ETH Zürich

Semesterarbeit am Institut für Computersysteme bei Prof. Jürg Gutknecht

Ein Programmiermodell für State Machines

Urs Fässler
ursf@student.ethz.ch

Betreut durch
Dr. Felix Friedrich

Frühjahr 2011

Danksagung

Mein Dank geht an alle die in irgendeiner Form zum gelingen dieser Arbeit beigetragen haben. Insbesondere Prof. Jürg Gutknecht, dass ich die Arbeit an seinem Institut durchführen konnte. Dr. Felix Friedrich für seine Unterstützung und Betreuung. Thomas Zehnder von der Firma B&R, dass er sich die Zeit genommen hat, Ideen und mögliche gemeinsame Arbeiten zu besprechen. Fritz Schönenberger für gute Diskussionen zum Thema. Allen Korrekturlesern, namentlich Ursina Rumetsch.

Schlussendlich möchte ich mich bei allen Personen oder Projekten bedanken, welche Ideen und Inspirationen gegeben haben oder Freie Software unterstützen, welche direkt oder indirekt für diese Arbeit benutzt wurde.

Bild Frontseite: ein Papageientaucher, erstellt von Aki G. Karlsson, Public Domain, www.openclipart.org

Abstract

Bei der Implementation vieler eingebetteter Systeme kommen Finite State Machines (FSM) zum Einsatz. Typischerweise werden diese in gängigen Programmiersprachen in fehleranfälliger oder umständlicher Weise implementiert. Um die Implementation auf Sprachebene besser und übersichtlicher unterstützen zu können, wird in dieser Semesterarbeit ein Programmiermodell entwickelt, welches speziell auf die Implementation von FSMs zugeschnitten ist. Als Grundlage für eine Implementierung dieses Modells wird die Programmiersprache Rizzly verwendet, welche eigens für eingebettete Systeme entworfen wurde. Die Realisierbarkeit wird gezeigt indem der bestehende Rizzly-zu-C Compiler um diese Funktionalität erweitert wird. Beispiele sollen schlussendlich die Anwendung demonstrieren.

Inhaltsverzeichnis

1. Einführung

Rizzly ist eine Programmiersprache für die reaktive Programmierung. Sie ist konzipiert für Anwendungen welche über wenig Hardware Ressourcen verfügen. Dies ist der Fall bei der Systemprogrammierung wie auch bei Embedded-Systemen.

In einem reaktiven System hat man eine Umkehrung der Kontrolle (Inversion of Control, siehe [5], eine deutsche Beschreibung findet man bei [11]). Im Gegensatz zur prozeduralen Programmierung bestimmt nicht der Programmierer den Programmfluss, sondern der Stream der Ereignisse welcher von ausserhalb des Systems kommt. Solch ein reaktives System lässt sich zwar mit prozeduralen Programmiersprachen entwickeln, diese sind jedoch nicht auf solche Anwendungen ausgelegt [20]. Dies zeigt sich auch darin, dass es Versuche gab, prozedurale Sprachen wie C [2] oder objektorientierte Sprachen wie Java [15] zu erweitern, um die reaktive Programmierung zu vereinfachen.

Eine weitere Eigenschaft der reaktiven Programmierung besteht darin, dass kooperatives Multitasking inhärent unterstützt wird. Der Overhead für ein Runtime System oder Betriebssystem reduziert sich im Vergleich zu Threads auf ein Minimum [8], was zu einer effizienteren Ausnutzung der Prozessorleistung führt.

Gegenüber anderen Programmiersprachen wie Scala [8] versucht Rizzly nicht die Inversion of Control zu verstecken, sondern unterstützt den Programmierer bei der Ausnutzung dieses Programmier-Paradigmas. Eine weit verbreitete Methode, die Komplexität in den Griff zu bekommen, ist mit Hilfe von State Machines.

1.1. Ähnliche Arbeiten

Mit C existiert eine etablierte Programmiersprache im Bereich der Embedded-Systeme und der Systemprogrammierung. Jedoch wurde C nicht für die reaktive Programmierung entworfen und bietet nicht die Features, welche sich in moderneren Programmiersprachen durchgesetzt haben. Insbesondere kennt ein C Compiler das Modell der State Machine nicht und kann darum weder Fehler entdecken noch bestimmte Optimierungen durchführen.

Frameworks wie z.B. qp [12] oder COMDES [1] können einen erheblichen Teil der Infrastruktur von State Machines übernehmen. Jedoch hat ein Framework nie die gesamte Kontrolle über den Kontrolfluss (z.B. kann ein *goto* nicht verhindert werden). Auch ist hier dem Compiler das Modell, welches der Programmierer benutzt, unbekannt.

Oft werden grafische Tools verwendet um State Machines zu erstellen, jedoch verliert man schnell die Übersicht [10]. Zwar kann eine grafische Repräsentation gerade für

State Machines sehr hilfreich sein, hingegen beschränkt die ständige Benutzung der Maus resp. den Wechsel zwischen der Tastatur und der Maus fortgeschrittene Benutzer in der Produktivität. Beispiele solcher Tools ist z.B. das CIP-Tool [4] oder Rational Rhapsody [17], welches unter anderen auch die UML Definition für State Machines [18] unterstützt.

Weiter existiert mit nesC [7] eine Programmiersprache, welche Komponenten und Kompositionen der Komponenten unterstützt, allerdings fehlt die Unterstützung von State Machines. Rebeca [14] setzt ebenfalls ähnliche Ideen um, aber auch hier werden State Machines nicht unterstützt.

1.2. Kurzbeschreibung Rizzly

Rizzly ist eine komponentenbasierte Programmiersprache. Eine Komponente implementiert, ähnlich wie ein Objekt, Interfaces. Dies beschreibt die Ereignisse, über welche das Objekt verändert wird. Für eine Komponente müssen auch Interfaces angegeben werden, welche diejenigen Ereignisse beschreiben, welche von der Komponente erzeugt werden. Erzeugt eine Komponente dieselben Ereignisse wie eine andere Komponente verwendet, so können diese zwei Komponenten mittels Sprachkonstrukten zusammengeschaltet werden. Solch eine Verbindung ist synchron, in einer späteren Version sollen auch asynchrone Verbindungen möglich sein.

Die Komponenten bestehen entweder aus anderen Komponenten und werden aus diesen zusammengesetzt, oder sie bestehen aus eigenständigem Code. Dieser kann momentan nur als als hierarchische State Machine implementiert werden.

Eine Komponente ist passiv - sie kann nur über Ereignisse (welche durch die implementierten Interfaces definiert sind) getriggert werden. Wird eine Komponente über ein Ereignis getriggert, so läuft die Ereignisbehandlung zu Ende. Dabei kann sie nicht durch ein anderes Ereignis unterbrochen werden (gleich wie synchroner Code in nesC [7]). Allgemein wird dies als Run-To-Completion bezeichnet. Dadurch ist das System einfacher verständlich und somit auch einfacher zu programmieren, was schlussendlich zu weniger Fehlern führt. Weiter ist der Zustand der State Machine konsistent, Deadlocks werden minimiert [16].

1.3. Beispiel

In Listings 1 und 2 ist jeweils eine Velolicht-Steuerung in C respektive in Rizzly realisiert. Das Velolicht hat 3 Modi: Ausgeschaltet, Leuchten und Blinken. Dieses Beispiel zeigt bereits die übersichtlichere Darstellung mit Rizzly State Machines. In den folgenden Kapiteln wird die Bedeutung der Syntax und die Semantik genauer erklärt.

Listing 1: Velolicht-Steuerung in C.

```c
#define    OFF          0
#define    ON           1
#define    BLINKOFF     2
#define    BLINKON      3

static int state = OFF;

void init()
{
  lampOff();
}

void tick()
{
  switch( state ){
    case BLINKON:
      lampOff();
      state = BLINKOFF;
      break;
    case BLINKOFF:
      lampOn();
      state = BLINKON;
      break;
  }
}

void mode()
{
  switch( state ){
    case OFF:
      lampOn();
      state = ON;
      break;
    case ON:
      state = BLINKON;
      break;
    case BLINKON:
      lampOff();
      state = OFF;
      break;
    case BLINKOFF:
      state = OFF;
      break;
  }
}
```

Listing 2: Velolicht-Steuerung in Rizzly.

```
Off = state()
  initial
  entry
    lamp.off()
  end
end

On = state()
  entry
    lamp.on()
  end
end

Blink = state()
end

BlinkOn = state( Blink )
  initial
  entry
    lamp.on()
  end
end

BlinkOff = state( Blink )
  entry
    lamp.off()
  end
end

transition Off to On
  by mode.pressed() end

transition On to Blink
  by mode.pressed() end

transition Blink to Off
  by mode.pressed() end

transition BlinkOn to BlinkOff
  by time.tick() end

transition BlinkOff to BlinkOn
  by time.tick() end
```

2. Syntax und Semantik

2.1. Allgemein

Die Definition der Syntax ist im Anhang in Listing 30 dargestellt. Generell orientiert sich die Syntax an Pascal. Allerdings wurde auf das Semikolon als Statement-Trenner verzichtet. Whitespaces wie Tabulator und Newlines können überall hinzugefügt werden. Kommentar kann wie in C deklariert werden. So startet ein Zeilenkommentar mit einem doppelten Slash (//), der Blockkommentar startet mit einem Slash-Stern (/*) und endet mit einem Stern-Slash (*/).

2.2. Interfaces

Listing 3: Interface Definition in Rizzly.

```
1 interface ExampleInterface
2
3 function newSymbol( symbol: Integer )
4 function finished()
5 query error():Boolean
```

Interfaces werden separat in je einer Datei definiert. Wie in Listing 3 zu sehen gibt es zwei verschiedene Typen von Einträgen. Function beschreibt ein Ereignis. Dieses kann zusätzlich Daten in Form von Argumenten beinhalten. Rückgabewerte sind nicht möglich. Query dient der Abfrage von Informationen und hat deshalb immer einen Rückgabewert. Parameter können im jetzigen Stand nicht übergeben werden.

2.3. Komponenten

Komponenten sind eigenständige Einheiten, welche nur über die verwendeten Eingangs- und Ausgangs-Interfaces definiert sind. In den Interfaces können Messages und Queries definiert sein. Eine Message berichtet über ein Ereignis und kann Daten enthalten. Meist löst eine Message eine Zustandsänderung und weitere Messages aus. Queries dagegen dienen der Abfrage von Komponenten und können keine Zustandsänderung an der abgefragten Komponente auslösen.

Listing 4: Header einer Komponente. Dieser sieht für die verschiedenen Komponenten immer gleich aus.

```
1  component Example
2
3  import
4    AnotherComponent
5    AnInterface
6
7  interface
8    input
9      in       : AnInterface
10
11   output
12     out      : AnInterface
```

2.4. Dateinamen, Namespace

Jede Komponente und jedes Interface muss in einer separaten Datei definiert werden. Dies hat zur Folge, dass ein Programm schnell aus vielen Dateien besteht. Zur bessern Übersicht sollte das Programm über mehrere Verzeichnisse verteilt werden. Die Verzeichnishierarchie entspricht den Namespaces im Programm. Um eine Datei zu importieren muss der gesamte Namespace angegeben werden. Bei der Verwendung einer Komponente (wie in Listing 5 Zeile 9 bis 11) oder eines Interfaces kann der Bezeichner verwendet werden. Ist dieser nicht eindeutig muss so viel vom Namespace mitangegeben werden bis sich die beiden Namespaces unterscheiden.

Listing 5: Verwendung von Namespaces

```
1  import
2    comp.os.Timer
3    comp.io.system.Base
4    comp.os.system.Base
5
6  implementation composition
7
8  component
9    timer          : Timer
10   io             : io.system.Base
11   os             : os.system.Base
```

2.5. Komposition

Eine Komposition ist eine Komponente, welche aus anderen Komponenten zusammengesetzt ist. Die Komponenten können untereinander sowie mit den Interfaces der äusseren Komponente verdrahtet werden. Die Reihenfolge, in welcher die Komponenten aufgeführt sind, entspricht der Reihenfolge der Initialisierung der Komponenten. Wird eine Message von mehreren Empfängern verwendet (siehe Listing 6 Zeile 16 und 17), so entspricht die Reihenfolge der Zustellung der Reihenfolge der Definition.

Listing 6: Komposition

```
1  interface
2    input
3      ifaceIn  : SomeInterface
4
5    output
6      ifaceOut : SomeInterface
7
8  implementation composition
9
10 component
11   instA    : classA
12   instB    : classB
13
14 connection
15   ifaceIn          -> instA.ifaceIn
16   instA.ifaceOut   -> ifaceOut
17   instA.ifaceOut   -> instB.ifaceIn
```

Eine Komposition kann als Graph aufgefasst werden (siehe Abbildung 11, Seite 27). Die Komponenten sind dabei Knoten und die Verbindungen Kanten. Es ist nicht erlaubt, in dem Graphen Zyklen zu erstellen. Da die Verbindungen synchron sind, würde dies die Run to Completion Semantik verletzen. In Zukunft sollen auch asynchrone Verbindungen sowie aktive Komponenten realisiert werden. In solch einem Setting sind dann auch Zyklen möglich.

2.6. Zustände

Jeder Zustand wird mit einem separaten Eintrag deklariert. In der einfachsten Ausführung wird nur der Name des Zustands benötigt. Ein Zustand kann auch innerhalb eines anderen Zustands definiert werden. Dazu wird der Zustand angegeben, in welchem der neue Zustand definiert wird. In Listing 7 sind die Zustände BA und BB Unter-Zustände

von B, B ist ein zusammengesetzter Zustand. Die Hierarchie der Zustände bilden wie in Abbildung 1 dargestellt einen Baum. Intern verwendet Rizzly einen Wurzelknoten welcher die Zustände ohne übergeordneten Zustand als Kinder hat. Die State Machine kann nur in Zuständen sein, welche den Blättern entsprechen, implizit ist sie dann auch in allen übergeordneten Zuständen. Jeweils ein Kind-Zustand muss als der initiale Zustand deklariert sein.

Listing 7: Zustand Deklaration

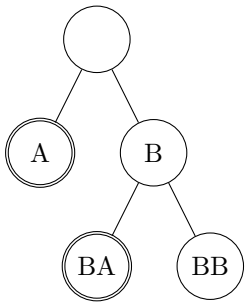
```
A = state()
  initial
end

B = state()
end

BA = state(B)
  initial
end

BB = state(B)
end
```

Abbildung 1: State Machine im Querschnitt. Initiale Zustände sind doppelt gezeichnet.



Innerhalb eines Zustands können, wie in Listing 8 dargestellt, Variablen deklariert werden. Diese sind solange gültig wie man sich in dem Zustand befindet. Ein Verlassen des Zustands zerstört den Inhalt der Variablen. Variablen sind in dem Zustand, in welchem sie definiert sind, sowie allen Unter-Zuständen sichtbar.

Ein Zustand kann Entry und Exit Funktionen besitzen, diese werden ausgeführt wenn man in einen Zustand hineinkommt respektive ihn verlässt. Listing 9 zeigt die Deklaration von Entry/Exit Funktionen.

Listing 8: Zustands-Variablen

```
A = state()
  var
    a : Integer
    b : Boolean
end
```

Listing 9: Entry/Exit Funktionen

```
A = state()
  entry
    ...
  end
  exit
    ...
  end
end
```

2.7. Transitionen

Eine Transition beschreibt, wann von einem Zustand in einen anderen gewechselt wird. Sie bietet die einzige Möglichkeit den Zustand einer Komponente zu ändern. Dies gilt auch für die Variablen von Zuständen. Transitionen werden durch Ereignisse, welche von ausserhalb der Komponente kommen, ausgelöst. Jedes Ereignis kann in einer Komponente genau eine oder keine Transition auslösen.

Listing 10: Eine einfache Transition

```
transition A to B by time.tick() end
```

Wie in Listing 10 dargestellt wird jede Transition separat spezifiziert. Minimal notwendig ist der Zustand den man verlässt (Source), der Zustand zu welchem man wechselt (Destination) sowie das triggernde Ereignis. Diese Lösung wurde gewählt, weil eine Transition im Modell ebenfalls eine atomare Einheit ist und z.B. eine Gruppierung nach Zustand oder Ereignis eine künstliche Ordnung erzwingen würde. Der Programmierer ist frei wie Transitionen sortiert werden sollen.

Source und Destination können beliebige Zustände sein, auch zusammengesetzte. Ist der Source Zustand ein zusammengesetzter wird die Transition dann durchgeführt, wenn kein enthaltener Zustand bereits eine Transition durchführen kann. Falls die Destination ein zusammengesetzter Zustand ist, wird jeweils zum initialen Zustand

gewechselt. Bei einer Transition werden alle Entry/Exit Funktionen bis zum tiefsten gemeinsamen Zustand ausgeführt.

Mit Hilfe von Guards können Transition bedingt benutzt werden. Ein Guard ist ein Prädikat, welches bei den Transition zusätzlich angegeben wird (siehe auch Listing 11). Das Prädikat kann auf Variablen vom Source-Zustand[1] oder Parameter des Ereignis zugreifen. Es können alle pure Functions benutzt werden. Sind zwei oder mehrere Transitionen mit demselben Source- sowie Destination-Zustand definiert und die Guards nicht disjunkt, so wird erst das Prädikat derjenigen Transition geprüft, welche zuoberst in der Datei definiert wurde. Ist das Prädikat erfüllt, wird die Transition durchgeführt. Wenn nicht wird mit den verbleibenden Transitionen fortgefahren.

Listing 11: Eine Transition mit einem Guard

```
1 transition B to A by time.tick() if time = 0 end
```

Schlussendlich kann eine Transition auch einen Body haben. Dies ist eine Liste von Statements, welche ausgeführt werden, wenn eine Transition benutzt wird. Im Body kann auf Variablen des Source- und Destination-Zustand sowie des auslösenden Ereignis zugegriffen werden. Auch können Aktionen ausgelöst sowie Variablen des Destination-Zustand beschrieben werden. Listing 12 zeigt solch eine Transition.

Listing 12: Eine Transition mit einem Body

```
1 transition B to A by time.tick() do
2   time := time - 1
3 end
```

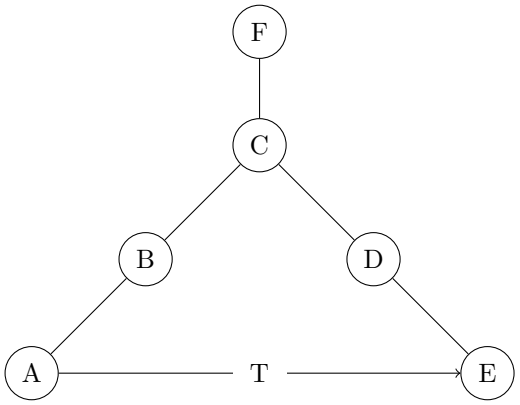
2.8. Sichtbarkeitsbereich und Ausführungsreihenfolge bei einer Transition

Der Sichtbarkeitsbereich und die Reihenfolge der Code-Ausführung einer umfangreicheren Transition ist nicht automatisch klar. Für die in Abbildung 2 dargestellte Transition T von A nach E gilt folgendes:

[1] beziehungsweise eines höheren Zustand

1. A.exit()
2. B.exit()
3. D sichtbar
4. E sichtbar
5. T.body()
6. A ungültig
7. B ungültig
8. D.entry()
9. E.entry()

Abbildung 2: Ablauf einer umfangreicheren Transition in einer hierarchischen State Machine. Die Kreise mit den Buchstaben sind Zustände. Die Linien zwischen den Zuständen zeigen die Hierarchie.



Dabei ist es egal, ob die Transition A, B, C oder F als Source-Zustand hat (insofern keine andere Transition T überschreibt). Da C und F nicht verlassen werden, wird bei einer Transition von A nach E niemals eine Entry/Exit Funktion von C oder F aufgerufen.

2.9. Hierarchische endliche State Machines

Die Implementation des Verhaltens kann mittels hierarchical finite State Machines (HFSM) realisiert werden. Dazu werden Zustände und Transitionen, wie vorhin beschrieben, verwendet. Es wäre auch möglich, dieselbe Funktionalität mit "einfachen" State Machines (nicht hierarchische, ohne zusätzliche Funktionalität) zu realisieren. Dies würde jedoch in umfangreicherem Code enden. Die Funktionalität müssten mehrmals programmiert werden, was schlussendlich fehleranfällig ist. Wie in Kapitel 3.2 erwähnt übernimmt dies Rizzly.

2.10. Queries und HFSM

Queries werden pro Zustand implementiert. Implementiert ein zusammengesetzter Zustand eine Query, so wird diese Implementation auch von den Unter-Zuständen be-

nutzt. Implementiert allerdings ein Unter-Zustand dieselbe Query wie ein übergeordneter Zustand, so wird immer die tiefstmögliche Implementation verwendet. Für jeden Zustand müssen alle in den verwendeten Interfaces definierten Queries implementiert sein. Dies kann entweder direkt im Zustand oder in einem übergeordneten geschehen. In Listing 13 ist die Implementation einer Query zu sehen.

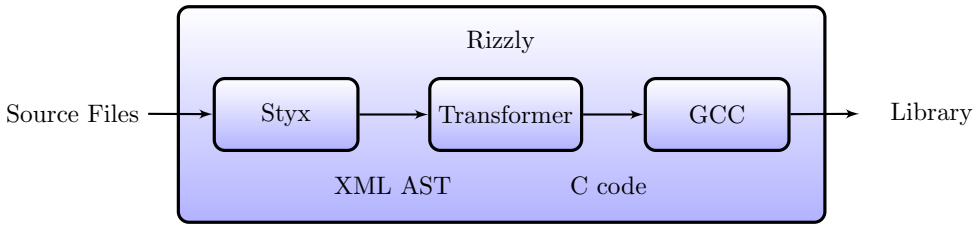
Listing 13: Deklaration einer Query in einem Zustand

```
1 A = state()
2   query info.getInfo():Integer
3     return 42
4   end
5 end
```

3. Technische Umsetzung

Wie in Abbildung 3 dargestellt besteht der Rizzly Compiler aus drei grösseren Modulen. Als Parser wird Styx und als Backend wird GCC verwendet. Die Transformierung vom AST zu C wurde selbst geschrieben. Der ganze Compiler wurde unter Debian GNU/Linux entwickelt. Der Source des Compilers kann unter `http://www.n.ethz.ch/~ursf/download/rizzly/compiler.zip` gefunden werden.

Abbildung 3: Aufbau und Workflow von Rizzly



3.1. Parser

Um die Rizzly Files zu parsen wird Styx [3] verwendet. Styx erfüllt die Kriterien einer Freien Lizenz (GNU General Public License, GPL), dass er ohne virtuelle Maschine auskommt, eine saubere Definition der Sprache zulässt sowie dass er einen generischen AST generieren kann, welcher sich einfach in ein XML schreiben lässt. Dass Styx bereits in den Debian Repositories vorhanden war, trug endgültig zu der Wahl bei.

Die Syntax-Definition von Rizzly ist im Anhang unter Listing 30 zu sehen. Styx generiert daraus C Code welcher den AST im Speicher aufbaut. Ein kleines C Programm verwendet den generierten Code, um eine Datei zu parsen und anschliessend das XML zu schreiben.

3.2. Transformierung

Wie bereits erwähnt wurde das Transformationsmodul selbst erstellt. Als Programmiersprache wurde Object Pascal verwendet. Entwickelt wurde unter Lazarus, kompiliert mit dem FreePascal Compiler.

Da der Compiler alle Source Files auf einmal braucht, ist der Compiler Driver ebenfalls im Transformationsmodul enthalten. Der Compiler Driver lädt alle Rizzly Sourcen,

aus welchen dann ein Abstract Semantic Graph (ASG) erstellt wird. Durch die semantische Prüfung und Verlinkung werden in dieser Phase erste Fehler und Probleme entdeckt. Dies umfasst fehlende Dateien, falsch benannte Namespaces, nicht gefundene Dateien, nicht vorhandene Interfaces, inkompatible Interfaces, mehrfach implementierte Queries, nicht vorhandene Parent-States, doppelt definierte States, nicht vorhandene Initiale States sowie inkompatible Typen.

Auf dem ASG werden diverse Fehlerprüfungen durchgeführt. So werden die Verbindungen der Komposition auf Zyklen überprüft und allenfalls wird ein Fehler ausgegeben. Innerhalb einer State Machine werden Zustände, welche nicht erreicht werden, gefunden und als Warnung ausgegeben. Weiter wird sichergestellt, dass alle Queries auf ein Input Interface verbunden sind und die Implemetationen der Queries den Zustand der Komponente nicht verändern.

Als Output produziert der Transformer C-Sourcecode. Die Gründe dafür sind die einfachere Generierung sowie die Plattformunabhängigkeit von C- gegenüber Assemblercode. Ausserdem ist C universell einsetzbar, da es für praktisch alle Mikrocontroller einen C Compiler gibt. Weiter ist das Debuggen und somit die Fehlersuche einiges einfacher, wenn der zu untersuchende Code in C ist. Einen Output in einer "höheren" Sprache ist ebenfalls nicht geeignet, da dadurch die Benutzbarkeit für Mikrocontroller verloren geht. Das Erstellen einer Library, welche in einer beliebigen Sprache benutzt werden kann, ist ebenfalls am einfachsten wenn der Code als C Source vorliegt.

Pro Interface wird eine C Headerdatei generiert. Diese spezifiziert eine Struktur mit Funktionspointern, welche den Funktionen entsprechen. Ein Beispiel ist in Listing 14 zu sehen.

Listing 14: Interface CalcActn.rzy als C Headerdatei. Die Rizzly Datei dazu ist in Listing 15 zu finden.

```
1  #ifndef __CalcActn_h__
2  #define __CalcActn_h__
3
4  typedef struct
5  {
6    void (*evt_showValue)( void *instance, const int n );
7    void (*evt_noOpError)( void *instance );
8  } __ifa_CalcActn_t;
9
10 #endif
```

Für jede Komponente wird eine Header- sowie eine Sourcedatei generiert. In der Headerdatei wird eine Struktur für das Interface definiert. Weiter ist eine Struktur mit allen Elementen enthalten, welche zu der Speicherung des Zustands benötigt werden oder

Listing 15: Das Interface CalcActn.rzy.

```
1 interface CalcActn
2
3 function showValue( n: Integer )
4 function noOpError()
```

der Verlinkung der Komponente dienen. Listing 16 zeigt die C Headerdatei einer State Machine Komponente. Auch sind darin immer die Pointer für die Callback-Funktionen gespeichert. Falls es sich um eine Composite Komponente handelt, werden die Strukturen der einzelnen Komponenten in die Headerdatei geschrieben. Handelt es sich jedoch um eine State Machine Komponente, so wird der Zustand sowie die Zustandsvariablen geschrieben. Die C Sourcedatei enthält den Code für die Verbindungen im Falle einer Komposite Komponente.

Da C keine Möglichkeit bietet, State Machines native zu definieren, kann eine State Machine Komponente nicht direkt übersetzt werden. Die hierarchische State Machine wird in eine "normale" State Machine transformiert. Mit "normal" ist hier eine State Machine gemeint, welche keine Hierarchie besitzt, alle zusammengesetzten Zustände entfallen. Transitionen, welche solch einen Zustand als Source-Zustand definiert haben, werden zu allen Blatt-Zustand kopiert. Entry/Exit Funktionen werden in die Transitionen kopiert. Die Zustands-lokalen Variablen bleiben enthalten. Pro Ereignis wird eine C Funktion generiert. Darin ist ein grosser Switch-Case enthalten, welcher anhand des Zustands den Code der entsprechenden Transition ausführt. Listing 17 zeigt den generierten Code solch einer Funktion.

3.3. Backend

Als Backend dient GCC, welcher alle C Files in eine Library kompiliert. Da GCC Extensions für die Initialisierung von Strukturen verwendet werden, können die Dateien nicht mit einem anderen Compiler kompiliert werden. Dagegen ist es möglich, die temporären C Files zu behalten und diese für ein anderes Target zu kompilieren.

Zur einfacheren Verwendung der Library in einem FreePascal Programm wird ein Pascal Modul generiert, welches die Funktionsprototypen sowie benötigte Typen enthält.

Listing 16: C Interface der Komponente Calc.rzy (Ausschnitt). Die Komponente Calc.rzy ist in Listing 21 zu finden.

```
typedef struct
{
  __ifa_CalcActn_t action;
} __ifa_Calc_t;

typedef struct
{
  const __ifa_Calc_t *callback;
  void *context;
  struct {
    int state;                // Zustand
    union {                   // Alle hoechsten Zustaende
      struct {                // Zustand "Top"
        int leftNum;          // Variablen des Zustands "Top"
        int prevOp;
        union {               // Alle Kinder-Zustaende von "Top"
          struct {            // Zustand "ParseNum"
            int num;          // Variablen des Zustands "ParseNum"
          } ParseNum;
        };
      } Top;
    };
  } stor;
} __cmp_Calc_t;

void __evt_Calc_cmd_clear( __cmp_Calc_t *instance );
void __evt_Calc_cmd_operand( __cmp_Calc_t *instance, const int op
    );
void __evt_Calc_cmd_number( __cmp_Calc_t *instance, const int n )
   ;
```

Listing 17: C Code der Komponente Calc.rzy (Ausschnitt). Der relevante Code der Komponente Calc.rzy ist in Listing 18 dargestellt.

```
1  void __evt_Calc_cmd_number( __cmp_Calc_t *instance, const int n )
2  {
3    switch( instance->stor.state )
4    {
5      case ST_StartNum:
6      {
7        instance->stor.Top.ParseNum.num = n;
8        instance->callback->action.evt_showValue( instance->context
             , n   );
9        instance->stor.state = ST_ParseNum;
10       break;
11     }
12     case ST_ParseNum:
13     {
14       instance->stor.Top.ParseNum.num = ((instance->stor.Top.
             ParseNum.num*10)+n);
15       instance->callback->action.evt_showValue( instance->context
             , instance->stor.Top.ParseNum.num   );
16       instance->stor.state = ST_ParseNum;
17       break;
18     }
19   }
20 }
```

Listing 18: Rizzly Code des in Listing 17 dargestellten C Codes.

```
1  transition StartNum to ParseNum by cmd.number( n: Integer ) do
2    num := n
3    action.showValue( n )
4  end
5
6  transition ParseNum to ParseNum by cmd.number( n: Integer ) do
7    num := num * 10 + n
8    action.showValue( num )
9  end
```

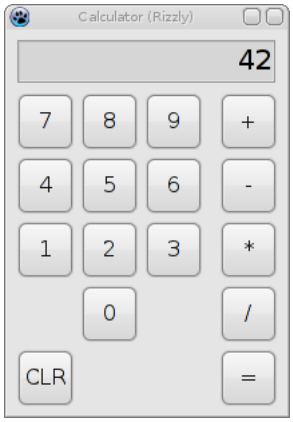
4. Beispiele

Im Folgenden werden anhand verschiedener Beispiele die vielfältigen Anwendungsgebiete von Rizzly aufgezeigt. Die erwähnten Listings zu den Beispielen sind im Anhang enthalten, der Code aller Beispiele kann von `http://www.n.ethz.ch/~ursf/download/rizzly/examples.zip` heruntergeladen werden.

4.1. Rechner

Grafische Oberflächen sind reaktive Systeme, deshalb lassen sich diese gut mit Rizzly umsetzten. Ein einfacher Rechner wurde von Java[2] nach FreePascal/Lazarus portiert. Dieselbe Funktionalität wurde auch mit Rizzly umgesetzt. Abbildung 4 zeigt die Oberfläche des Rechners. Die Zahlen 0 bis 9 lösen ein "number" Ereignis aus, + - * / = ein "operand" Ereignis und CLR das "clear" Ereignis.

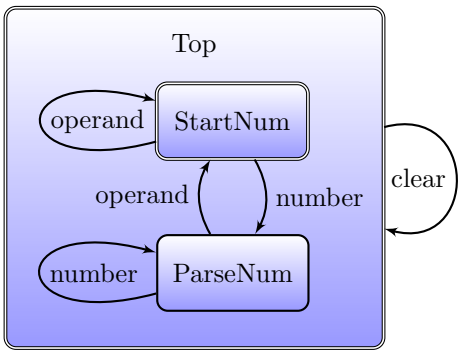
Abbildung 4: Das Benutzerinterface des Rechners.



Listing 20 zeigt die FreePascal-Version, Listing 21 das Rizzly Pendant und Abbildung 5 schliesslich die State Machine dazu. Da in Rizzly die Zustände explizit definiert und Ereignisse je nach Zustand unterschiedlich behandelt werden, ist der Code in Rizzly einiges übersichtlicher geworden.

[2] Original von `http://www.leepoint.net/notes-java/examples/components/calculator/calc.html`

Abbildung 5: Die State Machine des Rechners.



4.2. Förderband

Dieses Beispiel zeigt verschiedene Aspekte von Rizzly. Darunter fallen hierarchische State Machines, Kompositionen, Queries und mehr.

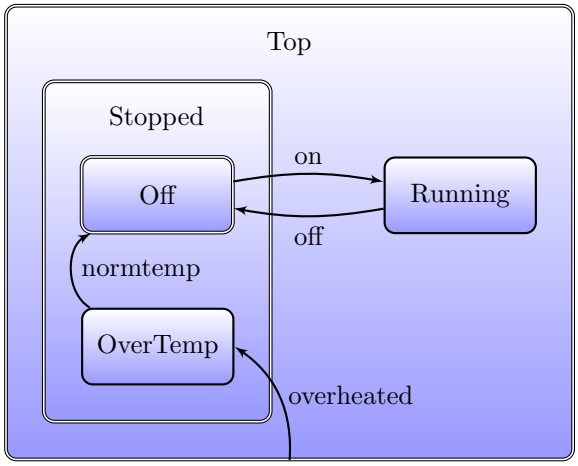
Die Anwendung ist ein Förderband welches Kisten transportiert. Dabei läuft das Förderband nur, wenn mindestens eine Kiste auf dem Förderband ist. Ein Sensor am Anfang des Förderbandes meldet das Auflegen einer Kiste. Ein anderer am Ende meldet das Entnehmen. Der Motor des Förderbandes verfügt ausserdem über eine automatische Übertemperaturabschaltung. Sobald der Temperatursensor die Übertemperatur meldet, schaltet der Motor aus. Solange die Temperatur nicht im Normalbereich ist, kann der Motor nicht wieder angeschaltet werden. Nach einer Übertemperaturabschaltung muss das Förderband so lange manuell bedient werden, bis keine Kiste mehr auf dem Förderband ist[3].

Die Motorsteuerung ist in Abbildung 6 dargestellt, in Listing 22 ist der Code dazu. Der Zustand Top wird für die overheated Transition benötigt. Da keine andere Transition das Ereignis overheated benutzt, wird immer in den Zustand OverTemp gewechselt wenn das Ereignis overheated eintritt.

Die Logik des Förderbandes zeigt Abbildung 7, Listing 23 zeigt den Code. Auf der obersten Ebene kann zwischen dem automatischen und manuellen Modus umgeschaltet werden. Im manuellen Modus läuft das Förderband immer, im automatischen nur dann, wenn eine Kiste auf dem Förderband ist. Dazu zählt die Variable items die Anzahl Kisten auf dem Förderband. Das Ereignis load inkrementiert items, free dekrementiert.

[3] Die Übertemperatur ist wahrscheinlich entstanden weil zu viele Kisten auf dem Förderband waren. Wenn nun Kisten manuell entfernt werden hat die Steuerung keine Ahnung, wie viele Kisten auf dem Förderband verbleiben.

Abbildung 6: Motor mit Übertemperatur-Abschaltung



Die Zusammenschaltung der Komponenten zeigt Abbildung 8 und Listing 24 zeigt den Code. Das Interface motorState der Komponente Motor wird nicht benutzt. Da das Interface nur eine Query beinhaltet ist auch so die gesamte Funktionalität von Motor gewährleistet.

Schliesslich zeigen die Listings 25, 26, 27 und 28 die verwendeten Interfaces.

Abbildung 7: Förderband mit manueller Bedienung

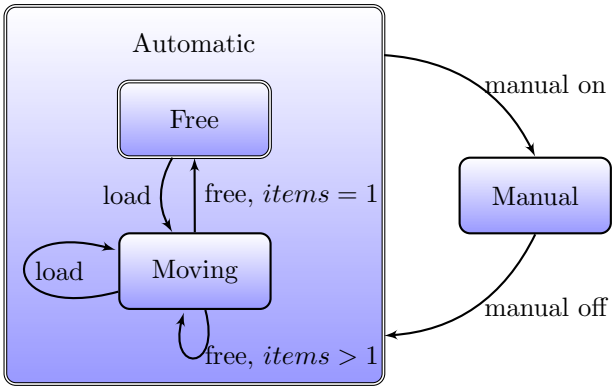
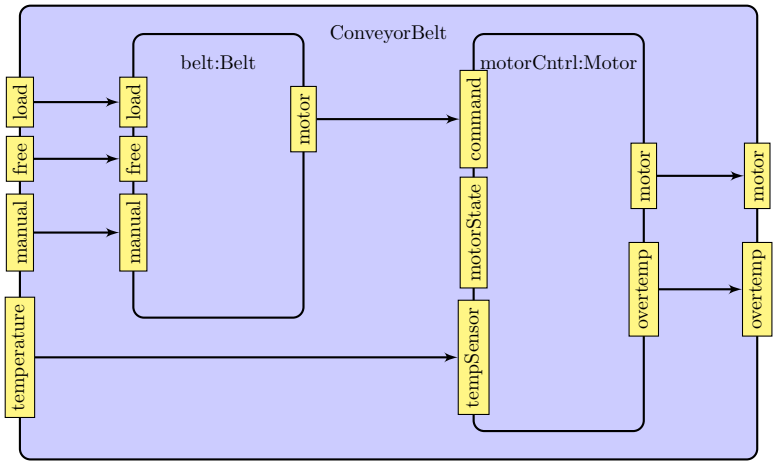


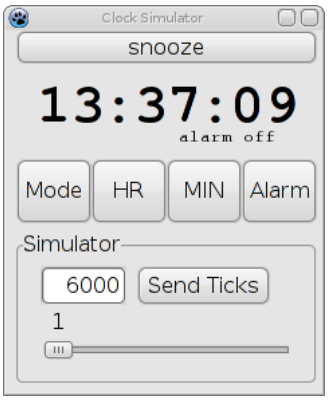
Abbildung 8: Zusammenschaltung des Motors und des Förderbandes



4.3. Wecker

Dieses Beispiel beschreibt die komplette Implementation eines Weckers. Die Funktionalität besteht aus der Anzeige der Uhrzeit, Einstellen der Uhrzeit, Einstellen der Weckzeit, dreifache Wiederholung eines einminütigen Alarms wenn die Weckzeit erreicht wurde, Aktivierung/Deaktivierung des Weckers und Snooze Knopf um den Alarm 4 Minuten zu unterdrücken.

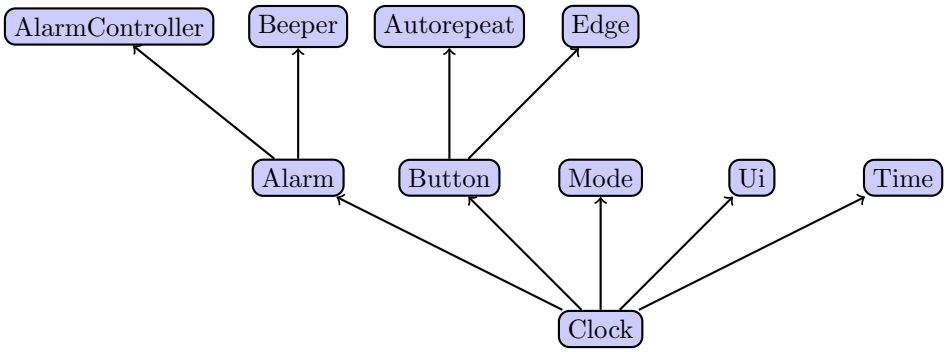
Abbildung 9: GUI des Simulators des Weckers. Mit Mode kann zwischen der Anzeige der Uhrzeit, Einstellen der Uhrzeit und Einstellen der Weckzeit umgeschaltet werden. Mit HR und MIN können die Stunden resp. Minuten eingestellt werden. Alarm aktiviert/deaktiviert den Alarm.



Um die Funktionalität zu testen wurde ein Simulator, dessen GUI in Abbildung 9 dargestellt ist, erstellt. Die Ereignisse der Buttons werden direkt zu dem von Rizzly generierten Code geschickt. Die Information für die Anzeige kommt ebenfalls direkt von dem generierten Code. Nicht sichtbar ist ein Timer, welcher alle 10 Millisekunden einen Tick generiert. Der Simulator stellt also nur das GUI zur Verfügung. Die gesamte Logik wurde mit Rizzly realisiert.

Der Wecker besteht aus 21 Source Dateien, davon 11 Interfaces, 7 State Machines und 3 Composite Komponenten. Abbildung 10 zeigt den Importgraph. Die Blätter entsprechen den State Machines und die inneren Knoten den Composite Komponenten. Dabei wird auch eine weiteren Stärke von Rizzly ersichtlich. Wie in Abbildung 11 zu sehen, sind die Komponenten Alarm, Button, Mode, Ui und Time stark voneinander abhängig, es besteht jedoch keine direkte Abhängigkeit zwischen ihnen. Die einzigen Abhängigkeiten zwischen Komponenten bestehen dann, wenn eine Composite Komponente andere Komponenten verwendet.

Abbildung 10: Import-Graph des Wecker Beispiels. Die Interfaces sind wegen der Übersichtlichkeit nicht dargestellt.



Die Zuständigkeiten der Komponenten sind wie folgt realisiert:

Clock ist die oberste Komponente und realisiert den gesamten Wecker.

Time zählt die aktuelle Uhrzeit anhand der Ticks und löst jede neue Sekunde ein Ereignis aus.

Ui Kontrolliert das Benutzerinterface. Im Normalfall zeigt es die Uhrzeit an, durch eine Modus-Umschaltung kann eine Zeit manipuliert werden.

Mode kontrolliert den Modus für das Ui und stellt sicher, dass die eingestellte Zeit an die entsprechende Komponente (Alarm oder Time) weitergeleitet wird.

Button dekodiert die Tasten Ereignisse "Pressed" und "Released" zu einem "Click" Ereignis.

Autorepeat Wiederholt das Click Ereignis automatisch wenn man länger auf einer Taste bleibt.

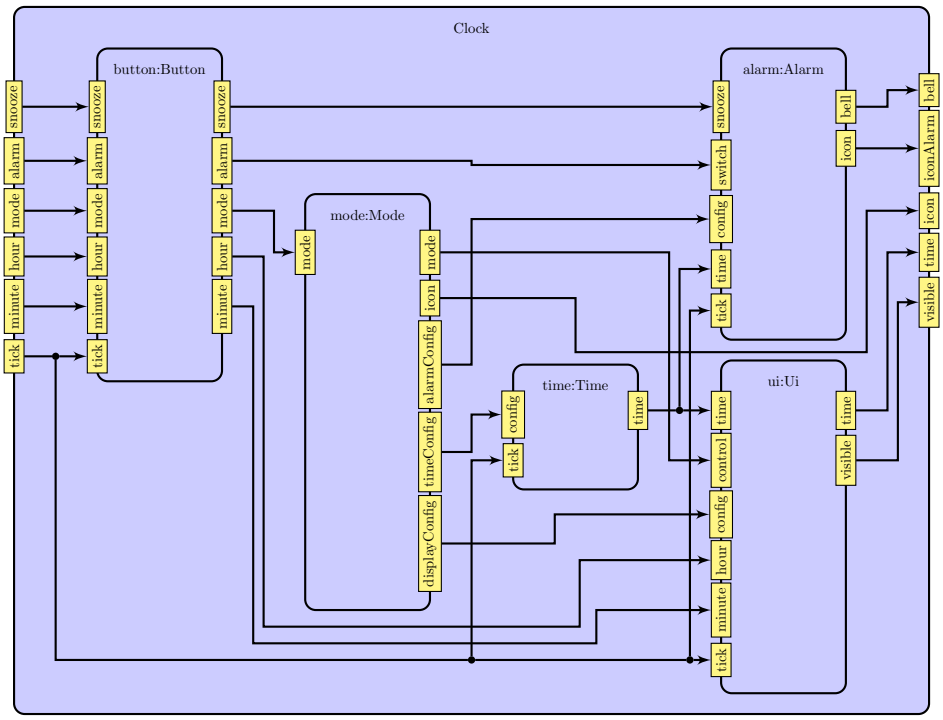
Edge implementiert eine Kantendetektion. In diesem Fall generiert es aus einem "Pressed" Ereignis einen "Click" Ereignis

Alarm implementiert die Weck-Funktion.

AlarmController löst den Alarm für eine Minute (oder weniger wenn der Snooze Button getätigt wurde) aus, wenn die Weckzeit erreicht wurde und der Alarm aktiv ist. Danach pausiert er für 4 Minuten und wiederholt den Alarm. Dies geschieht 3 mal.

Beeper generiert einen Takt für den Summer des Weckers wenn der Alarm aktiv ist.

Abbildung 11: Komponenten und Verbindungen aus der Datei Clock.rzy.



4.4. Computerspiel

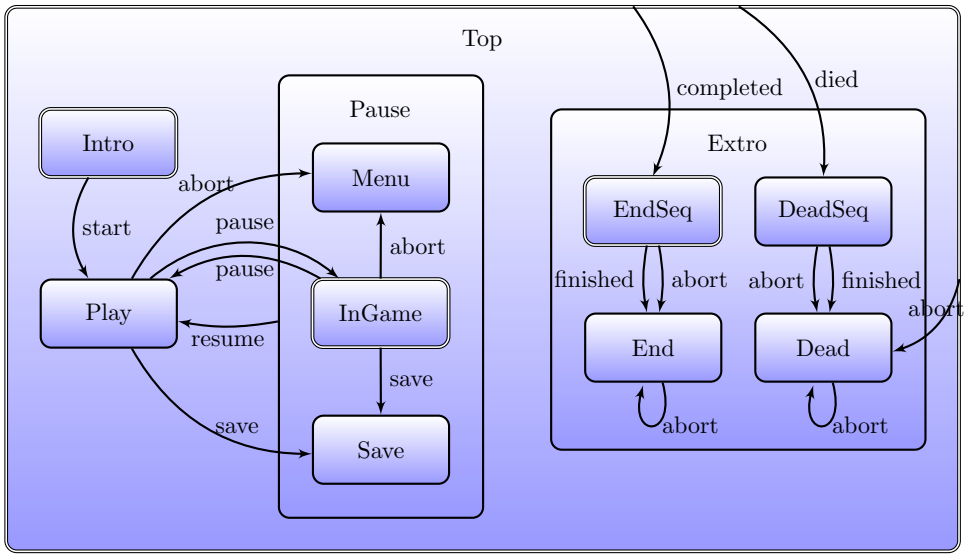
In Computerspielen werden oft State Machines verwendet, um das Verhalten von künstlichen Gegnern zu beschreiben, wie auch um den gesamten Spielablauf zu steuern [13, 9, 19]. So benutzt auch das Spiel Cavitation Velocity [6] mehrere State Machines. Im Spiel geht es darum, in einem "Unterwasser-Flugzeug" gegnerische Ziele zu zerstören. Stationäre und mobile Gegner versuchen den Spieler daran zu hindern. State Machines wurden z.B. für das Verhalten des Spiels verwendet. Darin ist beschrieben, wie sich das Spiel verhält, wenn der Benutzer in das Menü wechselt, das Spiel pausiert oder die Mission erfolgreich abgeschlossen hat. Das Verhalten der mobilen Gegner ist ebenfalls als State Machine realisiert. Dabei ist der Gegner entweder im Zustand Angriff, falls er zu nahe an den Spieler kommt geht er in den Zustand Ausweichen, um nach einer gewissen Zeit wieder anzugreifen.

Die erwähnte State Machine, welche das Verhalten des Spiels definiert, wurde auf Rizzly umgeschrieben. Abbildung 12 zeigt die State Machine, Listing 29 den Code. Das Verhalten der Gegner konnte im Rahmen dieser Arbeit nicht in Rizzly umgesetzt werden. Das hauptsächliche Problem lag darin, dass diese State Machine nicht

auf Ereignisse reagiert, sondern anhand zyklisch aufgerufenen Berechnungen Transitionen durchführt. Für eine Implementation mit Rizzly hätten grosse Teile des Spiels umgeschrieben werden müssen.

Allgemein bestanden in diesem Beispiel drei Probleme die eine Portierung auf Rizzly erschweren. Diese sind, (1) dass die Funktionalität von Rizzly noch nicht genug mächtig ist (z.B. um Matrix Operationen durchzuführen), (2) dass asynchrone Verbindungen fehlen und die Zusammenschaltung von Komponenten deshalb sehr eingeschränkt ist und (3) dass die jetzige Implementation des Spiels nicht durchgängig mit Ereignissen arbeitet.

Abbildung 12: State Machine des Spiels Cavitation Velocity. Es gibt drei verschiedene Möglichkeiten, das Spiel zu pausieren: die "normale" Pause (InGame), speichern des Spiels (Save) und ein Wechsel zum Menu (Menu). Das Spiel wird je mit einer Sequenz beendet, abhängig ob man die Mission erfüllt hat (EndSeq) oder ob man gestorben ist (DeadSeq).



5. Fazit

In dieser Semesterarbeit wurde eine Syntax definiert, welche erlaubt, State Machines direkt in einer Source-Datei zu beschreiben. Der bestehende, experimentelle Rizzly Compiler wurde um diese Funktionalität erweitert. Anhand von Beispielen aus den Gebieten Embedded-Systeme, GUI und Spiele wurde der Syntax sowie die Implementation getestet.

Während der Arbeit konnten vielseitige Erkenntnisse gewonnen werden. Gegenüber einer Implementierung von Hand können in Rizzly die State Machines übersichtlicher beschrieben werden. Die Gründe dafür sind vielfältig. So bewirkt das explizite definieren der Zustände eine Kapselung der Funktionalität. Entry und Exit Funktionen müssen nur einmal geschrieben werden und sind direkt dem betreffenden Zustand zugeordnet. Da Variablen immer im Scope eines Zustands leben, können sie nicht fälschlicherweise aus einem anderen Zustand referenziert werden.

Hierarchische State Machines können mit Rizzly direkt im Source Code umgesetzt werden. Die dadurch möglich gewordenen default-Transitionen tragen viel zu der Übersichtlichkeit sowie zur einfacheren und schnelleren Erstellung bei.

Da der Compiler das Modell der State Machine versteht und verarbeitet, kann er mehr und bessere Fehlerüberprüfungen vornehmen. In der aktuellen Implementation werden Verletzungen der Run to Completion Semantik erkannt, unerreichbare Zustände werden detektiert und es wird sichergestellt, dass Queries keine Seiteneffekte haben.

Alles in allem zeigte sich während dieser Arbeit der Nutzen einer State Machine-Implementation auf Programmiersprachenebene. Es macht den Code übersichtlicher, die Ereignisbehandlung wird natürlicher und durch den hierarchischen Aufbau lässt sich eine Reduktionen des Codes erreichen.

6. Ausblick

Anhand der Umsetzung der Beispiele zeigten sich Gebiete für mögliche Erweiterungen. Die Übersichtlichkeit gegenüber einer Implementation von Hand steigert sich zwar, jedoch bleibt es ein Graph, welcher in einer linearen Form definiert wird. Die einzige sinnvolle Alternative ist eine grafische Repräsentation des Programms. Ein rein grafischer Editor ist jedoch nicht die Lösung. In solch einem können nie alle Informationen dargestellt werden. Eine sinnvolle Erweiterung wäre ein Programm, welches die in Rizzly erstellten State Machines grafisch darstellen kann.

Mit dem Wecker in Kapitel 4.3 wurde ein etwas grösseres Beispiel komplett in Rizzly erstellt, allerdings zeigte sich anhand des Spiels in Kapitel 4.4, dass asynchrone Verbindungen unerlässlich sind. Mit diesen wären auch Zyklen in der Komposition erlaubt. Mit asynchronen Verbindungen bietet sich auch ein Backend an, welches Code für einen Modell-Checker, z.B. Promela/Spin, generiert.

Es zeigte sich ebenfalls, dass in Rizzly die noch fehlenden Konstrukte wie Enumeratoren, Arrays und Loops implementiert werden müssen. Werden State Machines grösser, so wäre es praktisch, wenn sich Sub-State Machines in separate Dateien auslagern lassen. Ebenfalls wäre eine Transition-To-History (shallow history und deep history in UML) wünschenswert.

Transitionen werden in Rizzly als eigene Einheit deklariert. Da sie die einzige Möglichkeit sind den Zustand zu ändern, bietet es sich an, Pre- und Postconditions bei den Transitionen zu definieren. Syntaktisch würde dies wie folgendermassen aussehen: Im Moment wird in Rizzly ein Kommentar wie in C eingefügt. Die Idee ist jedoch, dass es keinen eigentlichen Kommentar gibt, sondern vielmehr eine Methode, um Metadaten im Sourcecode einzubetten. Ein Rizzly Compiler müsste nur die Metadaten interpretieren die ihn interessieren, andere kann er ignorieren. Als Metadaten bieten sich z.B. folgende an: Pre- und Postconditions, Source-Dokumentation, zusätzliche Informationen für das Backend und genereller Kommentar. Listing 19 zeigt mögliche Metadaten. Wird dies umgesetzt, so muss genau spezifiziert werden, zu welchem Teil des Source-Codes welcher Kommentar gehört.

Der Compiler generiert den C Code so, dass jede Komponente einzeln übersetzt werden könnte. Dies ist nicht nur unnötig, es benötigt zur Laufzeit auch zu viel RAM. Hier liegt viel Potential für Optimierungen, z.B. kann die Verbindung zwischen Komponenten statisch im Code enthalten sein. Weiter kann man sich vorstellen, dass die benutzten Komponenten einer Composite Komponente zu einer neuen Komponente zusammengesetzt werden (Produkt aller erreichbaren Zustände). Dadurch reduziert man die Tiefe des Call-Stacks und potentiell auch die Anzahl der Zustände. Dies ist dann sinnvoll, wenn nur wenige Instanzen einer Komponente bestehen. Es wird vermutet, dass dies bei den vorgeschlagenen Anwendungsgebieten der Fall ist.

Listing 19: Metadaten in Rizzly

```
1  //require n >= 0                    // Precondition
2  //ensure  result = n!               // Postcondition
3  ///Berechnet die Fakultaet von n    // Source Dokumentation
4  //gcc attribute(hot)                // Information fuer gcc-Backend
5  function fac( n: Integer ):Integer
6    result : Integer = 1
7    while n > 0 do                    // normaler Kommentar
8      result := result * n
9      n      := n - 1
10   end
11 end
```

6.1. Arbeiten

Die Firma B&R Industrie Automation AG ist ein Hersteller von Automatisierungstechnik, insbesondere von Steuerungssystemen und deren Entwicklungsumgebung. Es besteht das Interesse, während einer Masterarbeit Rizzly in die zukünftige, modellbasierte Entwicklungsumgebung aufzunehmen. Es wird im Moment an einer Lösung basierend auf der CIP Methode [4] gearbeitet. Rizzly wäre eine zweite Implementation. Einerseits würde dies die Erweiterbarkeit der Entwicklungsumgebung demonstrieren, anderseits würde es gegenüber der CIP Methode eine textuelle Entwicklung erlauben. Eine Transformation eines in Rizzly erstellten Programms nach CIP und eventuell zurück ist ebenfalls denkbar.

Das Interesse des Autors liegt an einer sinnvollen Anwendung mit Rizzly. Aus diesem Grund werden die Anstrengungen dahin gehen, dass die fehlenden Sprachkonstrukte implementiert werden, sowie in eine bessere C Code Generierung.

7. Literatur

[1] C. Angelov u. a. “Reconfigurable State Machine Components for Embedded Applications”. In: *Software Engineering and Advanced Applications, 2008. SEAA '08. 34th Euromicro Conference*. 2008, S. 51 –58. DOI: 10.1109/SEAA.2008.38.

[2] Frédéric Boussinot. “Reactive C: An Extension of C to Program Reactive Systems”. In: (1991).

[3] Lars Dölle und Heike Manns. *Styx*. URL: http://speculate.de/.

[4] Hugo Fierz. “The CIP Method: Component- and Model-Based Construction of Embedded Systems”. In: (1999).

[5] Martin Fowler. *http://martinfowler.com/bliki/InversionOfControl.html*. Juni 2005. URL: http://martinfowler.com/bliki/InversionOfControl.html.

[6] Urs Fässler. *Cavitation Velocity*. 2008. URL: http://cavitationvelocity.origo.ethz.ch.

[7] David Gay u. a. “The nesC language: A holistic approach to networked embedded systems”. In: *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. ACM, 2003, S. 1–11. ISBN: 1-58113-662-5. DOI: http://doi.acm.org/10.1145/781131.781133.

[8] Philipp Haller und Martin Odersky. “Event-Based Programming Without Inversion of Control”. In: *Modular Programming Languages* (2006), S. 4–22.

[9] Ki-Duk Kwon und Koo-Rack Park. “Behavior Control of Intelligent Multi NPCs Using Vickrey Auction System and Hierarchical Finite State Machine”. In: *SICE-ICASE, 2006. International Joint Conference*. Okt. 2006, S. 3821 –3826. DOI: 10.1109/SICE.2006.314669.

[10] S. Prochnow und R. von Hanxleden. “Comfortable Modeling of Complex Reactive Systems”. In: *Proc. Design, Automation and Test in Europe DATE '06*. Bd. 1. 2006, S. 1–2. DOI: 10.1109/DATE.2006.243970.

[11] Golo Roden. *Event Based Components: Architektur von Software in neuem Licht*. Mai 2011. URL: http://heise.de/-1240993.

[12] Miro Samek. *Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems*. 2. Aufl. Newnes, 2008.

[13] Brian Schwab. “Implementation Walkthrough of a Homegrown 'Abstract State Machine' Style System in a Commercial Sports Game”. In: *AIIDE*. 2008.

[14] M. Sirjani u. a. “Extended Rebeca: a component-based actor language with synchronous message passing”. In: *Application of Concurrency to System Design, 2005. ACSD 2005. Fifth International Conference on*. Aug. 2005, S. 212 –221. DOI: 10.1109/ACSD.2005.12.

[15] Jean-Ferdy Susini. “The reactive programming approach on top of Java/J2ME”. In: *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*. ACM, 2006, S. 227–236. ISBN: 1-59593-544-4. DOI: http://doi.acm.org/10.1145/1167999.1168037.

[16] Jennifer Tenzer und Perdita Stevens. “Modelling Recursive Calls with UML State Diagrams”. In: *Fundamental Approaches to Software Engineering*. Hrsg. von Mauro Pezzè. Bd. 2621. Lecture Notes in Computer Science. 10.1007/3-540-36578-8_10. Springer Berlin / Heidelberg, 2003, S. 135–149. URL: http://dx.doi.org/10.1007/3-540-36578-8_10.

[17] Wikipedia. *Rational Rhapsody — Wikipedia, Die freie Enzyklopädie*. [Online; Stand 3. Juni 2011]. 2011. URL: http://de.wikipedia.org/w/index.php?title=Rational_Rhapsody&oldid=88324251.

[18] Wikipedia. *UML state machine — Wikipedia, The Free Encyclopedia*. [Online; accessed 3-June-2011]. 2011. URL: http://en.wikipedia.org/w/index.php?title=UML_state_machine&oldid=429010904.

[19] Z.D. Wu. “Performance modeling of multicast groups for multiplayer games in peer-to-peer networks”. In: *Distributed Simulation and Real-Time Applications, 2005. DS-RT 2005 Proceedings. Ninth IEEE International Symposium on*. Okt. 2005, S. 105 –112. DOI: 10.1109/DISTRA.2005.35.

[20] Guoqing Yang u. a. “SmartC: A Component-Based Hierarchical Modeling Language for Automotive Electronics”. In: *Control, Automation, Robotics and Vision, 2006. ICARCV ’06. 9th International Conference on*. Juni 2006, S. 1 –6. DOI: 10.1109/ICARCV.2006.345212.

8. Abbildungsverzeichnis

9. Listings

A. Code der Beispiele

A.1. Rechner

Listing 20: Umsetzung des Rechners mit FreePascal (Ausschnitt).

```
var
  displayField : TStaticText;
  startNumber  : Boolean;
  previousOp   : Char;
  currentTotal : Integer;

procedure actionClear();
begin
  startNumber           := true;
  displayField.Caption  := '0';
  previousOp            := '=';
  currentTotal          := 0;
end;

procedure opHandler( cmd: Char );
var
  displayNumber   : Integer;
begin
  if startNumber then begin
    actionClear();
    displayField.Caption := 'ERROR - No operator';
  end else begin
    startNumber    := true;
    displayNumber  := StrToInt( displayField.Caption );

    case previousOp of
      '=': currentTotal := displayNumber;
      '+': currentTotal := currentTotal + displayNumber;
      '-': currentTotal := currentTotal - displayNumber;
      '*': currentTotal := currentTotal * displayNumber;
      '/': currentTotal := currentTotal div displayNumber
          ;
    end;

    displayField.Caption := IntToStr( currentTotal );
    previousOp := cmd;
  end;
end;

procedure numHandler( digit: Integer );
begin
  if startNumber then begin
    displayField.Caption := IntToStr( digit );
    startNumber          := false;
  end else begin
    displayField.Caption := displayField.Caption +
        IntToStr(digit);
  end
end;

procedure clearHandler;
begin
  actionClear();
end;

procedure init( display: TStaticText );
begin
  displayField := display;
  actionClear();
end;
```

Listing 21: Umsetzung des Rechners mit Rizzly (Ausschnitt).

```
Top = state()
  initial
  var
    leftNum : Integer
    prevOp  : Integer
end

StartNum = state( Top )
  initial
end

ParseNum = state( Top )
```

```
13    var
14      num      : Integer
15  end
16
17  transition Top to Top by cmd.clear() do
18    action.showValue( 0 )
19  end
20
21  transition StartNum to StartNum by cmd.operand( op:
        Integer ) do
22    action.noOpError()
23  end
24
25  transition ParseNum to StartNum by cmd.operand( op:
        Integer ) do
26    if prevOp = 0 then            //  =
27      leftNum := num
28    ef prevOp = 1 then            //  +
29      leftNum := leftNum + num
30    ef prevOp = 2 then            //  -
31      leftNum := leftNum - num
32    ef prevOp = 3 then            //  *
33      leftNum := leftNum * num
34    ef prevOp = 4 then            //  /
35      leftNum := leftNum / num
36    end
37    action.showValue( leftNum )
38    prevOp := op
39  end
40
41  transition StartNum to ParseNum by cmd.number( n: Integer
        ) do
42    num := n
43    action.showValue( n )
44  end
45
46  transition ParseNum to ParseNum by cmd.number( n: Integer
        ) do
47    num := num * 10 + n
48    action.showValue( num )
49  end
```

A.2. Förderband

Listing 22: Source des Motors mit Übertemperatur-Abschaltung

```
1   component Motor
2
3   import
4     iface.Switch
5     iface.SwitchQuery
6     iface.Temperature
7
8   interface
9     input
10      command        : Switch
11      motorState     : SwitchQuery
12      tempSensor     : Temperature
13
14    output
15      motor          : Switch
16      overtemp       : Switch
17
18  implementation hfsm
19
20  Top = state()
21    initial
22  end
23
24  Stopped = state( Top )
25    initial
26    entry
27      motor.off()
28    end
29    query motorState.isOn():Boolean
30      return false
31    end
32  end
33
34  Running = state( Top )
35    entry
36      motor.on()
```

```
  end
  query motorState.isOn():Boolean
    return true
  end
end

Off = state( Stopped )
  initial
end

OverTemp = state( Stopped )
  entry
    overtemp.on()
  end
  exit
    overtemp.off()
  end
end

transition Top to OverTemp by tempSensor.overheated() end

transition OverTemp to Off by tempSensor.normtemp() end

transition Off to Running by command.on() end

transition Running to Off by command.off() end
```

Listing 23: Source des Förderbands mit manueller Bedienung

```
component Belt

import
  iface.Switch
  iface.GenericEvent

interface
  input
    load          : GenericEvent
    free          : GenericEvent
    manual        : Switch

  output
    motor         : Switch

implementation hfsm

Automatic = state()
  initial
end

Free = state( Automatic )
  initial
  entry
    motor.off()
  end
end

Moving = state( Automatic )
  var
    items   : Integer
  entry
    motor.on()
  end
end

Manual = state()
  entry
    motor.on()
  end
end

transition Free to Moving by load.evt() do
  items := 1
end

transition Moving to Moving by load.evt() do
  items := items + 1
end

transition Moving to Moving by free.evt() if items > 1 do
  items := items - 1
end
```

```
54
55 transition Moving to Free by free.evt() if items = 1 end
56
57 transition Automatic to Manual by manual.on() end
58
59 transition Manual to Automatic by manual.off() end
```

Listing 24: Source der zusammenschaltung des Motors und des Förderbandes

```
1  component ConveyorBelt
2
3  import
4    iface.Switch
5    iface.GenericEvent
6    iface.Temperature
7    Belt
8    Motor
9
10 interface
11   input
12     load        : GenericEvent
13     free        : GenericEvent
14     manual      : Switch
15     temperature : Temperature
16
17   output
18     motor       : Switch
19     overtemp    : Switch
20
21 implementation composition
22
23 component
24   belt          : Belt
25   motorCntrl    : Motor
26
27 connection
28   temperature         -> motorCntrl.tempSensor
29   load                -> belt.load
30   free                -> belt.free
31   manual              -> belt.manual
32
33   belt.motor          -> motorCntrl.command
34
35   motorCntrl.motor    -> motor
36   motorCntrl.overtemp -> overtemp
```

Listing 25: Das Interface GenericEvent

```
1 interface iface.GenericEvent
2
3 function evt()
```

Listing 26: Das Interface Switch

```
1 interface iface.Switch
2
3 function on()
4 function off()
```

Listing 27: Das Interface SwitchQuery

```
1 interface iface.SwitchQuery
2
3 query isOn():Boolean
```

Listing 28: Das Interface Temperature

```
1 interface iface.Temperature
2
3 function normtemp()
4 function overheated()
```

A.3. Computerspiel

Listing 29: Code der State Machine des Menus.

```
1 component Game
2
3 import
4   iface.GameCommand
5   iface.UserCommand
```

```
  iface.Switch
  iface.GameState
  iface.Message

interface
  input
    cmdGame      : GameCommand
    cmdUser      : UserCommand
    stateQuery   : GameState

  output
    sound        : Switch
    msg          : Message


implementation hfsm

Top = state()
  initial
end

Intro = state( Top )
  initial
  query stateQuery.gameState():Integer
    return 0
  end
end

Play = state( Top )
  query stateQuery.gameState():Integer
    return 1
  end
end

Pause = state( Top )
  entry
    sound.off()
  end
  exit
    sound.on()
  end
end

Extro = state( Top )
end


InGame = state( Pause )
  initial
  query stateQuery.gameState():Integer
    return 5
  end
end

Menu = state( Pause )
  query stateQuery.gameState():Integer
    return 6
  end
end

Save = state( Pause )
  query stateQuery.gameState():Integer
    return 7
  end
end


EndSeq = state( Extro )
  initial
  entry
    msg.msgNr( 0 )
  end
  query stateQuery.gameState():Integer
    return 2
  end
end

End = state( Extro )
  query stateQuery.gameState():Integer
    return 3
  end
end
```

```
88
89  DeadSeq = state( Extro )
90    entry
91      msg.msgNr( 1 )
92    end
93    query stateQuery.gameState():Integer
94      return 2
95    end
96  end
97
98  Dead = state( Extro )
99    query stateQuery.gameState():Integer
100     return 4
101   end
102 end
103
104
105 transition Intro to Play by cmdGame.start() end
106
107 transition Play to InGame by cmdUser.pause() end
108
109 transition InGame to Play by cmdUser.pause() end
110
111 transition Play to Menu by cmdUser.abort() end
112
113 transition Play to Save by cmdUser.save() end
114
115 transition Pause to Play by cmdUser.resume() end
116
117 transition InGame to Menu by cmdUser.abort() end
118
119 transition InGame to Save by cmdUser.save() end
120
121 transition Top to EndSeq by cmdGame.completed() end
122
123 transition EndSeq to End by cmdGame.finished() end
124
125 transition Top to DeadSeq by cmdGame.died() end
126
127 transition DeadSeq to Dead by cmdGame.finished() end
128
129 transition EndSeq to End by cmdUser.abort() end
130
131 transition DeadSeq to Dead by cmdUser.abort() end
132
133 transition End to End by cmdUser.abort() end
134
135 transition Dead to Dead by cmdUser.abort() end
136
137 transition Top to Dead by cmdUser.abort() end
```

B. Syntax von Rizzly

Listing 30: Syntax von Rizzly in der Definition von Styx.

```
Language rizzly

Regular Grammar

  ign Ign         = ' \t\n\r'
  com Comment     = "/*" ({Byte}-({Byte}"*/"{Byte})) "*/"
  com LineComment = "//" ({Byte}-({Byte}("\n"){Byte})) "\n"
  let Byte        = '\00' .. '\ff'
  tok String      = '\'' { Byte - '\'' } '\''
  tok Tok         = ',()+*/.;[]'
  tok Mul         = ':<>=-'+
  tok Int         = ('0'..'9')+
  tok Id          = ('a'..'z'|'A'..'Z')('a'..'z'|'A'..'Z'|'0'..'9'|'_')*

Context Free Grammar

; ---- Header ----

start File
  :cmp: FileComponent
  :ifa: FileInterface

let FileComponent
  :all: "component" Designator Import Properties Interface Implementation

let FileInterface
  :all: "interface" Designator Import InterfaceDecl

let Import
  :nil:
  :imp: "import" DesignatorSeq

let Interface
  :nil:
  :use1: "interface" IfaceProvide
  :use2: "interface" IfaceUse
  :use3: "interface" IfaceUse IfaceProvide

let IfaceUse
  :o: "input" IfaceSeq

let IfaceProvide
  :o: "output" IfaceSeq

let Implementation
  :elm:  "implementation" "elementary" ImplElem
  :comp: "implementation" "composition" ImplComp
  :fsm:  "implementation" "hfsm" ImplHfsm
  :nat:  "implementation" "native"

let IfaceSeq
  :ifs: Iface IfaceSeq
  :ifa: Iface

let Iface
  :ifa: Id ":" Designator

let Properties
  :act: "active"
  :nil:

; ---- Implementation HFSM ----

let ImplHfsm
  :sta: StateDecl ImplHfsm
  :tra: TransDecl ImplHfsm
  :nil:

let StateDecl
  :sta: Id "=" "state" "(" ParentState ")" StateFlags StateVar StateFuncSeq "end"

let StateFlags
  :ini: "initial"
  :nil:
```

```
let StateFuncSeq
  :seq: StateFunc StateFuncSeq
  :nil:

let StateFunc
  :nom: Func
  :int: "entry" StmtSeq "end"
  :ext: "exit" StmtSeq "end"

let StateVar
  :var: VarDeclBlock
  :nil:

let ParentState
  :nam: Id
  :nil:

let TransDecl
  :tra: "transition" Id "to" Id "by" FuncHeader Guard
      TransBody "end"

let TransBody
  :bdy: "do" StmtSeq
  :nil:

let Guard
  :grd: "if" Expr
  :nil:

let FuncHeader
  :hdr: Designator Params

; ---- Implementation Composition ----

let ImplComp
  :both: "component" ComponentSeq "connection"
      ConnectionSeq
  :comp: "component" ComponentSeq
  :con:  "connection" ConnectionSeq
  :nil:

let ComponentSeq
  :seq: ComponentUse ComponentSeq
  :end: ComponentUse

let ComponentUse
  :cmp: Id ":" Designator

let ConnectionSeq
  :seq: Connection ConnectionSeq
  :end: Connection

let Connection
  :con: Designator "->" Designator

; ---- Interface Declaration ----

let InterfaceDecl
  :ifd: IfaceMessageDeclSeq

let IfaceMessageDeclSeq
  :seq: IfaceMessageDecl IfaceMessageDeclSeq
  :end: IfaceMessageDecl

let IfaceMessageDecl
  :evt: "function" Id Params
  :msg: "message" Id Params
  :que: "query" Id Params ReturnType

let ReturnType
  :nil:
  :typ: ":" Type

let ImplElem
  :elm: ImplTopSeq
  :nil:

let ImplTopSeq
  :var: VarDeclBlock ImplTopSeq
  :van: VarDeclBlock
  :fuc: Func ImplTopSeq
```

```
  :fun: Func

let VarDeclBlock
  :var: "var" VarDeclSeq

let VarDeclSeq
  :seq: Param VarDeclSeq
  :fin: Param

let Func
  :evt: "function" Designator Params StmtSeq "end"
  :msg: "message" Designator Params StmtSeq "end"
  :que: "query" Designator Params ReturnType StmtSeq "end
      "

let Params
  :nil: "(" ")"
  :par: "(" ParamSeq ")"

let ParamSeq
  :pse: Param ";" ParamSeq
  :psq: Param

let Param
  :par: VarSeq ":" Type

let VarSeq
  :vas: Id "," VarSeq
  :var: Id

let StmtSeq
  :stm: Stmt StmtSeq
  :nil:

let Stmt
  :fun: FunCall
  :if:  IfStmt
  :ass: Assignment
  :ret: Return

let FunCall
  :all: Designator ParamList

let ParamList
  :emp: "(" ")"
  :seq: "(" ActualParamSeq ")"

let ActualParamSeq
  :seq: Expr "," ActualParamSeq
  :end: Expr

let IfStmt
  :if:  "if" Expr "then" StmtSeq IfEfElse "end"

let IfEfElse
  :ef:   "ef" Expr "then" StmtSeq IfEfElse
  :else: "else" StmtSeq
  :nil:

let Assignment
  :ass: Designator ":=" Expr

let Return
  :val: "return" Expr

; ---- common ----

let Type
  :sim: Id
  :arr: Id "[" "]"

let DesignatorSeq
  :dsq: Designator DesignatorSeq
  :dse: Designator

let Designator
  :dsg: Id "." Designator
  :dsf: Id

; ---- Expressions ----

let Expr
```

```
  :compare:     SimpleExpression CompOp SimpleExpression
  :simple:      SimpleExpression

let SimpleExpression
  :weak:        SimpleExpression WeakOp Term
  :term:        Term

let Term
  :strong:      Term StrongOp Factor
  :factor:      Factor

let Factor
  :brace:       "(" Expr ")"
  :unary:       UnaryOp Factor
  :const:       Constant
  :var:         Designator
  :query:       QueryCall

let QueryCall
  :cal: Designator "(" ")"

let UnaryOp
  :pos:  "+"
  :neg:  "-"
  :not:  "not"

let StrongOp
  :mul:  "*"
  :fdiv: "/"
  :idiv: "div"
  :mod:  "mod"
  :and:  "and"

let WeakOp
  :add: "+"
  :sub: "-"
  :or:  "or"

let CompOp
  :equ: "="
  :low: "<"
  :gre: ">"
  :leq: "<="
  :geq: ">="
  :neq: "<>"

let Constant
  :bool:    BooleanConst
  :int:     Int
  :string:  String

let BooleanConst
  :true:  "true"
  :false: "false"
```