfter. Die Drehzahlen der Lüfter sind jeweils wie folgt codiert: U/min des Lüfters wird berechnet durch *Drehzahl* * (11,31 / 60000) * 4096 / 10.

```
Drehzahl_Lüfter1: Uint12 := 0
Drehzahl_Lüfter2: Uint12 := 0
```

Outputs

Einschalten der beiden Lüfter:

```
Lüfter1_Ein: Bool := False
Lüfter2_Ein: Bool := False
```

Einschalten der Beamerlampe:

```
Lampe_Ein: Bool := False
```

Fehlermeldungen (zur Anzeige z.B. am Schirm und/oder eine LED am Gerät):

```
Fehler: enum {Kein_Fehler, Lüfter1_Defekt, Lüfter2_Defekt,
Beide_Lüfter_Defekt}
```

Anforderungen

Einschalten ist nur möglich, falls der Beamer mindestens seit 60 Sekunden ausgeschaltet war (Schutz vor schnellem An-/Ausschalten der Lampe).

Zum Ausschalten muss der OnOffButton zweimal gedrückt werden.

Solange der Beamer eingeschaltet ist, werden die Lampe und beide Lüfter eingeschaltet.

Wird der Beamer ausgeschaltet, wird die Lampe sofort ausgeschaltet, die Lüfter sollen jedoch noch 30 Sekunden nachlaufen.

Die Lüfter werden über die Eingänge Drehzahl_Lüfter1/2 auf korrekte Funktion überwacht: Erreicht ein Lüfter 100 ms nach dem Einschalten nicht die Mindestdrehzahl von 8000 U/min oder unterschreitet er nach dem Ausschalten diese Drehzahl nicht innerhalb von 300 ms, gilt er als defekt.

Sind beide Lüfter defekt, wird der Beamer ausgeschaltet und muss vom Netz getrennt werden, bevor er wieder angeschaltet werden kann.

Sequenz 1

Test Step	Wait Cycles	Input 1 Drehzahl_Luefter1	Input 2 Drehzahl_Luefter2	Input 3 OnOffButtonPressed	Output 1 Fehler	Output 2 Lampe_Ein	Output 3 Luefter1_Ein	Output 4 Luefter2_Ein	Description	Tested Requirement
1		0	0	False	Kein_Fehler	False	False	False	initial state	
2		0	0	True	Kein_Fehler	True	True	True	start beamer	Req 3
3	8	0	0	True	Kein_Fehler	True	True	True	wait 8 cycles	Req 5
11		617	617	False	Kein_Fehler	True	True	True	release OnOff Button set Drehzahl_Luefter1(2) exactly after 100ms to Threshold to simulate normal behaviour	Req 5
12		617	617	False	Kein_Fehler	True	True	True	normal operation	
13		617	617	True	Kein_Fehler	True	True	True	shut down beamer press OnOff Button first time	Req 2
14		617	617	False	Kein_Fehler	True	True	True	release OnOff Button	Req 2
15	299	617	617	False	Kein_Fehler	True	True	True	wait 299 cycles (>3s)	Req 2
314		617	617	True	Kein_Fehler	True	True	True	press OnOff Button second time but one step too late. So it is the first time again	Req 2
315		617	617	False	Kein_Fehler	True	True	True	release OnOff Button	Req 2
316	298	617	617	False	Kein_Fehler	True	True	True	wait 298 cycles (=3s)	Req 2
614		617	617	True	Kein_Fehler	False	True	True	press OnOff Button second time	Req 2
615	3000	617	617	False	Kein_Fehler	False	False	False	wait 2999 further cycles for fans to go out	Req 4
3615	30	616	616	False	Kein_Fehler	False	False	False	set Drehzahl_Luefter1(2) < Threshold to simulate normal behaviour	Req 5
3645		0	0	False	Kein_Fehler	False	False	False	initial state	
3646		0	0	True	Kein_Fehler	False	False	False	start beamer but no reaction due to 60s dead time (since Lampe_Ein = False)	Req 1
3647	2966	0	0	False	Kein_Fehler	False	False	False	release button	Req 1

6613		0	0	True	Kein_Fehler	False	False	False	dead time (on step before timer expires)	Req 1
6614		0	0	False	Kein_Fehler	False	False	False	release button	Req 1
6615		0	0	True	Kein_Fehler	True	True	True	start beamer (60s over)	Req 1
6616	8	0	0	True	Kein_Fehler	True	True	True	wait 8 cycles	Req 5
6624		617	617	False	Kein_Fehler	True	True	True	release OnOff-Button set Drehzahl_Luefter1(2) exactly after 100ms to Threshold to simulate normal behaviour	Req 5
6625		616	617	False	Luefter1_Defekt	True	True	True	simulate error Drehzahl_Luefter1 < 8000 U/min	Req 5
6626		617	617	False	Luefter1_Defekt	True	True	True	remove failure failure message remains	Req 5
6627		617	616	False	Beide_Luefter_Defekt	False	True	True	simulate error Drehzahl_Luefter2 < 8000 U/min now both fans are defect	Req 5
6628		617	617	False	Beide_Luefter_Defekt	False	True	True	remove failure failure message remains	Req 5

Sequenz 2

Test Step	Wait Cycles	Input 1 Drehzahl_Luefter1	Input 2 Drehzahl_Luefter2	Input 3 OnOffButtonPressed	Output 1 Fehler	Output 2 Lampe_Ein	Output 3 Luefter1_Ein	Output 4 Luefter2_Ein	Description	Tested Requirement
1		0	0	False	Kein_Fehler	False	False	False	initial state	
2		0	0	True	Kein_Fehler	True	True	True	start beamer	Req 3
3	9	0	0	True	Kein_Fehler	True	True	True	wait 8 further cycles	Req 5
12		616	617	False	Luefter1_Defekt	True	True	True	release OnOff Button set Drehzahl_Luefter1 < Threshold to simulate error	Req 5

Sequenz 3: Variante für Luefter2_Defekt
Sequenz 4: Variante für Beide_Luefter_Defekt

Sequenz 5

Test Step	Wait Cycles	Input 1 Drehzahl_Luefter1	Input 2 Drehzahl_Luefter2	Input 3 OnOffButtonPressed	Output 1 Fehler	Output 2 Lampe_Ein	Output 3 Luefter1_Ein	Output 4 Luefter2_Ein	Description	Tested Requirement
1		0	0	False	Kein_Fehler	False	False	False	initial state	
2		0	0	True	Kein_Fehler	True	True	True	start beamer	Req 3
3	8	0	0	True	Kein_Fehler	True	True	True	wait 8 cycles	Req 5
11		617	617	False	Kein_Fehler	True	True	True	release OnOff Button set Drehzahl_Luefter1(2) exactly after 100ms to Threshold to simulate normal behaviour	Req 5
12		617	617	False	Kein_Fehler	True	True	True	normal operation	
13		617	617	True	Kein_Fehler	True	True	True	shut down beamer press OnOff Button first time	Req 2
14		617	617	False	Kein_Fehler	True	True	True	release OnOff Button	Req 2
15	298	617	617	False	Kein_Fehler	True	True	True	wait 298 cycles (=3s)	Req 2
313		617	617	True	Kein_Fehler	False	True	True	press OnOff Button second time	Req 2
314	3000	617	617	False	Kein_Fehler	False	False	False	release OnOff Button wait 2999 further cycles for fans to go out	Req 4
3314	30	617	616	False	Luefter1_Defekt	False	False	False	set Drehzahl_Luefter1 > Threshold to simulate error	Req 5
3344		0	0	False	Luefter1_Defekt	False	False	False	initial state	

Sequenz 6: Variante für Luefter2_Defekt
Sequenz 7: Variante für Beide_Luefter_Defekt

Literaturverzeichnis

[AhoUllman86] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, 1986.

[LL07] J. Ludewig, H. Lichter. *Software Engineering – Grundlagen, Menschen, Prozesse, Techniken*. dpunkt.verlag, 2007.

[Grune] Dick Grune, Ceriel J.H. Jacobs. *Parsing Techniques - A Practical Guide*, First Edition. `http://dickgrune.com/Books/PTAPG_1st_Edition`, 1990.

[Sulzmann] Martin Sulzmann. *A Tabular Requirement Specification Language – Syntax and Semantics*. Internes Papier der ICS AG (auf der CD enthalten).

[Heninger] K. L. Heninger et. al. *Specifying software requirements for complex systems: New techniques and their application*. In *IEEE Trans. Softw. Eng.*, vol. SE-6, Jan. 1980.

[FaulkBracket] S.R. Faulk, J. Bracket et. al. *The CoRE method for real-time requirements*. In *IEEE Transactions on Software Engineering*, Sept. 1992.

[Heitmeyer02] Constance L. Heitmeyer. *Software Cost Reduction*. Technical Report NRL 02-1221.1-1419.

[Heitmeyer07] Constance L. Heitmeyer. *Formal Methods for Specifying, Validating, and Verifying Requirements*. 2007.

[ParnasMadey] David L. Parnas and Jan Medey. *Functional documentation for computer systems*. In *Science of Computer Programming*, Okt. 1995.

[ANTLR] *ANTLR 3 Documentation*. `http://www.antlr.org/wiki/display/ANTLR3/`.

[MSDN] Microsoft *.NET Framework 4* documentation. `http://msdn.microsoft.com/en-us/library/w0x726c2(v=vs.100).aspx`.

[C-sharp] *C# Language Specification 4.0*. `http://www.microsoft.com/en-us/download/details.aspx?id=7029`, 2010.

[C90] *C Language Reference Manual*. Silicon Graphics, 2003.

[GnuC] *The GNU C Reference Manual*. `http://www.gnu.org/software/gnu-c-manual/`.

[Harel86] David Harel. *Statecharts: A Visual Formalism For Complex Systems*. In *Science of Computer Programming*, Volume 8 Issue 3, 1987.

[SMD] *State Machine Diagrams*. `http://www.uml-diagrams.org/state-machine-diagrams.html`

[CTesting] Practical Testing of a C99 Compiler Using Output Comparison. Access Systems Americas Inc. In *Software: Practice and Experience*, 2006.

[Helmer] Guy Helmer. *Safety Checklist for Four-Variable Requirements Methods*. Iowa State University, 1998.

[Santillan] Mayte Santillan, Kritika Sharma. *Software Cost Reduction* (Lecture).

[gcc52478] *-ftrapv calls the wrong functions in libgcc*. `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=52478`.

[Seacord] Robert C. Seacord. *Secure Coding in C And C++*. Addison-Wesley, 2005.

[Leitner] Felix von Leitner: *Catching Integer Overflows in C*. `http://www.fefe.de/intof.html`, 2007.

Abbildungsverzeichnis

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Wolfgang Fellger)