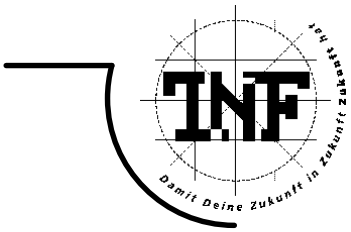
JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis



Eine objektorientierte virtuelle Maschine für die Programmiersprache Monaco

Bakkalaureatsarbeit
(Projektpraktikum)

zur Erlangung des akademischen Grades

Bakkalaureus der Technischen Wissenschaften

in der Studienrichtung

Informatik

Angefertigt am *Institut für Systemsoftware*

Betreuung:

o.Univ.-Prof. Dr. Dr. h.c. Hanspeter Mössenböck
Dipl.-Ing. Dr. Herbert Prähofer

Eingereicht von:

Roland Schatz, 0355521

Linz, 10. April 2007

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Bakkalaureatsarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, April 2007 Roland Schatz

Kurzfassung

Die Programmiersprache Monaco ist eine domänenspezifische Sprache für Maschinensteuerungen. Sie wurde als Basis für Endbenutzer-Programmiersysteme entwickelt. Monaco ist eine imperative Sprache mit einer ähnlichen Ausdruckskraft wie Statecharts. Sie ist komponentenbasiert und hat spezielle Sprachkonstrukte für Parallelität und asynchrone Ereignisbehandlung. Diese Arbeit beschreibt eine objektorientierte Datenstruktur (CodeDOM) für Monaco-Programme und eine virtuelle Maschine, die diese Datenstruktur ausführen kann. Die virtuelle Maschine wurde sprachunabhängig gestaltet. Sie stellt mehrere einfache Dienste wie kooperatives Multitasking und bedingtes Warten bereit. Die Ausführungslogik selbst ist nicht in der VM sondern im CodeDOM implementiert. Diese Sprachunabhängigkeit, zusammen mit der objektorientierten Struktur des CodeDOM, hat den Vorteil, dass das Experimentieren mit neuen Sprachkonstrukten und alternativen Notationen vereinfacht wird. Am Ende der Arbeit wird im Detail beschrieben, wie die einzelnen Sprachkonstrukte der Sprache Monaco unter Verwendung der von der virtuellen Maschine bereitgestellten Dienste ausgeführt werden.

Abstract

The language Monaco is a domain specific language for automation control. It was developed as a base of an end-user programming system. Monaco is an imperative language with expressive power similar to Statecharts. It is component-based and has special language constructs to support parallelism and asynchronous event handling. This thesis describes an object oriented data structure (CodeDOM) for storing Monaco programs and a virtual machine for executing this data structure. The virtual machine is defined in a language independent way. It defines several low-level services like cooperative multitasking and conditional waiting. The execution logic itself is not defined in the VM, it is implemented by the CodeDOM. This language independence, together with the object-oriented structure of the CodeDOM, has the advantage of making it very easy to experiment with new language constructs or alternative notations. At the end of this thesis, the execution of the most important language constructs of the Monaco language is described in terms of the services provided by the Monaco VM.

Inhaltsverzeichnis

1 Einleitung

Diese Bakkalaureatsarbeit wurde im Rahmen des Moduls „Domain-specific Languages for Industrial Automation“ des Christian Doppler Labors für Automated Software Engineering an der Johannes Kepler Universität Linz erstellt. In diesem Projekt wird in Zusammenarbeit mit der KEBA AG (www.keba.at) an einem Endbenutzer-Programmiersystem für Maschinenautomation gearbeitet.

Ziel des Projekts ist es, ein Framework für Endbenutzer-Programmiersysteme zu entwickeln. Dieses Framework soll es den Herstellern von Maschinen (OEMs) erleichtern, für ihre Kunden Programmiersysteme zu erstellen, mit denen die Maschinensteuerung von Endbenutzern in einfacher Weise parametrisiert und auch umprogrammiert werden kann.

1.1 Monaco

Als erster Schritt in Richtung Endbenutzer-Programmierung wurde die domänenspezifische Sprache „Monaco“ entwickelt [3]. Wichtige Merkmale dieser Sprache sind eine imperative Notation und Ereignisbehandlung. Maschinen arbeiten in einzelnen Fertigungsschritten, die sequentiell aufeinander folgen. Wie Gespräche mit Endbenutzern und Domänenexperten zeigten, ist auch das Denkmodell eines Maschinenbedieners imperativ.

Die Sprache Monaco ist komponentenbasiert. Damit lässt sich die Struktur der zu steuernden Maschinen direkt im Code abbilden. Die Komponenten sind streng hierarchisch angeordnet, d.h. eine Komponente kann bei der Durchführung ihrer Aufgaben nur die unmittelbar unter ihr liegenden Komponenten verwenden. Direkte Kommunikation zwischen Komponenten auf der gleichen Ebene ist nicht erlaubt. Das ermöglicht eine hierarchische Abstraktion auf mehreren Ebenen.

Ein wichtiger Vorteil dieses komponentenbasierten Ansatzes ist in dieser hierarchischen Abstraktion von Steuerungsaufgaben zu sehen. Auf der untersten Ebene sind Komponenten die nur Hardwaresignale setzen oder Regelprozesse anstoßen. Diese Komponenten sind sehr einfach, da sie nur für kleine Teile der Maschine zuständig sind. Weiter oben werden schrittweise immer komplexere Aufgaben durch Koordination der weiter unten liegenden Komponenten gelöst. Auf oberster Ebene werden dann nur mehr die in den unteren Ebenen implementierten Teilschritte aneinandergereiht. Das ist auch die Ebene, an der die Endbenutzer eventuell noch Änderungen vornehmen wollen.

Ein weiteren Aspekt von Monaco ist eine einfache Behandlung von asynchronen Er-

eignissen. Die meisten Steuerungsaufgaben in der Automatisierungsdomäne sind sehr einfach und rein sequentiell aufgebaut. Komplexität wird nur durch die Behandlung von Ausnahmezuständen und Fehlern verursacht. Gelöst wird dieses Problem durch sogenannte ON-Handler, die ähnlich zum Exception-Handling in objektorientierten Sprachen funktionieren. Dadurch bleibt der normale Steuerungscode einfach und die Fehlerabfragen werden in einem eigenen Codeabschnitt realisiert.

In weiterer Folge wurde für die Sprache Monaco ein Compiler und eine virtuelle Maschine in Java entwickelt. Der Compiler übersetzt textuellen Monaco-Code in eine objektorientierte Zwischensprache (*CodeDOM*). Diese Zwischensprache kann von der virtuellen Maschine ausgeführt werden. In der derzeitigen Implementierung der virtuellen Maschine erfolgt die Ausführung interpretiert, ein Compiler nach C-Code ist allerdings angedacht.

In dieser Bakkalaureatsarbeit werden die Architektur und die Implementierung des objektorientierten CodeDOM und der virtuellen Maschine behandelt. In [8] wird ein visueller Editor beschrieben, mit dem Monaco-Programme graphisch dargestellt und bearbeitet werden können.

1.2 Architektur der VM

Die virtuelle Maschine selbst ist weitgehend sprachunabhängig gestaltet. Sie stellt eine Ausführungsumgebung für reaktive Programme mit Ereignisverarbeitung und Parallelität bereit.

Die VM besteht aus einer Menge von Diensten. Wichtige Dienste sind zum Beispiel die Verwaltung des Callstacks und des Speicherplatzes für Variablen. Weiters enthält die VM einen Scheduler für kooperatives Multitasking. Dieser unterstützt auch das Warten auf Bedingungen und asynchrone Ereignisse.

Die eigentliche Ausführungslogik von Monaco-Programmen ist allerdings nicht in der VM realisiert. Jede Klasse im CodeDOM weiß selbst, wie sie ausgeführt werden soll. Die VM benachrichtigt nur die einzelnen Elemente des CodeDOM. Daraufhin führt das Element eine entsprechende Aktion aus. Dazu verwendet es die von der VM zur Verfügung gestellten Dienste (siehe Abbildung 1).

Diese Architektur hat den Vorteil, dass sehr einfach mit neuen Sprachkonstrukten experimentiert werden kann. Es muss nur eine neue Klasse im CodeDOM erstellt werden, die zur Laufzeit entsprechende Aktionen setzt. Die VM selbst ist von Änderungen in der Sprache nicht betroffen. Außerdem können ganz einfach komplett neue Notationen für

Endbenutzer erstellt werden, die dann trotzdem mit der gleiche VM ausgeführt werden können.

Der Nachteil dieses Ansatzes ist, dass die Performance etwas niedriger ist. Allerdings ist das in der aktuellen Ausbaustufe noch nicht relevant. Um Echtzeitfähigkeit zu erreichen ist in weiteren Folge ohnehin ein Compiler geplant.

1.3 Struktur der Arbeit

Die vorliegende Arbeit ist wie folgt aufgebaut:

- In Kapitel 2 wird die Sprache Monaco vorgestellt. Am Ende dieses Kapitels wird ein Beispielprogramm präsentiert, anhand dessen später die Funktionsweise der Ausführungsumgebung erläutert wird.
- In Kapitel 3 wird eine objektorientierte Datenstruktur (*CodeDOM*) vorgestellt, in die Monaco-Programme übersetzt werden. Diese Datenstruktur wird für die Ausführung und die visuelle Repräsentation von Monaco-Programmen verwendet.
- In Kapitel 4 wird eine virtuelle Maschine definiert, mit der es möglich ist den CodeDOM auszuführen. Diese virtuelle Maschine ist sprachunabhängig gestaltet.
- Kapitel 5 erläutert dann, wie die einzelnen Sprachkonstrukte von Monaco mit Hilfe dieser virtuellen Maschine realisiert werden.

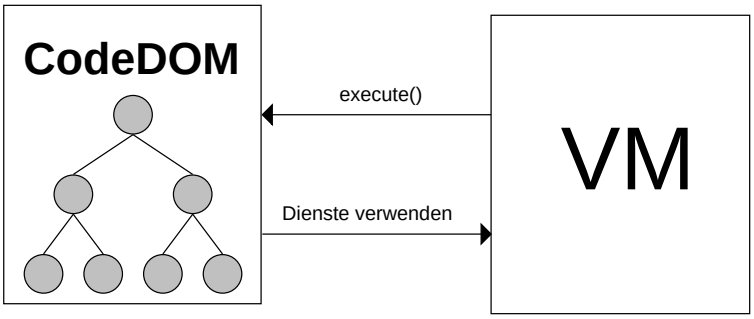


Abbildung 1: Beziehung CodeDOM - VM

2 Die Sprache Monaco

In diesem Kapitel wird die Sprache Monaco [6] vorgestellt.

2.1 Entwurfskriterien

Das Design der Programmiersprache Monaco basiert auf den folgenden Kriterien:

Imperative Notation

Die Hauptmotivation für den Entwurf einer neuen Notation für ereignisbasierte Kontrollsysteme statt beispielsweise der Verwendung des bekannten Statechart-Formalismus [2] war die Beobachtung, dass Domänenexperten und Endbenutzer Schwierigkeiten haben, in Zuständen und Zustandsübergängen zu denken. Domänenexperten denken eher in Sequenzen von Kontrollaufgaben und deren Koordinierung. Eine weitere Beobachtung war, dass in der normalen Ausführung, also ohne Ausnahme- und Fehlerbehandlung, der zugrunde liegende Kontrollfluss oft sequentiell und sehr einfach ist. Nur bei der Behandlung von Ausnahmefällen und Fehlern kommt asynchrones Verhalten ins Spiel. Unsere Sprache erlaubt folglich, den normalen Ablauf als Sequenz von Operationen anzugeben. Zusätzlich übernimmt die Sprache bewährte Konzepte von imperative Sprachen wie prozedurale Abstraktion, synchrone Funktionsaufrufe, Parameter, Blockstruktur und statische Gültigkeitsbereiche.

Ereignisbehandlung

Automatisierungssysteme sind charakterisiert durch asynchrone Ereignisse und parallele Aktivitäten. Deswegen haben wir eigene Kontrollstrukturen eingeführt, mit denen Ereignisbehandlungen, Parallelität und Synchronisation, Fehlerbehandlung und Zeitüberschreitungen übersichtlich abgebildet werden können. Ereignisbehandlungen werden in separaten Codeabschnitten erledigt. Dieser Ansatz ist ähnlich zum Exception-Handling in modernen objektorientierten Programmiersprachen. Er vermeidet, normalen Code mit Ausnahmebehandlungen unnötig kompliziert zu machen.

Komponentenbasierte, hierarchische Kontrollarchitektur

Monaco verfolgt einen komponentenbasierten Ansatz mit strenger Modularisierung. Komponenten sind modulare Einheiten (Black-Boxes), die ausschließlich über definierte Schnittstellen kommunizieren. Im Gegensatz zu anderen komponentenbasierten Ansätzen in diesem Bereich, wie zum Beispiel UML/RT [7], forciert Monaco eine strenge hierarchische Kontrollarchitektur mit über- und untergeordneten Komponenten. Interaktion ist nur zwischen einer übergeordneten und ihren unmittelbar untergeordneten Komponenten möglich. Eine Komponente baut auf Operationen, Zuständen und Ereignissen von untergeordneten Komponenten auf. Sie koordiniert das Verhalten ihrer untergeordneten Komponenten und stellt eine abstrakte und vereinfachte Sicht für ihre übergeordnete Komponente zur Verfügung. Auf diesem Weg unterstützt die Sprache hierarchische Abstraktion von Kontrollfunktionalität. Das resultiert in präziseren, leichter verständlichen Kontrollprogrammen und entspricht auch den Vorstellungen der Domänenexperten.

Statische Konfiguration

Das Setup eines Monaco-Programms (d.h. Instantiierung und Zusammensetzen der Komponenten und Einstellung der Parameter) wird in einer separaten Konfigurierungsphase vor dem Programmstart erledigt. Zur Laufzeit ist also das ganze System statisch konfiguriert. Alle Komponenten, deren Parameter und auch die Größen aller Arrays sind festgelegt. Nach dem Programmstart kann diese Konfiguration nicht mehr verändert werden. Das gibt Monaco-Programmen eine gewisse statische Natur. Zum Beispiel sind Subkomponenten zwar prinzipiell polymorph, allerdings werden sie bereits zur Setup-Zeit zusammengesetzt. Daher ist dynamische Methodenbindung ist nicht notwendig.

Diese statische Konfiguration von Monaco-Programmen ist eine wichtige Eigenschaft. Sie erlaubt es erstens, optimierten Code zu generieren, und zweitens, eine statische Programmanalyse durchzuführen.

2.2 Sprachkonstrukte

2.2.1 Interfaces

Interfaces werden benutzt, um den statischen Kontrakt zwischen Komponenten zu spezifizieren. Sie haben daher eine ähnliche Aufgabe wie Interfaces in modernen objektorientierten Sprachen. Monaco-Interfaces erlauben die Erstellung einer hierarchischen Ab-

straktion in Kontrollanwendungen. Einerseits definieren Interfaces in Form von Routinen die Operationen, die auf einer Komponente möglich sind, andererseits definieren sie, wie eine Komponente Rückmeldungen an die übergeordnete Komponente liefert. Dazu definiert es Ereignisse, die die Komponente auslösen kann, und Funktionen, über die Zugriff auf den Laufzeit-Zustand der Komponente ermöglicht wird.

Listing 1 zeigt ein Beispiel einer Interface-Definition.

2.2.2 Komponenten und Subkomponenten

Interfaces werden von Komponenten implementiert, d.h., Komponenten müssen alle Routinen, Funktionen und Ereignisse die im Interface definiert sind implementieren. Eine Komponente kann Parameter und interne Variablen enthalten, und sie kann Subkomponenten referenzieren, auf die sie bei der Erfüllung ihrer Kontrollaufgaben aufbaut. Referenzen auf Subkomponenten werden als `SUBCOMPONENT`-Variablen deklariert und haben einen Interface-Typ. Komponenten können also nur auf Routinen, Funktionen und Ereignisse zugreifen, die im Interface der Subkomponente deklariert wurden. Andererseits sind Subkomponenten polymorph, d.h., jede Komponente die das korrekte Interface implementiert, kann als Subkomponente verwendet werden.

Listing 2 zeigt Ausschnitte aus der Deklaration von zwei Komponenten.

2.2.3 Funktionen

Eine Funktion ist ähnlich einer Funktion in prozeduralen Programmiersprachen wie Pascal. Sie gibt Eigenschaften des Laufzeitzustands einer Komponente zurück. Monaco-Funktionen dürfen keine Nebeneffekte haben.

```
INTERFACE IDrillCtrl
    EVENTS error;
    FUNCTION position() : REAL;

    ROUTINE startDriller();
    ROUTINE stopDriller();
    ROUTINE down(depth : REAL);
    ROUTINE up();
    ROUTINE stop();
END IDrillCtrl
```

Listing 1: Deklaration eines **`INTERFACE`**

Normalerweise werden Funktionen verwendet, um wichtige Zustandseigenschaften zu berechnen und diese in einer abstrakteren, konzentrierteren Form an die übergeordnete Komponente weiterzuleiten. Listing 3 zeigt ein Beispiel einer Funktionsimplementierung. Die Funktion liest einen relativen Wert aus einer Subkomponente und rechnet diesen in einen absoluten Wert um.

2.2.4 Ereignisse

Ereignisse sind ein Konzept, das in ähnlicher Form auch von anderen reaktiven Programmiersprachen zur Verfügung gestellt wird. Sie haben auch Ähnlichkeiten zu Exceptions in objektorientierten Sprachen.

Ereignisdeklarationen in Monaco sind ähnlich zu Variablendeklarationen und werden mit dem EVENTS-Schlüsselwort eingeleitet. In Routinen können Ereignisse mit dem

```
COMPONENT DrillingMachine IMPLEMENTS IDrillingMachine
    SUBCOMPONENTS
        driller : IDrillCtrl;
        cooler : ICooler;

    ...
END DrillingMachine

COMPONENT DrillCtrl IMPLEMENTS IDrillCtrl
    PARAMETERS
        reactionTimeout : INT := 500;
        upTimeout : INT := 5000;
        downTimeout : INT := 10000;
        upPosition : REAL := 300.0;

    SUBCOMPONENTS
        driller : IDriller;

    ...
END DrillCtrl
```

Listing 2: Deklaration von Komponenten und Subkomponenten

```
FUNCTION position() : REAL
BEGIN
    RETURN driller.position() * upPosition;
END
```

Listing 3: Beispiel einer Funktionsimplementierung

FIRE-Statement ausgelöst werden. Mittels WAIT- und ON-Bedingungen kann auf ein Ereignis reagiert werden.

Die Codefragmente in den Listings 4 und 5 demonstrieren die Verwendung von Ereignissen. Listing 4 zeigt die Deklaration eines Fehler-Ereignisses. Listing 5 zeigt einen Code-Block, in dem das Ereignis gefeuert wird, und einen ON-Handler der darauf reagiert. Dieser Handler stoppt den Bohrer und meldet das Ereignis an die übergeordnete Komponente weiter.

2.2.5 Routinen

Routinen werden verwendet, um Kontrollalgorithmen zu implementieren. Die Implementierung erfolgt in einer prozeduralen Notation mit Parametern, lokalen Variablen, Anweisungssequenzen, Blockstruktur und normalen Kontrollflussstatements wie Verzweigungen und Schleifen. Zusätzlich gibt es spezielle Statements für parallele Ausführung und Ereignisbehandlung.

Listing 6 zeigt ein Fragment der drill-Routine. Die Routine deklariert einen Parameter depth mit Datentyp REAL und definiert den Rumpf mit durch die Schlüsselwörter BEGIN und END erzeugter Blockstruktur. Jeder Block definiert einen Anweisungssequenz, in der die Anweisungen sequentiell ausgeführt werden.

Routinenaufrufe starten üblicherweise Kontrollaufgaben, die durch Wartebedingungen (siehe WAIT-Statement weiter unten) angehalten werden können. Das heißt, Routi-

```
COMPONENT DrillingMachine IMPLEMENTS IDrillingMachine
    EVENTS error;
    ...
END DrillingMachine
```

Listing 4: Deklaration eines Ereignisses

```
BEGIN
    ...
    FIRE error;
    ...
ON error.FIRED
    driller.stop();
    FIRE error;
END
```

Listing 5: Verwendung eines Ereignisses

nen laufen eine gewisse Zeit und die Terminierung hängt von Rückmeldungen der Maschine ab. Die Aufrufsemantik von Routinen ist synchron, das heißt der Aufrufer wartet bis die Routine zurückkehrt. Routinen können allerdings durch asynchrone Ereignisse unterbrochen werden (siehe `ON`-Handler weiter unten).

2.2.6 Kontrollanweisungen

Im folgenden Abschnitt werden die wichtigsten Kontrollanweisungen präsentiert.

WAIT Das `WAIT`-Statement wird verwendet, um die Ausführung des aktuellen Threads zu unterbrechen, bis eine gewisse Bedingung erfüllt wird. Listing 7 zeigt eine Routine, die den Bohrer startet, dann wartet bis eine gewisse Tiefe erreicht ist und anschließend den Bohrer wieder stoppt.

ON `ON`-Statements werden verwendet, um Situationen zu behandeln die asynchron zur normalen Programmausführung auftreten. Sie spezifizieren eine beliebige Ereignisbedingung und hängen an einem `BEGIN/END`-Block. Die Ereignisbedingung kann entweder eine simple Bedingung wie ein Integer-Vergleich sein, oder auch ein Ereignis oder ein Timeout. Die Bedeutung ist, dass immer wenn die Bedingung wahr wird, während die Programmausführung auf einem `WAIT`-Statement innerhalb des Blocks steht, der Block verlassen und das `ON`-Statement ausgeführt wird.

```
ROUTINE drill(depth : REAL)
BEGIN
    IF NOT driller.isDrilling()
    BEGIN
        driller.startDriller();
    END
    ...
END drill
```

Listing 6: Beispiel einer Routine

```
ROUTINE down(depth : REAL)
BEGIN
    driller.down();
    WAIT driller.position() <= depth;
    driller.stop();
END down
```

Listing 7: Verwendung des **`WAIT`**-Statements

Listing 8 zeigt ein Beispiel eines `ON`-Handlers. Wenn die Eigenschaft `isDrilling` des Bohrers irgendwann während der Ausführung der Routine falsch wird, oder die Ausführungszeit der Routine `downTimeout` überschreitet, wird die Routine abgebrochen, der Bohrer gestoppt und ein Fehlerereignis gefeuert.

RESUME `ON`-Handler brechen normalerweise die Ausführung des Blocks ab, in dem sie sich befinden. Das `RESUME`-Statement ermöglicht es, nach dem `ON`-Handler die Ausführung an der Stelle im Block fortzusetzen, an der sie unterbrochen wurde. Listing 9 demonstriert das an einem Beispiel. Statt den Bohrer komplett zu stoppen, wenn er stecken bleibt, wird er nach oben gefahren und neu gestartet. Anschließend wird der Prozess fortgesetzt.

PARALLEL Das `PARALLEL`-Statement wird verwendet, um parallel laufende Zweige zu realisieren. Jeder parallele Ablauf wird durch ein Statement oder einen Statement-Block spezifiziert. Die Ausführung des Hauptablaufs wird erst fortgesetzt, wenn jeder Zweig des `PARALLEL`-Statements terminiert ist. Listing 10 demonstriert die Verwendung des `PARALLEL`-Statements. Ein Ablauf kontrolliert den Bohrprozess, der andere den Kühlungsprozess.

```
ROUTINE down(depth : REAL)
BEGIN
    ...
ON NOT driller.isDrilling() OR TIMEOUT(downTimeout)
    driller.stop();
    FIRE error;
END
```

Listing 8: Verwendung eines **ON**-Handlers

```
ROUTINE down(depth : REAL)
BEGIN
    ...
ON NOT driller.isDrilling()
    driller.up();
    driller.startDriller();
    WAIT driller.isDrilling();
    driller.down();
    RESUME;
END down
```

Listing 9: Fortsetzen eines Prozesses mit **RESUME**

2.2.7 SETUP

Um ein vollständiges Monaco-Programm zu erstellen, müssen die Komponenten instantiiert und die Subkomponenten-Beziehungen aufgebaut werden. Außerdem müssen die Komponenten-Parameter gesetzt werden, sofern sie von den Standardwerten abweichen. Diese statische Konfiguration des Systems wird in der SETUP-Routine erledigt. Sie wird ausgeführt, bevor das eigentliche Monaco-Programm gestartet wird.

Listing 11 zeigt einen Ausschnitt aus dem SETUP einer Bohrmaschinen-Anwendung. Im VARS-Abschnitt werden die Komponenten instantiiert. Die ersten zwei Instanzen sind die Kontrollkomponente für den Bohrer und die Koordinator-Komponente. Beide wurden in Monaco implementiert. Die nächsten zwei sind Instanzen von NATIVE-Komponenten die in einer anderen Programmiersprache geschrieben wurden (in diesem Fall C). Sie stellen das Interface zu den niedrigeren Steuerungsebenen und zur Hardware dar. Die letzte Instanz ist die oberste Komponente, die den Ablauf des Gesamtsystems steuert.

Im BEGIN/END-Block können die Komponenten konfiguriert werden, indem Parameter gesetzt und Subkomponentenbeziehungen hergestellt werden. Am Ende des SETUP wird die Hauptroutine des Programms gestartet. Ab diesem Zeitpunkt sind die Einstellungen des SETUP fix und können nicht mehr verändert werden.

```
ROUTINE drill(depth : REAL)
BEGIN
    driller.startDriller();
    PARALLEL
        BEGIN
            driller.down(depth);
            WAIT 2000;
            driller.up();
        END
    ||
        BEGIN
            cooler.on();
            WAIT cooler.temp() < 60.0;
            cooler.off();
        END
    END
    ...
END drill
```

Listing 10: Parallele Abläufe mit **PARALLEL**

2.3 Beispiel

Im Folgenden soll ein kleines Beispielprogramm die Verwendung der Sprache zeigen. Es handelt sich dabei um ein Steuerungsprogramm für ein Bohrsystem wie in Abbildung 2 gezeigt [6]. Ein Bohrer soll verwendet werden, um Löcher in Metallstücke zu bohren. Das System besteht aus der Bohreinheit selbst und einer Kühleinheit.

Das Steuerungsprogramm besteht aus mehreren Komponenten. Die untersten Kom-

```
SETUP
VARS
    machine : DrillingMachine;
    drillctrl : DrillCtrl;
    oilCooler : NATIVE CCode("cooler");
    driller : NATIVE CCode("driller");
    ...
    supervisor : Supervisor;

BEGIN
    driller.upPosition := 250.0;
    drillctrl.driller := driller;
    machine.cooler := oilCooler;
    machine.driller := drillctrl;
    ...

    supervisor.main();
END SETUP
```

Listing 11: Aufbau der Komponentenhierarchie

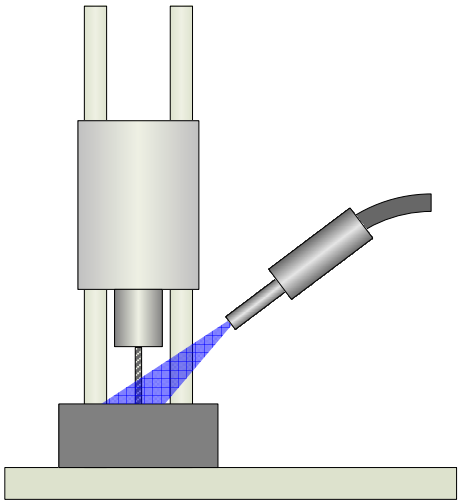


Abbildung 2: Bohrsystem

ponenten realisieren eine Verbindung zur Hardware oder zu den unteren Regelungsebenen, die nicht in Monaco realisiert sind. Dann gibt es eine eigene Steuerungskomponente für den Bohrer (*drill controller*). Eine übergeordnete Steuerungskomponente wird für die Koordination und Überwachung des ganzen Prozesses verwendet.

Listing 12 zeigt die Interfaces der einzelnen Komponenten. `ICooler` und `IDriller` stellen die Interfaces für die untersten Ebenen dar. `ICooler` stellt Routinen für das Ein- und Ausschalten der Kühlung und eine Funktion für das Abfragen der aktuellen Temperatur zur Verfügung. `IDriller` stellt analog Routinen zum Start des Bohrers und zur Steuerung der Auf- und Abbewegung bereit. Rückmeldung wird über die Funktionen `position` und `isDrilling` bereit gestellt.

Das Interface `IDrillCtrl` stellt ein ähnliches Interface wie `IDriller` zur Verfügung. Allerdings sind hier die Vorgänge schon gegen Fehler abgesichert und es wird daher zusätzlich ein Event `error` deklariert, welches Fehler in dieser Komponente anzeigt.

`IDrillingMachine` ist dann das Interface der obersten Komponente, die nur mehr eine Routine `drill` für das Bohren eines Loches bestimmter Tiefe anbietet und ein Event `error` für das Anzeigen eines Fehlers im Bohrvorgang deklariert.

```
INTERFACE IDrillingMachine
    EVENTS error;
    ROUTINE drill(depth : REAL);
END IDrillingMachine

INTERFACE IDrillCtrl
    EVENTS error;
    FUNCTION position() : REAL;
    ROUTINE startDriller();
```

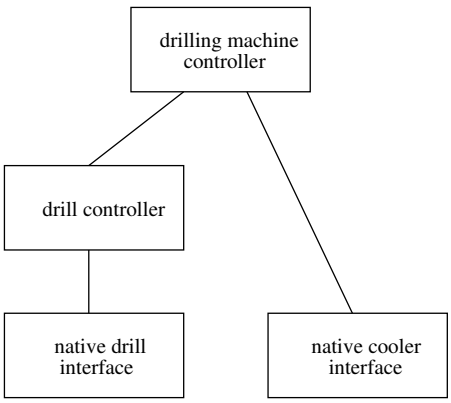


Abbildung 3: Komponentenhierarchie

```
    ROUTINE stopDriller();
    ROUTINE down(depth : REAL);
    ROUTINE up();
    ROUTINE stop();
END IDrillCtrl

INTERFACE ICooler
    FUNCTION temp() : REAL;
    ROUTINE off();
    ROUTINE on();
END ICooler

INTERFACE IDriller
    FUNCTION isDrilling() : BOOL;
    FUNCTION position() : REAL;
    ROUTINE startDriller();
    ROUTINE stopDriller();
    ROUTINE down();
    ROUTINE up();
    ROUTINE stop();
END IDriller
```

Listing 12: Interfaces `IDrillingMachine`, `IDrillCtrl`, `IDriller`, `ICooler`

Listing 13 zeigt die Realisierung der Steuerungskomponente `DrillingMachine`, welche das Interface `IDrillingMachine` implementiert. Die Komponente definiert folgende Elemente:

- Es wird ein Parameter downTimeout deklariert, welcher für die Überwachung der Abwärtsbewegung verwendet wird.
- Es werden Variablen für die zwei direkten Subkomponenten `driller` und `cooler` deklariert.
- Es wird das Event `error` deklariert.
- Es wird die Routine drill implementiert.

Die Routine `drill` ist wie folgt aufgebaut:

- Sie hat einen Parameter `depth`, der die Bohrtiefe vorgibt.
- Im Rumpf wird zuerst der Bohrer gestartet.
- Dann werden in zwei parallelen Zweigen der Bohrprozess und gleichzeitig die Kühlung gesteuert.

- Im Bohrprozess wird der Bohrer nach unten bewegt (`driller.down(depth)`). Es wird für 2 Sekunden gewartet und danach der Bohrer wieder nach oben bewegt.
- Im Kühlprozess wird gewartet, bis eine Temperatur von 80 °C erreicht wird. Dann wird die Kühlung eingeschaltet. Fällt die Kühlung wieder unter 60 °C wird sie wieder ausgeschaltet.
- Sind beide Prozesse beendet, wird der Bohrer ausgeschaltet.
- Die beiden `ON`-Handler am Ende des Blocks überwachen das Auftreten eines Errors in einer der Subkomponenten bzw. die Überhitzung des Bohrers. In beiden Fällen wird die Maschine sofort gestoppt und das Event `error` gefeuert.

```
COMPONENT DrillingMachine IMPLEMENTS IDrillingMachine
    PARAMETERS
        downTimeout : INT := 2000; // 2 sec

    SUBCOMPONENTS
        driller : IDrillCtrl;
        cooler : ICooler;

    EVENTS
        error;

    ROUTINE drill(depth : REAL)
    BEGIN
        driller.startDriller();

        PARALLEL
            BEGIN
                driller.down(depth);
                WAIT TIMEOUT(downTimeout);
                driller.up();
            END
        ||
            BEGIN
                WAIT cooler.temp() > 80.0;
                cooler.on();
                WAIT cooler.temp() < 60.0;
                cooler.off();
            END
        END
        driller.stopDriller();

    ON driller.error.FIRED
        driller.stop();
        FIRE error;
```

```
    ON cooler.temp() > 180.0
        driller.stop();
        FIRE error;
    END drill
END DrillingMachine
```

Listing 13: Komponente `DrillingMachine`

In Listing 11 wurde bereits das Setup für das Programm gezeigt. Hier werden die Komponenten angelegt und ihre Parameter gesetzt. Dann werden die Komponenten miteinander verbunden, indem die Subkomponenten den Subkomponenten-Variablen zugewiesen werden. Das Programm ist somit fertig zur Ausführung. Am Ende des Setup steht ein Routinenaufruf, der die Hauptroutine der Supervisor-Komponente startet.

3 CodeDOM

Als Zwischensprache zur Ausführung und Bearbeitung von Monaco-Code wurde ein objektorientiertes Datenmodell (in Folge *CodeDOM*) definiert.

In dieser Datenstruktur werden alle Deklarationen und der abstrakte Syntaxbaum des Codes abgelegt. Ein mit dem Parser-Generator Coco/R [5] erzeugter Parser liest den textuellen Monaco-Code und erstellt daraus den CodeDOM. Dieser kann dann mit der in Kapitel 4 vorgestellten Infrastruktur ausgeführt werden (siehe Kapitel 5).

Außerdem kann der CodeDOM mit einem visuellen Editor [8] dargestellt und bearbeitet werden. Das Ergebnis kann wieder in textueller Form ausgegeben werden.

Abbildung 4 gibt einen Überblick über die Vererbungshierarchie des CodeDOM. Im Folgenden werden die Struktur des CodeDOM mit den wichtigsten Basisklassen sowie exemplarisch einige konkrete Subklassen vorgestellt.

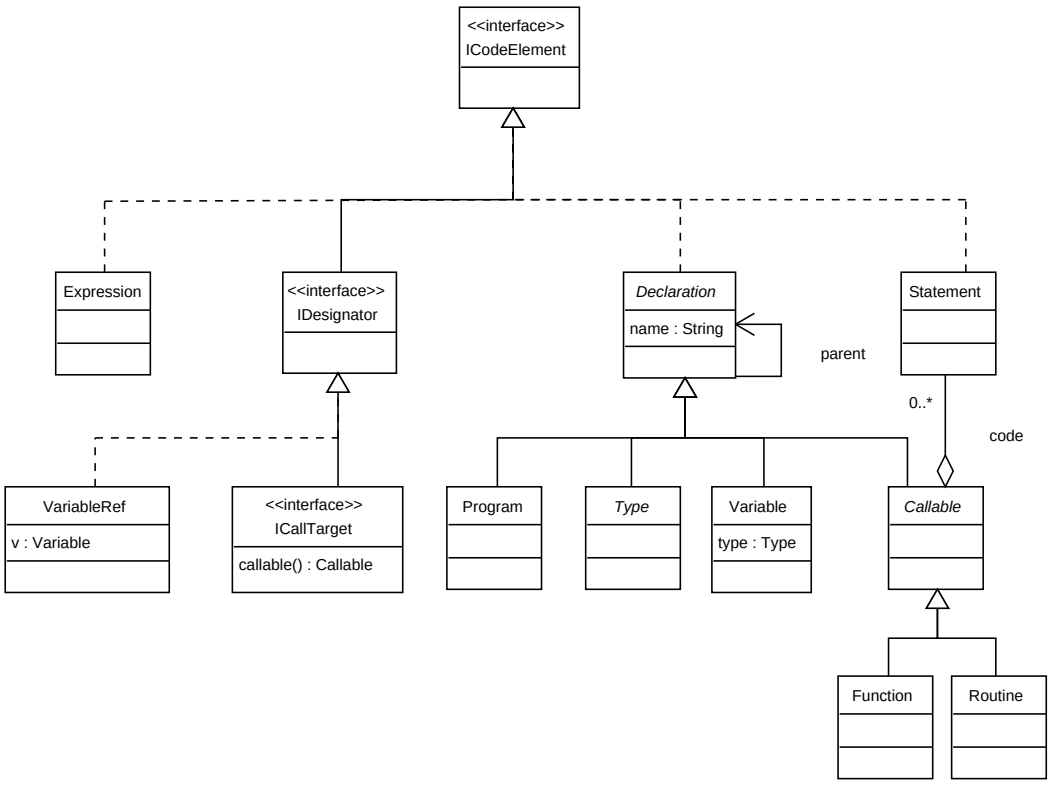


Abbildung 4: Vererbungshierarchie des CodeDOM

3.1 ICodeElement

An oberster Stelle steht das Interface `ICodeElement`. In diesem Interface sind Funktionen, mit denen zu jedem Element im CodeDOM die entsprechende Position im Quellcode ermittelt werden kann. Hier ist auch die `accept`-Methode des Visitor-Patterns [1] definiert, welches für unterschiedliche Verarbeitungsalgorithmen verwendet wird.

3.2 Deklarationen

Die abstrakte Klasse `Declaration` ist die Basisklasse für alle Deklarationen eines Monaco-Programms. Eine Deklaration stellt einen Scope für enthaltene Deklarationen dar. Jede Deklaration hat eine Parent-Deklaration, in der sie deklariert ist, und kann selber als Scope für weitere Child-Deklarationen dienen.

Wesentliche Subtypen von `Declaration` sind `Program`, `Type`, `Variable` und `Callable`. Letzteres hat wiederum `Function` und `Routine` als Subtypen.

Abbildung 5 zeigt die Beziehungen zwischen den wichtigsten Deklarationen und den

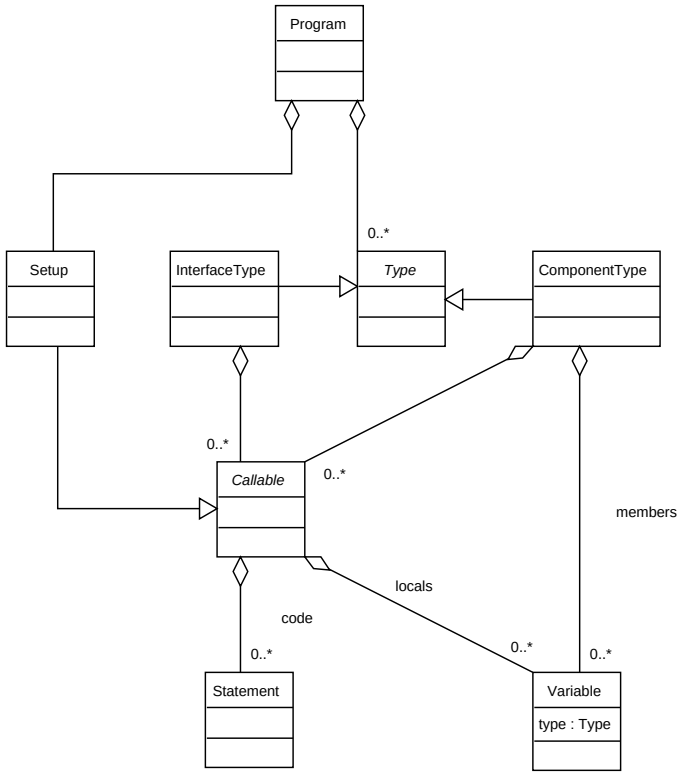


Abbildung 5: Deklarationen im CodeDOM

Statements. Die Wurzel des CodeDOM ist `Program`. Es enthält alle globalen Deklarationen, insbesondere Interfaces und Komponenten.

Interfaces und Komponenten enthalten Callables, konkret Routinen und Funktionen. Die Callables der Komponenten enthalten dann in Form von Statements den Code.

Die `Variable`-Klasse dient zur Repräsentation aller Variablen. Sie wird sowohl für Membervariablen von Komponenten als auch für lokale Variablen verwendet. Auch Subkomponenten, Events und Parameter werden intern als Variablen implementiert.

3.2.1 Beispiel

Abbildung 6 zeigt einen Teil der Deklarationen des Beispielprogramms aus Kapitel 2. An oberster Stelle ist das `Program`. Unter ihm sind die Komponente `DrillingMachine` und deren Interface.

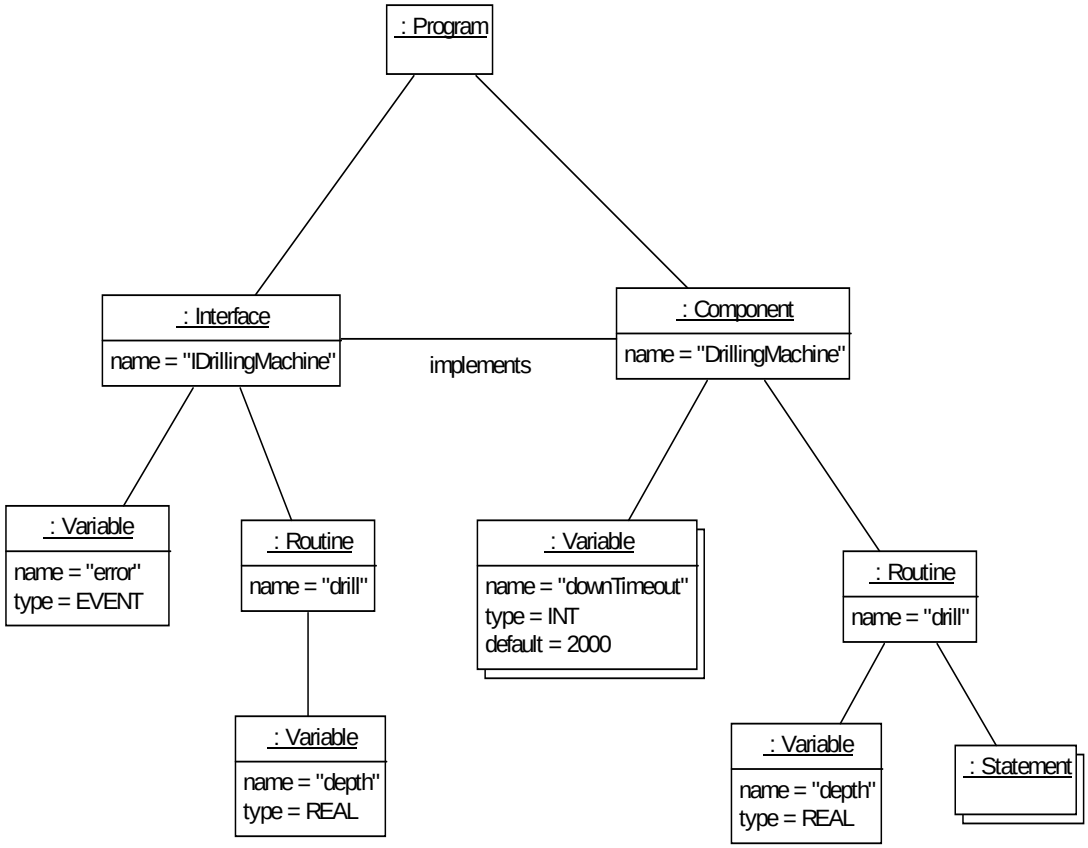


Abbildung 6: Deklarationen im Beispielprogramm

Das Interface besteht aus einer Variable für das `error`-Ereignis und einer abstrakten Routine. Die Routine enthält wiederum eine Variable, den lokalen Parameter `depth`.

Die Komponente besteht aus mehreren Variablen, von denen in der Abbildung nur der Parameter `downTimeout` als Beispiel angeführt ist. Außerdem enthält sie die Implementierung der Routine `drill`. Wie beim Interface enthält die Routine einen Parameter. Zusätzlich enthält sie noch den Code in Form von Statements (siehe Abschnitt 3.5.5).

3.3 Datentypen

Eine wichtige Subklasse von `Declaration` ist die Klasse `Type`. Für jeden Datentyp gibt es eine Instanz einer `Type`-Subklasse. Abbildung 7 zeigt einen Überblick über die Subklassen von `Type`. Diese sind:

- `ComponentType` dient zur Darstellung von Komponenten.
- `InterfaceType` dient zur Darstellung von Interfaces.
- `SimpleType` dient zur Darstellung der simplen Datentypen `INT`, `REAL`, `BOOL` und `STRING`.
- `StructType` dient zur Darstellung benutzerdefinierter Record-Typen.
- `ArrayType` wird für Array-Datentypen verwendet. Er besteht aus einem Elementtyp und einer laufzeitkonstanten Expression für die Länge.

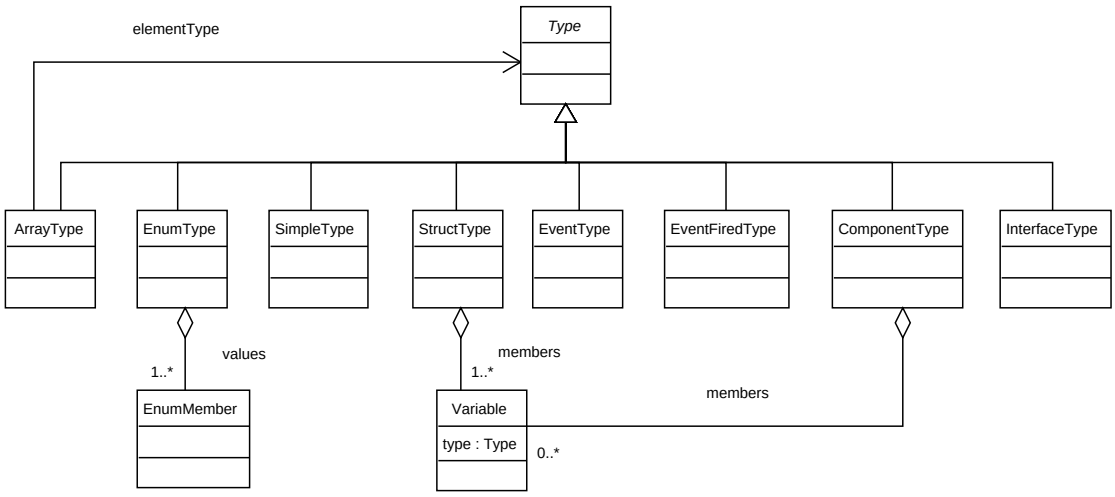


Abbildung 7: Subklassen der Klasse `Type`

- `EnumType` wird für benutzerdefinierte Enumerationstypen verwendet.
- `EventType` wird zur Darstellung des internen Datentyps von Ereignisvariablen verwendet.
- `EventFiredType` wird zur Darstellung des internen Datentyps des `FIRED`-Members von Ereignisvariablen verwendet.

3.4 Designator

Das Interface `IDesignator` wird verwendet, um Referenzen auf eine konkrete Instanz einer Deklaration darzustellen. Abbildung 8 zeigt einen Überblick über die Implementierungen.

Objekte der Subklasse `VariableRef` referenzieren Variablen. Sie bestehen aus der Deklaration der Variable. Membervariablen enthalten noch ein `VariableRef`-Objekt, das den Container (Struct, Komponente oder Array) angibt, in dem die Variable abgelegt ist.

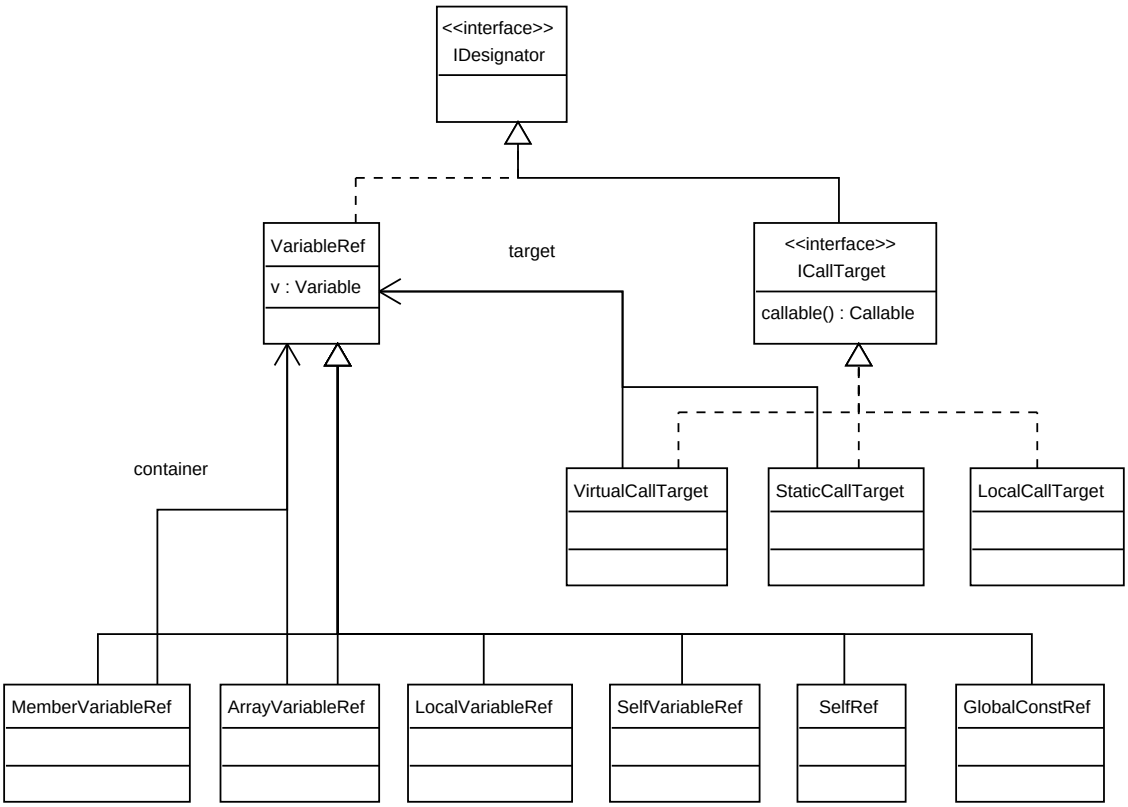


Abbildung 8: Implementierungen des Interface `IDesignator`

Die wichtigsten `VariableRef`-Subklassen sind:

- `LocalVariableRef` wird verwendet, um eine lokale Variable zu referenzieren.
- `MemberVariableRef` wird verwendet, um eine Membervariable eines Struct oder einer Komponente zu referenzieren.
- `SelfVariableRef` wird verwendet, um eine Membervariable der aktuellen Komponente zu referenzieren.
- `SelfRef` wird verwendet, um die aktuelle Komponente selbst zu referenzieren.
- `ArrayVariableRef` wird verwendet, um ein einzelnes Element eines Arrays zu referenzieren.
- `GlobalConstRef` wird verwendet, um eine globale Konstante zu referenzieren.

Ein weiteres Subinterface von `IDesignator` ist `ICallTarget`. Es referenziert ein `Callable`. Die wichtigsten Implementierungen sind:

- `LocalCallTarget` wird verwendet, um eine Memberfunktion innerhalb der aktuellen Komponente zu referenzieren.
- `StaticCallTarget` wird verwendet, um eine Memberfunktion einer anderen Komponente zu referenzieren.
- `VirtualCallTarget` wird verwendet, um eine Memberfunktion eines Interfaces zu referenzieren.

3.5 Statements

Ausführbarer Code wird im CodeDOM durch Subklassen der Klasse `Statement` dargestellt. Listing 14 zeigt das öffentliche Interface von Statements.

```
public abstract class Statement implements ICodeElement {
    public abstract void execute(IContext context);
    public Statement getNext();
}
```

Listing 14: Die Klasse `Statement`

Statements werden im CodeDOM als verkettete Liste abgelegt, die Methode `getNext` liefert das nächste Element dieser Liste. Jedes Statement kennt also seinen unmittelbaren Nachfolger.

`Statement` enthält eine abstrakte Methode `execute(IContext context)`. Diese Methode wird von der VM aufgerufen, wenn das Statement ausgeführt werden soll (siehe Kapitel 4).

3.5.1 BlockStatement

Eine wichtige Subklasse von `Statement` ist `BlockStatement`. Der Körper von Routinen und Funktionen ist immer ein einzelnes `BlockStatement`.

Neben einer Liste von Statements enthält es auch noch eine Liste von ON-Handlern, die solange aktiv sind, solange der Block ausgeführt wird. Wenn während der Ausführung des Blocks die Bedingung eines ON-Handlers wahr wird, wird die Ausführung des Blocks unterbrochen und mit dem Handler-Code fortgesetzt.

3.5.2 AssignmentStatement

Die Klasse `AssignmentStatement` wird verwendet, um eine Zuweisung darzustellen.

Sie enthält eine `Expression` und eine `VariableRef`. Die `Expression` wird ausgewertet und das Ergebnis in die referenzierte Variable gespeichert. Die Zuweisung erfolgt immer mit Wertsemantik, nie mit Referenzsemantik.

Für jeden Basisdatentyp (`INT`, `STRUCT`, ...) existiert eine eigene Subklasse.

3.5.3 Kontrollflussstatements

Für jedes Kontrollflussstatement existiert eine eigene `Statement`-Subklasse.

Als Beispiel sei hier nur das `IF`-Statement angeführt. Es besteht aus einer Liste von `IfThen`-Klauseln und optional einem einzelnen Statement für den `ELSE`-Zweig. Jede `IfThen`-Klausel besteht aus einer Bedingung und einem `Statement`.

Schleifen, `PARALLEL` und `ALTERNATIVE` sind analog aufgebaut.

3.5.4 Weitere Statements

Es gibt noch einige weitere Statements, auf die hier nicht im Detail eingegangen wird. Der Aufbau dieser Statements sollte auch ohne nähere Beschreibung klar sein.

`RoutineCall` ruft eine durch ein `ICallTarget` referenzierte Routine auf. Es enthält eine Liste von Expressions, deren Auswertungsergebnis als Parameter übergeben wird.

`WaitStatement` wartet auf das Eintreten einer Bedingung. Die Bedingung kann eine normale boolesche Expression sein, insbesondere auch ein Ereignis oder ein Timeout (siehe Abschnitt 3.7).

`ReturnStatement` bricht die Ausführung einer Funktion ab und gibt den Rückgabewert an.

`FireStatement` feuert ein Ereignis.

`DebugStatement` gibt eine Meldung auf der Konsole aus.

3.5.5 Beispiel

Abbildung 9 zeigt die Statements der Routine `drill` (siehe Listing 13). Direkt an der Routine hängt ein `BlockStatement`. Dieser Block enthält einen `RoutineCall`, ein `ParallelStatement` und einen weiteren `RoutineCall`. Die beiden parallelen Zweige werden wieder durch ein `BlockStatement` dargestellt.

Am obersten `BlockStatement` hängen außerdem noch zwei `Handler`. Im Diagramm ist zur Wahrung der Übersichtlichkeit nur einer dargestellt, der zweite ist in gleicher Weise aufgebaut. Am `Handler` hängen wieder ein `RoutineCall` und ein `FireStatement`.

Abbildung 10 zeigt den Aufruf der Routine `driller.down(depth)` im Detail. Das `RoutineCall`-Statement enthält ein `VirtualCallTarget` und eine Parameterliste. Das `VirtualCallTarget` besteht aus einer Expression, die zur Subkomponente `driller` evaluiert, und der Deklaration der aufzurufenden Routine `down`. Die Parameterliste enthält die Expression `depth`.

3.6 Expressions

Die Basisklasse aller Expressions ist `Expression` (siehe Listing 15).

Die Methode `evaluateValue` evaluiert die Expression und liefert ein `Value` zurück.

Für jeden Basisdatentyp existiert eine separate Subklasse. Als Beispiel sei hier nur `IntegerExpression` angeführt (siehe Listing 16). Die anderen Subklassen sind analog aufgebaut.

Die Methode `evaluate` der jeweiligen Subklassen führt die eigentliche Berechnung durch. Die Methode `evaluateValue` ruft nur die `evaluate`-Methode auf und verpackt den Rückgabewert in einem `Value`-Objekt.

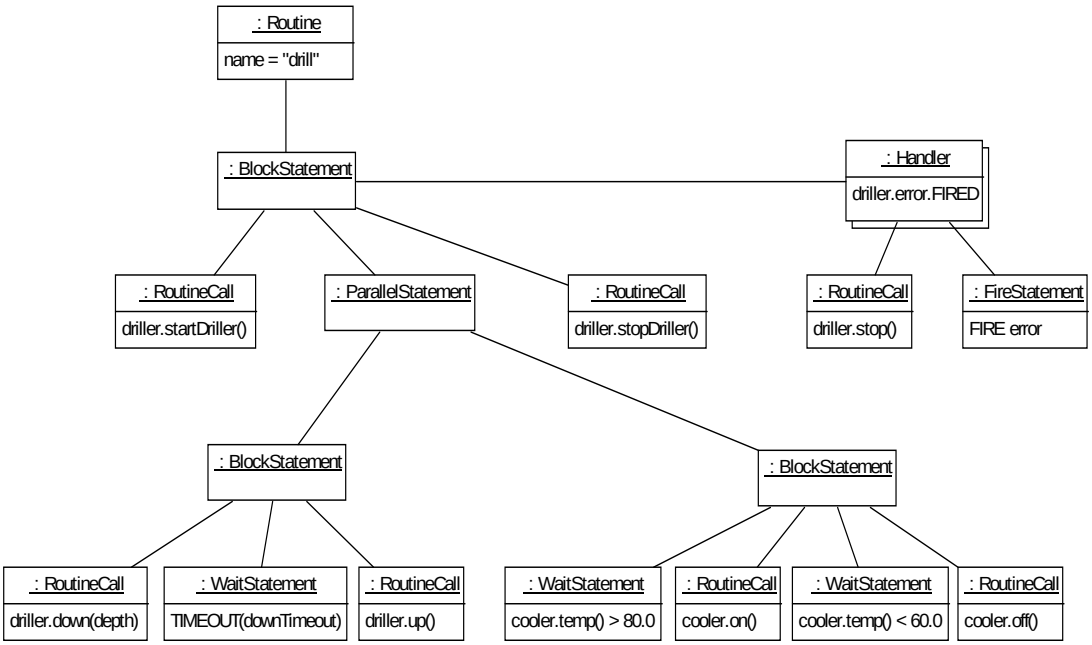


Abbildung 9: Statements der Routine `drill`

```
public class Expression implements ICodeElement {
    public Value evaluateValue(IContext context);
    public Type getType();
    public boolean isConstant();
}
```

Listing 15: Die Klasse `Expression`

3.6.1 Rechenoperationen

Für jede gültige Rechenoperation existiert jeweils eine Subklasse. Für die Basisklasse `IntegerExpression` sind das beispielsweise:

- `IntegerNegExpression`
- `IntegerAddExpression`
- `IntegerSubExpression`

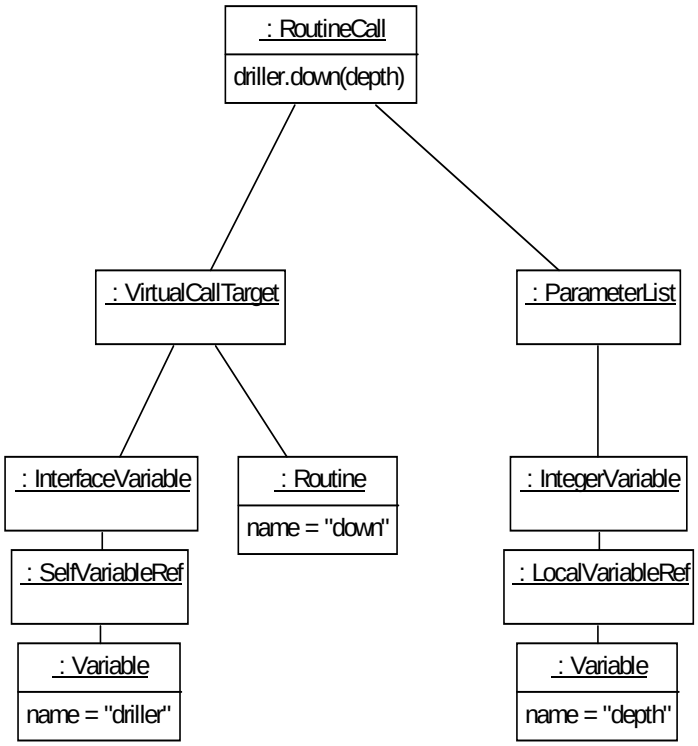


Abbildung 10: Aufruf der Routine `driller.down(depth)`

```
public class IntegerExpression extends Expression {
    public abstract int evaluate(IContext context);

    public Value evaluateValue(IContext context) {
        return new IntegerValue(evaluate(context));
    }
}
```

Listing 16: Die Klasse `IntegerExpression`

- `IntegerMulExpression`
- `IntegerDivExpression`
- `IntegerModExpression`

Jede dieser Klassen enthält weitere `IntegerExpression`-Objekte als Operanden.

3.6.2 Typumwandlungen

In der Sprache Monaco werden Typumwandlungen grundsätzlich nur explizit in Form von eingebauten Funktionen durchgeführt.

Im CodeDOM gibt es für jede zulässige Typumwandlung eine eigene Expression. Für die `IntegerExpression` sind das beispielsweise:

- `CeilExpression`
- `FloorExpression`
- `RoundExpression`

Diese Expressions wandeln jeweils einen `REAL`-Wert in einen `INT`-Wert um, indem sie ihn aufrunden, abrunden oder mathematisch korrekt runden.

3.6.3 Werte

Direkt im Sourcecode angegebene Konstanten werden durch eine eigene `Expression`-Subklasse dargestellt (zum Beispiel `IntegerConstExpression`).

Auch zum Lesen von Variablen, Parametern, globalen Konstanten und so weiter (im Allgemeinen alles was durch die Klasse `Variable` dargestellt wird) gibt es für jeden Datentyp eine eigene Subklasse (z.B. `IntegerVariableExpression`). Diese Klassen enthalten jeweils eine `VariableRef`. Beim Auswerten der `Expression` wird der Wert aus der referenzierten Variable ausgelesen.

3.6.4 Funktionsaufrufe

Der Aufruf einer Funktion wird ebenfalls durch eigene `Expression`-Subklassen (zum Beispiel `IntegerFunctionCall`) dargestellt. Wie das `RoutineCall`-Statement enthalten sie ein `ICallTarget` und eine `Expression`-Liste als Parameter.

3.6.5 Beispiel

Abbildung 11 zeigt die Repräsentation der Expression `cooler.temp() > 80.0`. Das oberste Element, eine `RealGtExpression`, stellt den Operator `>` dar. Diese Klasse ist von der Klasse `BoolExpression` abgeleitet. Das Ergebnis ist also vom Datentyp `BOOL`.

Das linke Kind ist ein Funktionsaufruf in eine Subkomponente. Er ist ähnlich wie ein Routinenaufruf aufgebaut. Die Parameterliste ist in diesem Beispiel leer. Das rechte Kind ist eine `RealConstExpression`, die zur konstanten Zahl `80.0` evaluiert.

3.7 Wartebedingungen

Die `Expression`-Subklasse `BoolExpression` stellt einen Sonderfall dar. Neben der herkömmlichen Verwendung als `Expression` in Zuweisungen oder Kontrollflussanweisungen können boolesche Ausdrücke auch in `WAIT`-Statements und in `ON`-Handlern verwendet werden, um auf das Eintreten einer Bedingung zu warten bzw. asynchron auf das Eintreten einer Bedingung zu reagieren.

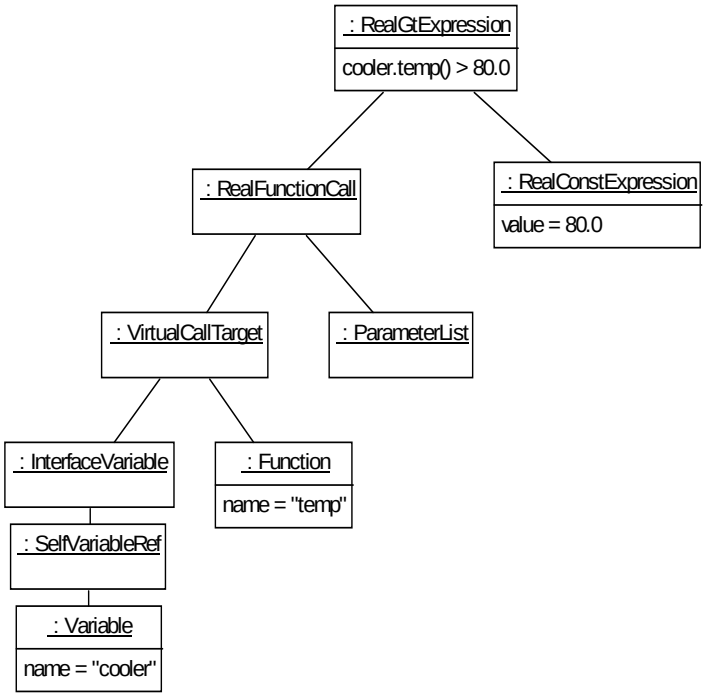


Abbildung 11: CodeDOM der Expression `cooler.temp() > 80.0`

Listing 17 zeigt die wichtigsten Methoden der Basisklasse `BoolExpression`. Die Methoden `evaluate` und `evaluateValue` dienen zur Auswertung der Bedingung, wie bereits vorher am Beispiel der `IntegerExpression` erläutert wurde. Die anderen Methoden werden zur Behandlung von zeit- und ereignisgesteuerten Wartebedingungen verwendet.

Zusätzlich zu den bereits im vorigen Abschnitt genannten Subklassen existieren noch die speziellen Klassen `EventFiredVariable` und `EventFiredWrapper` zur Darstellung von ereignisgesteuerten Wartebedingungen, sowie `RelativeTimerExpression` und `AbsoluteTimerExpression` zur Darstellung von zeitgesteuerten Wartebedingungen. Diese Bedingungen sind nur in WAIT-Statements und ON-Handlern, nicht aber in normalen Kontrollflussstatements erlaubt. Mit der Methode `containsEvent` kann festgestellt werden, ob eine zusammengesetzte Expression diese speziellen Subklassen enthält.

Die Methode `getWakeupTime` gibt eine Schätzung ab, wann eine Bedingung frühestens wahr werden kann. Diese Information wird von der VM nur zur Optimierung verwendet.

3.7.1 Aktivieren von Bedingungen

Bei zeit- und ereignisgesteuerten Wartebedingungen ist es wichtig feststellen zu können, wann die Bedingung aktiv ist. Die Bedingung eines `WAIT`-Statements ist aktiv, wenn die

```
public class BoolExpression extends Expression {
    public boolean evaluate(IContext context);

    public Value evaluateValue(IContext context) {
        return new BoolValue(evaluate(context));
    }

    public boolean containsEvent();
    public void clearEvents();

    public BoolExpression createWakeupCondition(
            IContext context);
    public void destroyWakeupCondition();

    public long getWakeupTime();
}
```

Listing 17: Die Klasse `BoolExpression`

Ausführung gerade auf dem `WAIT`-Statement steht. Die Bedingung eines `ON`-Handlers ist aktiv, solange sich die Ausführung innerhalb des Code-Blocks befindet, zu dem der `ON`-Handler gehört.

Durch parallele Ausführung kann es auch passieren, das eine einzige Wartebedingung mehrmals gleichzeitig aktiv ist. In diesem Fall müssen beide Instanzen der Bedingung voneinander unabhängig behandelt werden.

Dieses Problem wird dadurch gelöst, dass immer wenn eine Bedingung aktiv wird mit der Methode `createWakeupCondition` eine lokale Kopie dieser Bedingung angelegt wird. Diese Methode kopiert nur, wenn unbedingt notwendig. Normale Bedingungen, die keine Ereignisse oder Timer enthalten, werden dabei nicht kopiert.

So angelegte Bedingungen müssen mit `destroyWakeupCondition` freigegeben werden, sobald sie nicht mehr aktiv sind.

3.7.2 Ereignisse

In Monaco werden Ereignisse durch ein spezielles Konstrukt namens `EVENT` dargestellt. Ein Ereignis kann mit dem `FIRE`-Statement ausgelöst werden. Mit der Pseudo-Membervariable `FIRED` kann man auf ein Ereignis warten. Semantisch verhält sich ein Ereignis wie eine Bedingung, die vom Zeitpunkt des Feuerns bis zum Ende des nächsten logischen Zeitschritts (siehe Abschnitt 4.2.2) wahr ist.

Im CodeDOM werden Ereignisse als Variablen vom Datentyp `EventType` dargestellt. Die Klasse `EventFiredVariable` stellt eine Bedingung dar, die auf ein Ereignis wartet. Die aktive Form dieser Bedingung ist ein `EventFiredWrapper`. Er registriert sich beim Ereignis und wird ab diesem Zeitpunkt bei jedem `FIRE` benachrichtigt. Die Bedingung ist solange falsch, bis das Ereignis gefeuert wird. Ab dann ist sie solange wahr, bis der Zeitschritt zu Ende ist. Ab dann ist sie wieder falsch.

3.7.3 Zeitgesteuerte Bedingungen

In Monaco kann mit der eingebauten Funktion `TIMEOUT` eine bestimmte Zeit gewartet oder in Verbindung mit `ON`-Handlern ein Timeout für einen Codeblock festgelegt werden. Im CodeDOM wird diese Funktion durch eine `RelativeTimerExpression` dargestellt. Diese Bedingung selbst evaluiert immer zu `FALSE`.

Mit `createWakeupCondition` wird der Parameter der `TIMEOUT`-Funktion ausgewertet und daraus eine `AbsoluteTimerExpression` erzeugt. Diese enthält den abso-

luten Zeitpunkt, ab dem die Bedingung wahr wird.

Beispiel

Abbildung 12 demonstriert die Aktivierung einer `TIMEOUT`-Bedingung in einem `WAIT`-Statement. Bei Erreichen des Statements wird die `RelativeTimerExpression` aktiviert.

Diese evaluiert zuerst die `IntegerExpression` in der die Wartezeit gespeichert ist. Im konkreten Beispiel ist das der Komponentenparameter `downTimeout`, das Ergebnis ist `2000`. Dann wird aus der VM die aktuelle Zeit ausgelesen, im diesem Fall `10000`. Daher ist die absolute Zeit, zu der die Bedingung eintreten soll, `12000`. Mit diesem Wert wird eine `AbsoluteTimerExpression` angelegt.

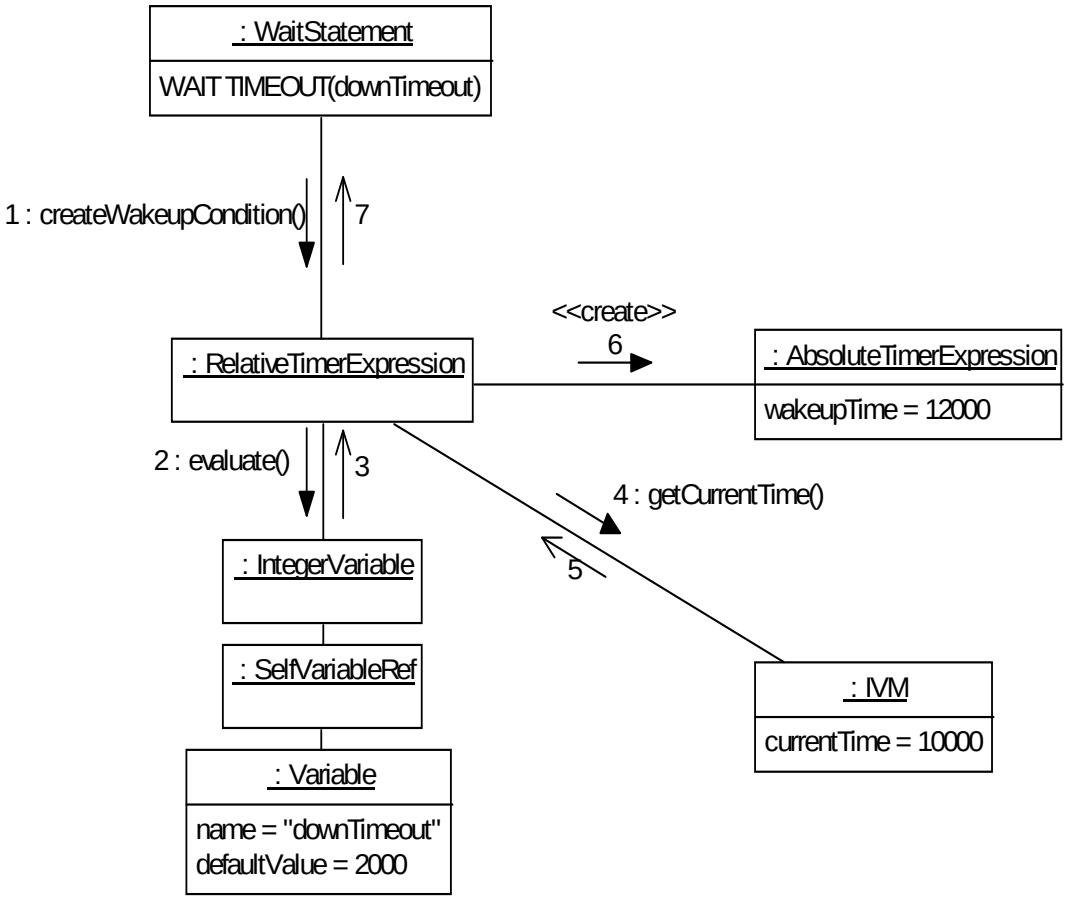


Abbildung 12: Aktivierung einer zeitgesteuerten Wartebedingung

4 Spezifikation der VM

Die Semantik von Monaco ist stark an Statecharts [2] angelehnt. Wichtige Features sind Parallelität und asynchrone Ereignisse. Außerdem müssen Monaco-Programme voll deterministisch und echtzeitfähig sein.

Im folgenden Kapitel wird eine „virtuelle Maschine“ (VM) spezifiziert, die diese Voraussetzungen erfüllen kann. Diese VM stellt eine Ausführungsumgebung zur Verfügung, auf deren Basis Monaco-Programme ausgeführt werden können.

Die hier beschriebene Implementierung der VM wurde in Java geschrieben und ist daher auch noch nicht echtzeitfähig.

In Kapitel 5 wird beschrieben, wie die einzelnen Sprachkonstrukte von Monaco basierend auf dieser VM realisiert werden.

4.1 Struktur

Die VM für Monaco ist sprachunabhängig gestaltet. Sie stellt allgemeine Dienste bereit, die es erlauben, Monaco-Programme auszuführen. Die eigentliche Ausführung definieren dann die Elemente des CodeDOM. Dadurch ist es einfach möglich, neue Sprachkonstrukte einzuführen und mit unterschiedlichen Sprachkonstrukten zu experimentieren.

Die Monaco VM stellt folgende Dienste zur Verfügung:

kooperativer Scheduler: Damit kann sowohl Parallelität als auch das Warten auf asynchrone Ereignisse realisiert werden. Durch das kooperative Multitasking wird Determinismus sicher gestellt.

Callstack: Damit können die Blockstruktur, Routinenaufrufe und Scopes realisiert werden.

Datenspeicher: Die VM stellt auch Speicherplatz für lokale Variablen und Objektinstanzen zur Verfügung.

Der CodeDOM kann über die Interfaces `IVM`, `IContext` und `IValueStore` auf die Dienste der VM zugreifen. Ein `IContext`-Objekt stellt immer die Ausführungsumgebung eines Threads dar. Über dieses Interface kann der Scheduler gesteuert werden. Innerhalb eines Context können geschachtelte *Code-Blöcke* angelegt werden, mit denen ein Callstack und strukturierte Kontrollanweisungen realisiert werden können. An die

Code-Blöcke können noch *Scopes* angehängt werden, in denen lokale Variablen gespeichert werden können. Abbildung 13 zeigt die Beziehung zwischen diesen Konzepten.

In den folgenden Abschnitten werden diese Teile der VM näher erläutert.

4.1.1 VM

Das Interface `IVM` (siehe Listing 18) wird verwendet, um auf eine Instanz der Monaco-VM zuzugreifen. Mit der Methode `createContext` kann ein Context angelegt werden. Ein so angelegter Context läuft asynchron zu allen anderen Contexts (siehe Abschnitt 4.2.6).

Die Methoden `getGlobalContext` und `initGlobalContext` verwalten einen globalen Context, der verwendet wird, um globale Konstanten abzuspeichern (siehe nächster Abschnitt).

4.1.2 Context

Das zentrale Element der Ausführungsumgebung ist der „Context“. Ein Context entspricht einem Thread im laufenden Programm. Der Verwender der VM kann über das Interface `IContext` auf die Elemente eines Context zugreifen.

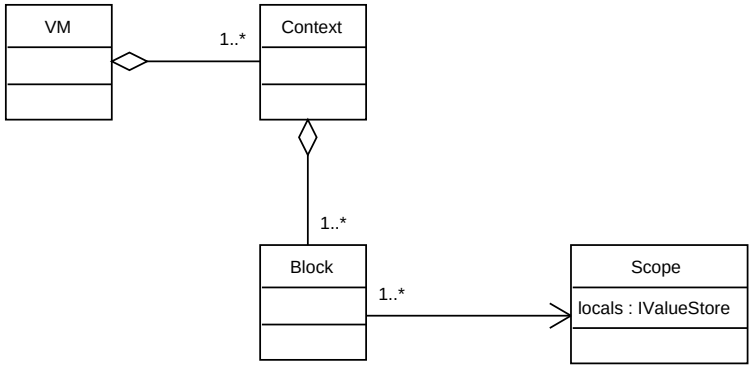


Abbildung 13: Beziehung VM-Context-Block-Scope

```
public interface IVM {
    public IContext createContext();

    public IContext getGlobalContext();
    public void initGlobalContext(Value[] consts);
}
```

Listing 18: Das Interface `IVM`

Über den Context sind alle lokalen Daten des Programms zugänglich. Kontrollstrukturen, Parallelität und asynchrone Ereignisse können durch Manipulation des Context realisiert werden.

Listing 19 zeigt das Interface `IContext`. Die Methode `getVM` liefert die Instanz der VM, in der dieser Context läuft. Die Methoden `activate`, `deactivate` und `suspend` steuern den kooperativen Scheduler. `split` und `createHandler` legen neue Contexte an. Mit `terminate`, `execute`, `openBlock` und `setNextInstruction` wird der Kontrollfluss gesteuert. Die Methoden `createScope` und `openScope` manipulieren den aktuellen Scope, mit `getSelf` und `getLocals` kann auf den Inhalt des Scope zugegriffen werden.

Diese Methoden werden in den folgenden Abschnitten noch genauer beschrieben.

```
public interface IContext {
    public IVM getVM();

    public void activate();
    public void deactivate();

    public IContext[] split(int childCount);
    public IContext createHandler(boolean resume);

    public void suspend();
    public void suspend(BoolExpression wakeup);

    public void terminate();
    public void terminate(Value returnValue);

    public Value execute();

    public void openBlock();
    public void setNextInstruction(Statement statement);

    public void createScope(ComponentInstance self,
        Callable callable);
    public void openScope(IContext other);

    public ComponentInstance getSelf();
    public IValueStore getLocals();
}
```

Listing 19: Das Interface `IContext`

Globaler Context

Zum Speichern globaler Konstanten unterstützt die Monaco VM einen globalen Context, auf den mit der `getGlobalContext`-Methode des `IVM`-Interface zugegriffen werden kann. Dieser Context kann keinen ausführbaren Code und keine Blöcke enthalten. Er kann nicht suspendiert oder aktiviert werden und keine Kinder oder Handler erzeugen. Im Scope des globalen Context sind global sichtbare Variablen, die zur Laufzeit nicht verändert werden dürfen. Die Werte dieser Variablen werden mit `initGlobalContext` festgelegt. Diese Methode kann nur einmal beim Start der VM aufgerufen werden.

4.1.3 IValueStore

Zum Ablegen von Daten wird in der VM das Interface `IValueStore` (Listing 20) bereitgestellt. Einen `IValueStore` kann man sich als flaches Array von nicht typisierten Werten vorstellen. Für die Typtests ist der Verwender der VM oder der Compiler verantwortlich.

Die Basisklasse aller Daten, mit denen in der Monaco VM gearbeitet werden kann, ist `Value`. Davon abgeleitet sind alle Basisdatentypen wie z.B. `IntegerValue` oder `RealValue`.

Lokale Variablen werden direkt im Context abgelegt. Die Methode `getLocals` liefert einen für diesen Zweck angelegten `IValueStore`. Die `Value`-Subklassen für komplexe Datentypen implementieren `IValueStore`, sodass auch auf deren Elemente über das gleiche Interface zugegriffen werden kann.

4.1.4 Blöcke

Der Kontrollfluss innerhalb eines Context wird über „Code-Blöcke“ gesteuert. Ein Block entspricht dabei in etwa einem Stackframe [4]. Allerdings wird dieses Konzept nicht nur für Routinenaufrufe sondern allgemein für alle Kontrollstrukturen verwendet.

Jeder Block hat einen Instruction-Pointer, der auf das nächste auszuführende State-

```
public interface IValueStore {
    public Value get(int index);
    public void set(int index, Value value);
}
```

Listing 20: Das Interface `IValueStore`

ment zeigt, und einen übergeordneten Block, zu dem nach Abschluss des aktuellen Blocks zurück gesprungen wird.

Blöcke werden über die Methoden `openBlock` und `setNextInstruction` des `IContext`-Interface manipuliert.

Wird ein Context ausgeführt, wird immer das Statement, auf das der Instruction-Pointer des innersten Blocks zeigt, benachrichtigt. Das Statement kann jetzt seine Aktion ausführen. Es muss zumindest den Instruction-Pointer auf das nächste Statement setzen. Kontrollstatements wie zum Beispiel Verzweigungen oder Schleifen setzen typischerweise den Instruction-Pointer des aktuellen Blocks auf das nachfolgende Statement und legen einen neuen geschachtelten Block für ihren Körper an. In Kapitel 5 wird im Detail beschrieben, wie die einzelnen Statements den Instruction-Pointer setzen.

Zeigt der Instruction-Pointer eines Blocks auf **`null`**, ist der Block zu Ende und die Ausführung wird beim Instruction Pointer des nächst äußeren Blocks fortgesetzt. Gibt es keinen weiteren Block, terminiert der Context.

4.1.5 Scopes

Ein Scope ist ein Gültigkeitsbereich von lokalen Variablen. Scopes können in der Monaco VM nicht geschachtelt werden, allerdings enthält ein Scope zusätzlich zu seinen lokalen Variablen immer eine Referenz auf die aktuelle Komponente (`SELF`-Referenz).

Scopes sind Blöcken zugeordnet, das heißt, ein Block hat immer Zugriff auf genau einen Scope. Allerdings können mehrere Blöcke Zugriff auf denselben Scope haben (siehe Abbildung 13).

Wenn ein neuer Block angelegt wird, übernimmt er automatisch den Scope des übergeordneten Blocks. Mit den Methoden `createScope` und `openScope` kann man den Scope dann wechseln. `createScope` legt einen neuen Scope an und reserviert Speicherplatz für lokale Variablen einer Routine. `openScope` übernimmt den aktuellen Scope eines anderen Context. Scopes werden implizit mit den sie enthaltenden Blöcken geschlossen.

Mit den Methoden `getSelf` und `getLocals` kann auf den Inhalt des aktuellen Scopes zugegriffen werden.

4.1.6 Beispiel

Abbildung 14 zeigt die wichtigsten Datenstrukturen der VM während der Ausführung des Beispielprogramms aus Abschnitt 2.3. Das Programm steht gerade in der Routine `drill` unmittelbar vor dem `PARALLEL`-Statement.

Zu diesem Zeitpunkt gibt es nur einen aktiven Context (links in der Abbildung). Der aktuelle Block dieses Context entspricht dem Block-Statement, aus dem der Rumpf der Routine besteht. Dieser Block ist mit dem Scope der Routine `drill` verbunden. Der Instruction-Pointer zeigt auf das `PARALLEL`-Statement.

Der Scope enthält eine `SELF`-Referenz auf die Komponente `DrillingMachine` mit ihren Variablen `downTimeout`, `driller`, `cooler` und `error`. Weiters enthält der Scope einen `IValueStore`, in dem die lokale Variable `depth` gespeichert wird.

Der aktive Block hat weitere äußere Blöcke. Diese Blöcke kann man sich wie einen Callstack vorstellen. Der nächstäußere Block entspricht dem Rumpf der Routine und zeigt auf den gleichen Scope. Der äußerste Block ist der Block des `SETUP`. Sein Scope enthält einen `IValueStore`, in dem die Instanzen aller Komponenten gespeichert sind.

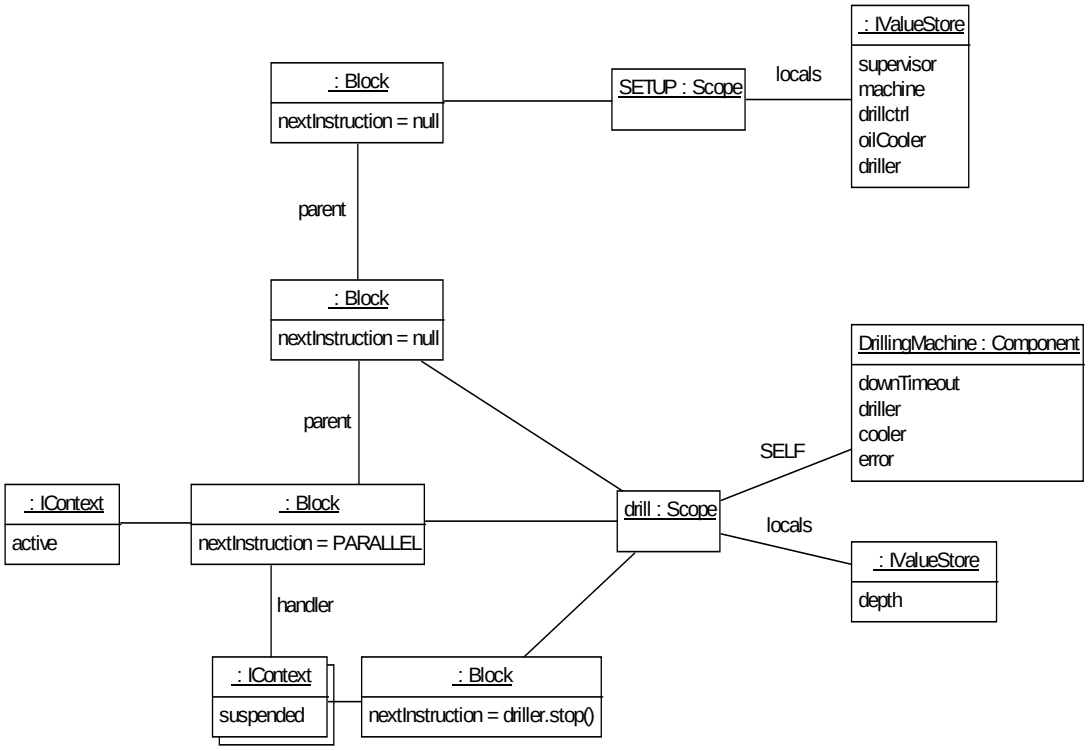


Abbildung 14: Datenstruktur der VM im Beispielprogramm

Am aktiven Block hängen zwei suspendierte Handler-Contexte, die die ON-Handler realisieren (siehe Abschnitt 4.2.4). Sie haben jeweils einen Block, dessen Instruction-Pointer auf die erste Instruktion des Handlers zeigt. Die Blöcke der Handler zeigen ebenfalls auf den Scope der `drill`-Routine.

4.2 Ausführung

Im vorigen Abschnitt wurden die Datenstrukturen der VM vorgestellt. Diese betreffen immer nur die Ausführung eines einzelnen Context. Im folgenden Abschnitt wird die Dynamik der VM behandelt. Zuerst wird das Zusammenspiel mehrerer Contexte mittels kooperativem Multitasking erläutert. Dann wird auf asynchrone Ereignisbehandlung und Parallelität eingegangen.

4.2.1 Kooperatives Multitasking

Um kooperatives Multitasking zu unterstützen, muss es möglich sein, einen Context zu suspendieren. Das wird von der Methode `suspend` erledigt. Ein suspendierter Context wartet auf das Eintreten einer Wartebedingung, bevor er seine Ausführung fortsetzen kann. Ein Context kann auch ohne Wartebedingung suspendiert werden. In diesem Fall wird seine Ausführung unterbrochen und der nächste aktive Context ausgeführt. Der suspendierte Context bleibt aber trotzdem aktiv. Das wird zum Beispiel am Ende einer Routine implizit gemacht, um eventuell vorhandene Handler überprüfen zu können.

Zusätzlich kann ein Context vollkommen deaktiviert werden. In diesem Fall wird er weder ausgeführt, noch eine eventuell vorhandene Wartebedingung überprüft, bis er wieder aktiviert wird. Ein Context kann entweder implizit terminieren, indem sein letzter Block geschlossen wird, oder explizit durch Aufruf der `terminate`-Methode. Die Terminierung kann aus jedem Zustand erfolgen.

Abbildung 15 zeigt die Zustände, in denen sich ein Context befinden kann, und die Methodenaufrufe im `IContext`-Interface, mit denen in einen anderen Zustand gewechselt wird.

4.2.2 Scheduling

Das Scheduling in der VM erfolgt mit einem simplen prioritätsgesteuerten Algorithmus.

Die Ausführung wird in einzelne logische Zeitschritte zerlegt. Der Scheduler entfernt einen aktiven Context aus dem aktuellen Zeitschritt und führt ihn wie bereits in Ab-

schnitt 4.1.4 beschrieben aus. Wenn dabei ein Context neu angelegt, aktiviert oder suspendiert wird, landet er nicht im aktuellen sondern im nächsten logischen Zeitschritt.

Das wird solange wiederholt, bis kein aktiver Context im aktuellen Zeitschritt übrig ist. Damit ist der Zeitschritt abgeschlossen, alle übrigen (suspendierten) Contexte werden in den nächsten Zeitschritt übernommen und die Ausführung dort fortgesetzt.

Dieses System stellt sicher, dass jeder aktive Context garantiert einmal ausgeführt wird, bevor ein anderer ein zweites Mal dran kommt. Das ist eine wichtige Voraussetzung für die Echtzeitfähigkeit.

Bei der Auswahl eines aktiven Context wird die folgende Prioritätsordnung verwendet:

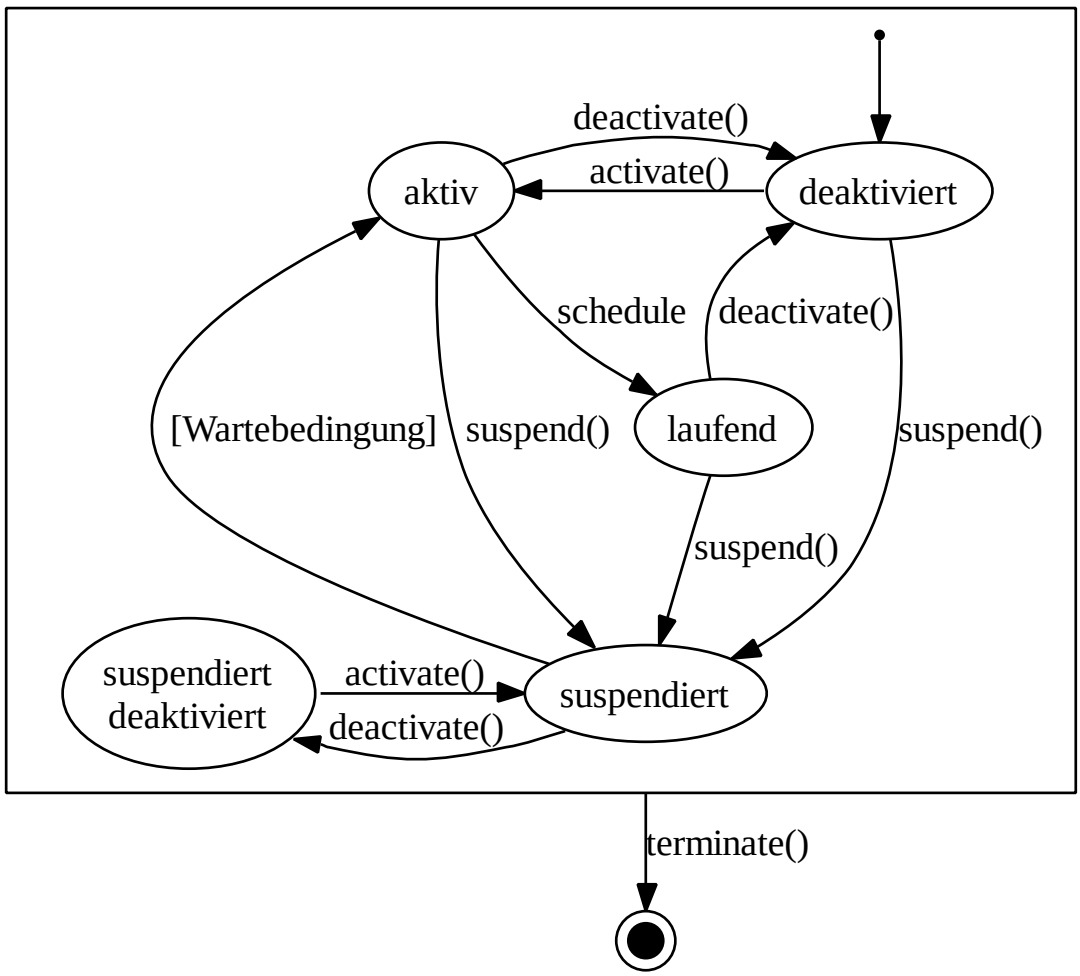


Abbildung 15: Zustände eines Context

- Handler haben immer Vorrang vor normalen Contexten
- Handler an inneren Blöcken haben Vorrang vor Handlern an übergeordneten Blöcken

Bei gleicher Priorität werden die Contexte in der Reihenfolge des Eintreffens in der Queue (d.h. der Suspendierung) ausgeführt.

4.2.3 Wartebedingungen

In Monaco ist es möglich, auf das Eintreten einer beliebigen booleschen Bedingung zu warten. Dazu muss einfach nur der Context suspendiert werden. Der Context bleibt solange im suspendierten Zustand, bis die Bedingung wahr wird. Dann wird der Context aktiv, und die VM führt ihn bei nächster Gelegenheit, d.h. spätestens am Ende des aktuellen logischen Zeitschritts, aus.

4.2.4 Asynchrone Ereignisse

Asynchrone Ereignisbehandlung kann mit einem Handler-Context realisiert werden. Dieses Konzept wird verwendet, um `ON`-Handler zu implementieren.

Ein Handler-Context hat genau einen Parent-Context. Er ist fest mit einem Block seines Parent verbunden. Mit der Methode `createHandler` von `IContext` kann ein neuer Handler-Context angelegt und mit dem aktuellen Block verbunden werden. Der Context wird sofort nach dem Anlegen suspendiert, um auf das Eintreten des asynchronen Ereignisses zu warten.

Ein `ON`-Handler verhält sich also in etwa so, wie ein paralleler Thread mit `WAIT` am Anfang. Wenn allerdings ein Handler-Context aktiv wird, deaktiviert er automatisch seinen Parent und alle weiteren am selben Block oder weiter unten hängenden Handler. Während der Ausführung verhält sich der Handler-Context wie ein ganz normaler Context. Wenn er terminiert, wird eine der folgenden Aktionen gesetzt:

Wenn der Handler-Context mit `resume = false` erzeugt wurde, werden die Blöcke des Parent-Context bis zu dem Block abgebaut, in dem der Handler erzeugt wurde. Alle weiter unten liegenden Kinder werden terminiert. Die Ausführung setzt am unmittelbar übergeordneten Block fort. Das entspricht einem normalen `ON`-Handler.

Wenn der Handler-Context mit `resume = true` erzeugt wurde, werden der Parent und alle seine Kinder wieder aktiviert und der Handler erneut installiert. Die Ausfüh-

rung setzt genau dort fort, wo der Parent unterbrochen wurde. Das entspricht dem Verhalten der `RESUME`-Anweisung in Monaco.

4.2.5 Parallele Ausführung und Synchronisation

Zur parallelen Ausführung und anschließenden Synchronisation kann ein Context in mehrere Kinder aufgespalten werden. Das geschieht mit der Methode `split`. Die Ausführung des Parent wird erst fortgesetzt, wenn alle seine Kinder terminiert sind. Dieses Konzept wird verwendet, um das `PARALLEL`-Statement zu realisieren.

Der Parent-Context ist untrennbar mit seinen Kindern verbunden. Wenn der Parent durch einen aktiv werdenden Handler deaktiviert und nachher wieder aktiviert wird, werden auch alle Kinder deaktiviert bzw. aktiviert. Wenn der Parent vorzeitig terminiert, werden auch die Kinder terminiert.

4.2.6 Asynchrone Ausführung

Über das Interface `IVM` kann jederzeit ein neuer Context angelegt werden. Dieser Context ist vollkommen unabhängig von allen anderen. Sobald er aktiviert wird, läuft er asynchron zum Rest des Programms. Dieser Mechanismus wird auch beim Programmstart zum Erstellen des ersten Context verwendet.

Die Sprache Monaco besitzt momentan kein Sprachkonstrukt zur asynchronen Ausführung von Code.

4.2.7 Synchrone Ausführung

Mit der Methode `execute` kann ein Context synchron ausgeführt werden. Der Aufruf blockiert solange, bis der Context terminiert.

Ein synchroner Context darf während er läuft nicht suspendiert oder deaktiviert werden. Er darf keine Handler anlegen und nicht gesplittet werden. Ein synchron laufender Context muss mit der Methode `terminate(Value ret)` terminiert werden. Der Parameter wird von der `execute`-Methode zurückgegeben.

Dieser Mechanismus kann zur synchronen Ausführung von Funktionen verwendet werden. Deren Rückgabewert wird sofort benötigt, um die aktuelle `Expression` fertig berechnen zu können.

4.2.8 Beispiel

Die folgende Tabelle zeigt den Ablauf der Routine `drill` des Beispielprogramms aus Listing 13. Die Spalten stellen die beteiligten Contexte dar. Die horizontalen Linien zeigen die Grenzen der logischen Zeitschritte. Der Code in den Routinenaufrufen wird nicht dargestellt. Der Rücksprung der Routinen wird mit `RETURN` angedeutet. Die Handler-Contexte wurden zur Erhöhung der Übersichtlichkeit ausgelassen. Es wird angenommen, dass ihre Bedingungen nicht eintreten.

Hauptablauf	`PARALLEL` Zweig 1	`PARALLEL` Zweig 2
`driller.startDriller()`		
⋮	⋮	⋮

`RETURN startDriller()`
`PARALLEL`
deaktiviert
	start
	aktiv
		start
		aktiv

	`driller.down(depth)`	`WAIT ...` *suspendiert*
⋮	⋮	⋮
		aktiv `cooler.on()`
		`WAIT ...` *suspendiert*
⋮	⋮	⋮

	`RETURN down(depth)`
	`WAIT ...`
	suspendiert
		aktiv
		`cooler.off()`

		terminiert
⋮	⋮	⋮
	aktiv `driller.up()`	
⋮	⋮	⋮
aktiv	`RETURN up()` *terminiert*	
`driller.stopDriller()` *terminiert*		

Zuerst wird im Hauptablauf die Routine `driller.startDriller` aufgerufen. Wie jeder Routinenaufruf suspendiert er den Context, es vergeht mindestens ein logischer Zeitschritt. Nach dem Rücksprung wird das `PARALLEL`-Statement ausgeführt. Zwei neue Contexte werden angelegt und der alte Context wird deaktiviert.

Der erste parallele Ablauf beginnt mit einem Routinenaufruf. Dieser läuft bis zum ersten `WAIT`. Anschließend kommt noch im selben logischen Zeitschritt der zweite Zweig dran und wird vom `WAIT` suspendiert. Gewisse Zeit später tritt die Bedingung des `WAIT` ein und der zweite Zweig wird wieder aktiv. Er schaltet den Kühler ein und wird wieder suspendiert.

Etwas später kehrt der Routinenaufruf des ersten Zweigs zurück. Das nächste Statement, in diesem Fall ein `WAIT`, wird ausgeführt und der Context wird wieder suspendiert. Während der Ausführung wurde der zweite Zweig aktiv, das heißt noch im selben Zeitschritt wird dessen Code ausgeführt, der Kühler wird wieder ausgeschaltet. Anschließend terminiert der Context des zweiten Zweigs.

Nach Ablauf der Wartezeit später wird wieder der erste Zweig aktiv. Nun wird die Routine `driller.up` aufgerufen. Nach dem Rücksprung dieser Routine terminiert auch der erste Zweig. Da nun alle Zweige des `PARALLEL`-Statement terminiert sind, wird der erste Context wieder aktiviert. Die Routine `driller.stopDriller` wird noch aufgerufen, dann terminiert der Ablauf.

5 Ausführung von Monaco-Code

In Kapitel 4 wurde das Ausführungsmodell der Monaco-VM bereits ausführlich erläutert. In diesem Kapitel wird nun im Detail beschrieben, wie Monaco-Programme durch die Monaco-VM ausgeführt werden.

Die Ausführung von Monaco-Programmen basiert auf dem in Kapitel 3 vorgestellten CodeDOM. Die Semantik der einzelnen Statements ist im CodeDOM implementiert, die VM stellt nur eine Ausführungsumgebung bereit. Bei der Ausführung eines Context wird der Reihe nach die `execute`-Methode der einzelnen Statements aufgerufen. Die Statements können dann die von der VM zur Verfügung gestellten Dienste nutzen um ihre Aufgabe zu erfüllen.

Die meisten dieser Dienste werden durch Interface `IContext` bereitgestellt. Die in diesem Kapitel erwähnten Methoden beziehen sich, wo nicht explizit anders erwähnt, immer auf dieses Interface.

Kooperatives Multitasking wird bereits von der Monaco-VM unterstützt. Dazu muss im CodeDOM nichts mehr explizit implementiert werden. Interessant sind allerdings die Punkte, an denen eine Unterbrechung des Ablaufs erfolgen kann. Diese Punkte sind insofern bereits durch die VM vorgegeben, als der Ablauf eines Context nur unterbrochen werden kann, wenn er explizit suspendiert oder deaktiviert wird (siehe Abbildung 15 auf Seite 39).

Da sich aber die Ausführung von manchen Statements über mehrere Contexte erstreckt, ergeben sich durch den Contextwechsel weitere Unterbrechungspunkte im logischen Ablauf. Um Verwirrung durch diese implizite Semantik zu vermeiden, wird im Folgenden auf alle Unterbrechungspunkte explizit hingewiesen.

5.1 Datenmanipulation

Wie bereits im Kapitel 3 beschrieben, erfolgt jeder Datenzugriff (sowohl lesend als auch schreibend) über Subklassen der Klasse `VariableRef`.

Bei der Ausführung ist jede Variable in einem `IValueStore` gespeichert. Das heißt, jede Variable ist durch einen `IValueStore` und einen Integer-Index eindeutig bestimmt (siehe Listing 20 auf Seite 35). Die einzelnen `VariableRef`-Subklassen unterscheiden sich dadurch, woher sie den `IValueStore` bekommen und wie sie den Index bestimmen.

5.1.1 lokale Variablen

Lokale Variablen werden direkt im aktuellen Scope des Context abgespeichert. Der Zugriff auf den Scope erfolgt mit der Methode `getLocals`. Der Index innerhalb des `IValueStore` ist konstant im CodeDOM abgelegt.

5.1.2 Membervariablen

Mit Membervariablen sind hier alle Variablen gemeint, die Teil eines anderen „Container-Objekts“ sind. Beispiele dafür sind Ereignisse und Member von Strukturen, Komponenten und Subkomponenten. Die Container-Objekte werden durch `Value`-Subklassen dargestellt, die das `IValueStore`-Interface implementieren.

Der Zugriff auf Membervariablen erfolgt über die Klasse `MemberVariableRef`. Sie enthält ist eine geschachtelte `VariableRef`, die auf das Objekt verweist, in dem die Membervariable abgelegt ist, und einen konstanten Index.

5.1.3 SELF

Das Schlüsselwort `SELF` bezeichnet die aktuelle Komponente. `SELF` ist an sich keine echte Variable und daher auch nicht in einem `IValueStore` gespeichert.

Die Zugriffsklasse `SelfRef` wird nur zur Schachtelung verwendet, um auf die Membervariablen der aktuellen Komponente zugreifen zu können. Abgelegt wird der Wert im aktuellen Scope. Der Lesezugriff erfolgt mit der Methode `getSelf`. Schreibzugriff ist nicht möglich.

5.1.4 Arrayelemente

Arrays werden in Monaco durch die Klasse `ArrayValue` dargestellt. Diese Klasse implementiert auch das Interface `IValueStore`.

Die Klasse `ArrayVariableRef` implementiert den Zugriff auf einzelne Elemente eines Arrays. Wie bei der `MemberVariableRef` gibt es eine geschachtelte `VariableRef`, die auf das Array zeigt. Allerdings ist der Index hier nicht konstant. Er ist durch eine `IntegerExpression` gegeben, die bei jedem Zugriff ausgewertet wird.

5.1.5 globale Konstanten

Globale Konstanten sind ähnlich implementiert wie lokale Variablen, allerdings ist nur lesender Zugriff erlaubt. Außerdem wird nicht der Scope des aktuellen Context sondern der Scope des globalen Context verwendet (Methode `getGlobalContext` im Interface `IVM`).

5.1.6 Beispiel

Abbildung 16 zeigt die Zuweisung `driller.upPosition := 250.0` aus dem Setup des Beispielprogramms (siehe Listing 11). Die rechte Seite der Zuweisung ist eine konstante Expression. Diese wird am Anfang evaluiert.

Die linke Seite ist eine Referenz auf die Membervariable `driller.upPosition`, dargestellt durch ein `MemberVariableRef`-Objekt. Dieses ist wiederum zusammengesetzt aus einer Referenz auf `driller` (eine lokale Variable) und der Deklaration der Variable `upPosition`.

Beim Schreiben eines Wertes in die `MemberVariableRef` (3) wird zuerst der Wert aus der `LocalVariableRef` gelesen (4). Diese holt sich die Adresse der lokalen Variable

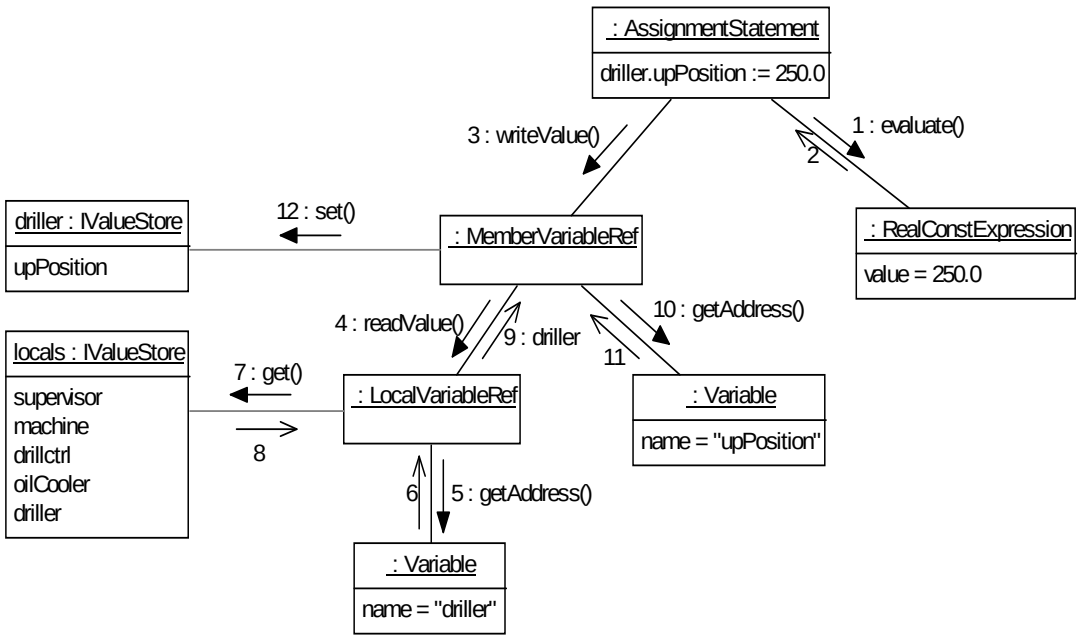


Abbildung 16: Zuweisung des Parameters `driller.upPosition := 250.0`

`driller` aus deren Deklaration (5), und den `IValueStore` mit den lokalen Variablen aus dem aktuellen Context. Dann wird die Variable aus dem `IValueStore` gelesen (7).

Das Ergebnis dieser Leseoperation (9) ist die Instanz der Komponente `driller`. Diese implementiert das Interface `IValueStore`. Nun wird aus der Deklaration der Variable `upPosition` deren Adresse ausgelesen (10) und an dieser Adresse der Wert 250.0 in die Komponente geschrieben (12).

5.2 Kontrollfluss

Kontrollflussstatements nutzen zur Ausführung die Blöcke der VM. Die Ausführung ist immer ähnlich:

- Setzen des Instruction-Pointers auf die nächste Instruktion.
- Anlegen eines neuen Blocks (Methode `openBlock`).
- Setzen des Instruction-Pointers auf die erste Instruktion im Körper des Statements.

5.2.1 Blöcke

Monaco-Blöcke werden eins zu eins auf VM-Blöcke abgebildet. Der Körper eines Monaco-Blocks wird einfach in einem geschachtelten VM-Block ausgeführt.

Zusätzlich können noch `ON`-Handler installiert werden. Siehe dazu Abschnitt 5.4.

5.2.2 Schleifen

Anstatt den Instruction-Pointer auf die nächste Instruktion zu setzen, setzt ihn das `LOOP`-Statement auf sich selbst. Das hat zur Folge, dass wieder mit dem selben `LOOP`-Statement fortgesetzt wird, nachdem der Block des Körpers geschlossen wurde

Die `WHILE`-Schleife funktioniert im Prinzip genauso. Zusätzlich wird zu Beginn die Bedingung evaluiert. Wenn das Ergebnis `TRUE` ist, wird wie bei der `LOOP`-Schleife vorgegangen. Sonst wird der Instruction-Pointer einfach nur auf das Statement gesetzt, das der Schleife folgt.

5.2.3 Verzweigungen

Das `IF`-Statement evaluiert der Reihe nach alle Bedingungen der `IfThen`-Zweige, bis eine gefunden wird, die zu `TRUE` evaluiert. Dann wird ein neuer Block angelegt und der Instruction-Pointer auf den Beginn des Körpers des entsprechenden Zweigs (oder wenn vorhanden des `ELSE`-Zweigs) gesetzt.

5.2.4 ALTERNATIVE

Das `ALTERNATIVE`-Statement soll gleichzeitig auf mehrere Bedingungen warten. Sobald eine der Bedingungen aus den `WHEN`-Zweigen eintritt, soll mit diesem Zweig fortgesetzt werden.

Dieses Verhalten wird mit Handlern implementiert. Zuerst wird ein neuer Block angelegt. Jeder `WHEN`-Zweig wird in diesem Block als Handler installiert (siehe Abschnitt 5.4). Anschließend wird der aktuelle Context mit der konstanten Wartebedingung `FALSE` suspendiert, er kann also nur weiterlaufen nachdem einer der Handler aktiv wurde.

Durch diese Implementierung ergibt sich sowohl am Anfang als auch am Ende eines `ALTERNATIVE`-Statements ein impliziter Unterbrechungspunkt.

5.3 WAIT

Das `WAIT`-Statement ist eines der wichtigsten Statements der Sprache Monaco. Die Implementierung des `WAIT`-Statements ist aber relativ einfach.

Zuerst wird der Instruction-Pointer auf das nächste Statement gesetzt. Dann wird der aktuelle Context mit der im Statement enthaltenen Wartebedingung suspendiert, sodass die Ausführung erst nach Eintreten dieser Bedingung fortgesetzt wird.

Klarerweise ist jedes `WAIT`-Statement ein Unterbrechungspunkt. Wichtig ist, dass die Ausführung immer unterbrochen wird, auch wenn die Bedingung beim Erreichen des Statements bereits wahr ist. Das ist notwendig, um den Vorrang von `ON`-Handlern vor `WAIT`-Statements sicherzustellen. Außerdem wird dadurch sichergestellt, dass jeder parallele Ablauf regelmäßig aktiviert wird.

5.4 ON-Handler

Wenn ein Monaco-Block einen `ON`-Handler enthält, bedeutet das, dass die Ausführung des Blocks bei Eintreten einer Bedingung unterbrochen und stattdessen mit dem Handler fortgesetzt werden soll. Genau diese Semantik kann mit einem Handler-Context erreicht werden.

Wenn ein `BlockStatement` ausgeführt wird, wird zuerst ein neuer Block angelegt. Dann wird in diesem Block für jeden `ON`-Handler ein Handler-Context angelegt (Methode `createHandler(boolean resume)`). Dieser Context wird anschließend mit der Wartebedingung des `ON`-Blocks suspendiert.

Anschließend läuft der aktuelle Context normal weiter. Die bereits in Kapitel 4 beschriebene Semantik des Handler-Context sorgt automatisch dafür, dass die Ausführung des Parent-Context bei Eintreten der Wartebedingung unterbrochen und nachher wieder korrekt fortgesetzt wird.

Durch die Abtrennung der Handler in einen eigenen Context ergibt sich am Ende jedes Handlers ein Unterbrechungspunkt. Wenn allerdings der Block normal durchläuft und kein Handler feuert, gibt es auch keinen Unterbrechungspunkt am Ende.

5.4.1 RESUME

Das `RESUME`-Statement wird vom Compiler im CodeDOM nicht als Statement, sondern als Eigenschaft eines `ON`-Handlers abgelegt.

Je nachdem, ob ein `ON`-Handler mit `RESUME` endet, wird der `resume`-Parameter beim Erstellen des Handler-Context gesetzt. Die Semantik von abbrechenden und wiederaufsetzenden Handlern ergibt sich automatisch aus der in Kapitel 4 beschriebenen Fortsetzungsstrategie eines Handler-Context mit oder ohne `resume`.

5.5 Parallelität

Parallelität wird in der Monaco-VM durch parallel laufende Contexte realisiert.

Das `PARALLEL`-Statement legt zuerst mit der Methode `split(int count)` einen neuen Context für jeden parallelen Zweig an. Der aktuelle Scope wird mit der Methode `openScope` auf jeden Child-Context übertragen. Dann wird der Instruction Pointer jedes Child-Context auf das erste Statement des entsprechenden Zweigs gesetzt.

Dann wird noch der Instruction Pointer des aktuellen Context auf das nachfolgende

Statement gesetzt. Die Semantik der `split`-Methode sorgt dafür, dass der aktuelle Context solange deaktiviert bleibt, bis alle untergeordneten Contexte terminiert sind. Das nachfolgende Statement wird also erst ausgeführt, wenn alle parallelen Zweige fertig sind.

Durch die Contextwechsel ergibt sich sowohl am Anfang als auch am Ende eines `PARALLEL`-Statements jeweils ein Unterbrechungspunkt.

5.6 Beispiel

Abbildung 17 zeigt die Interaktion zwischen der VM und dem CodeDOM anhand des Hauptablaufs der Routine `drill` des Beispielprogramms. Die Objekte links in der Abbildung (`IVM` und die Contexte) sind Teil der VM. Die anderen Objekte (der Handler und die beiden Statements) sind Teil des CodeDOM.

Nach dem Start der VM wird zuerst der Context für den Hauptablauf erzeugt (2). Dieser wird von der VM ausgeführt (3). Das nächste auszuführende Statement ist das oberste `BlockStatement`, also wird von diesem Statement die `execute`-Methode aufgerufen (4). Wie in Abschnitt 5.2 beschrieben setzt der Block zuerst den Instruction-Pointer auf das nächste Statement (5). Dann wird ein neuer VM-Block geöffnet (6).

Wie in Abschnitt 5.4 beschrieben, werden dann die ON-Handler installiert (7). Jeder Handler erzeugt mit `createHandler` einen neuen Context (8), setzt seinen Instruction-Pointer auf das erste Statement im Handler (11) und suspendiert ihn anschließend (12). Dann wird noch der Instruction-Pointer auf das erste Statement im Block gesetzt (13). Damit ist die Ausführung des `BlockStatement` abgeschlossen (14).

Um die Übersichtlichkeit zu wahren, wird hier auf den jetzt folgenden Routinenaufruf und dessen Inhalt nicht näher eingegangen. Wir nehmen an, dass jetzt sofort das `PARALLEL`-Statement dran kommt. Dieses wird von der VM ausgeführt (15).

Zuerst wird wieder der Instruction-Pointer auf das nächste Statement gesetzt (16). Dann werden wie in Abschnitt 5.5 beschrieben mit der Methode `split` neue Contexte für die parallelen Ausführunszweige angelegt (17). Der Instruction-Pointer der neuen Contexte wird jeweils auf das erste Statement des entsprechenden Zweigs gesetzt (20).

Die Ausführung des `ParallelStatement` ist damit abgeschlossen (21). Da durch den `split`-Aufruf der aktuelle Context deaktiviert wurde, wird nun kein weiteres Statement ausgeführt, sondern zum Scheduler der VM zurückgekehrt (22). Dieser führt nun einen der beiden parallelen Contexte aus (23). Erst nachdem beide parallelen Contexte terminiert sind (angedeutet durch 24), wird der erste Context wieder aktiv und kann

erneut ausgeführt werden (26).

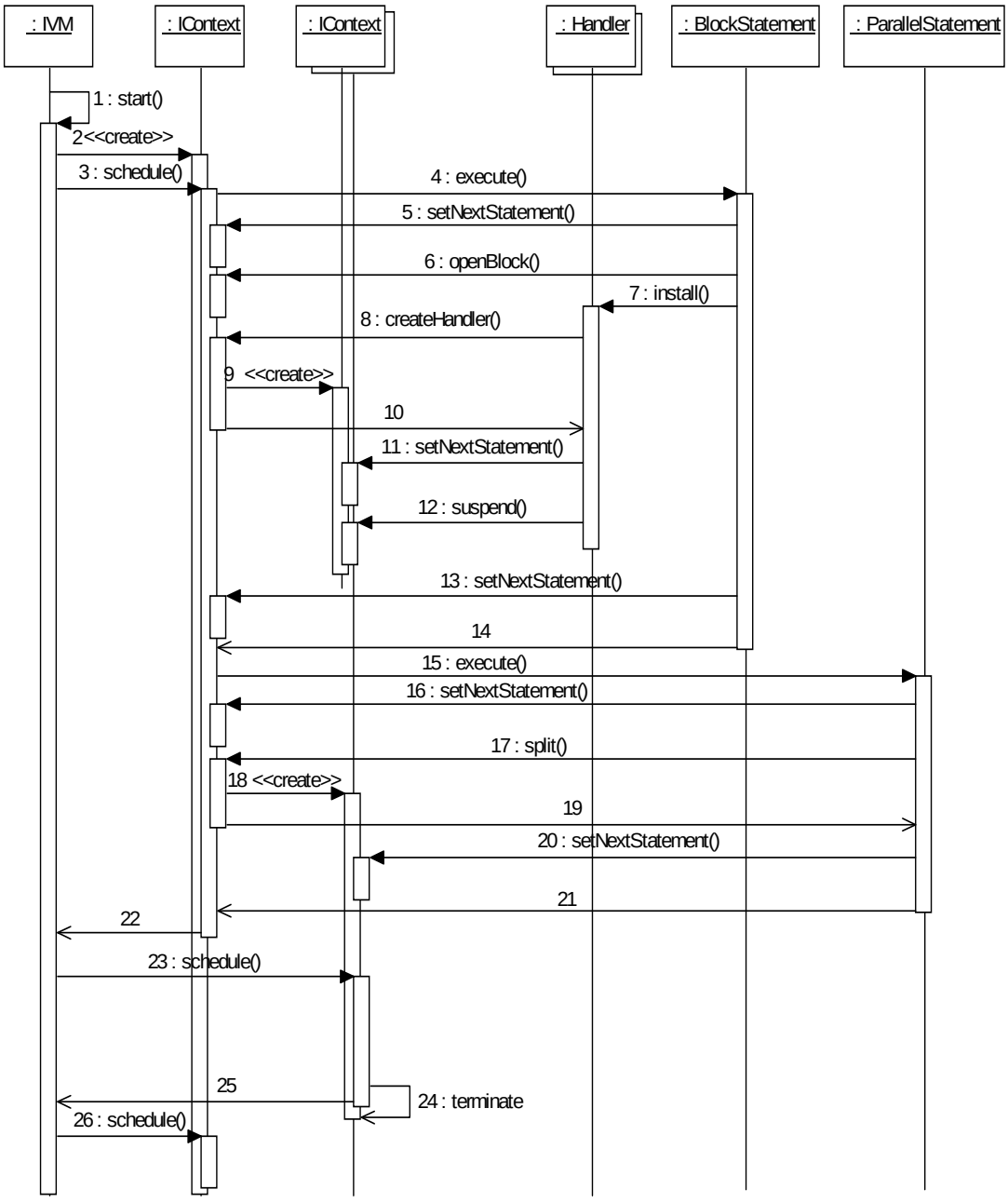


Abbildung 17: Interaktion zwischen VM und CodeDOM

5.7 Routinenaufrufe

Semantisch ist ein Routinenaufruf immer äquivalent zum Inlining der betreffenden Routine. Allerdings gibt es am Ende jeder Routine einen impliziten Unterbrechungspunkt.

Das `RoutineCall`-Statement evaluiert zuerst alle Parameter-Expressions und setzt die nächste Instruktion auf ein implizites `WAIT`-Statement. Anschließend wird ein neuer Block und in diesem Block ein neuer Scope angelegt. Die ersten Variablen des Scopes werden mit den Werten der Parameter initialisiert und der Instruction Pointer des Scopes wird auf das erste Statement der Routine gesetzt.

Die aufzurufende Routine wird über ein `ICallTarget` referenziert. In den folgenden Abschnitten werden die unterschiedlichen Implementierungen dieses Interface erläutert.

5.7.1 Lokaler Routinenaufruf

Der einfachste Fall einen Routinenaufrufs ist der Aufruf in die aktuelle Komponente (`SELF`). Er wird vom `LocalCallTarget` realisiert.

Dieses `ICallTarget` enthält einfach eine direkte Referenz auf das `Callable` das aufgerufen werden soll. Daraus kann das erste Statement direkt ausgelesen werden.

5.7.2 Statischer Routinenaufruf

Der Aufruf einer Routine in einer anderen Komponente wird von `StaticCallTarget` vorgenommen.

Auch hier wird der Aufruf direkt über das `Callable` der Komponente erledigt. Allerdings muss beim Erstellen des Scopes eine neue `SELF`-Referenz angegeben werden. Diese stammt von einer `ComponentExpression`.

Diese Form des Routinenaufrufs ist nur im `SETUP` erlaubt (siehe Abschnitt 5.9).

5.7.3 Virtueller Routinenaufruf

Der Aufruf einer Routine in einer Subkomponente ist der schwierigste Fall. Die Implementierung erfolgt in der Klasse `VirtualCallTarget`.

Wie beim statischen Routinenaufruf muss die `SELF`-Referenz geändert werden. Der neue Wert stammt aus einer `InterfaceExpression`.

Außerdem kann in diesem Fall zur Compilezeit noch nicht festgestellt werden, welches `Callable` wirklich ausgeführt werden wird. Das `VirtualCallTarget` enthält nur das abstrakte `Callable` des Interface. Die Implementierung kann erst zur Laufzeit ermittelt werden, wenn der Inhalt der Subkomponenten-Variable bekannt ist.

Genau genommen könnte durch den statischen Systemaufbau jeder virtuelle Aufruf bereits zur `SETUP`-Zeit in einen statischen Aufruf umgewandelt werden. Diese Optimierung wurde in der aktuellen Version der VM noch nicht implementiert.

5.7.4 Beispiel

Abbildung 18 zeigt, wie der Routinenaufruf `driller.down(depth)` ausgeführt wird. Das ist der Routinenaufruf, dessen Datenstruktur bereits in Abschnitt 3.5.5 vorgestellt wurde.

Zuerst wird wie bei fast jedem Statement der Instruction-Pointer auf das nächste Statement gesetzt (3). Dann wird noch im alten Scope die Parameterliste ausgewertet (4)

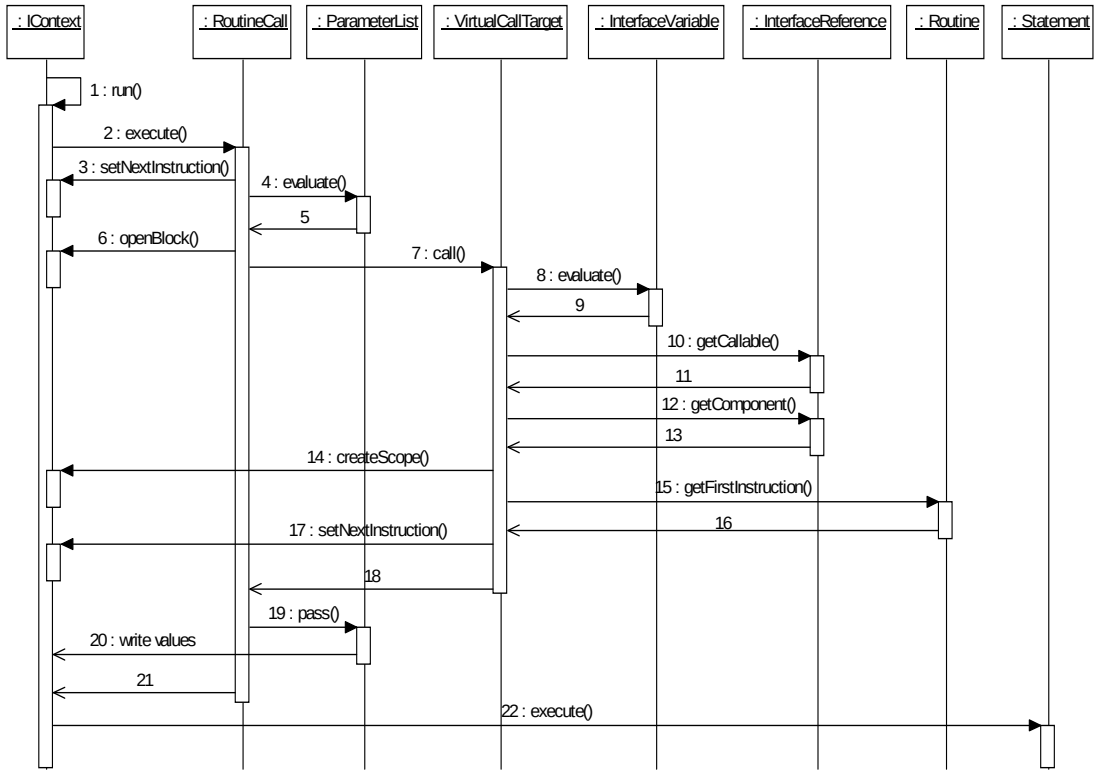


Abbildung 18: Ausführung eines Routinenaufrufs

und das Ergebnis zwischengespeichert. Anschließend wird ein neuer Block für die Routine angelegt (6) und der Aufruf vorbereitet (7).

Dazu wird zuerst die `InterfaceVariable` ausgelesen (8). Das Ergebnis ist eine `InterfaceReference`. Diese Klasse enthält eine Referenz auf die Komponente `driller`. Mit diesem Objekt kann jetzt die Implementierung der virtuellen Routine gesucht werden (10). Außerdem wird die neue `SELF`-Referenz bestimmt (12).

Mit diesen Informationen wird im Context ein neuer Scope angelegt (14). Dann wird aus der Routine das erste `Statement` gelesen (15) und der Instruction-Pointer gesetzt (17). Nun müssen nur noch die vorher vorbereiteten Parameter in den neu angelegten Scope geschrieben werden (19). Die Ausführung des `RoutineCall` ist damit beendet (21), der Context setzt die Ausführung mit dem ersten Statement der Routine fort (22).

5.8 Funktionsaufrufe

Im Gegensatz zum Routinenaufruf genügt es beim Funktionsaufruf nicht, einfach das nächste Statement zu ermitteln. Funktionsaufrufe sind im CodeDOM als `Expression`-Subklasse modelliert. Der Rückgabewert der Funktion muss noch während der Evaluierung dieser `Expression` bestimmt werden.

Allerdings gibt es bei Funktionen Einschränkungen. Innerhalb von Funktionen sind keine `WAIT`-Statements, Routinenaufrufe, `PARALLEL`-Statements oder `ON`-Handler erlaubt. Allgemein gesagt ist in einer Funktion nichts erlaubt, was den Context suspendieren oder deaktivieren könnte. Durch diese Einschränkungen ist es möglich, den Code der Funktion in einem weiteren Context synchron auszuführen (siehe Abschnitt 4.2.7).

Die `FunctionCall`-Expressions legen also einen neuen Context an. In diesem neuen Context wird genauso wie beim Routinenaufruf ein neuer Scope angelegt, der neue Instruction-Pointer gesetzt und die Parameter übergeben. Anschließend wird der Context mit der `execute`-Methode synchron ausgeführt und das Ergebnis zurückgegeben.

5.9 Setup und Programmstart

In Monaco wird die gesamte Komponentenhierarchie sowie die Komponentenparameter und der Inhalt der globalen Konstanten im `SETUP` festgelegt.

Implementiert wird das `SETUP` als ein spezielles `Callable`. Alle Komponenten werden als lokale Variablen des `SETUP` angelegt. Die Hierarchie und die Parameter werden

durch Zuweisungen im Körper des SETUP realisiert. Das letzte Statement im SETUP ist ein Routinenaufruf in die obersten Komponente, der das eigentliche Monaco-Programm startet.

Beim Start des Systems wird zuerst der globale Context angelegt und mit den Werten der globalen Konstanten initialisiert. Dann wird ein ausführbarer Context erzeugt. In diesem Context wird genau wie beim Routinenaufruf ein Scope für das SETUP angelegt und der Instruction Pointer auf das erste Statement gesetzt. Anschließend wird der Context aktiviert und der Scheduler der VM gestartet.

6 Zusammenfassung

Monaco ist eine domänenspezifische Programmiersprache für Maschinensteuerungen, die als Basis für ein Endbenutzer-Programmiersystem entwickelt wurde. In dieser Arbeit wurde eine objektorientierte Datenstruktur (CodeDOM) als Zwischensprache zur Ausführung von Monaco-Programmen vorgestellt. Weiters wurde eine virtuelle Maschine vorgestellt, um Monaco-Programme ausführen zu können. Dabei wurden auch die wesentlichen Sprachkonstrukte der Programmiersprache Monaco kurz vorgestellt und deren Ausführung auf der virtuellen Maschine erklärt.

Der Vorteil der Aufteilung in einen objektorientierten CodeDOM und eine weitgehend sprachunabhängige virtuelle Maschine ist, dass man ohne großen Aufwand neue Sprachkonstrukte und Notationen testen kann. Es genügt, neue Klassen im CodeDOM einzuführen. Die virtuelle Maschine ist von diesen Änderungen nicht betroffen. Außerdem kann der CodeDOM von einem graphischen Editor direkt bearbeitet werden, was einen wichtiger Schritt in Richtung Endbenutzer-Programmierung darstellt.

Im Rahmen dieser Arbeit wurden auch ein Compiler, der Monaco-Code in den CodeDOM übersetzt, sowie eine Implementierung der virtuellen Maschine in Java entwickelt. Dieser Java-Prototyp ist klarerweise noch nicht echtzeitfähig. Aufbauend auf den Erfahrungen an diesem Prototypen ist jedoch eine optimierte Implementierung geplant, die Monaco-Programme in echtzeitfähigen C-Code übersetzt.

Literatur

[1] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, January 1995.

[2] HAREL, D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming 8*, 3 (June 1987), 231–274.

[3] HURNAUS, D. Eine domänenspezifische Programmiersprache für Maschinensteuerungen: Entwurf eines Compilers und einer Ausführungsumgebung. Diplomarbeit, Fachhochschule Hagenberg, 2006.

[4] MÖSSENBÖCK, H. Unterlagen zur Vorlesung: Fortgeschrittene Techniken des Übersetzerbaus. Institut für Systemsoftware, Johannes Kepler Universität Linz, 2006.

[5] MÖSSENBÖCK, H., WÖSS, A., AND LÖBERBAUER, M. The compiler generator Coco/R. http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/, September 2005.

[6] PRÄHOFER, H., HURNAUS, D., SCHATZ, R., AND MÖSSENBÖCK, H. The domain-specific language Monaco. Tech. rep., Christian Doppler Laboratory for Automated Software Engineering, Johannes Kepler University, Austria, 2007.

[7] SELIC, B. Using UML for modeling complex real-time systems. In *LCTES '98: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems* (London, UK, 1998), Springer-Verlag, pp. 250–260.

[8] WIRTH, C. Ein visueller Editor für die Programmiersprache Monaco. Bakkalaureatsarbeit, Institut für Systemsoftware, Johannes Kepler Universität Linz, 2007.

Codeverzeichnis

Abbildungsverzeichnis